INSTITUTO
UNIVERSITÁRIO
DE LISBOA

AirQual-IOT – Air Quality Monitoring and Data Analysis for Indoor Environments

Henrique Vieira de Sousa

Master in Telecommunications and Computer Engineering

Supervisor:
PhD Octavian Adrian Postolache, Full Professor
Iscte – Instituto Universitário de Lisboa

Co-Supervisor:
PhD Joaquim Gabriel Magalhães Mendes, Full Professor,
FEUP – Faculdade de Engenharia da Universidade do Porto

September, 2025

TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

AirQual-IOT – Air Quality Monitoring and Data Analysis for Indoor Environments

Henrique Vieira de Sousa

Master in Telecommunications and Computer Engineering

Supervisor:
PhD Octavian Adrian Postolache, Full Professor
Iscte – Instituto Universitário de Lisboa

Co-Supervisor:
PhD Joaquim Gabriel Magalhães Mendes, Full Professor,
FEUP – Faculdade de Engenharia da Universidade do Porto

September, 2025

All we have to decide is what to do
with the time that is given to us.
- J.R.R. Tolkien.

Acknowledgment

This thesis would have not been possible without the guidance and support of several people, who directly or indirectly helped me through this process.

That being said, I would like to express my deepest gratitude to my supervisors, Professor Octavian Adrian Postolache and Professor Joaquim Magalhães Mendes, for all the contributions, understanding and for providing me part of their immense knowledge.

To Instituto de Telecomunicações - IUL, for the continuous support, and to Faculdade de Engenharia da Universidade do Porto, for the opportunity.

To ISCTE – Instituto Universitário de Lisboa, for all the amazing years and for enabling me to meet new and interesting people along my journey.

From those people, I can call some friends and to whom I would like extend a warm thank you. To Filipe Gonçalves, or should I say "Carlitos", Duarte Casaleiro (or Dudu), Miguel Romana, Oleksandr Kobelyuk, Pedro Arsénio and Rodrigo Guerreiro, for all the help and companionship throughout the years. I know I can always count on you.

Lastly, I would like to mention and thank Diana Ribeiro, a very wonderful person that I have the chance to call not only my best friend but also my lover, for all the unconditional love and support she provided me during these times and for always being there when I most needed it. Without her, this work would have taken a lot longer. Thank you for being present through everything.

Resumo

O aumento da urbanização e a redução das áreas verdes têm representado um risco para a nossa saúde respiratória devido ao aumento da poluição do ar que respiramos.
Esta dissertação apresenta o desenvolvimento de um sistema de monitorização da qualidade do ar e análise de dados para ambientes interiores. Para ajudar nesta questão, propomos um sistema composto por uma placa e sensores que proporcionam ao utilizador a capacidade de detetar impurezas no ar e avaliar a qualidade do ar no ambiente desejado. Este sistema é composto por uma placa baseada em ESP32 e diversos sensores que permitem a captura e o encaminhamento de dados. Para isso, foi necessário escrever um firmware para a placa, a fim de fornecer a capacidade de se comunicar com os sensores e, em seguida, encaminhar os dados coletados, através do uso de Wi-Fi, para um contentor Docker, onde são processados e exibidos ao utilizador. No contentor de visualização, é possível informar o utilizador, através do uso de alarmes e mensagens automatizadas, sobre a qualidade do ar atual.
Os resultados desta dissertação mostram a placa com o firmware personalizado a fornecer dados dos sensores e a encaminhá-los através do uso de Wi-Fi para processamento em um ambiente auto-hospedado, de forma económica a longo prazo.

Palavras Chave: *Sensores*, *Sistemas embebidos*, *Qualidade de ar*, *Índices de qualidade do ar*, *Monitorização contínua*

Abstract

The increase in urbanisation and reduction of green areas has posed a risk to our respiratory health due to the increased pollution of the air we breath.
This dissertation showcases the development of an Air Quality Monitoring and Data Analysis for Indoor Environments. To assist in this issue, we propose a system composed of a board and sensors, providing the user with the capability of detecting Air impurities and the ability to evaluate the air quality in their desired environment.
This system is composed of a board based on ESP32 and diverse sensors which allow for the capture and forwarding of data. To achieve this, it was necessary to write firmware for the board to provide it with the ability to communicate with the sensors and then forward the data gathered, through the use of Wi-Fi, to a Docker container where it was processed and then displayed to the user. In the visualisation container, it is possible to inform the user through the usage of alarms and automated messages of the current air quality.
The results of this dissertation showcase the board with the custom firmware providing sensor data and forwarding it through Wi-Fi for processing on a self-hosted environment whilst being cost-efficient in the long term.

KEYWORDS: *Sensors*, *Embedded Systems*, *Air quality*, *Air quality indexes*, *Continuous monitoring*

Contents

List of Figures

List of Tables

List of Acronyms

ACME	Automatic Certificate Management Environment
AQI	Air Quality Index
CA	Certificate Authority
C_2H_6O	Ethyl
C_3H_8	Propane
$C_6H_5CH_3$	Toluene
C_6H_6	Benzene
CH_2O	Formaldehyde
CH_4	Methane
CLI	Command-line interface
CO_2	Carbon Dioxide
CO	Carbon Monoxide
COPD	Chronic Obstructive Pulmonary Diseases
CORS	Cross-origin resource sharing
CSS	Chirp Spread Spectrum
DB	Data Base
DNS	Domain Name Server
DSR	Design Science Research
eCO_2	Estimated Carbon Dioxide
ED	End Devices
EEA	European Environment Agency
EPA	Environmental Protection Agency
EU	European Union
FEUP	Faculty of Engineering of the University of Porto
GW	Gateway
H_2	Hydrogen
IoT	Internet of Things
LPG	Liquefied Petroleum Gas
LPWAN	Low-power wide-area network
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
MAC	Medium Access Control
MCU	Microcontroller Unit
MOS	Metal Oxide Semiconductor

MTU	Maximum Transmission Unit
MCU	Microcontroller Unit
NH_3	Ammonia
NS	Network Server
NO_2	Nitrogen Dioxide
NO_X	Nitrogen Oxides
NoSQL	Not Only Structured Query Language
O_3	Ozone
PM	Particulate Matter
PAN	Personal Area Network
PSRAM	Pseudostatic Random Access Memory
RDBMS	Relational Database Management System
RQ	Research Question
SnO_2	Tin(IV) oxide
SF	Spread Factor
SO_2	Sulfur Dioxide
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
TSDB	Time-Series Database
TVOC	Total Volatile Organic Compounds
UI	User Interface
USA	United States of America
VM	Virtual Machine
VOC	Volatile Organic Compound
VPN	Virtual Private Network
VPS	Virtual Private Server
WHO	World Health Organization

CHAPTER 1

Introduction

The bad Air Quality seriously affects the health of a population, as it can potentiate many diseases and limit the breathe ability. Taking this into consideration, the European Union (EU) has introduced several standards to regulate air quality and its pollutants thus making imperative to continuos monitoring the environment.

This chapter will start by presenting the motivation behind this dissertation in sub-chapter 1.1, followed by the context of it in sub-chapter 1.2, explain the end objective which we want to attain in sub-chapter 1.3 and our research approach in sub-chapter 1.4. Finally, the outline of this dissertation's structure can be found in sub-chapter 1.5.

1.1. Motivation

The air we breathe in can be classified as indoor and outdoor and it's becoming gradually more polluted due to the increase of the population and production processes. With the known fact that people spend 90% of their time indoors [1], we will be focusing on that in this dissertation, using a bedroom as our environment site. With that, the motivation for this work stems from providing the ability to sense these gases in indoor environments, at a low cost, and inform the user to take the proper actions.

The usage of sensors, with the help of software, allows us to detect the notorious pollutants that plague the air, by processing and presenting the data as information regarding the current air composition.

1.2. Context

Air quality refers to the level of hygiene in a particular area or environment [2] and is a key factor to a healthy life. Air is composed of 77.3% nitrogen, 20.7% oxygen, 1% water vapour and inert gases [3]. However, air pollution is one of the greatest environmental risks as it has adverse effects on the population's health.

The pollution can cause or aggravate existing respiratory illnesses in the population, such as Asthma - 300 million people are affected by it worldwide -, Chronic Obstructive Pulmonary Diseases (COPD) [4], bronchitis and viral infections [5].

The main pollutants in urban areas are Particulate Matter (PM), Carbon Monoxide (CO), Ozone (O_3), Nitrogen Dioxide (NO_2), Sulfur Dioxide (SO_2) [1] and Volatile Organic Compounds (VOCs). PM has different sizes: super coarse for particles with diameters larger than $10\mu m$, coarse particles with diameters between 2.5 - $10\mu m$ (PM_{10}), fine particles with diameters less than $2.5\mu m$ ($PM_{2.5}$) and ultra-fine particles, with diameters less than $0.1\mu m$ [5], [6]. Moreover, cancerogenic substances such as Formaldehyde (CH_2O)

can also be found in the air. However, the EU has very strict regulations on its usage [7], whilst other countries, like China [8], need to worry about its presence in the air.

Symptoms of exposure to PM include persistent coughing, sore throat, chest tightness and burning eyes, which can trigger asthma or lead to death [5].
This is due to the fact that PM deposition can occur within any of the three regions of the respiratory system: the extrathoracic area, tracheobronchial tree or the alveolar spaces [5], which can be critical if an individual has pre-existing respiratory diseases.

Ozone smog is another concern for air pollution since it forms when Nitrogen Oxides (NO_X) - which is formed through fuel combustion alongside SO_2 and CO [6] - are enhanced by sunlight and heat [5], [6].

VOCs, as the name implies, are a group of organic chemicals which evaporate into the air at room temperature. They can occur in natural or artificial sources, encompassing plants, fungi, animals, industrial products [9] and cleaning products, cooking practices, fragrances and fresheners, hobbies and at-home work behaviours [10]. As Kyle L. Alford et al said in their work - "VOC exposure was associated with asthma and related symptoms, including wheezing and throat irritation." [11].

High concentration of Carbon Dioxide (CO_2) in the air leads to an increase in pollen production and that, coupled with NO_X and O_3, can trigger the release of allergens from pollen granules, leading to an increased concentration of pollen in the air. Allergic diseases, such as asthma or rhinitis, are very common in children and young adults and are triggered by this increase in pollen production [5], as these particles are present as PM.

Due to air exchange methods, such as diffusion, convection, pressure equalization or stack effect, polluted air is able to infiltrate the indoor environment. **T**his environment may have people who could, for example, wear perfumes and clothing with pet dander [4], which may bring in some contaminants, such as allergens. It may also be affected by some activities that take place in said environment, such as cleaning - dusting, sweeping, vacuuming - or simply the movement of people [5], which will inevitably increase the pollutants.

This becomes more worrying when humidity is also present, since humidity whose values are above 60% will make the air harder to breathe and turns the air stagnant, trapping said pollutants and allergens, which can trigger asthma attacks [1] and worsen the condition.

1.3. Objectives

The aim of this dissertation is the development of a prototype of an Internet of Things (IoT) system that is capable of detecting Temperature, Humidity, CO_2, VOC, PM of all sizes and incorporate, in the future, the other gases, with a small footprint in energy

consumption, the ability to send alerts to its users to inform them of current air quality, all whilst being a low cost implementation. To achieve this, we will be resorting to small form factor boards and virtualized services, that will be used to gather and present our data.

Based on the factors exposed, this dissertation aims to answer the following Research Questions (RQs):

- RQ1: Can we produce a viable device that can be simple and easy to use while being cost-affordable and have low power consumption?
- RQ2: Can this device be used to inform and send alerts to the user about the current air composition?

1.4. Research Methodology

The methodology Design Science Research (DSR) - Figure 1.1 - approaches research by defining a problem. This is then expanded onto developing and evaluating existing and new solutions and propagating the results.

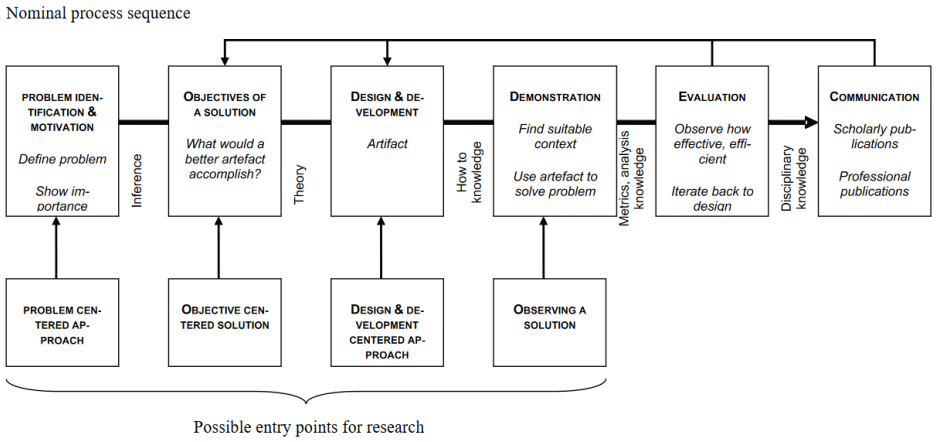


FIGURE 1.1. Design Science Research process model. [12]

In this first chapter of the dissertation, we introduce the first two key steps: identifying the problem (or opportunity for improvement) and outlining the main objectives and research questions. The next three steps - design and development, demonstration, and evaluation - involve building the solution (or prototype), describing its structure and functionality, and then testing and analysing it to assess its effectiveness.

Although there are four possible beginnings in the process, researchers aren't required to begin with problem identification and motivation. However, since this dissertation takes a problem centred approach, we will start by defining the problem and then follow the steps in the order they are typically presented.

Finally, we share the outcomes of this work through this thesis.

1.5. Outline

This section depicts the structure of this dissertation and its organisation as well as a brief preview of each chapter.

Chapter 1 introduces the motivation behind this work and the goals to achieve.

Chapter 2 aims to introduce complex and existing concepts and provide comprehensive usage.

Chapter 3 provides extensive exposure to the process of creating our custom board, achieved through the usage of Arduino IDE and Docker.

Chapter 4 depicts the results obtained from our custom solution, such as values, alerts, power consumption and ease of reproduction.

Chapter 5, lastly, showcases the conclusions of this work and considerations for future work.

CHAPTER 2

State of the Art

In this Chapter, we will describe the state of the art related to all of the research done on the topics that will be covered in this dissertation: issues brought upon by air pollution, sensors explored and researched, communication protocols used for IoT and, finally, the databases for storing data.

2.1. Air Pollution

As we previously stated, the air we breathe can be contaminated with pollutants from many sources which can influence our well-being to the point of sickness. The air is mainly polluted from human activities, such as farming, industrial processes and fossil fuel combustion [6], and this extends to our environment. The pollutants generated from these activities are PM, CO, O_3, NO_2 and SO_2 [1], [13], and VOCs. This has led to 3.2 million people dying prematurely from illnesses which are caused by air pollution in households. From those, 19% are from COPD [14], as it remains the fourth leading cause of death worldwide and is the eighth cause of poor health [15], 6% from lung cancer, 21% from lower respiratory infections such as pneumonia and, in 2019, it led to the loss of around 86 million people [14].

Fortunately, the World Health Organization (WHO) has made available some guidelines, in 2021, [16] that have recommended safe concentrations levels of each pollutant. These offer guidance to governments allowing them to understand and take actions to reduce the issues that originate the air pollution [16], as seen in Figure 2.1.

Pollutant	Averaging Time	2005 AQGs	2021 AQGs
$PM_{2.5}$, $\mu g/m^3$	Annual	10	5
	24-hour[a]	25	15
PM_{10}, $\mu g/m^3$	Annual	20	15
	24-hour[a]	50	45
O_3, $\mu g/m^3$	Peak season[b]	-	60
	8-hour[a]	100	100
NO_2, $\mu g/m^3$	Annual	40	10
	24-hour[a]	-	25
SO_2, $\mu g/m^3$	24-hour[a]	20	40
CO, mg/m^3	24-hour[a]	-	4

FIGURE 2.1. Air quality Guidelines. [16]

With such serious problems originating from exposure to indoor air pollution, there have been efforts to reduce indoor contamination through quality monitoring using an air quality monitoring system [2]. These systems will employ several types of sensors, such as

particle sensors to measure PM in the air, gas sensors that are based on electrochemistry [2] or other gas sensors like Tin Oxide Sensors - Figaro and Nemoto -, but they require some processing of their data to avoid cross sensitivities and temperature and humidity dependence.

There exists commercial items that provide user with information regarding the air quality around them, such as Apple, Netatmo, Evehome, Awair and uHoo. In the table 2.1, its showcased their ability to detect the notorious gases.

TABLE 2.1. Comparison of solutions offered.

	Netamo	Evehome	Awair	Uhoo	Our's
Temperature	x	x	x	x	x
Humidity	x	x	x	x	x
CO_2	x		x	x	x
O_3				x	
CO				x	
NO_2				x	
SO_2					
VOC		x	x	x	x
$PM_{2.5}$			x	x	x
PM_{10}					x

Netamo offers two products, an indoor air quality monitor that provides insight on the levels of CO_2, Humidity, Temperature and Sound [17], and a smart CO alarm that can only inform the user if carbon monoxide is present in their environment [18].

Evehome has what they call the Eve Room, which will monitor room climate and air quality. The product provides the user with information in the form of a small clock about temperature, humidity and VOC [19].

Awair can track 5 environmental factors, temperature, Humidity, CO_2, TVOC and $PM_{2.5}$ [20].

Uhoo provides a smart air monitor, which is capable of detecting CO_2, $PM_{2.5}$, NO_2, Relative Humidity, Air pressure, CO, Temperature, VOCs and O_3 [21]. Uhoo is, by far, the brand that can provide the most data regarding air quality in an environment. However, through investigation, we found they provide two plans that come with their products, a free and a premium plan. This can limit the users that want to analyse the air in their environment, because even when provided with the plan for a year, which you are granted if you purchase the item, you then have to pay monthly to get the same data [22].

With our proposed solution, users don't have to pay a yearly subscription. Besides that, alerts can be customisable as desired, data visualisation can be done through mobile or website and the data will always be on their device.

2.1.1. Air Quality Index

The Air Quality Index (AQI) is a measure designed to detect the level of air quality in locations [23] by using numerical indicators that depend on the concentration of pollutants. The European Environment Agency (EEA) defines the pollutants for their AQI as being: $PM_{2.5}$,PM_{10}, NO_2, O_3 SO_2 [23], as illustrated by the Figure 2.2, which contains the AQI levels and their respective colour. These levels are determined via hourly concentrations and the bands that determine the category thresholds are linked to health impacts, specifically to the risk factor of mortality due to short-term exposure. It is important to mention that the air quality can be estimated when not all the parameters are present, this is done through methods depending on the pollutant: for $PM_{2.5}$,PM_{10}, NO_2 a difference method is used, for O_3 it is a multiplicative method and none for SO_2 [23].

	Good	Fair	Moderate	Poor	Very poor	Extremely poor
Particles less than 2.5 µm ($PM_{2.5}$)	0-10	10-20	20-25	25-50	50-75	75-800
Particles less than 10 µm (PM_{10})	0-20	20-40	40-50	50-100	100-150	150-1200
Nitrogen dioxide (NO_2)	0-40	40-90	90-120	120-230	230-340	340-1000
Ozone (O_3)	0-50	50-100	100-130	130-240	240-380	380-800
Sulphur dioxide (SO_2)	0-100	100-200	200-350	350-500	500-750	750-1250

FIGURE 2.2. Air Quality Index in µg/m^3. [23]

As presented in the Figure 2.2, there are different values for each pollutant that form a scale for the AQI. As these values increase, their representative colour changes, with the highest value being the poorest quality of air, represented as dark purple, meaning that the air is unfit for any type of activity outdoors for people of a sensitive group. For the AQI issued by Environmental Protection Agency (EPA) [24], as seen in Figure 2.3, it's a conversion of pollutant concentrations to a standard scale.

For the EEA, the calculation of their index depends on the location of the station, if it's a traffic station the concentration of NO_2 and $PM_{2.5}$,PM_{10} or both are used for the

AQI Basics for Ozone and Particle Pollution

Daily AQI Color	Levels of Concern	Values of Index	Description of Air Quality
Green	Good	0 to 50	Air quality is satisfactory, and air pollution poses little or no risk.
Yellow	Moderate	51 to 100	Air quality is acceptable. However, there may be a risk for some people, particularly those who are unusually sensitive to air pollution.
Orange	Unhealthy for Sensitive Groups	101 to 150	Members of sensitive groups may experience health effects. The general public is less likely to be affected.
Red	Unhealthy	151 to 200	Some members of the general public may experience health effects; members of sensitive groups may experience more serious health effects.
Purple	Very Unhealthy	201 to 300	Health alert: The risk of health effects is increased for everyone.
Maroon	Hazardous	301 and higher	Health warning of emergency conditions: everyone is more likely to be affected.

FIGURE 2.3. Air Quality Index in the USA in µg. [24]

calculation. This is due to the SO_2 concentrations being high in localised areas which can distort the local air quality, whilst O_3 levels are very low at these stations. For industrial and background stations, NO_2 and $PM_{2.5}$,PM_{10} or both are used as well as O_3 [23].

2.2. MOS Sensors - Gas and Particulate Matter Sensors

In this section, we will introduce sensors that were used by other authors and their capabilities, which form the first layer of IoT system that is called the Perception Layer or the Sensing Layer. This layer's purpose is to acquire data about its surrounding environment.

These sensors are essential for this dissertation, as they will provide a better understanding of how to properly detect harmful pollutants.

An important thing to reference is that many sensors used by the authors, and in this dissertation, are based on a Metal Oxide Semiconductor (MOS) technology as it's able to detect many gases such as Benzene (C_6H_6), Toluene ($C_6H_5CH_3$), Ethyl (C_2H_6O), Propane (C_3H_8), Hydrogen (H_2), CO_2, Ammonia (NH_3), NO_X, CH_2O and smoke [1], [25], [26]. These sensors are comprised of two units, a heating one and an electrochemical one. For the latter, the most common metal oxide used is Tin(IV) oxide (SnO_2) although others link Zinc, Titanium and Tungsten oxides. For this to work, the heating unit heats the sensing surface and, as the clean air environment has low electrical conductivity, if foreign gases are present, the conductivity will increase.

We will present two tables regarding the pollutant gas sensors. To better understand them, it's important to note that the desired parameters for this system are detection of PM, VOCs, CO, CO_2, O_3, NO_2 and SO_2. The first table - table 2.2 - will have data regarding the sensitivity each sensor has to a determined pollutant and their drawbacks,

provided from their datasheet, and the second table - table 2.3 - will be regarding their relevancy for our environment.

Table 2.2. Sensor's comparison

	Gases Detected	Drawbacks
MQ-4	Methane Natural Gas	Limited range of detection.
MQ-7	Methane Hydrogen Carbon Monoxide Liquifed Petroleum Gas	Needs alternating heating cycles for better CO accuracy; Affected by humidity and temperature; Cross-sensitivity.
MQ-9	Methane Carbon Monoxide Liquefied Petroleum Gas	Affected by humidity and temperature; cross-sensitivity.
MQ-135	Smoke Alcohol Benzene Ammonia Sulphur Dioxide Carbon Dioxide Nitrogen Oxides	Affected by humidity and temperature; Cross-sensitivity.
MQ-138	Toluene Alcohol Propane Benzene Hydrogen Formaldehyde	Affected by humidity and temperature; High cross-sensitivity.
CCS811	VOCs eCO_2	Not real CO_2 measured.
BME280	Temperature Humidity Pressure	-

Table 2.3. Sensor's for our indoor environment

	Indoor Environment
MQ-4	Less ideal unless installed in places with gases (Labs; Kitchens)
MQ-7	Ideal for CO
MQ-9	Ideal for labs industrial kitchens and household kitchens
MQ-135	-
MQ-138	Ideal for VOC
CCS811	Ideal for VOC
BME280	Ideal for basic data

All the MQ sensors mentioned require calibration to operate and suffer from cross-sensitivity. This is achieved through the usage of an algorithm to estimate the tension generated by the gases in the electrochemical unit, outputting the "correct" gas. The MOS sensors are affected by humidity and heat due to their nature. After careful consideration of our requirements, studying the data presented and limited availability, the sensors for detecting gases that will provide the data fit for our model are the CCS811 and BME280. However, the MQ-7, MQ-135 and the MQ-138 are all strong candidates.

2.2.1. MQ-7

The MQ-7 is able to detect CO, Methane (CH_4), Liquefied Petroleum Gas (LPG) and H_2. This makes this sensor useful for detecting CO at Faculty of Engineering of the University of Porto (FEUP)'s environment. However, one of its drawbacks is the need to be set to high heat to burn off contaminants, which in turn will reset the sensor, making it prone to detect CO without hindrances.

2.2.2. MQ-135

The MQ-135 is a gas sensor that is more in tune for general usage and makes it able to detect NO_X, NH_3, C_2H_6O, C_6H_6, CO_2 and smoke [1], [25]. This makes this sensor ideal for our environment, due to its ability to detect many gases. One of the downsides is the inability to provide precise measurements compared to dedicated sensors.

2.2.3. MQ-138

The MQ-138, just like the MQ-135, can detect a variety of gases, which include C_6H_6, $C_6H_5CH_3$, C_2H_6O, C_3H_8, CH_2O and H_2. Due to this, it is the ideal sensor for an environment made up of organic vapors [8], [26]. As our environment has VOCs, the MQ-138 is useful because of its ability to detect them.

2.2.4. CCS811

The CCS811 is able to detect VOCs and eCO_2. This estimate is based on the amount of VOCs present which can be induce errors with high amounts. A slight drawback of this sensor is the necessity to provide it with data for humidity and temperature to calibrate it.

2.2.5. BME280

The BME280 is a sensor that is capable to detect humidity, pressure and temperature with precision. This is ideal for our needs due to the nature of the prototype and avoids having to rely on multiple sensors to obtain the same data.

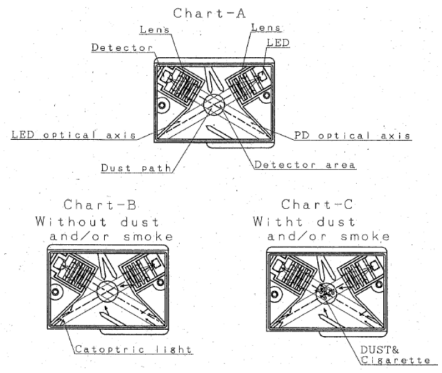
2.2.6. Particle Matter Sensors

The sensors in this section work based on the principle of Light Scattering and they employ an infrared light source, a photodiode and a detection chamber. The light source is always turned on providing a constant steady source of light. When PM passes through

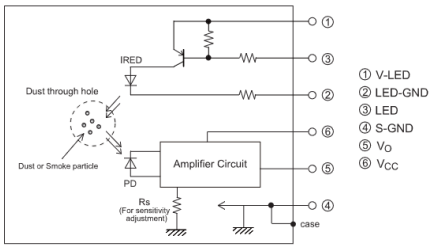
the chamber, the light from the source is reflected off of it, and it bounces inside the chamber which is eventually detected by the photodiode. This technique can be subjected to varied readings depending on the temperature and humidity in the environment where it is located.

2.2.7. Compact Optical Dust Detector - GP2Y1010AUOF

The GP2Y1010AUOF is an optical sensor that allows for the detection of $PM_{2.5}$ in the indoor air. This sensor provides real-time dust concentration monitoring [8]. This can be seen in Figure 2.4a in chart C. Once a constant output is achieved, it is internally amplified through its circuitry, as shown in Figure 2.4b and we are given a signal proportional to the particles in suspension [8], [27].



(A) GP2Y1010AUOF blueprint. [27]



(B) GP2Y1010AU0F internal structure. [28]

FIGURE 2.4. Compact Optical Dust Detector.

2.2.8. SPS30 Particulate Matter sensor

The SPS30 is a PM sensor that also employs the Light Scattering technique [29]. This sensor has three operational modes: idle, sleep and measurement. Since idle and sleep are similar - both aim to reduce sensor energy consumption when not in use -, our focus will be on the measurement mode. Once this mode is on, it works by leveraging the light scattering technique [30].

We can see how the SPS30 works in Figure 2.5. The fan blows in fresh air from the environment and funnels it down a path where the light diode is found. As the particles go through the path, they pass through the laser and the light sensor, which detects them based on the scattered light. With this, we are given a signal proportional to the particles in suspension and, through internal processing of the sensor, we can know which type of particle went through produces an output of the composition.

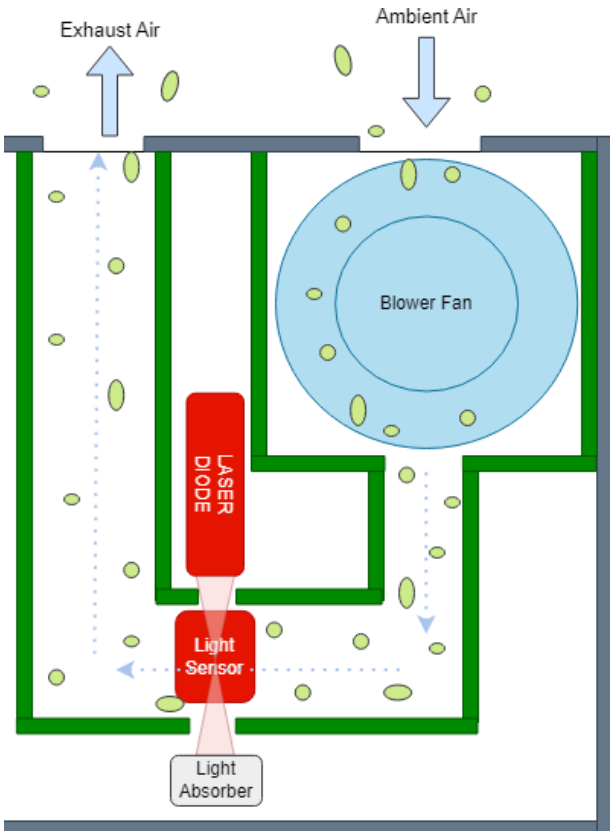


FIGURE 2.5. SPS30 internal working. [31]

2.3. Communication Protocols

In this section, we will present the Network layer of the developed IoT ecosystem.

This layer is about the data transmission and processing and it is responsible for forwarding data that the Perception layer gathers to external services such as databases or visualisation platforms. The communication is achieved by using existing types of network like Personal Area Networks (PANs), Wi-Fi, mobile networks or Low-power wide-area networks (LPWANs).

With the advancements made in IoT and considering the other technologies' limitations, the focus has shifted to the usage of 5G technology and the development of LPWANs making usage of the licensed and unlicensed spectrum bands, achieving lower latency, reducing costs related to hardware, lower energy consumption for batteries and overall usage [32], [33].

LPWANs, such as Long Range (LoRa) and Sigfox, use a star topology allowing devices to connect directly to base stations which promotes a better usage of their energy as they don't have to listen to other devices that relay data through them [33]. An important fact to mention is that LPWANs do not use conventional bandwidths, they instead use sub-GHz bands that have other several benefits: less energy used, less signal attenuation and greater communication range [33].

With this information, it's possible to understand that LPWANs are ideal for bigger indoor environments, for example, large offices or large-scale commercial buildings. As our

dissertation focuses on indoor buildings, like houses and/or apartments, and it's purpose to build an IoT ecosystem to monitor air quality, which sends only sensor information over a short-range network, the usage of LPWANs might be unnecessary. Instead, we can focus on the other types of networks, like PANs, as they are easier to deploy, focused on smaller environments and provide the same characteristics as LPWAN.

2.3.1. Zigbee

Zigbee has been considered by many people in the IoT industry - such as Factory Automation, Agriculture and Precision Farming and Supply Chain and Logistics - as an advanced and the best communication technology. This technology is preferred over Wi-Fi due to its low power consumption, increased security and stability [34]. This technology supports multiple network structures such as Star, Tree and Mesh networks, as shown by Figure 2.6 [35], [36].

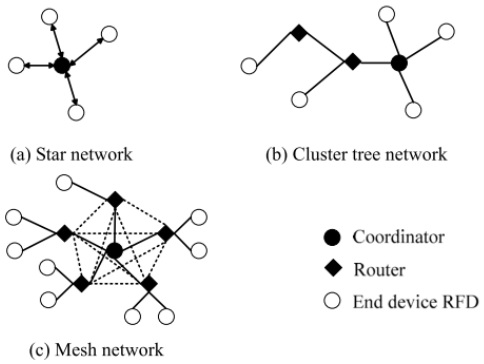


FIGURE 2.6. Zigbee Network Topology. [35]

This allows the network to take a different shape depending on the necessities of it's user, but most of the time the Mesh Network is mostly used and relied upon for having less points of failure and allowing 65000 nodes as Zigbee uses a 16-bit address space for device addressing, and mesh networking allows devices to relay messages over long distances. For a Star network, the number of nodes is considerably small as it's mostly used for small environments as the range is limited by the coordinator's transmission power. In the Cluster Tree network, there is a hierarchical structure where some devices called routes act as intermediaries to relay messages to other devices.

Zigbee can reach 10 to 100 meters in indoor environments, depending on the materials that are around, such as walls and metals. For outdoor environments, this range can extend up to 100 to 300 meters, in line-of-sight conditions.

As this network shares the 2.4GHz band with Wi-Fi, it can be considered a shortcoming, since many homes rely on this default band and do not use the 5GHz bandwidth to its fullest, which can make Zigbee not the most reliable to relay data on this band. To overcome this, it uses two other additional bands, 868MHz and 915MHz, with an effective range of 300 meters [34], [37]. These sub-GHz bands are used in Europe and America, respectively [36].

2.3.2. Z-wave

Z-wave is a wireless protocol created by Zensys and it can be used for many indoor appliances such as cooking, televisions and home security . For a device to be recognised inside a network that uses the Z-wave protocol, it needs to have a Network ID. Another of its key features is that it uses a sub-GHz band to operate - 900MHz to be precise -, and can reach a range of 30 meters, but also transforms the devices in the network into repeaters which further increases this range [37]. This is achieved, since this technology works in a Mesh configuration where each device in the networks sends and receives control commands and uses intermediate nodes to find the best route to reach their destination [38].

2.3.3. LoRa

The LoRa is a LPWAN which uses a proprietary bandwidth in the sub GHz bands. In Europe, the bandwidth is 868 MHz, in North America 915 MHz and 433 MHz in Asia. LoRa uses a Chirp Spread Spectrum (CSS) modulation that provides bidirectional communication and spreads the signal over a wide bandwidth [33], [39]. This technology, with the usage of Spread Factors (SFs), can change the data range, the rate of data transfer and the lifetime of batteries. The SFs found in LoRa range from SF7 to SF12. As the SF increases, so does the transmission range; however, a trade-off is that the data rate gets reduced as well as the battery life [33], [39], [40].

Long Range Wide Area Network (LoRaWAN) is an open standard protocol that uses unlicensed Industrial, Scientific and Medical Frequency bands [29], [41]. It can adjust itself accordingly, based on the data it collects from analysing the metrics to optimise the device's performance. We can say that the LoRa acts on the physical layer and that LoRaWAN acts on the Medium Access Control (MAC) layer [29].

LoRaWAN is made up of three devices: the End Devicess (EDs), which send information via LoRa to their Gateways (GWs), then the information leaves the GW via TCP/IP to reach a Network Server (NS), where it is forwarded to application platforms [29].

2.3.4. Basic Wi-Fi

Wi-Fi is found everywhere, from user's terminals, like laptops and handheld devices, to embedded devices and appliances. Due to this, Wi-Fi plays the role of our method of communication because of it's availability and because power consumption for our node is not a factor. With new protocols, like 802.11g, it provides the ability to invoke Power Save making the Access point buffer data for a node until this one requires it, and 802.11n, where dual-band was introduced, providing the ability to use 2.4GHz and 5GHz simultaneously, depending on the equipment, with 20MHz or 40MHz bandwidth and MIMO capabilities [42]. We can introduce our prototype to any existing Wi-Fi network with ease due to it being abundantly common, easier to setup and without disturbing

any current ecosystem as it only uses 2.4GHz which allows for longer reach at the cost of slightly lower speeds.

2.4. Data Storage

In this section, we will present the Application layer in IoT, as show in the 3-layer presented by the Figure 2.7.

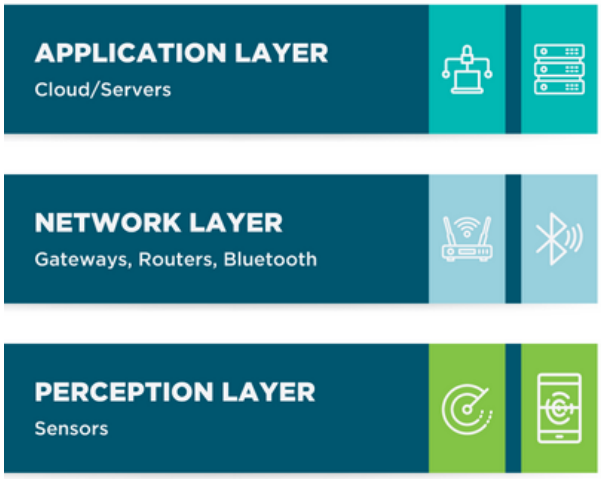


FIGURE 2.7. 3-Layer IoT. [43]

To process and data storage, we can use Relational or Not Only Structured Query Language (NoSQL) DBs. Relational DBs make use of Structured Query Language (SQL) as its a standard programming language for handling data in this type of model and considered the most systematic tool for analysing queries and writing scripts. This comes with the ability to Insert, Query, Update and Delete data, but requires it to be structured as it cannot handle, for example, documents and emails efficiently. This fuelled the creation of the non-relation databases which is used for Big Data [44].

With the conception of NoSQL, it acquired some properties such as non-relational, distributed and horizontally scalable. This means that NoSQL databases do not use Relational Database Management System (RDBMS) principles and do not support SQL as their query language which leads to the loss of the Join operation. Another aspect is that the data can be stored in different servers which allows for the capability to add or remove servers [44], [45]. NoSQL also offers no schema definition before inserting data which means that the following databases can be used:

- Key-Value Store;
- Document Store - JSON, XML, PDF;
- Graph;
- Time Series.

As sensors produce data at every possible time interval, some in the milliseconds, SQL's structure would be extremely inefficient in processing the amount of data that would be flowing in. To overcome this issue, in this dissertation, the usage of NoSQL DBs focused on Time Series was chosen.

2.4.1. NoSQL

In this work, we will be making use of NoSQLDBs. Although there are several DBs options available, the aim is to identify a lightweight and optimised DB, prompting the exploration of both InfluxDB and MongoDB.

MongoDB is more open and more lax in terms of data that can be stored in it. This is, in part, due to the usage of JSON-like format as its schema [46].

InfluxDB is a Time Series DB, offers a schema-less data model and uses specialised query languages called InfluxQL and Flux - we can also use standard SQL, but this does not make it a SQLDB [47].

Due to how our data is being retrieved from the sensors, we have data near real time, which is a characteristic of time series data. This type of data is arranged in a chronological order which helps many sectors - healthcare, industrial and construction [48], [49] - track their work through the collection, processing and analysing their data. Other characteristics, besides its chronological generation, are the high frequency of data collection, the processing and analysis based on time alignment, time difference and others, as well as regularity such as periodic changes [47].

2.4.2. MongoDB

This section will introduce MongoDB, a Document DB, as well as its usage and the research done by other authors, such as R.Aghi et al [46], A. Nayak [50] and M. M. Patil et al [45].

Storing and analysing large amounts of data is becoming mandatory due to the amount of smart equipment and, as we make use of said equipment, we will be required to analyse the data before we can present to users, since we are able to obtain insight into the current environment and gain the ability to calibrate the used sensors, due to some having burnin required. As we expand our system with more nodes, eventually, the time needed to fetch data at a certain point in time will be extremely high and near-like unresponsive using traditional SQL. On account of that, we will be using Time Series DBs in this dissertation. This is done in a way that each data point will be tied to a specific timestamp so that it can retrieve values based on certain time events, such as an hour before the current time, making it a viable candidate for IoT applications and sensor-generated data.

MongoDB, being a NoSQL DB, offers a flexible schema, but also because of how it was conceived, its faster to read and write [50]. MongoDB allows one to input data manually as well as to automatically feed it data [46]. This allows testing to verify if the data is being parsed and retrieved correctly. The same authors, Rajat Aghi et al[46], made a detailed comparison of MongoDB and SQL databases, in terms of insertions, joins and retrievals. The results of this comparison were that SQL databases were faster for simple queries and that MongoDB is more suitable for large and complex queries [45].

2.4.3. InfluxDB

This section will present and explore InfluxDB based on research provided by other authors, such as Q.Liu et al [47] and U. Kayaduman et al [51].

As mentioned before, InfluxDB is a Time-Series Database (TSDB), which is an open source distributed DB that was designed to process high write and query loads [51], [52]. It was developed using Go, a fast programming language, and the primary benefit of it is the ability to aggregate data without manual interference.

We will be using Grafana to visualise and present data to users. Since Grafana supports InfluxDB as a data source [52] - and InfluxDB allows queries in both standard SQL and its native Flux Query Language [47] -, users who are familiar with SQL can adopt the system with minimal learning curve. The Figure 2.8 showcases the top DBs used over the past months and we are able to see that InfluxDB continues to hold the first place with an extremly high score.

Rank			DBMS	Database Model	Score		
May 2025	Apr 2025	May 2024			May 2025	Apr 2025	May 2024
1.	1.	1.	InfluxDB	Time Series, Multi-model	21.78	+0.24	-4.05
2.	↑ 3.	2.	Prometheus	Time Series	6.77	+0.28	-1.65
3.	↓ 2.	3.	Kdb	Multi-model	6.58	+0.09	-0.96
4.	4.	↑ 5.	Graphite	Time Series	4.25	0.00	-0.32
5.	5.	↓ 4.	TimescaleDB	Time Series, Multi-model	3.59	+0.00	-1.05
6.	6.	↑ 9.	QuestDB	Time Series, Multi-model	3.26	+0.13	+0.73
7.	7.	7.	Apache Druid	Multi-model	3.18	+0.13	-0.17
8.	8.	↓ 6.	DolphinDB	Multi-model	2.16	-0.05	-1.98
9.	9.	↑ 10.	GridDB	Time Series, Multi-model	2.00	+0.00	+0.05
10.	10.	↓ 8.	TDengine	Time Series, Multi-model	1.68	+0.02	-0.91

FIGURE 2.8. Top DBs used. [53]

With this stated, we can conclude that the best DB option for this prototype is InfluxDB. This is, in part, due to its practicality to begin and/or transition into it, as well as its algorithm to compress data into small pieces, allowing it to store vast amounts of data in environments that cannot afford a wide array of storage.

2.5. Virtualization

In this section, we will introduce Virtualization. This approach was deemed necessary in case the user has limited computing capabilities and it also provides ease of up-gradability, modularity and transferability. The concept of virtualization has the ability to allocate the resources available while maintaining proper security and availability for the applications. We will make use of Docker as our engine and another machine as our test environment.

2.5.1. Docker

Docker is an open source project that provides faster deployment of applications inside containers written in GO [54], [55].

A container is a lightweight, standardised unit of software. In other terms, it's a smaller Virtual Machine (VM) with very limited amount of resources. They use the Kernel from the Host instead of having their own. Docker expands on that to run processes in isolation, such as CPU, memory, I/O, network and others [54]. A beneficial usage of Docker is the ability to have multiple containers running in the same machine, which in a Virtual Private Server (VPS) is a viable solution as our resources are constrained without spending currency to upgrade. With Docker, we are given the ability to shut down and boot our containers in case of any malfunction or, in the worst case scenario, a breach. This ability can help us update, upgrade or fix containers that are facing problems which can prevent downtimes while promoting safety and modularity, as seen in Appendix A.9.

We installed Docker in a separate virtual machine to be our test environment with limited resources, 6 GB of RAM and 1 Thread from the CPU to push the system to the limits and understand the requirements for a self hosting environment to allow replication for individuals.

CHAPTER 3

System Description

3.1. Hardware Component

This section will present the hardware used to create this prototype. The hardware changed from the initial ones scouted and reviewed, but the goal was still achieved through the usage of other sensors which we will present and discuss.

3.1.1. Embedded System

For this prototype, we required a board capable to handle a Wi-Fi connection, multiple sensors sending data and a CPU to process the data and forward it. To achieve the desired prototype, we used the LILYGO TTGO T-Beam board as it is based on the ESP32, providing seamless integration with code written for Arduino. The board's Microcontroller Unit (MCU) is a dual core running at 240MHz, it is equipped with Wi-Fi and Bluetooth 4.2, 4MB of flash memory, 8MB of Pseudostatic Random Access Memory (PSRAM) and a LoRa antenna with a very high sensitivity -148dBm.

This board, as seen in Figure 3.1, provides a lot of potential for our prototype while only drawing between 0.8W and 1.1W. This can be changed by using a NCR18650B battery, as recommended by LiLyGo themselves.

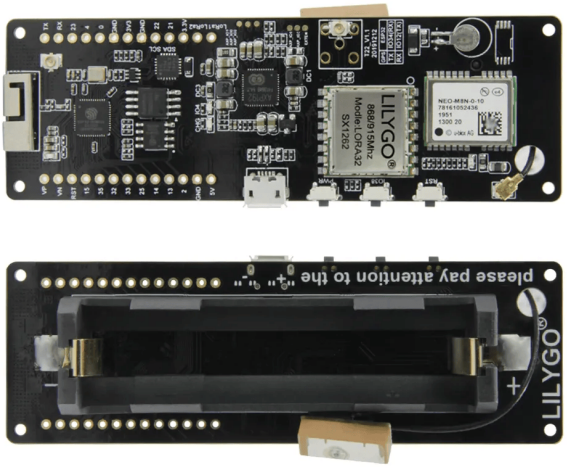


FIGURE 3.1. LILYGO TTGO T-Beam board. [56]

For our prototype, we made use of Pins 13 and 14 as our SDA and SCL pins, as well as the 5V and 3V3 pins to power our sensors as they required different voltages to operate. The sensor SPS30 required a 5V line with pull-up resistors to work, while the other sensor only required a 3V3 line. However, we will dive into more details regarding the sensors in sub-chapter 3.1.2.

Finally, with the sensors, data and power connected to the board, as seen in Figures 3.2 and 3.3, it was necessary to create a firmware for them to interact with each other and send data back to our DB for further processing. This will be discussed and presented in greater detail in sub-chapter 3.2.1.

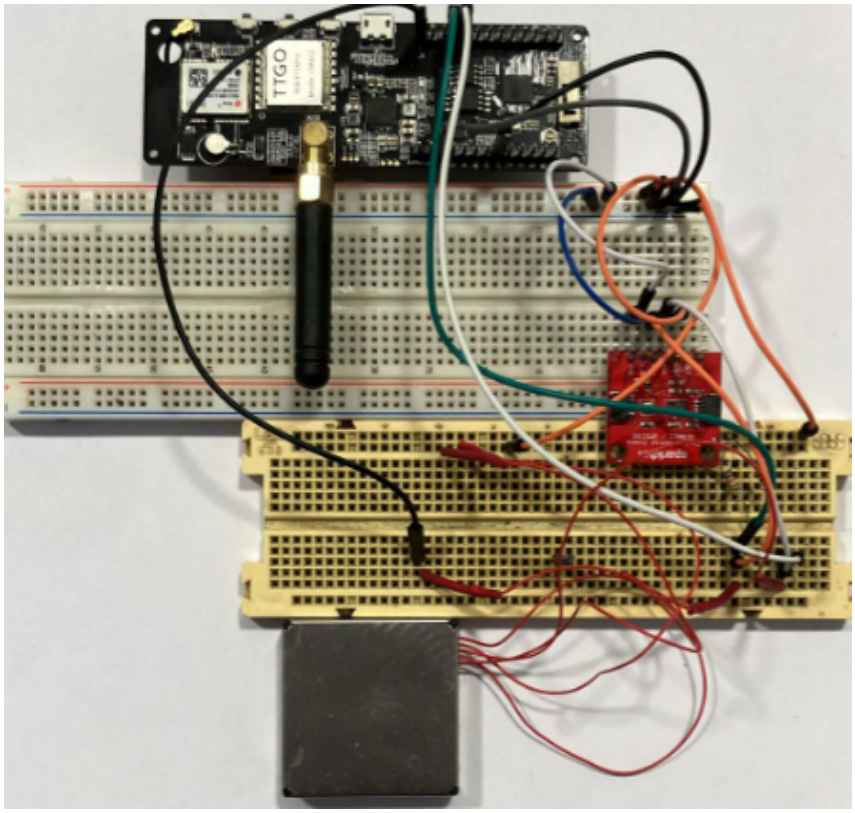


FIGURE 3.2. Prototype Assembled.

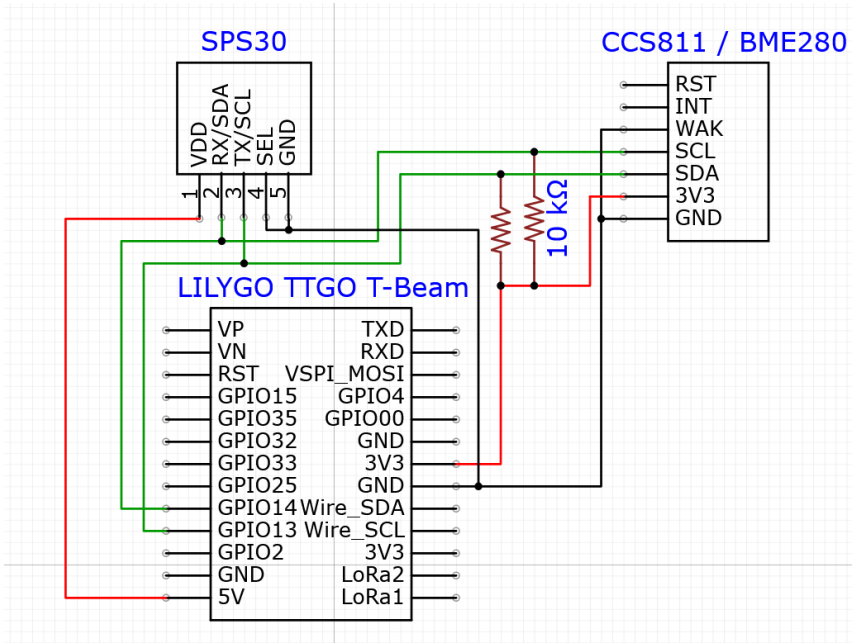
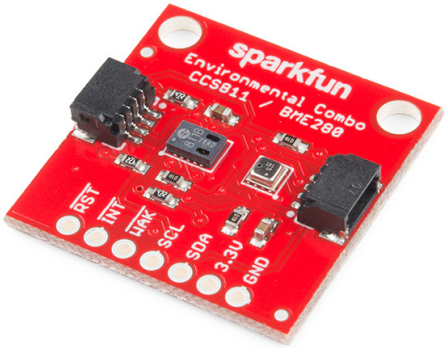


FIGURE 3.3. System Block diagram based schematic.

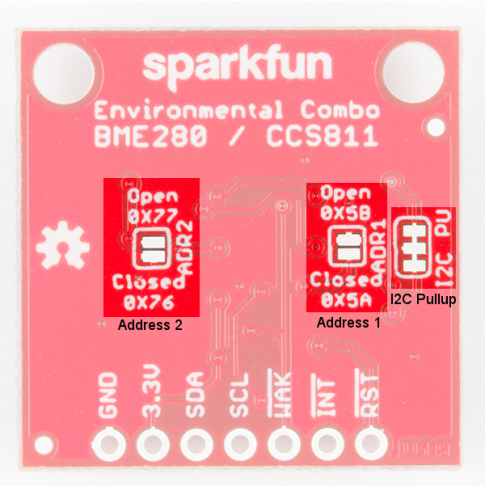
3.1.2. Air Quality Sensors

The sensors that were used for our prototype were the SPS30 and a combination of the BME280 and CCS811 from Sparkfun due to physical constraints with the equipment provided.

The sensors BME280 and CCS811, as seen in Figure 3.4a, were a two-in-one, which allowed us to reduce the footprint of the final prototype. However, this can come with a disadvantage because if one fails, we will need to replace the whole board. In Figure 3.4b, we can see an option to choose betwen two I^2C addresses for each sensor, depending on whether the connection is open or closed.



(A) BME280 and CCS811 Sensor. [57]



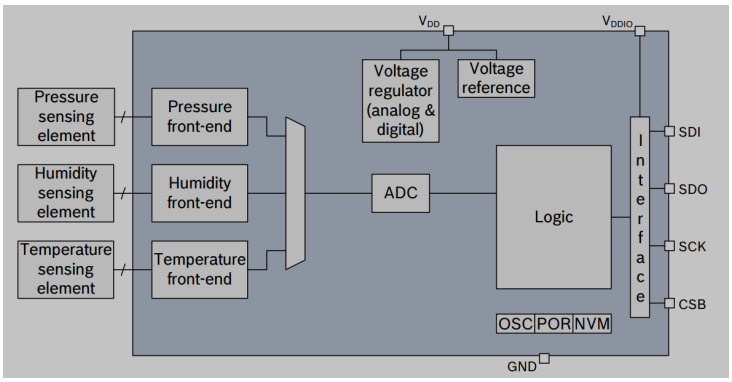
(B) Sensors I^2C address. [57]

FIGURE 3.4. BME280 and CCS811 front and back.

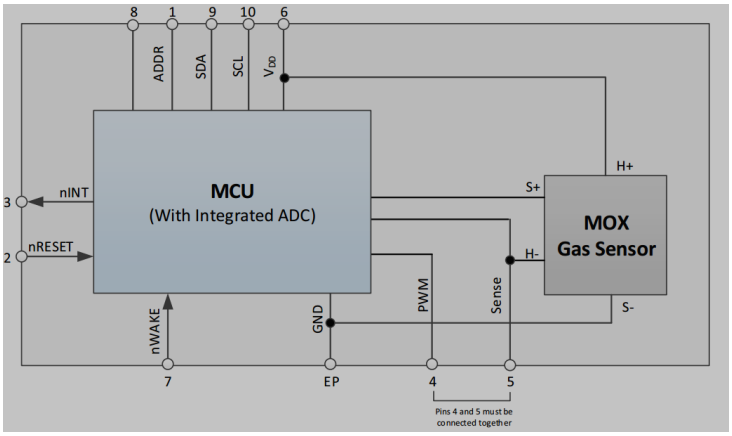
The sensor BME280 provides readings, apart from pressure, which are important as they measure the current humidity and temperature in our surroundings as these values can affect the readings of our sensor CCS811.

The sensor CCS811 required some configuration in the code before we could make use of it. These were based on the temperature and humidity, which we were able to provide through the BME280. With the adjustments made, we were able to gather TVOC and eCO_2 readings from the bedroom's air and have an understanding of how much of it was polluted from day to day activities, for example, sleeping, eating and outside movement.

Figures 3.5a and 3.5b present the block diagram of each sensor as well as the logic behind it.



(A) BME280 block diagram.

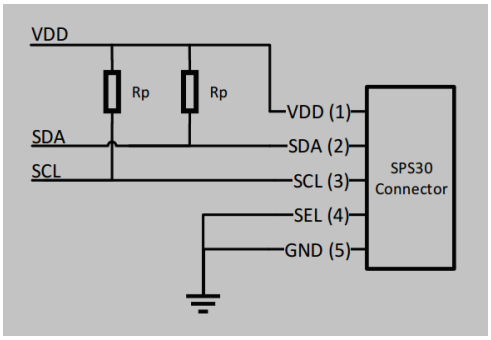


(B) CCS811 block diagram.

FIGURE 3.5. BME280 and CCS811 block diagrams.

The SPS30, as seen in Figure 3.6a, was a sensor that required 5V to operate and pull up resistors, as shown in Figure 3.6b and the SEL pin connected to Ground, to provide readings to our board while using the I^2C mode. This meant that we need to pull the 5V line from our Board to power it independently of the other sensor. Additionally, it boasts 10 plus years of lifetime with a 24/7 operation and requires no calibration or input from the user to start working. The table 3.1 holds more detailed specifications of the sensor.

(A) SPS30.



(B) SPS30 Diagram.

FIGURE 3.6. SPS30 sensor and its internal diagram.

TABLE 3.1. Sensirion SPS30 Particulate Matter Sensor Specifications.

Parameter	**Specification**
Technology	Optical laser scattering with contamination-resistance
Lifetime	>10 years (24/7 operation)
Dimensions	$41 \times 41 \times 12\,\text{mm}^3$
Weight	$26.3 \pm 0.3\,\text{g}$
Noise level	25 dB(A) max @ 0.2 m
Measurement Specifications	
Mass concentration range	0 – 1000 $\mu\text{g/m}^3$
Number concentration range	0 – 3000 particles/cm^3
Particle size range	PM1.0: 0.3–1.0 μm; PM2.5: 0.3–2.5 μm; PM4: 0.3–4.0 μm; PM10: 0.3–10.0 μm
Precision (PM1, PM2.5)	$\pm[5\,\mu\text{g/m}^3 + 5\%\,\text{m.v.}]$ (0–100 $\mu\text{g/m}^3$) $\pm 10\%\,\text{m.v.}$ (100–1000 $\mu\text{g/m}^3$)
Precision (PM4, PM10)	$\pm 25\,\mu\text{g/m}^3$ (0–100 $\mu\text{g/m}^3$) $\pm 25\%\,\text{m.v.}$ (100–1000 $\mu\text{g/m}^3$)
Sampling interval	1 ± 0.04 s
Start-up time	8–30 s (depending on concentration)
Electrical Specifications	
Supply voltage	4.5 – 5.5 V (typ. 5 V)
Current consumption	Sleep: 38–50 μA; Idle: 300–360 μA; Measurement: 45–65 mA (up to 80 mA at fan start)
Interfaces	
Communication	UART (115200 bps), I^2C (0x69 address)
Environmental Conditions	
Operating temperature	-10 to +60 °C
Recommended range	10–40 °C, 20–80 %RH
Operating humidity	0–95 %RH
Storage temperature	-40 to +70 °C
Additional Features	
Fan cleaning	Automatic (default every 168 h, configurable)
Calibrated	TSI DustTrak DRX 8533 ($PM_{2.5}$ mass) TSI OPS 3330 ($PM_{2.5}$ number)

With both sensors connected to our board and powered on, we needed to verify their respective I^2C address so we could set them up correctly in the code. To do that, we made use of an I^2C address lookup code, as seen in Appendix B.7 and it returned 3 addresses, 0x5B, 0x77 and 0x69, which are in conformity with their respective documentation.

Finally, with these addresses in hand, we could move towards creating the code for each individual sensor. This code will be explained in detail in sub-chapter 3.2.1.

3.2. Software

In this section, we will be talking extensively about the firmware developed for our microcontroller and the software used for this dissertation. This is what allowed us to gather the data from the sensors, transmit it to the DB that is being hosted in a separate machine, process this data and, finally, through the usage of Caddy set as a reverse Proxy, display for the user to interact with.

We used five containers - Caddy, Unbound, InfluxDB, Grafana and Wireguard Easy -, to make this process a lot simpler. All of their respective images can be found in Docker hub, except Wireguard Easy, which can be found in GitHub. We briefly showcase the usage of Wireguard in a scenario where a VPS is used, although, for our final prototype a VPS wasn't used, since we had to pay a monthly subscription to store our data. However, in this dissertation, it was decided to keep the solution low cost in the long term.

3.2.1. Embedded System Firmware

In this subsection, we will extensively present the code that was written for our board. This board is based on the ESP32 which allows us to use the software Arduino IDE to program its firmware to provide us values from the sensors. This software and board allows for the usage of high-level programming languages such as C or C++. The Figure 3.7 illustrates the process described in this subsection.

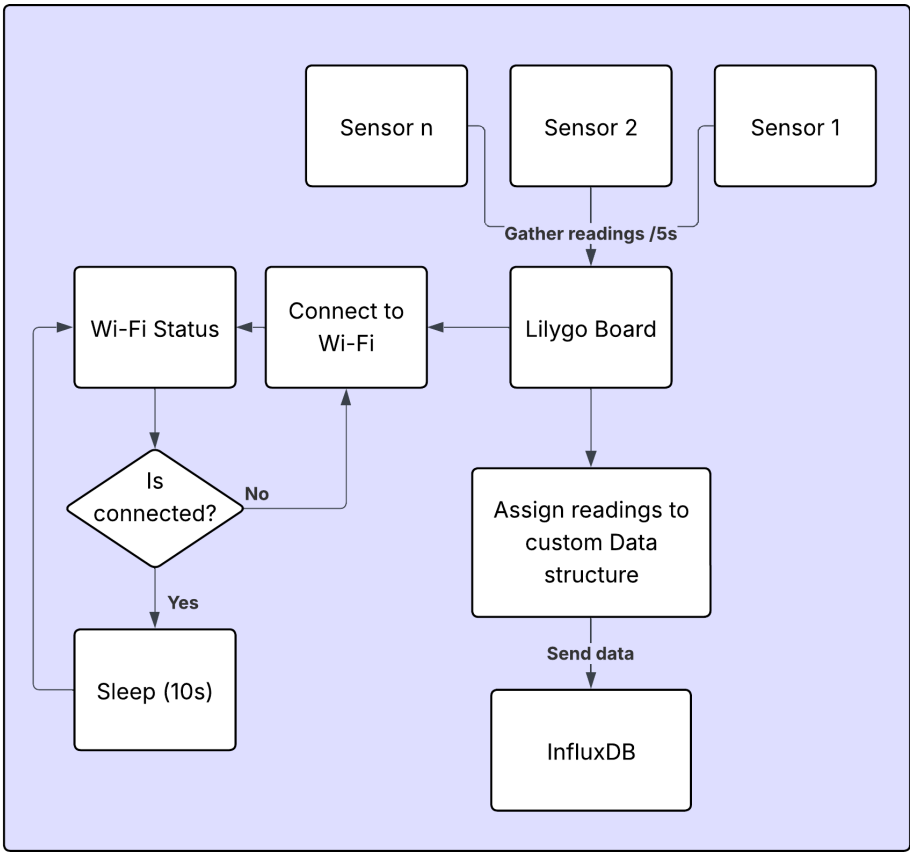


FIGURE 3.7. Firmware diagram.

There are some requirements that need to be fulfilled before being able to send the data and process it. First, it was required to discover the physical I^2C address of our sensors, mentioned in sub-chapter 3.1.2. With the addresses discovered, through the usage of Appendix B.7, the next requirement was to verify if the sensors required any additional libraries besides the default one of ESP32. For the sensors that we made use of, they required the Sensirion-sps [58] and SparkFun library [59].

After these requirements were fulfilled, we needed connect to our network. We decided to use Wi-Fi over, for example, MQTT, as it was simpler to implement and required no extra configuration from the user's part and the data was already being streamed in a secure way through HTTPS. We made use of the board's integrated Wi-Fi module which also required a library installation. Having this in mind, we then created a small code to start connectivity to our network as well as making sure it reconnects in case of timeout or disconnects. This can be seen in Appendix B.6.

With the Wi-Fi connected and tested, we proceeded to connect the sensors individually to our board to start creating the code for them. This approach was chosen instead of creating everything together, to avoid any potential bugs or possible cross readings from improperly setting up the sensors.

The sensor BM280's code can be found in Appendix B.1. This was the first code created for the sensors which can be found and used for the CCS811 sensor. This code is rather simple as it retrieves 3 measurements, atmospheric pressure, humidity and temperature. It then presents these measurements to the user over the Serial Monitor.

For the sensor CCS811, we used and adapted the library's code to our needs and it can be found in Appendix B.3. As stated in sub-chapter 3.1.2, this sensor required the BM280 readings to be setup to provide the correct readings for the eCO_2 and TVOC.

The SPS30, which the code can be found in Appendix B.5, was generated by the library. The code was quite extensive for our needs, which required us to tweak before having our final version of the code, as seen in Appendix B.10.

Finally, with the code of each sensor written, tested and adjusted to our needs, we were able to start working on the code needed to send the readings to our DB which we will explore and dive deeper in sub-chapter 3.2.5.

3.2.2. Reverse Proxy - Caddy

For this work, we will require a reverse proxy to provide us with the ability to reach our services without relying on an IP address. To achieve that, we can make use of publicly available reverse proxies such as Nginx Proxy Manager, Caddy and Traefik.

Caddy is an open-source web server written in Go. It is a simple and basic application with many features, such as being able to host files and webpages, but most importantly it has the ability to be used as a Reverse Proxy and provide Secure Sockets Layer (SSL) certificates.

Traefik is an open-source Application Proxy written in Go. Like Caddy, it's able to provide SSL certificates and allows for a more granulated configuration such as advanced routing and load balancing.

Nginx Proxy Manager, also an open-source application, is written in JavaScript. Which also has the ability to provide SSL. However, it's more geared towards making use of it's visual interface for any type of setup, limiting the modularity.

In this work, Caddy will be used. This choice was made due to Caddy's simple approach to reverse proxying and ease of modularity. Coupled with its ability to provide SSL certificates, allowing for the generation of encryption and security to access Grafana's interface and InfluxDB without needing to use an IP address, relying on a domain name instead. However, we will need to open ports 443, which is used for HTTPS and 80 for HTTP to the public. This can bring unwanted attention from other sources, such as scrapping bots which work by analysing the internet and looking for open ports to exploit and malicious users, who will try to gain access to our private self hosted server. Caddy has the ability to sign certificates via a DNS challenge which avoids opening the ports mentioned above, but requires purchasing a domain, although it is possible to use it's ability to self sign certificates which requires no domain. Both of these will require extra configuration, such as adding Unbound which is a DNS server that we will be using for our domains. In Figure 3.8, we can visualise this process happening.

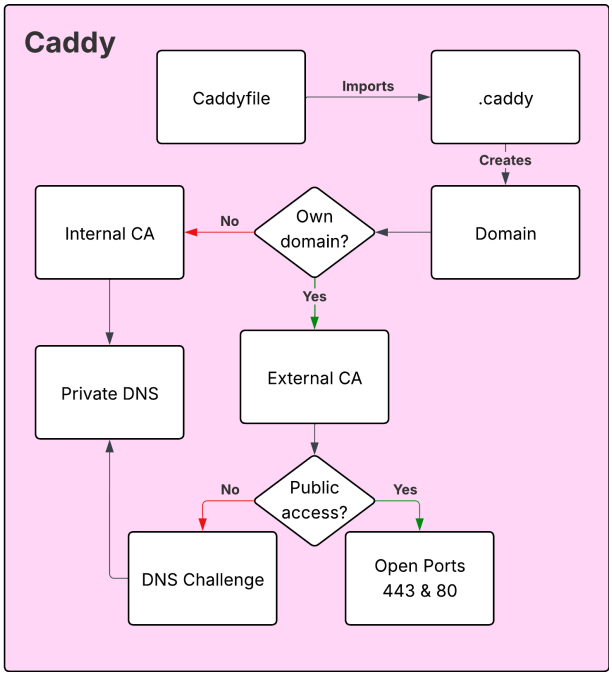


FIGURE 3.8. Caddy workflow.

We can now proceed to use Caddy without having to worry about external personnel accessing and seeing our data. To use Caddy, we need to pick its Docker image. We decided to use *caddy:2.10.0-alpine*, a lightweight and most up-to-date version available. For other environments, such as a dev environment, the image that could be used would

be with the *latest* tag. With our image decided, we have two options, run it on a single liner in the command prompt or create a Docker compose file. To provide clarity and ease of upgrading, a Docker compose file was chosen for this approach, which can be found in Appendix A.1. An important fact to mention in Docker is that the ports are mapped in the following way: *HOST_PORT:CONTAINER_PORT*. This exposes the container port and binds it to a port in the host. This, however, does not mean it's open to the outside world unless the same host port is opened in the firewall, but it does allow for local hosts to see and interact with it. With our directories created in our local server and our Docker compose file filled, all we have to do is run *Docker compose build* to build our current Docker compose in the current working directory. Afterwards, we can boot it by using *Docker compose up -d* to run in detached mode, allowing us to continue using our Command-line interface (CLI). This is needed so that we can have the image populate the directories we specified in its Docker compose while also having it run in detached mode.

After verifying that our Caddy is running, we can shut it down temporarily, using the command *Docker compose down*, so that we can start directly modifying the caddyfile. This file is the configuration part of our reverse proxy and we will modify it to achieve our desired results. This file is very sensitive to any changes, requiring us to restart the container for it to assume them. To avoid potentially breaking configurations, we would always do a backup of the file to be able to revert. This process allowed us to test our changes until we were confident on the way it works. We started by centralising everything on this one file, which could lead to a very long scroll to find our desired service. Initially, and for such a short file, it wouldn't be a bad outcome, but assuming this can be replicated to increase its size to several other services, we decided against this approach and went with one where we import other files with the extension *.caddy*, where, in each of these, we create the dedicated reverse proxying for our services. This increases modularity and simplicity as we can just copy one file and replicate it as many times as needed.

In a production setting, the security headers we created in our main Caddyfile are required to avoid any forgery or Cross-origin resource sharing (CORS) attacks. However, for a self-hosting environment, without opening the ports to the internet, they aren't required, but it's always in the best interest to have them available. Concerning our environment, which could be considered a production environment, they will be present and, due to how caddy works, we can call those headers with an import in our files allowing for an easier integration to all of our subsequent services.

Caddy by default serves all services over HTTPS. For internal ones, it uses self-signed certificates which will automatically be trusted locally if permitted - e.g. localhost, 127.0.0.1, .internal or .home.arpa -. For public Domain Name Server (DNS) names, it serves using certificates obtained from Automatic Certificate Management Environment (ACME), Certificate Authority (CA) such as Let's Encrypt or ZeroSSL. To provide our service with an SSL certificate, we first need to define the domain or, in our case,

subdomain. We selected a subdomain structure such as https://grafana.example.com. This approach provides intuitive naming and a user-friendly structure while preventing issues associated to directory traversal with a URL path like https://example.com/grafana. This approach also preserves the root domain for potential integration with any institution's domain.If such integration were to happen, this is an example of how it could look, https://www.grafana.up.pt/.

As this setup was done over a self hosted server, we will be using a domain which was bought and used to simulate the integration using a subdomain. For that, we will be using cloudflare as a CA by making use of the API key retrieved from Zones configuration and using it as a resolver. To do this, we can simply add it to the TLS tab in the caddy file of our grafana while also adding a propagation delay to avoid timeouts during the time it retrieves a certificate via DNS challenge.

As shown in Appendix A.3, we set our Caddyfile to listen to ports 80 and 443. This was necessary, as it would allow us to listen to all the HTTP requests and then automatically convert them to HTTPS. We defined the security headers and imported all files with the extension *.caddy*.

Finally, in Appendix A.4, which was also replicated for InfluxDB, we added the SSL certificate via Cloudflare and our Grafana is now reversed proxied at the URL defined as well as a logging function, which comes from caddy itself. This function is merely to log the IP addresses of who is accessing the interface and this can be used to monitor for any unusual activity.

3.2.3. Unbound

In this prototype, we made use of Pi-Hole with Unbound, which could imply different file locations in a fresh installation and, for the purpose of simplicity, we will only focus on Unbound. Unbound is a recursive and caching DNS resolver created by NLnet Labs with the purpose of hosting a decentralised DNS for better privacy.

With our Caddy setup completed, we now must look at how to tell our clients to reach the domains we have just setup and, to achieve that, we can make changes to a file in each of them, but that would be unrealistic to do at a large scale. So, instead, we will be making use of a DNS such as Unbound. The reasoning behind this is to enable the URLs we created in Caddy to point to the correct IP address, which, in this case, is the Grafana and InfluxDB IP. This is due to the fact that we don't have the ports opened to the exterior and didn't use a public DNS. We will need to tell the services that a certain name exists and that they need to look it up on our own DNS to reach it.

To achieve this, we need to edit a configuration file in Unbound and configure the network to route all the traffic to the Unbound container for querying. The Unbound file that we need to alter is the *unbound.conf* which holds the configuration for our DNS. Once inside it, we can create a zone for our locally hosted domains, as seen in Appendix C.1, where we start by defining a local-zone with our upper domain as static, meaning it

will not be resolved by external DNS servers, followed by a local-data entry with an A record to Caddy's current LAN IP.

Once these two are setup, we then create CNAME records for our subdomains and point them to our A record. This allows for ease of modularity as we can simply create more services and add them to the file and any IP change would simply need to be corrected in the A record.

With Unbound setup, we can now reach our services from any machine, as long as they use Unbound as their DNS and know the domain name to reach the services. With a Wireguard setup, we can define the DNS for the peers to use in its docker compose, bypassing the previous step of manual selection of the DNS.

3.2.4. Grafana

In this section, we will present the need for an interface. This need stems from a necessity to show the users the data processed in a single space whilst allowing for them to have a simple interaction with it. To achieve, this we made use of an open source web application - Grafana.

Grafana showcases analytics and interactive visualization and was written in GO and Rust for Version 3.x. It can create charts and graphs when connected to supported data sources.

Grafana comes with the ability to connect many data sources such as Amazon services, Google analytics, our data source of choice, InfluxDB and many others. Many of these services come pre-installed while others require installation to be usable, such as Amazon services. Fortunately, ours does not require installation and can be used right away. With our DB setup and receiving data, we can connect our Grafana to it. To achieve that, we need to create a new connection. From the available connection, it's possible to select InfluxDB, which then prompts us to fill the fields, with careful attention to the Query language as this will tell Grafana which version of InfluxDB we are connecting to. The available Query languages to connect to InfluxDB are InfluxQL, Flux and SQL. However, as mentioned previously, these are limited by the version that InfluxDB is currently on. For InfluxQL, we can use versions 1.x and 2.x. For Flux Query, our DB can be in versions 1.8+ to 3.x. For SQL, InfluxDB needs to be running in versions 3.x or higher.

For this work, we decided to use Flux as our query language due to using version 2.x as our InfluxDB's version and, through that choice, we can make use of the token we previously generated to connect to our DB. We now can specify at the URL if we want to use an HTTP or HTTPS connection. For the initial test, as presented in sub-chapter 3.2.5, we chose to use an HTTP connection. However, after Caddy's reverse proxying was setup, and to provide more security, we then chose to use an HTTPS connection. With this, our connection is configured and we can now use queries, as seen in Appendix B.8, to fetch data to present to the user.

With data being retrieved, we can now configure how to present it to the user. To achieve that, we can create a dashboard and add visualisations to it. The available visualisations range from Gauges, Time Series, Bar chart, Table, Histogram, Heatmap, Pies and many more. We decided to use Gauges and Time Series for ease of visualisation for each sensor, but we are able to mix them into a single Time Series chart to provide the user with multiple inputs from the same sensor, such as the SPS30.

One feature of Grafana is the ability to send alerts when a sensor's values reach a certain threshold, for example, eCO_2 reaching values higher than 1000 ppm. These alerts can be modular, but they have a slight issue. We can create templates to be used for all the other alerts, as long as we keep the same fields used, whilst the broker, which is what will be sending us the notification.

In our case, it can only use one template, which limits the usage of that broker - we created and used a Telegram Bot. To overcome this problem, we can create several brokers using the same Bot to send alerts with different templates.

Another drawback that we were able to verify is that, with the new Alert model, it's not possible to have several conditions to trigger an Alert. Ideally, we would create an alert that would verify if the latest value from a reading is within a certain range, for example, eCO_2 between 600 and 1000 ppm, then 1000 and 1500, etc, and then send a notification accordingly. However, this is not possible and, instead, we need to rely on multiple alerts, which can be quite an encumbrance if they are not properly named.

Finally, this prototype will be used by many people which, some for security and safety reasons, aren't Admins, so we needed to create other users and provide them with the ability to only visualise the data. Grafana provides fine-grained user roles and permissions per dashboard and data source, so adding users with the appropriate roles was quite simple. As seen in Figures 3.9 and 3.10, we can create a user and then assign that user the correct permissions.

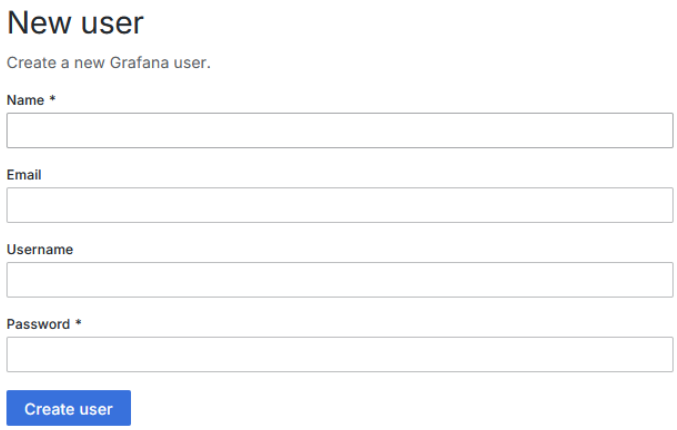


FIGURE 3.9. User Creation.

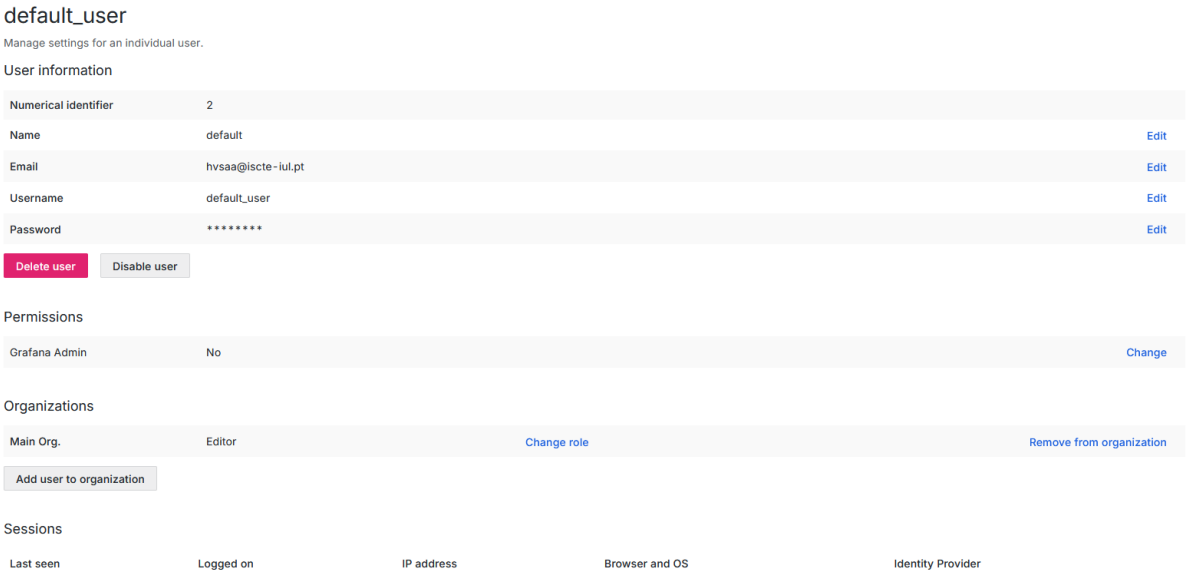


FIGURE 3.10. Configuration for the user "default".

This user can be furthermore enhanced through the usage of Teams, as illustrated in Figure 3.11, Grafana provides functionally to assign the user to a specific team, thereby association with multiple boards. In Figure 3.12, it displays all existing members of a specific team, accessible only if the current user has administrative privileges.

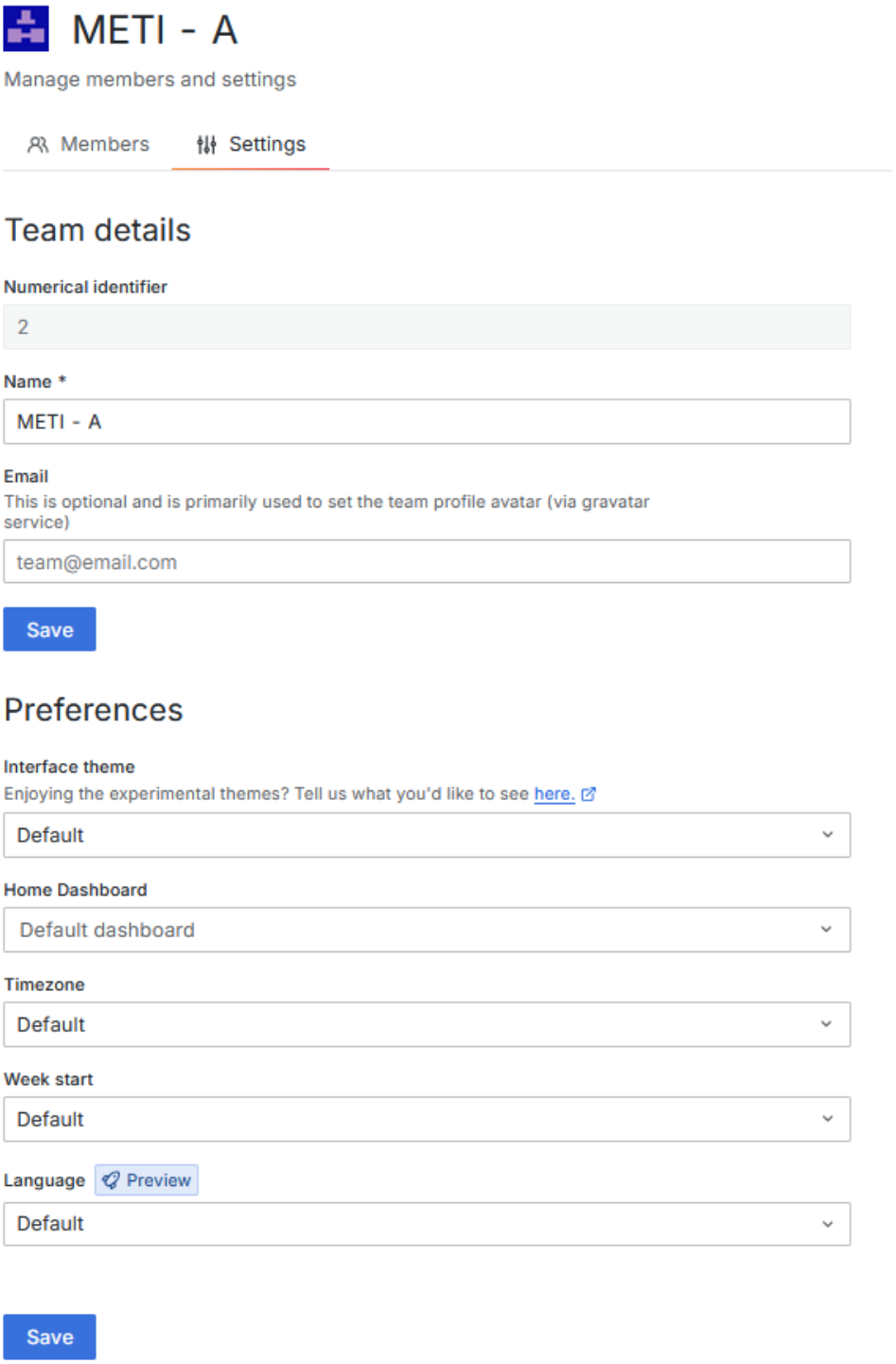


FIGURE 3.11. Team Settings.

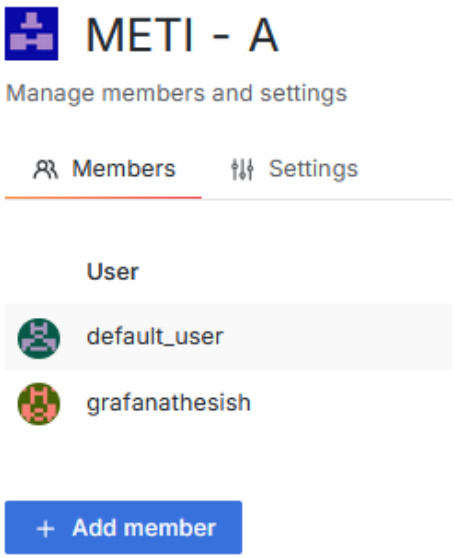


FIGURE 3.12. Existing Members for the team METI - A.

3.2.5. InfluxDB

InfluxDB itself is a time series database written in Go for versions 1.x and 2.x and Rust for version 3.x. We decided to use this database due to the amount of data that we will be generating over a small interval of time from our sensors.

To setup our data base, we needed to create a bucket as well as an organisation, as we can see in Appendix A.7. However, this is not mandatory as all of the environment variables in the Docker compose are not required and can be omitted. By doing so, we will be presented with a Wizard during the first boot where we will be able to define our Organization (ORG), our Bucket name, admin username and password, while the token will be automatically generated after this initial configuration.

With this set up completed, we can now access our DB where we are presented ways of uploading our data. InfluxDB provides a variety of ways to upload data through CSV, influx CLI and libraries. One of those libraries is our method of choice for gathering data, as to InfluxDB provided a step by step on how to incorporate. Prior to this, it was necessary to test whether our bucket was able to receive data and, to do it, we made use of an HTTP POST request, as seen in Appendix B.9 and in Figure 3.13.

In this request, we provided all of our measurements retrieved from the sensors, then sent it to our DB where we were met with a 204 response - 204 No Content successful response status code indicates that a request has succeeded, but the client doesn't need to navigate away from its current page -, a successful response, followed by a *No Content* response from the DB signifies that the DB received the data successfully and replied with no content.

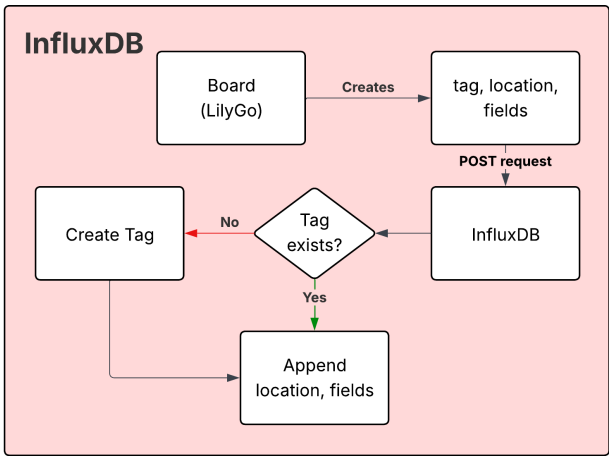


FIGURE 3.13. Data transmission.

Following this successful test, our caddy setup to provide SSL and our Unbound redirecting domain requests, we can now use the library by InfluxDB to send data via HTTPS, as seen in the final version of our code in Appendix B.10.

As our DB acquires data, it starts building a timeline of the events which can increase the size used for storage. Incidentally, InfluxDB uses a time-series data compression algorithm, which can significantly reduce the size of the stored data effectively saving space in our storage, to the point that a month's data doesn't exceed 200MB and sits comfortably at 161MB.

Finally, to make use of this newly acquired data we, can create a dashboard to see the data. However, this is not ideal since InfluxDB has limited customisation towards data presentation and user management. To overcome these limitations, we will be using Grafana which provides a lot more customisation, such as different charts, alerts, notifications for the alerts and more, which can be found in detail in sub-chapter 3.2.4.

3.2.6. WireGuard

For this work, it was intended to use a VM from Oracle to host our containers. To access and expose the services from the containers, it was necessary to use Wireguard to avoid unauthorized access. However, we were unable to continue with this approach due to several complications.

WireGuard is a communication protocol that implements encrypted Virtual Private Networks (VPNs) and passes traffic over UDP, but it requires opening port 51820. It might seem like a vulnerability, but it offers a significantly more secure communication channel compared to ports 80 and 443 - that is achieved by only allowing access to a few selected peers.

This solution is ideal if we host our services in a VPS as it creates a layer of protection around them and allows ease of accessibility. In sub-chapter 3.2.2, we mentioned the need to open ports so we could reach our data, however, said ports weren't open so we are not able to access the data from the outside. This creates an issue on how to safely access our data, which will be presented in Grafana and it can be solved by making use of WireGuard, as we can see in the diagram illustrated in Figure 3.14.

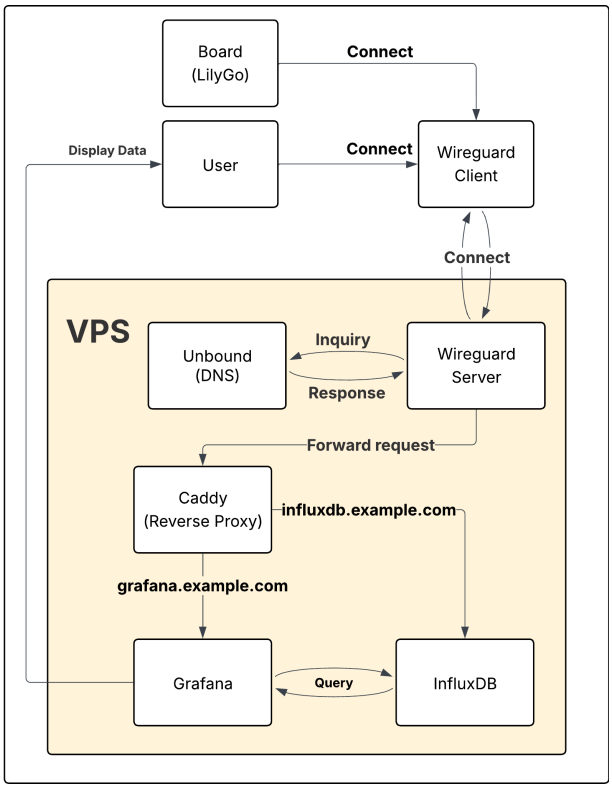


FIGURE 3.14. Wireguard connection.

The Wireguard server generates two types of files - a Server and Peer file. The server file, as seen in Figure 3.15, contains its private key, the private network address its creating (for example, *10.0.0.1/24*), the listening port 51820, the Maximum Transmission Unit (MTU), a list of peer blocks of all peers with their respective public key and allowed IPs, and commands for before starting and stopping and after starting and stopping, connoted by PreUp, PostUp, PreDown and PostDown.

FIGURE 3.15. Wireguard Server configuration file.

The peer file contains, as shown in Figure 3.16, the server's public key, the peer's private key, the server endpoint - in this case, the public IP of our machine - and routing info, such as DNS and allowed IPs. Once generated, it can then be sent to the selected peer to allow communication between the client and the server, effectively avoiding exposing the data to any one.

To create our own WireGuard, we will be making use of the Docker image of wg-easy. This docker compose, as seen in Appendix A.5, contains a lot of environment variables that we can make use of, such as defining the public IP address, the Wireguard port - in case we

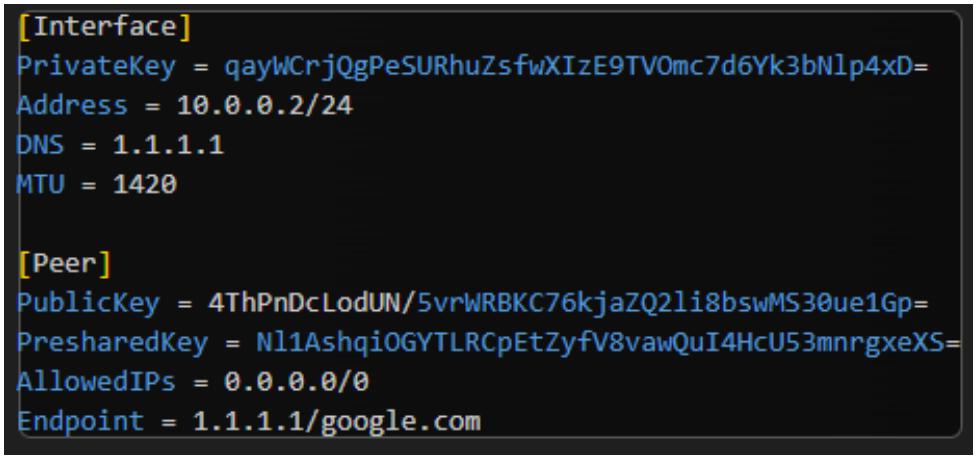


FIGURE 3.16. Peer configuration file.

want to obfuscate it to avoid using the default 51820 -, the User Interface (UI) for ease of peer management, custom commands for before and after setting up and taking down the VPN, among others. Once our changes were made, we could start our Wireguard server and provide a peer's details to our machine to access it remotely via Secure Shell (SSH) with the Wireguard client turned on. Once connected, we could provide these values to our Board. The board itself doesn't work with Wireguard, but making use of a library, we can install it and allow it to communicate through it.

Finally, due to the costs presented by the VPS, we decided against using one and instead relied on a self hosted VM to gather and process our data, but, nonetheless, this solution can be easily implemented on a scenario that does make use of one.

In Figure 3.17, we are able to see a system diagram of how every part of this prototype interacts with each other.

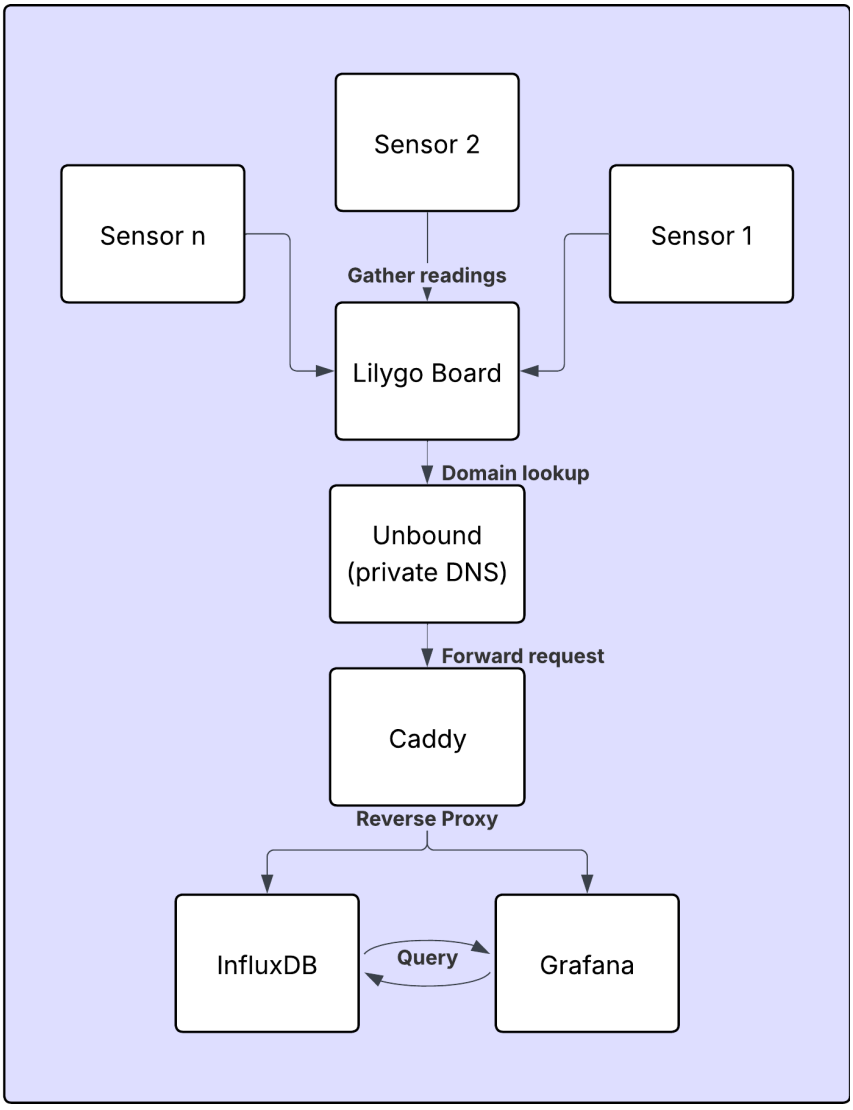


FIGURE 3.17. System diagram.

CHAPTER 4

Results and Discussion

In this chapter, we will present the results gathered from our prototype in the chosen testing environment, over the course of 30 days, where each sample was retrieved every second, starting in June 15th. Post testing, the retrieval interval was changed to every five seconds and can be further adjusted to the user's own discretion, depending on its needs. These results will be about the air quality and the alarms setup, including the bot creation, the user's access and the data display. We will also offer some remarks about the ease and process of setting everything up.

4.1. Air quality monitoring results

To provide context, our test environment was a bedroom facing the streets at a ground floor, with windows open about 80% of the time. It should be noted that, due to influences from the outside, the air quality wasn't always optimal, reinforcing the fact that the air should be purified through the usage of other equipments.

In the Figure 4.1, we can see the evolution of the temperature over the course of the mentioned time frame and, in Figure 4.2, we use a Gauge to visualise the temperature at the moment of reading.

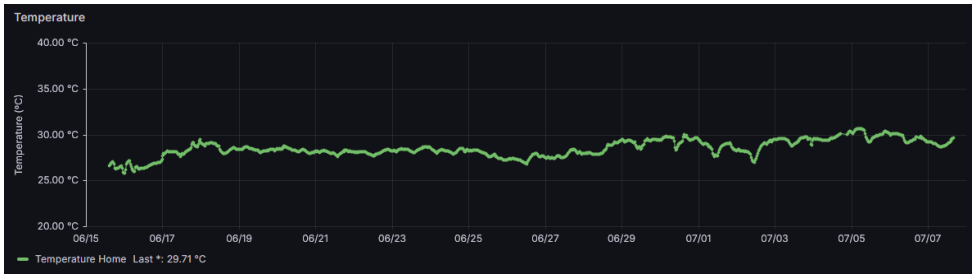


FIGURE 4.1. Temperature graph.

The Gauge display enables us to set visual thresholds that serve as cues to alert us to whether the current value is good or notorious. This type of display was used for all of our measurements in accordance to the AQI and PMs sizes.

For eCO_2, as seen in Figure 4.3, the values were higher than expected, especially when looking at determined times of the evening. These results are bound to the TVOCs sensor reading and, should this sensor detect high amounts, it estimates that the eCO_2 will also be found in high quantities. For future prototypes, this board will be replaced with 2 boards, where one has sensor that accurately reads eCO_2 and another to read TVOCs. Optionally, it can be preserved by pairing it with a sensor to accurately read eCO_2 and discard the reading from the eCO_2. It is also worth noting that, through Grafana, we

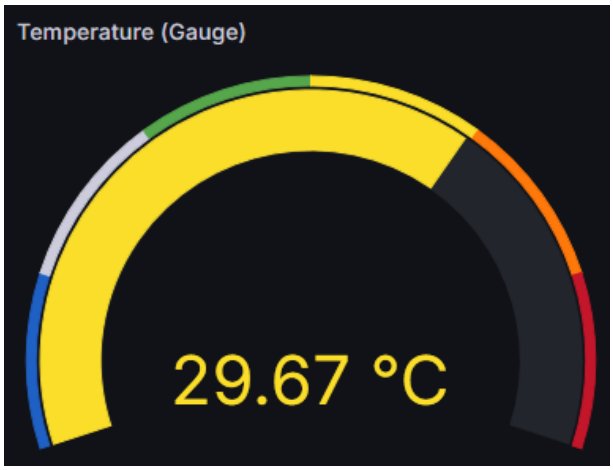


FIGURE 4.2. Temperature gauge.

are able to obtain the mean, median, max, min and last value and display them in the graph's legend, as seen in Figure 4.3. This allows the user to configure how they want to visualise their data, as well as the graph itself, as this one can also be rendered using those statistics. Nonetheless, we are able to track the values with some precision. It's important to note that these results suffer some interference from the outside, such as cars passing by, smoking and constructions, providing us real-time information regarding the effects these interferences have on the air even if these aren't noticed right away.

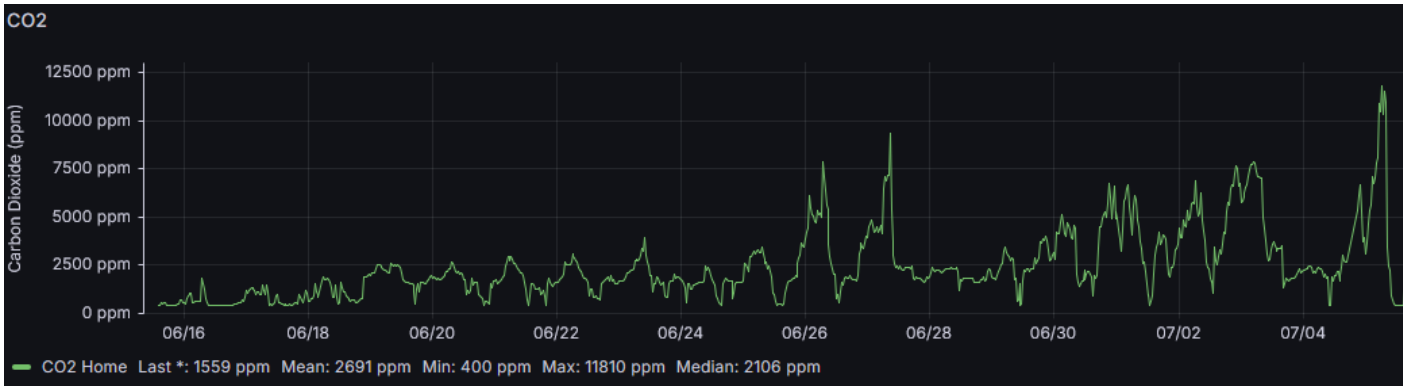


FIGURE 4.3. eCO_2 graph.

Following the same concept as the gauge for temperature, we can see the thresholds as well as the latest value. The query used to obtain these results was also used for our alerts, to provide us with some dynamic messages depending on the value readings.

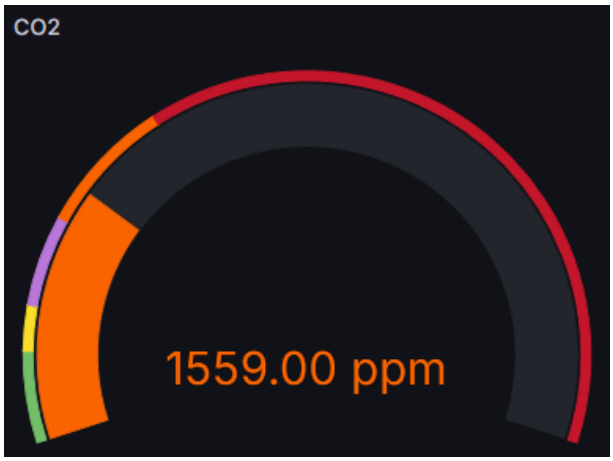


FIGURE 4.4. eCO_2 gauge.

Due to some limitations of the software, as seen in Figure 4.5, which restricted the alert condition to be within a single predefined range, we were required to create a set of three alerts. Each alert would send a message following a template, as seen in Figure 4.6, and depending on the last value reading, the correct one would trigger and send a message to our Telegram bot. In Figure 4.7, we have several labels - danger level, location and Alert name (which we dubbed as sensor) - that, through the template, Figure 4.6, we can send a message to our Telegram bot that will automatically replace the fields with the correct information as shown in Figures 4.8a and 4.8b.

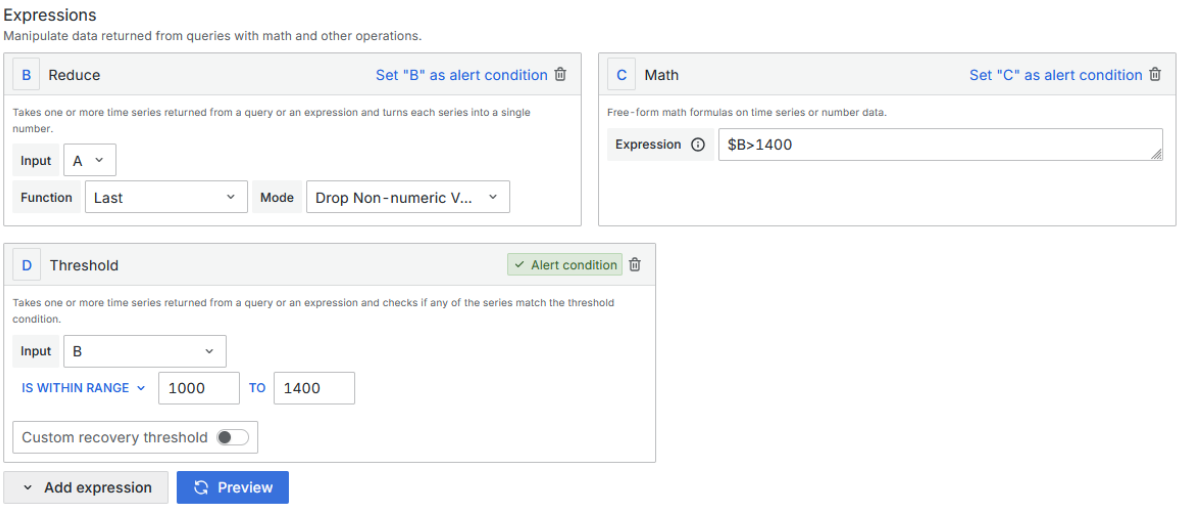


FIGURE 4.5. Alert trigger.

```
{{ define "custom.alerts" -}}
{{ len .Alerts }} alert(s)
{{ range .Alerts -}}
  {{ template "alert.summary_and_description" . -}}
{{ end -}}
{{ end -}}
{{ define "alert.summary_and_description" }}
  <b>Status:</b> {{ .Status }}
  <b>Sensor:</b> {{.Labels.alertname}}
  <b>Value:</b> {{.Annotations.Value}}
  <b>Severity:</b> {{.Labels.Severity}}
  <b>Location:</b> {{.Labels.location}}
  <b>Summary:</b> {{.Annotations.summary}}
  <b>Description:</b> {{.Annotations.description}}
{{ end -}}
```

FIGURE 4.6. Telegram template.

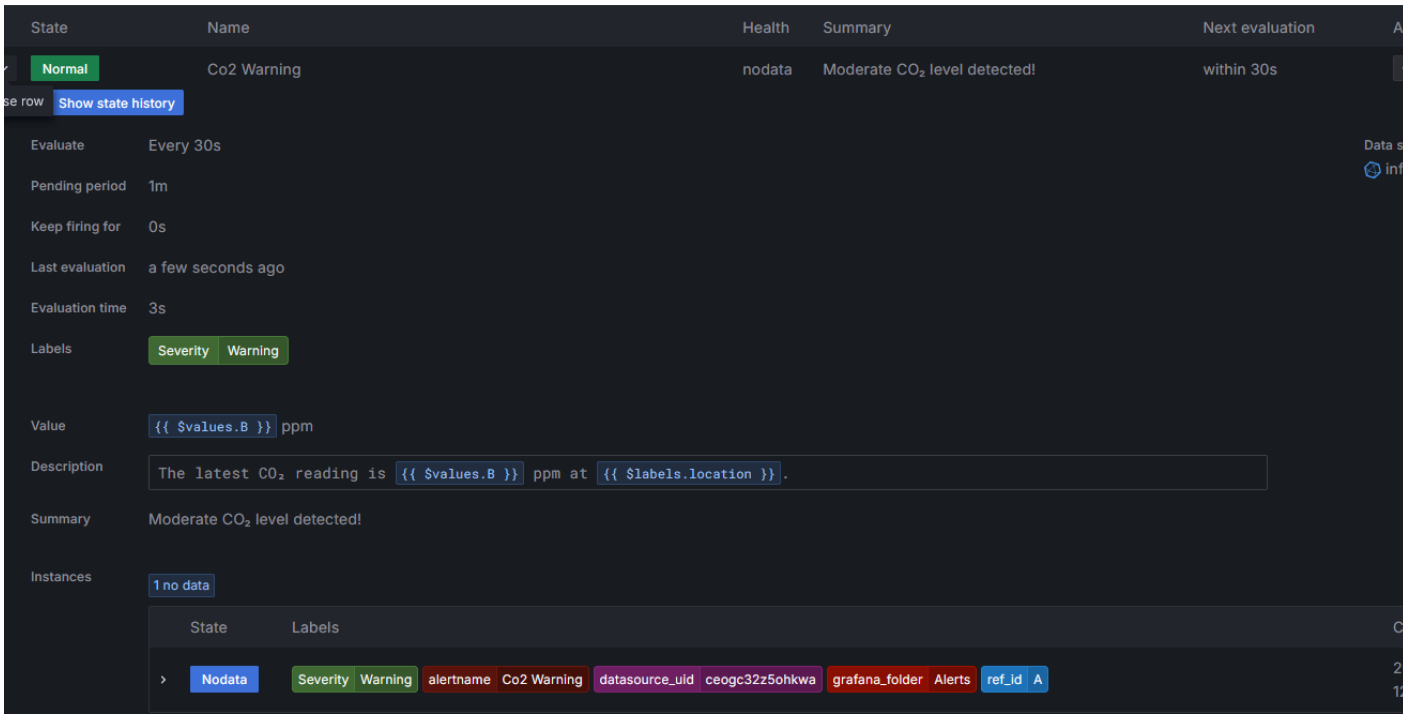
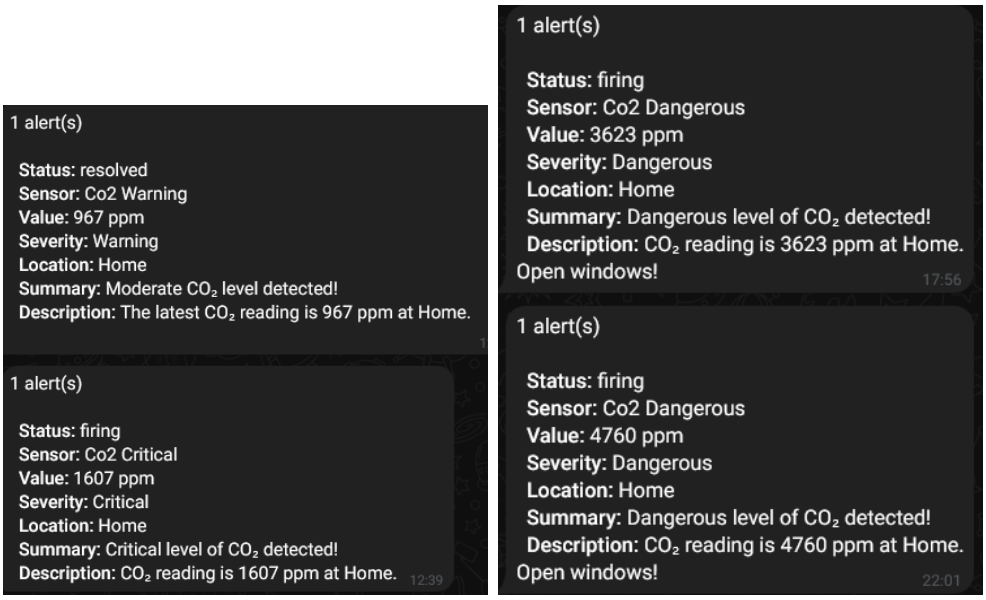


FIGURE 4.7. Warning alert for eCO_2 levels.

Through the same logic, we can see the values of TVOCs in Figures 4.11 and 4.12. These suffer from the same conditions as the previous readings due to being in the same board and requiring values to calibrate itself to be able to provide correct readings. As mentioned previously, these values were inputed into the code to allow the sensor to self calibrate.



(A) Normal and Critical messages. (B) Dangerous message.

FIGURE 4.8. Messages sent by the Telegram Bot in real time.

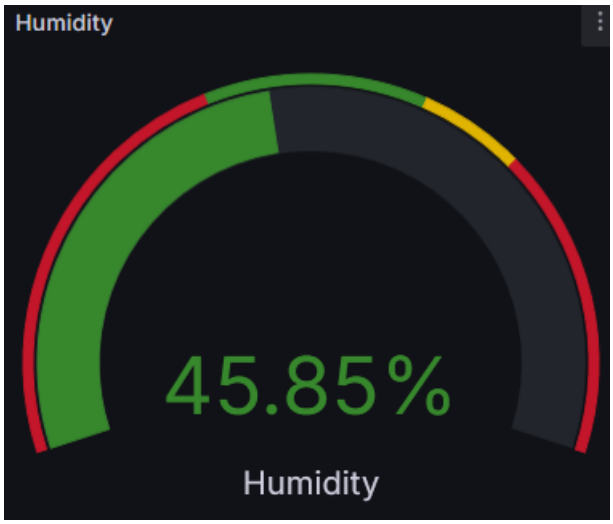


FIGURE 4.9. Humidity Gauge.

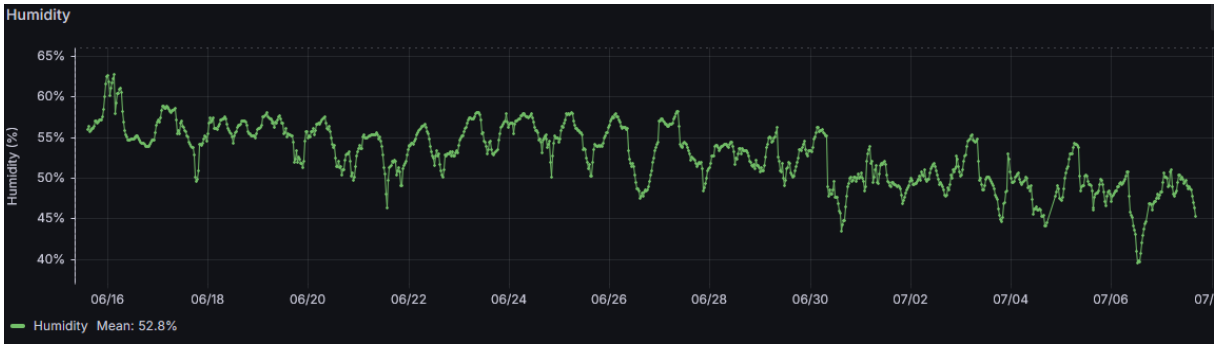


FIGURE 4.10. Humidity graph.

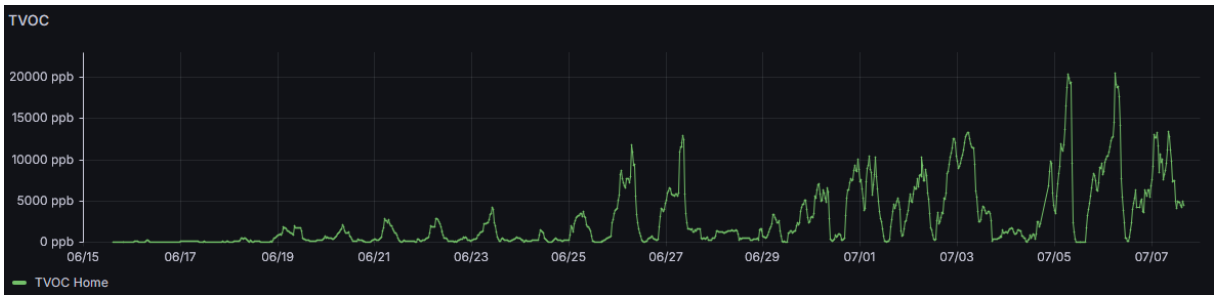


FIGURE 4.11. TVOC graph.

Initially, we can see a low amount of TVOC, but as the sensor spent more time gathering data, as seen in Figure 4.11, coming out of extensive non usage and getting affected by the air, the growth became apparent and alarming. Through the Figure 4.10, we are able to see that humidity gravitated towards 60% with the lowest point recorded being 40%. This amount of humidity will impact the air as it traps the pollutants and stagnates the air.

With this, humidity can influence the concentration of VOCs in the air by affecting how these particles are released and dispersed. When relative humidity reaches 60% or higher, the emission rates of VOCs from materials and products can increase, contributing to the odours commonly associated with fragrances, household items, and indoor pollutants. These compounds originate from a wide range of indoor sources, including household mould, upholstered furniture and cleaning supplies.

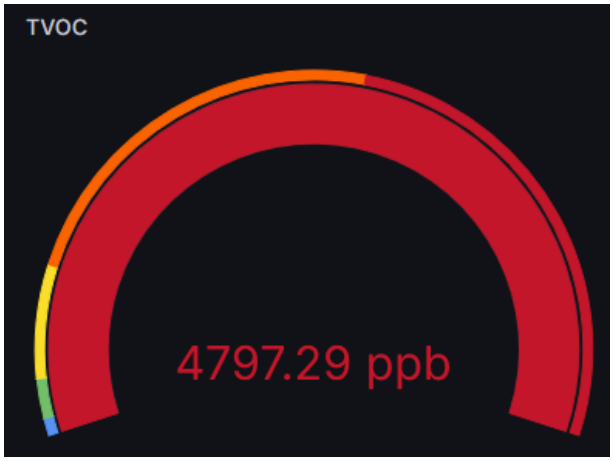


FIGURE 4.12. TVOC gauge.

While the other sensors showed elevated values, the sensor for one of the most concerning pollutant, PM, remained at a near constant rate of exposure through all of their sizes, as seen in Figure 4.13.

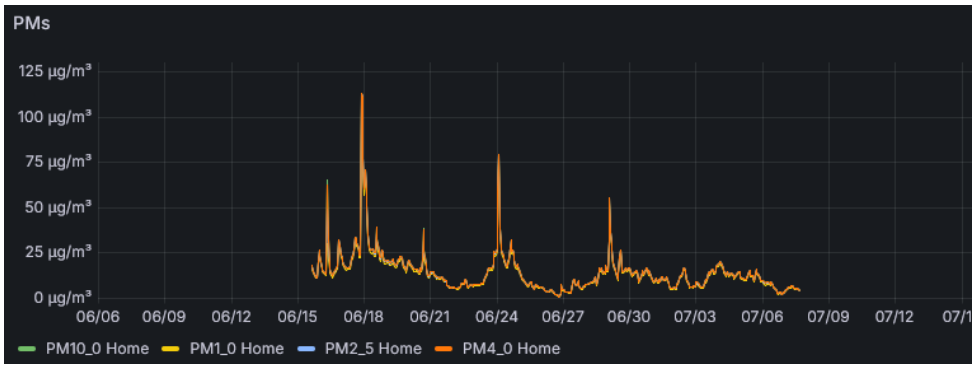


FIGURE 4.13. $PM_{1.0}$, $PM_{2.5}$, $PM_{4.0}$ and PM_{10} combined.

The sudden spikes shown in it are due to having the windows constantly open, tilted inwards, with constructions happening nearby, vehicles and garbage trucks passing by in the street. Consequently, the wind transports the particles related to these mentioned situations into the bedroom, allowing the sensor to detect them and explaning the results obtained.

CHAPTER 5

Conclusion and Future Work

5.1. Conclusions

In this section, we present our contributions on the topic of AirQual-IOT – Air Quality Monitoring and Data Analysis for Indoor Environments.

After analysing the prototype implemented and tested, we are able to provide the users with the ability to create their own system through the usage of the same technologies presented in this dissertation. It is possible to scale this system, setting the main board as a node aggregator connected to smaller nodes, such as the LilyGo Lora T3S3, which would then relay the sensor data. However, this was not attempted with this prototype. The initial cost for this specific prototype would be around €95.52 with a change to the BME/CCS811 sensor, since the one used is no longer in production, which could pose an issue to some users. In the long term, this cost would be negligible compared to issues that could arise from not being aware that the air we breath is polluted. Moreover, power consumption would be dependent on the board chosen. For this prototype, we are able to use either batteries to power it, which would reduce the cost to having to buy batteries over a long period, or use a USB cable with a wall adaptor, maintaining the power consumption to its lowest at a minimum of 0.8W, to answer RQ1 presented in sub-chapter 1.3.

This device, with the information presented in this dissertation, can be used to monitor the indoor air quality and, based on it, inform the user via custom alerts sent directly to their Phone - more specifically Telegram chat - about the current air composition. Subsequentially to this, there should be another system, especially one with the capabilities to filter the air. This is important because once we get an alert about the air around us and have the ability to filtrate it, we can help prevent health issues, such as asthma, thus answering RQ2 presented in sub-chapter 1.3.

Concluding, this dissertation is able to provide insight on the pollutants found in the air, how dangerous they are, their various degrees of quality through the AQI, and a working prototype that can be built by anyone due to the simplicity of the hardware required to assemble it, such as a board, sensors and the necessary wiring to connect both. Furthermore, this prototype, albeit small, can provide a lot of real-time insight of the air that is currently being circulated in our environment as well as ways to integrate this data with the InfluxDB and showcasing it to anyone that is interested in having visualisation of the air quality with alerts in real-time.

5.2. Future Research

For this work, we were able to successfully create a working prototype while maintaining a low cost and total control for the user without requiring any additional third party to provide us access. However, some areas for future research and development have emerged. To name a few:

(1) Adding more sensors:
The current implementation of our prototype does not have all the necessary sensors we set out to include. Future research could focus on adding sensors such as the MQ-135 and the MQ131 to monitor O_3, NO_2, SO_2 and thoroughly testing the system for breakpoints;

(2) Connecting to an air filtration system:
The addition of an air filtration system responding to the Telegram bot messages or the InfluxDB when certain values are within a threshold to start an air filtration process. This would allow greater control over the air we breath in our environment;

(3) Enhanced Security:
The current prototype boasts strong security, however, this can change with out of date libraries or components. This could be improved upon with regular maintenance or small nodes with sensors whose sole purpose is reporting to a main board to forward the traffic to the DB.

Ultimately, the future work outlined in this section aims to resolve current limitations, but also to overcome some of the existing boundaries in user-empowering ecosystems. Progressive research and development in these areas will shape IoT's next generation into a more secure and self-sovereign future.

References

[1] M. J. Rodrigues, O. Postolache, and F. Cercas, *2019 International Conference on Sensing and Instrumentation in IoT Era (ISSI)*. IEEE, 2019, ISBN: 9781728110226.

[2] Sholahudin, Y. Damey, W. Fauji, and M. A. S. Yudono, "Indoor air quality monitoring system with automation: A review," in *2023 IEEE 9th International Conference on Computing, Engineering and Design, ICCED 2023*, Institute of Electrical and Electronics Engineers Inc., 2023, ISBN: 9798350370126. DOI: 10.1109/ICCED60214.2023.10425129.

[3] O. A. Postolache, J. M. D. Pereira, and P. M. S. Girão, "Smart sensors network for air quality monitoring applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, pp. 3253–3262, 9 2009, ISSN: 00189456. DOI: 10.1109/TIM.2009.2022372.

[4] O. Postolache, P. Girão, M. D. Pereira, G. Ferraria, N. Barroso, and G. Postolache, *Indoor Monitoring of Respiratory Distress Triggering Factors Using a Wireless Sensing Network and a Smart Phone*. IEEE, 2009, ISBN: 9781424433537.

[5] M. P. Sierra-Vargas and L. M. Teran, *Air pollution: Impact and prevention*, Oct. 2012. DOI: 10.1111/j.1440-1843.2012.02213.x.

[6] O. K. Kurt, J. Zhang, and K. E. Pinkerton, *Pulmonary health effects of air pollution*, Mar. 2016. DOI: 10.1097/MCP.0000000000000248.

[7] E. Commission, *Chemicals: The eu restricts exposure to carcinogenic substance formaldehyde in consumer products*, 2023. [Online]. Available: https://single-market-economy.ec.europa.eu/news/chemicals-eu-restricts-exposure-carcinogenic-substance-formaldehyde-consumer-products-2023-07-14_en.

[8] B. Xu, *Design of an Intelligent Indoor Air Quality Monitoring And Purification Device*. IEEE Press, 2017, ISBN: 9781509053636.

[9] J.-j. Lv, X.-y. Li, Y.-c. Shen, *et al.*, "Assessing volatile organic compounds exposure and chronic obstructive pulmonary diseases in us adults," *Frontiers in Public Health*, vol. 11, 2023, ISSN: 2296-2565. DOI: 10.3389/fpubh.2023.1210136. [Online]. Available: https://www.frontiersin.org/journals/public-health/articles/10.3389/fpubh.2023.1210136.

[10] J. K. Wickliffe, T. H. Stock, J. L. Howard, *et al.*, "Increased long-term health risks attributable to select volatile organic compounds in residential indoor air in southeast louisiana," *Scientific Reports*, vol. 10, 1 Dec. 2020, ISSN: 20452322. DOI: 10.1038/s41598-020-78756-7.

[11] K. L. Alford and N. Kumar, "Pulmonary health effects of indoor volatile organic compounds—a meta-analysis," *International Journal of Environmental Research and Public Health*, vol. 18, pp. 1–12, 4 Feb. 2021, ISSN: 16604601. DOI: 10.3390/ijerph18041578.

[12] "Design Science Research Process: A Model for Producing and Presenting Information Systems Research," in *the Proceedings of the First International Conference on Design Science Research in Information Systems and Technology*, Claremont, California, USA, 2006, pp. 83–106. DOI: 10.48550/arXiv.2006.02763.

[13] W. H. Organization, *Air pollution*, Jul. 2019. [Online]. Available: https://www.who.int/health-topics/air-pollution#tab=tab_1.

[14] W. H. Organization, Oct. 2024. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/household-air-pollution-and-health.

[15] W. H. O. WHO and W. H. O. WHO, *Chronic obstructive pulmonary disease (copd)*, Nov. 2024. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/chronic-obstructive-pulmonary-disease-(copd).

[16] W. H. Organization, *What are the who air quality guidelines?* Sep. 2021. [Online]. Available: https://www.who.int/news-room/feature-stories/detail/what-are-the-who-air-quality-guidelines.

[17] Netatmo, *Netatmo - indoor air monitor*, en-US. [Online]. Available: https://www.netatmo.com/smart-indoor-air-quality-monitor/specifications.

[18] Netatmo, *Netatmo - carbon-monoxide*, en-US. [Online]. Available: https://www.netatmo.com/smart-carbon-monoxide-alarm/specifications.

[19] Evehome, *Evehome - eve room*, en-US. [Online]. Available: https://www.evehome.com/en/eve-room.

[20] Awair, *Awair - awair element*, en-US. [Online]. Available: https://uk.getawair.com/products/element.

[21] Uhoo, *Uhoo - smart air monitor*, en-US. [Online]. Available: https://getuhoo.com/smart-air-monitor.

[22] Uhoo, *Uhoo - premium*, en-US. [Online]. Available: https://getuhoo.com/premium.

[23] Accessed: January 11, 2025. [Online]. Available: https://airindex.eea.europa.eu/AQI/index.html (visited on 01/11/2025).

[24] Accessed: January 12, 2025. [Online]. Available: https://www.airnow.gov/aqi/aqi-basics/ (visited on 01/12/2025).

[25] [Online]. Available: https://components101.com/sensors/mq135-gas-sensor-for-air-quality.

[26] L. Zhengzhou Winsen Electronics Technology Co., *Tgs 842-for the detection of methane*. [Online]. Available: https://www.winsen-sensor.com/d/files/PDF/Semiconductor%20Gas%20Sensor/MQ138%20(Ver1.4)%20-%20Manual.pdf.

[27] SHARP, *Gp2y1010au0f*.

[28] D. Mark, *Die stamp marking : Ink stamp marking 1st digit 2nd digit year of production month of production a.*

[29] G. Morgado, O. Postolache, and J. D. Pereira, "Smart city air quality monitoring supported iot ecosystem," in *Proceedings of the International Conference on Sensing Technology, ICST*, IEEE Computer Society, 2023, ISBN: 9798350395341. DOI: 10.1109/ICST59744.2023.10460781.

[30] W. Y. Yi, K. M. Lo, T. Mak, K. S. Leung, Y. Leung, and M. L. Meng, *A survey of wireless sensor network based air pollution monitoring systems*, Dec. 2015. DOI: 10.3390/s151229859.

[31] Instructables, *How to: Sensirion sps30*, Jan. 2023. [Online]. Available: https://www.instructables.com/How-To-Sensirion-SPS30/.

[32] L. Chettri and R. Bera, *A comprehensive survey on internet of things (iot) toward 5g wireless systems*, Jan. 2020. DOI: 10.1109/JIOT.2019.2948888.

[33] U. Raza, P. Kulkarni, and M. Sooriyabandara, "Low power wide area networks: An overview," *IEEE Communications Surveys and Tutorials*, vol. 19, pp. 855–873, 2 Apr. 2017, ISSN: 1553877X. DOI: 10.1109/COMST.2017.2652320.

[34] G. Pan, Q. Wu, J. Cao, J. He, R. Fang, and D. Liao, *Automatic Stabilization of Zigbee Network*. IEEE Press, 2018, ISBN: 9781538669877.

[35] J. Li, N. Tang, X. Zhu, and J. Sui, *Study on ZigBee Network Architecture and Routing Algorithm*. IEEE, 2010, ISBN: 9781424468935.

[36] W. Wang, G. He, and J. Wan, *Research on Zigbee wireless communication technology*. IEEE, 2011, p. 3421, ISBN: 9781424481651.

[37] A. Varol and S. J. Danbatta, *Comparison of Zigbee, Z-Wave, Wi-Fi, and Bluetooth Wireless Technologies Used in Home Automation*. IEEE, 2019, ISBN: 9781728128276.

[38] M. B. Yassein, W. Mardini, and A. Khalil, *Smart Homes Automation using Z-wave Protocol*. IEEE, 2016, ISBN: 9781509055791.

[39] N. I. Osman and A. E. B, *Simulation and Modelling of LoRa and Sigfox Low Power Wide Area Network Technologies*. IEEE, 2018, ISBN: 9781538641231.

[40] M. Kais, B. Eddy, C. Frederic, and M. Fernand, *Overview of Cellular LPWAN Technologies for IoT Deployment: Sigfox, LoRaWAN, and NB-IoT*. Institute of Electrical and Electronics Engineers, 2018, p. 212, ISBN: 9781538632277.

[41] E. Katsiri, C. Karasoulas, and C. Keroglou, "Towards dense indoor environmental sensing with lorawan," Institute of Electrical and Electronics Engineers (IEEE), Sep. 2024, pp. 1–5, ISBN: 9798350308921. DOI: 10.1109/cnna60945.2023.10652627.

[42] T.-M. Carlos, A. Angel, C. Roberto, B. Ruben, and M. Alvaro, *Multi-Conference on Systems, Signals and Devices (SSD), 2014 11th International*. 2014, ISBN: 9781479938667.

[43] *Three-layer-architecture*. [Online]. Available: https://jelvix.com/wp-content/uploads/2022/01/three-layer-architecture.png.

[44] M. A. Hassan, "Relational and nosql databases: The appropriate database model choice," in *2021 22nd International Arab Conference on Information Technology,*

ACIT 2021, Institute of Electrical and Electronics Engineers Inc., 2021, ISBN: 9781665419956. DOI: 10.1109/ACIT53391.2021.9677042.

[45] M. M. Patil, A. Hanni, C. Tejeshwar, and P. Patil, *A qualitative analysis of the performance of MongoDB vs MySQL Database based on insertion and retriewal operations using a web/android application to explore Load Balancing – Sharding in MongoDB and its advantages*. IEEE, 2017, ISBN: 9781509032433.

[46] R. Aghi, S. Mehta, R. Chauhan, S. Chaudhary, and N. Bohra, "A comprehensive comparison of sql and mongodb databases," *International Journal of Scientific and Research Publications*, vol. 5, 2 2015, ISSN: 2250-3153. [Online]. Available: www.ijsrp.org.

[47] Q. Liu, J. Meng, D. Yu, Z. Qiao, J. Hou, and Z. Wu, "An implementation of power iot time series data based on influxdb," in *Proceedings - 2023 International Conference on Advances in Electrical Engineering and Computer Applications, AEECA 2023*, Institute of Electrical and Electronics Engineers Inc., 2023, pp. 34–39, ISBN: 9798350308082. DOI: 10.1109/AEECA59734.2023.00016.

[48] A. Aggoune and Z. Benratem, "ECG Data Visualization: Combining the power of Grafana and InfluxDB," *IEEE*, Mar. 2023. DOI: 10.1109/icaeccs56710.2023.10104857. [Online]. Available: https://doi.org/10.1109/icaeccs56710.2023.10104857.

[49] I. Hindi, M. Alyaman, A. AboZenah, A. Zaid, and M. Shrara, "Smart Alarm IoT System: Monitoring Elevator Traffic and Meteorological Data on Job Sites Using MQTT and InfluxDB integrated with Grafana," *IEEE*, pp. 1–6, Aug. 2024. DOI: 10.1109/icics63486.2024.10638309. [Online]. Available: https://doi.org/10.1109/icics63486.2024.10638309.

[50] A. Nayak, B. N. Merh, S. Chaudhari, and R. K. Agrawal, "Comparative analysis of nosql and time series databases for fast data acquisition system of indus-2," in *2024 3rd International Conference on Power, Control and Computing Technologies, ICPC2T 2024*, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 292–297, ISBN: 9798350349207. DOI: 10.1109/ICPC2T60072.2024.10475062.

[51] U. Kayaduman, T. Kale, and N. Sozen, "Transitioning iiot data processing: An experience with influxdb on kubernetes," in *Proceedings - 2024 11th International Conference on Future Internet of Things and Cloud, FiCloud 2024*, Institute of Electrical and Electronics Engineers Inc., 2024, pp. 247–252, ISBN: 9798331527198. DOI: 10.1109/FiCloud62933.2024.00045.

[52] *The best time-series databases compared*, Oct. 2024. [Online]. Available: https://www.timescale.com/learn/the-best-time-series-databases-compared.

[53] R. G. S. Ltd, *Db-engines ranking of time series dbms*, en-US. [Online]. Available: https://db-engines.com/en/ranking/time+series+dbms.

[54] D. BERNSTEIN, "From lxc to docker.pdf," Tech. Rep. [Online]. Available: https://linuxcontainers.org.

[55] T. Combe, T. P.-T. A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," Tech. Rep.

[56] LILYGO, *Lilygo ttgo t-beam devices | meshtastic*. [Online]. Available: `https://meshtastic.org/docs/hardware/devices/community-supported/lilygo/tbeam/`.

[57] SparkFun, *Ccs811/bme280 (qwiic) environmental combo breakout hookup guide*. [Online]. Available: `https://learn.sparkfun.com/tutorials/ccs811bme280-qwiic-environmental-combo-breakout-hookup-guide/all`.

[58] Sensirion, *Github - sensirion/arduino-avr-legacy-i2c-sps30: Arduino library for sensirion sps30*. [Online]. Available: `https://github.com/Sensirion/arduino-avr-legacy-i2c-sps30`.

[59] SparkFun, *Github - sparkfun/sparkfun_bme280_arduino_library: An arduino library to control the bme280 humidity and pressure sensor*. [Online]. Available: `https://github.com/sparkfun/SparkFun_BME280_Arduino_Library`.

APPENDIX A

Docker Composes

We will be showcasing a before and after for each docker compose, for ease of readability.

A.1. Caddy base docker compose

```
services:
  caddy:
    image: caddy:<version>
    restart: unless-stopped
    cap_add:
      - NET_ADMIN
    ports:
      - "80:80"
      - "443:443"
      - "443:443/udp"
    volumes:
      - $PWD/conf:/etc/caddy
      - $PWD/site:/srv
      - caddy_data:/data
      - caddy_config:/config

volumes:
  caddy_data:
  caddy_config:
```

A.2. Caddy final docker compose

```
services:
  caddy:
    image: caddy:2.10.0-alpine
    restart: unless-stopped
    cap_add:
      - NET_ADMIN
    ports:
      - 80:80
      - 443:443
    volumes:
      - $PWD/conf:/etc/caddy
      - $PWD/site:/srv
```

```
      - data/docker/caddy/data:/data
      - data/docker/caddy/config:/config
```

A.3. Caddyfile configuration

```
{
    http_port 80
        https_port 443
}

(security_headers) {
        header {
                Strict-Transport-Security "max-age=31536000; includeSubDomains"
                X-Frame-Options "DENY"
                X-XSS-Protection "1; mode=block"
                X-Content-Type-Options "nosniff"
                Referrer-Policy "no-referrer"
                X-Robots-Tag "noindex, nofollow, nosnippet, noarchive"
                Permissions-Policy "geolocation=(), microphone=(), camera=()"
        }
}

# Import
import caddyfiles/*.caddy
```

A.4. Grafana's caddy file configuration

```
grafana.customdomain.com {
    import security_headers

    log {
            output file /config/logs/grafana/access.log {
                    roll_size 10MB
                    roll_keep 5
                    roll_keep_for 7d
                }
        }
        tls {
                protocols tls1.2 tls1.3
                dns cloudflare {env.CF_API_TOKEN}
                propagation_delay 2m
                resolvers 1.1.1.1
        }

        encode gzip
        reverse_proxy <grafana_ip>
```

```
}
```

A.5. WireGuard default configuration

```
volumes:
  etc_wireguard:

services:
  wg-easy:
    environment:
      # Change Language:
      # (Supports: en, ua, ru, tr, no, pl, fr, de, ca, es, ko, vi, nl, is, pt, chs, cht,
        - LANG=en
        # Required:
        - WG_HOST=raspberrypi.local

        # Optional:
        - PASSWORD_HASH=
        - PORT=51821
        - WG_PORT=51820
        - WG_CONFIG_PORT=92820
        - WG_DEFAULT_ADDRESS=10.8.0.x
        - WG_DEFAULT_DNS=1.1.1.1
        - WG_MTU=1420
        - WG_ALLOWED_IPS=192.168.15.0/24, 10.0.1.0/24
        - WG_PERSISTENT_KEEPALIVE=25
        - WG_PRE_UP=echo "Pre Up" > /etc/wireguard/pre-up.txt
        - WG_POST_UP=echo "Post Up" > /etc/wireguard/post-up.txt
        - WG_PRE_DOWN=echo "Pre Down" > /etc/wireguard/pre-down.txt
        - WG_POST_DOWN=echo "Post Down" > /etc/wireguard/post-down.txt
        - UI_TRAFFIC_STATS=true
        - UI_CHART_TYPE=0 # (0 Charts disabled, 1 # Line chart, 2 # Area chart, 3 # Bar

    image: ghcr.io/wg-easy/wg-easy:14
    container_name: wg-easy
    volumes:
      - etc_wireguard:/etc/wireguard
    ports:
      - "51820:51820/udp"
      - "51821:51821/tcp"
    restart: unless-stopped
    cap_add:
      - NET_ADMIN
      - SYS_MODULE
     # - NET_RAW # Uncomment if using Podman
```

```
    sysctls:
      - net.ipv4.ip_forward=1
      - net.ipv4.conf.all.src_valid_mark=1
```

A.6. WireGuard configuration

```
networks:
  local_lan:
    name: local_lan
    driver: ipvlan
    driver_opts:
      ipvlan_mode: l2
      parent: <needschanging>
    ipam:
      driver: default
      config:
        - subnet: <needschanging>
          gateway: <needschanging>

services:
  wg-easy:
    image: ghcr.io/wg-easy/wg-easy:14
    container_name: wg-easy
    ports:
      - 51820:51820/udp
      - 51421:51421/tcp
    environment:
      - LANG=en
      - WG_HOST=
      - WG_DEVICE=
      - MAX_AGE=30
      - PASSWORD_HASH=<needschanging>
      - PORT=51421                # webui
      - WG_PORT=51820             # openWorld
      - WG_DEFAULT_ADDRESS=<needschanging>
      - WG_DEFAULT_DNS=1.1.1.1
      - WG_MTU=1420
      - WG_ALLOWED_IPS=0.0.0.0/0
      - WG_PERSISTENT_KEEPALIVE=25
      - UI_TRAFFIC_STATS=true
      - UI_CHART_TYPE=3
      - WG_ENABLE_ONE_TIME_LINKS=true
      - UI_ENABLE_SORT_CLIENTS=true
      - WG_ENABLE_EXPIRES_TIME=true
    volumes:
```

```
    - /data/docker/wgeasy:/etc/wireguard
  restart: unless-stopped
  cap_add:
    - NET_ADMIN
    - SYS_MODULE
  sysctls:
    - net.ipv4.ip_forward=1
    - net.ipv4.conf.all.src_valid_mark=1
    - net.ipv6.conf.all.disable_ipv6=1          # Disables IPv6 system-wide inside the c
    - net.ipv6.conf.default.disable_ipv6=1      # Prevents new interfaces from using IPv
    - net.ipv6.conf.lo.disable_ipv6=1           # Disables IPv6 on the loopback interfac
  networks:
    local_lan:
      ipv4_address: <needschanging>
  security_opt:
  - no-new-privileges:true
```

A.7. InfluxDB default configuration

```
services:
  influxdb2:
    image: influxdb:2
    ports:
      - 8086:8086
    environment:
      DOCKER_INFLUXDB_INIT_MODE: setup
      DOCKER_INFLUXDB_INIT_USERNAME_FILE: /run/secrets/influxdb2-admin-username
      DOCKER_INFLUXDB_INIT_PASSWORD_FILE: /run/secrets/influxdb2-admin-password
      DOCKER_INFLUXDB_INIT_ADMIN_TOKEN_FILE: /run/secrets/influxdb2-admin-token
      DOCKER_INFLUXDB_INIT_ORG: docs
      DOCKER_INFLUXDB_INIT_BUCKET: home
    secrets:
      - influxdb2-admin-username
      - influxdb2-admin-password
      - influxdb2-admin-token
    volumes:
      - type: volume
        source: influxdb2-data
        target: /var/lib/influxdb2
      - type: volume
        source: influxdb2-config
        target: /etc/influxdb2
secrets:
  influxdb2-admin-username:
    file: ~/.env.influxdb2-admin-username
```

```
  influxdb2-admin-password:
    file: ~/.env.influxdb2-admin-password
  influxdb2-admin-token:
    file: ~/.env.influxdb2-admin-token
volumes:
  influxdb2-data:
  influxdb2-config:
```

A.8. Grafana default configuration

```
services:
  grafana:
    image: grafana/grafana-enterprise
    container_name: grafana
    restart: unless-stopped
    # if you are running as root then set it to 0
    # else find the right id with the id -u command
    user: '0'
    ports:
      - '3000:3000'
    # adding the mount volume point which we create earlier
    volumes:
      - '$PWD/data:/var/lib/grafana'
```

A.9. Grafana and InfluxDB configuration

```
services:
  grafana:
    image: grafana/grafana-oss
    container_name: grafana
    restart: unless-stopped
    user: 1000:1000
    environment:
      - GF_SECURITY_ADMIN_USER=${grafana_admin}
      - GF_SECURITY_ADMIN_PASSWORD=${grafana_pwd}
    ports:
      - 3000:3000
    volumes:
      - /data/docker/grafana:/var/lib/grafana
    depends_on:
      - influxdb

  influxdb:
    container_name: influxdb
    image: influxdb:2.7.12
    ports:
```

```
  - 8086:8086
environment:
  - DOCKER_INFLUXDB_INIT_MODE=setup
  - DOCKER_INFLUXDB_INIT_USERNAME=${influxdb_username}
  - DOCKER_INFLUXDB_INIT_PASSWORD=${influxdb_password}
  - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=${influxdb_adminToken}
  - DOCKER_INFLUXDB_INIT_ORG=docs
  - DOCKER_INFLUXDB_INIT_BUCKET=home
volumes:
  - /data/docker/influxdb/data:/var/lib/influxdb2
  - /data/docker/influxdb/config:/etc/influxdb2
```

APPENDIX B

Code

B.1. BME280 Code

```
#include <SparkFunBME280.h>

#define SDA_PIN 13
#define SCL_PIN 14

BME280 myBME280;

void setup() {
    Serial.begin(115200);
    Wire.begin(SDA_PIN, SCL_PIN);
    delay(1000);
    myBME280.settings.commInterface = I2C_MODE;
    myBME280.settings.I2CAddress = 0x77;
    myBME280.settings.runMode = 3;  //Forced mode
    myBME280.settings.tStandby = 0;
    myBME280.settings.filter = 0;
    myBME280.settings.tempOverSample = 1;
    myBME280.settings.pressOverSample = 1;
    myBME280.settings.humidOverSample = 1;
    delay(10);
    if (!myBME280.begin()) {
        while (1);
    }
}

void loop() {
  sensorReading();
}

void sensorReading() {
    float temp = myBME280.readTempC();
    float humidity = myBME280.readFloatHumidity();
    float pressure = (myBME280.readFloatPressure() / 100.0);

    Serial.println("Temp: ");
```

```
    Serial.println(String(temp));
    Serial.println("C");

    Serial.println("Humidity: ");
    Serial.println(String(humidity));
    Serial.println("%");


    Serial.println("Pressure: ");
    Serial.println(String(pressure));
    Serial.println("Pa");
}
```

B.2. BME280 sending data to InfluxDB via HTTP

```
#include <SparkFunBME280.h>

#define SDA_PIN 13
#define SCL_PIN 14

const String influxToken = "OurSecretToken"
BME280 myBME280;

void setup() {
    Serial.begin(115200);
    Wire.begin(SDA_PIN, SCL_PIN);
    delay(1000);
    myBME280.settings.commInterface = I2C_MODE;
    myBME280.settings.I2CAddress = 0x77;
    myBME280.settings.runMode = 3;  //Forced mode
    myBME280.settings.tStandby = 0;
    myBME280.settings.filter = 0;
    myBME280.settings.tempOverSample = 1;
    myBME280.settings.pressOverSample = 1;
    myBME280.settings.humidOverSample = 1;
    delay(10);
    if (!myBME280.begin()) {
        while (1);
    }
}

void loop() {
  sensorReading();
}
```

```
void sensorReading() {
    float temp = myBME280.readTempC();
    float humidity = myBME280.readFloatHumidity();
    float pressure = (myBME280.readFloatPressure() / 100.0);

    sendToInflux(temp, humidity, pressure);
}

void sendToInflux(float temperature, float humidity, float pressure) {
if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    String serverUrl = "http://192.168.3.7:8086/api/v2/write?
    org=Thesis&bucket=Thesis&precision=ns";
    http.begin(serverUrl);
    http.addHeader("Authorization", "Token " + influxToken);
    http.addHeader("Content-Type", "text/plain");

    String data = "air_quality,location=Home Temperature=" +
    String(temperature) + ",Humidity=" + String(humidity) +
    ",Pressure=" + String(pressure);
    int httpResponseCode = http.POST(data);

    if (httpResponseCode != 204) {
      Serial.println("HTTP Response: " + String(httpResponseCode));
    }
    http.end();
  } else {
    Serial.println("WiFi status is down" + WiFi.status());
  }
}
```

B.3. CCS811 with BME280

```
#include <SparkFunBME280.h>
#include <SparkFunCCS811.h>

#define SDA_PIN 13
#define SCL_PIN 14

BME280 myBME280;

void setup() {
    Serial.begin(115200);
    Wire.begin(SDA_PIN, SCL_PIN);
```

```
    delay(1000);
    myBME280.settings.commInterface = I2C_MODE;
    myBME280.settings.I2CAddress = 0x77;
    myBME280.settings.runMode = 3;  //Forced mode
    myBME280.settings.tStandby = 0;
    myBME280.settings.filter = 0;
    myBME280.settings.tempOverSample = 1;
    myBME280.settings.pressOverSample = 1;
    myBME280.settings.humidOverSample = 1;
    delay(10);
    if (!myBME280.begin()) {
        while (1);
    }
    if (!myCCS811.begin()) {
        while (1);
  }
}

void loop() {
  sensorReading();
}

void sensorReading() {
  if (!myCCS811.dataAvailable()) {
    Serial.println("Failed to read from CCS811 sensor");
    return;
  }
  float temp = myBME280.readTempC();
  float humidity = myBME280.readFloatHumidity();
  float pressure = (myBME280.readFloatPressure() / 100.0);

  myCCS811.setEnvironmentalData(humidity, temp);
  myCCS811.readAlgorithmResults();
  float carbonDioxide = myCCS811.getCO2();
  float tvoc = myCCS811.getTVOC();
  delay(2000);
}
```

B.4. CCS811, BME280 sending data to InfluxDB via HTTP

```
#include <SparkFunBME280.h>
#include <SparkFunCCS811.h>

#define SDA_PIN 13
#define SCL_PIN 14
```

```
const String influxToken = "OurSecretToken"
BME280 myBME280;

void setup() {
    Serial.begin(115200);
    Wire.begin(SDA_PIN, SCL_PIN);
    delay(1000);
    myBME280.settings.commInterface = I2C_MODE;
    myBME280.settings.I2CAddress = 0x77;
    myBME280.settings.runMode = 3;  //Forced mode
    myBME280.settings.tStandby = 0;
    myBME280.settings.filter = 0;
    myBME280.settings.tempOverSample = 1;
    myBME280.settings.pressOverSample = 1;
    myBME280.settings.humidOverSample = 1;
    delay(10);
    if (!myBME280.begin()) {
        while (1);
    }
    if (!myCCS811.begin()) {
        while (1);
  }
}

void loop() {
  sensorReading();
}

void sensorReading() {
  if (!myCCS811.dataAvailable()) {
    Serial.println("Failed to read from CCS811 sensor");
    return;
  }
  float temp = myBME280.readTempC();
  float humidity = myBME280.readFloatHumidity();
  float pressure = (myBME280.readFloatPressure() / 100.0);

  myCCS811.setEnvironmentalData(humidity, temp);
  myCCS811.readAlgorithmResults();
  float carbonDioxide = myCCS811.getCO2();
  float tvoc = myCCS811.getTVOC();

  sendToInflux(temp, humidity, pressure, carbonDioxide, tvoc);
```

```
  delay(2000);
}

void sendToInflux(float temperature, float humidity,
float pressure, float carbonDioxide, float tvoc) {
if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    String serverUrl = "http://192.168.3.7:8086/api/v2/write?
    org=Thesis&bucket=Thesis&precision=ns";
    http.begin(serverUrl);
    http.addHeader("Authorization", "Token " + influxToken);
    http.addHeader("Content-Type", "text/plain");

    String data = "air_quality,location=Home Temperature=" + String(temperature) +
                                            ",Humidity=" + String(humidity) +
                                            ",Pressure=" + String(pressure) +
                                            ",CO2=" + String(carbonDioxide) +
                                            ",TVOC=" + String(tvoc);
    int httpResponseCode = http.POST(data);

    if (httpResponseCode != 204) {
      Serial.println("HTTP Response: " + String(httpResponseCode));
    }
    http.end();
  } else {
    Serial.println("WiFi status is down" + WiFi.status());
  }
}
```

B.5. SPS30 code example provided by Library

```
#include <sps30.h>

// Example arduino sketch, based on
// https://github.com/Sensirion/embedded-sps/blob/master/sps30-i2c/sps30_example_usage.c


// uncomment the next line to use the serial plotter
// #define PLOTTER_FORMAT

void setup() {
  int16_t ret;
  uint8_t auto_clean_days = 4;
  uint32_t auto_clean;
```

```
  Serial.begin(9600);
  delay(2000);

  sensirion_i2c_init();

  while (sps30_probe() != 0) {
    Serial.print("SPS sensor probing failed\n");
    delay(500);
  }

#ifndef PLOTTER_FORMAT
  Serial.print("SPS sensor probing successful\n");
#endif /* PLOTTER_FORMAT */

  ret = sps30_set_fan_auto_cleaning_interval_days(auto_clean_days);
  if (ret) {
    Serial.print("error setting the auto-clean interval: ");
    Serial.println(ret);
  }

  ret = sps30_start_measurement();
  if (ret < 0) {
    Serial.print("error starting measurement\n");
  }

#ifndef PLOTTER_FORMAT
  Serial.print("measurements started\n");
#endif /* PLOTTER_FORMAT */

#ifdef SPS30_LIMITED_I2C_BUFFER_SIZE
  Serial.print("Your Arduino hardware has a limitation that only\n");
  Serial.print("  allows reading the mass concentrations. For more\n");
  Serial.print("  information, please check\n");
  Serial.print("  https://github.com/Sensirion/arduino-sps#esp8266-partial-legacy-suppor
  Serial.print("\n");
  delay(2000);
#endif

  delay(1000);
}

void loop() {
  struct sps30_measurement m;
```

```
  char serial[SPS30_MAX_SERIAL_LEN];
  uint16_t data_ready;
  int16_t ret;

  do {
    ret = sps30_read_data_ready(&data_ready);
    if (ret < 0) {
      Serial.print("error reading data-ready flag: ");
      Serial.println(ret);
    } else if (!data_ready)
      Serial.print("data not ready, no new measurement available\n");
    else
      break;
    delay(100); /* retry in 100ms */
  } while (1);

  ret = sps30_read_measurement(&m);
  if (ret < 0) {
    Serial.print("error reading measurement\n");
  } else {

#ifndef PLOTTER_FORMAT
    Serial.print("PM  1.0: ");
    Serial.println(m.mc_1p0);
    Serial.print("PM  2.5: ");
    Serial.println(m.mc_2p5);
    Serial.print("PM  4.0: ");
    Serial.println(m.mc_4p0);
    Serial.print("PM 10.0: ");
    Serial.println(m.mc_10p0);

#ifndef SPS30_LIMITED_I2C_BUFFER_SIZE
    Serial.print("NC  0.5: ");
    Serial.println(m.nc_0p5);
    Serial.print("NC  1.0: ");
    Serial.println(m.nc_1p0);
    Serial.print("NC  2.5: ");
    Serial.println(m.nc_2p5);
    Serial.print("NC  4.0: ");
    Serial.println(m.nc_4p0);
    Serial.print("NC 10.0: ");
    Serial.println(m.nc_10p0);

    Serial.print("Typical particle size: ");
```

```
    Serial.println(m.typical_particle_size);
#endif

    Serial.println();

#else
    // since all values include particles smaller than X, if we want to create buckets w
    // need to subtract the smaller particle count.
    // This will create buckets (all values in micro meters):
    // - particles        <= 0,5
    // - particles > 0.5, <= 1
    // - particles > 1,   <= 2.5
    // - particles > 2.5, <= 4
    // - particles > 4,   <= 10

    Serial.print(m.nc_0p5);
    Serial.print(" ");
    Serial.print(m.nc_1p0  - m.nc_0p5);
    Serial.print(" ");
    Serial.print(m.nc_2p5  - m.nc_1p0);
    Serial.print(" ");
    Serial.print(m.nc_4p0  - m.nc_2p5);
    Serial.print(" ");
    Serial.print(m.nc_10p0 - m.nc_4p0);
    Serial.println();


#endif /* PLOTTER_FORMAT */

  }

  delay(1000);
}
```

B.6. Wifi testing

```
#include <WiFi.h>

#define WIFI_SSID "wifi_ssid"
#define WIFI_PASSWORD "wifi_pwd"


void setup() {
// WiFi initialization
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
```

```
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
}

void loop() {
  ensureWiFiConnected();
}


void ensureWiFiConnected() {
  if (WiFi.status() != WL_CONNECTED) {
    Serial.println("WiFi disconnected. Reconnecting...");
    WiFi.disconnect(); // optional: clean up state
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

    unsigned long startAttemptTime = millis();
    const unsigned long timeout = 10000; // 10 seconds

    while (WiFi.status() != WL_CONNECTED && millis() - startAttemptTime < timeout) {
      delay(500);
      Serial.print(".");
    }

    if (WiFi.status() == WL_CONNECTED) {
      Serial.println("\nWiFi reconnected successfully.");
      Serial.println("IP Address: " + WiFi.localIP().toString());
    } else {
      Serial.println("\nFailed to reconnect to WiFi.");
    }
  }
```

B.7. I2C Lookup

```
#include <Wire.h>
#define SDA_PIN 13
#define SCL_PIN 14
void setup() {
    Wire.begin(SDA_PIN, SCL_PIN);
    Serial.begin(115200);
    delay(1000);
    Serial.println("Scanning...");

    for (byte address = 1; address < 127; ++address) {
```

```
    Wire.beginTransmission(address);
        if (Wire.endTransmission() == 0) {
        Serial.print("Found device at 0x");
        Serial.println(address, HEX);
        }
    }
}

void loop() {}
```

B.8. CO_2 and PM_{10} Queries

```
from(bucket: "home")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "Air Quality")
  |> filter(fn: (r) => r["location"] == "Home")
  |> filter(fn: (r) => r["_field"] == "Temperature")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
```

```
from(bucket: "home")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "Air Quality")
  |> filter(fn: (r) => r["location"] == "Home")
  |> filter(fn: (r) => r["_field"] == "PM10_0")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
```

B.9. Code used for the POST

```
#include <SparkFunBME280.h>
#include <SparkFunCCS811.h>

#define SDA_PIN 13
#define SCL_PIN 14

const String influxToken = "OurSecretToken"
BME280 myBME280;

void setup() {
    Serial.begin(115200);
    Wire.begin(SDA_PIN, SCL_PIN);
    delay(1000);
    myBME280.settings.commInterface = I2C_MODE;
    myBME280.settings.I2CAddress = 0x77;
    myBME280.settings.runMode = 3;  //Forced mode
```

```
    myBME280.settings.tStandby = 0;
    myBME280.settings.filter = 0;
    myBME280.settings.tempOverSample = 1;
    myBME280.settings.pressOverSample = 1;
    myBME280.settings.humidOverSample = 1;
    delay(10);
    if (!myBME280.begin()) {
        while (1);
    }
    if (!myCCS811.begin()) {
        while (1);
  }
}

void loop() {
  sensorReading();
}

void sensorReading() {
  if (!myCCS811.dataAvailable()) {
    Serial.println("Failed to read from CCS811 sensor");
    return;
  }
  float temp = myBME280.readTempC();
  float humidity = myBME280.readFloatHumidity();
  float pressure = (myBME280.readFloatPressure() / 100.0);

  myCCS811.setEnvironmentalData(humidity, temp);
  myCCS811.readAlgorithmResults();
  float carbonDioxide = myCCS811.getCO2();
  float tvoc = myCCS811.getTVOC();

  sendToInflux(temp, humidity, pressure, carbonDioxide, tvoc);
  delay(2000);
}

void sendToInflux(float temperature, float humidity, float pressure, float carbonDioxide
if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    String serverUrl = "http://192.168.3.7:8086/api/v2/write?
    org=Thesis&bucket=Thesis&precision=ns";
    http.begin(serverUrl);
    http.addHeader("Authorization", "Token " + influxToken);
```

```
    http.addHeader("Content-Type", "text/plain");

    String data = "air_quality,location=Home Temperature=" + String(temperature) +
                                                ",Humidity=" + String(humidity) +
                                                ",Pressure=" + String(pressure) +
                                                ",CO2=" + String(carbonDioxide) +
                                                ",TVOC=" + String(tvoc);
    int httpResponseCode = http.POST(data);

    if (httpResponseCode != 204) {
      Serial.println("HTTP Response: " + String(httpResponseCode));
    }
    http.end();
  } else {
    Serial.println("WiFi status is down" + WiFi.status());
  }
}
```

B.10. Final Code used for deployment

```
#include <WiFi.h>
#include <SparkFunCCS811.h>
#include <SparkFunBME280.h>
#include <sps30.h>
#include <HTTPClient.h>
#include <InfluxDbClient.h>
#include <InfluxDbCloud.h>

#define DEVICE "ESP32"
#define SDA_PIN 13
#define SCL_PIN 14
#define CCS811_ADDR 0x5B
#define WIFI_SSID "wifi_ssid"
#define WIFI_PASSWORD "wifi_pwd"
#define INFLUXDB_URL "https://influx.example.com"
#define INFLUXDB_TOKEN "token"
#define INFLUXDB_ORG "organization"
#define INFLUXDB_BUCKET "home"
#define TZ_INFO "UTC1"

InfluxDBClient client(INFLUXDB_URL, INFLUXDB_ORG, INFLUXDB_BUCKET, INFLUXDB_TOKEN, Influx
Point sensor("Air Quality");
CCS811 myCCS811(CCS811_ADDR);
BME280 myBME280;
```

```
struct AirQualityData {
  float temp, humidity, pressure;
  float carbonDioxide, tvoc;
  float pm_1_0, pm_2_5, pm_4_0, pm_10_0;
  float nc_0_5, nc_1_0, nc_2_5, nc_4_0, nc_10_0;
  float typical_size;
};

void setup() {
  Serial.begin(115200);
  Wire.begin(SDA_PIN, SCL_PIN);
  delay(1000);

  // BME280 initialization
  myBME280.settings.commInterface = I2C_MODE;
  myBME280.settings.I2CAddress = 0x77;
  myBME280.settings.runMode = 3;
  myBME280.settings.tStandby = 0;
  myBME280.settings.filter = 0;
  myBME280.settings.tempOverSample = 1;
  myBME280.settings.pressOverSample = 1;
  myBME280.settings.humidOverSample = 1;
  if (!myBME280.begin()) {
    Serial.println("BME280 not found. Check wiring!");
    while (1);
  }
  // CCS811 initialization
  if (!myCCS811.begin()) {
    Serial.println("CCS811 not found. Check wiring!");
    while (1);
  }
  // SPS30 initialization
  sensirion_i2c_init();
  while (sps30_probe() != 0) {
    Serial.println("SPS30 not found. Retrying...");
    delay(500);
  }
  sps30_set_fan_auto_cleaning_interval_days(4);
  if (sps30_start_measurement() < 0) {
    Serial.println("SPS30 measurement start failed!");
    while (1);
  }
  Serial.println("Sensors initialized.");
  // WiFi initialization
```

```
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  timeSync(TZ_INFO, "pt.pool.ntp.org", "time.nis.gov");
  sensor.addTag("location", "Home");  // Add location tag for context in InfluxDB
  if (client.validateConnection()) {
    Serial.println(client.getServerUrl());
  } else {
    Serial.println(client.getLastErrorMessage());
  }

}

void loop() {
  ensureWiFiConnected();
  AirQualityData data = readSensors();
  sendToInflux(data);
  delay(5000);
}

AirQualityData readSensors() {
  AirQualityData d;

  // BME280 readings
  d.temp = myBME280.readTempC();
  d.humidity = myBME280.readFloatHumidity();
  d.pressure = myBME280.readFloatPressure() / 100.0;

  // CCS811 readings
  myCCS811.setEnvironmentalData(d.humidity, d.temp);
  if (myCCS811.dataAvailable()) {
    myCCS811.readAlgorithmResults();
    d.carbonDioxide = myCCS811.getCO2();
    d.tvoc = myCCS811.getTVOC();
  }

  // SPS30 readings
  struct sps30_measurement m;
  uint16_t data_ready;
  int16_t ret;
```

```
  do {
    ret = sps30_read_data_ready(&data_ready);
    delay(100);
  } while (ret == 0 && !data_ready);

  if (sps30_read_measurement(&m) >= 0) {
    d.pm_1_0 = m.mc_1p0;
    d.pm_2_5 = m.mc_2p5;
    d.pm_4_0 = m.mc_4p0;
    d.pm_10_0 = m.mc_10p0;
    d.nc_0_5 = m.nc_0p5;
    d.nc_1_0 = m.nc_1p0;
    d.nc_2_5 = m.nc_2p5;
    d.nc_4_0 = m.nc_4p0;
    d.nc_10_0 = m.nc_10p0;
    d.typical_size = m.typical_particle_size;
  }

  return d;
}

void sendToInflux(const AirQualityData& data) {
    sensor.clearFields();
    sensor.addField("Temperature", data.temp);
    sensor.addField("Humidity", data.humidity);
    sensor.addField("Pressure", data.pressure);
    sensor.addField("CO2", data.carbonDioxide);
    sensor.addField("TVOC", data.tvoc);
    sensor.addField("PM1_0", data.pm_1_0);
    sensor.addField("PM2_5", data.pm_2_5);
    sensor.addField("PM4_0", data.pm_4_0);
    sensor.addField("PM10_0", data.pm_10_0);
    sensor.addField("NC0_5", data.nc_0_5);
    sensor.addField("NC1_0", data.nc_1_0);
    sensor.addField("NC2_5", data.nc_2_5);
    sensor.addField("NC4_0", data.nc_4_0);
    sensor.addField("NC10_0", data.nc_10_0);
    sensor.addField("TypicalSize", data.typical_size);

  if (!client.writePoint(sensor)) {
    Serial.print("InfluxDB write failed: ");
    Serial.println(client.getLastErrorMessage());
  }
}
```

```
void ensureWiFiConnected() {
  if (WiFi.status() != WL_CONNECTED) {
    Serial.println("WiFi disconnected. Reconnecting...");
    WiFi.disconnect(); // optional: clean up state
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

    unsigned long startAttemptTime = millis();
    const unsigned long timeout = 10000; // 10 seconds

    while (WiFi.status() != WL_CONNECTED && millis() - startAttemptTime < timeout) {
      delay(500);
      Serial.print(".");
    }

    if (WiFi.status() == WL_CONNECTED) {
      Serial.println("\nWiFi reconnected successfully.");
      Serial.println("IP Address: " + WiFi.localIP().toString());
    } else {
      Serial.println("\nFailed to reconnect to WiFi.");
    }
  }
}
```

APPENDIX C

Files

C.1. Unbound local domains

```
# local zone
    local-zone: "local.home" static
    local-data: "local.home A 10.10.10.10"
    local-data: "grafana.local.home CNAME local.home"
    local-data: "influxdb.local.home CNAME local.home"
    local-data: "service1.local.home CNAME local.home"
    local-data: "serviceN.local.home CNAME local.home"
```