IoT Car Sensor and Mobile App for Carbon Monoxide and Temperature Monitoring

Ching Ziyuan

COM3610

Vitaveska Lanfranchi

2nd of May 2023

This report is submitted in partial fulfilment of the requirement for the degree of Computer Science BSc by Ching Ziyuan.

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Ching Ziyuan

Abstract

In the heat waves that hit Europe between June and August 2022, more than 20000 deaths occurred. Heat waves also disproportionately impact older people – 90% of heat-related deaths were among people that were >65 years old in the U.K [1].

Carbon monoxide (CO) poisoning is a risk often present in ill-maintained cars, and cars that run in a tight, enclosed space. Because CO is an odourless gas, accidents can happen very easily in environments with high CO concentrations. Long-term exposure to low-level CO can lead to neurological issues.

This project aims to combat those risks by developing a portable IoT (Internet of Things) system that can output live readings of temperature and CO, and warn users when those readings reach a dangerous level through an app.

Acknowledgements

My supervisor Vitaveska Lanfranchi has my undying gratitude. She has been nothing but supportive and helpful. Her words of encouragement and advice have shepherded me well throughout this dissertation.

I would like to thank my second marker Zhixiang Chen for making good points about my dissertation that I did not think about.

I am most grateful to Thomas David Plumpton for being patient with me when I was learning how to solder. Fixing up my earth-shatteringly terrible solder job on my logic level converter was no small feat. May you always roll nat20s.

I'd like to also thank Jason Yap Min Hsin, the big chemist in the sky, for helping me validate that smothering candles in a jar does indeed produce CO, which helped me with testing the MQ-7 sensor.

Special thanks to Daniel Lim Hui Yuan, Hayden Teh Ke Qing, and Praveen Araasu for wrecking my productivity by the simple virtue of being fun to be around.

Table of Contents

1. Introduction

1.1 Background

Heat stroke is a condition where one's body temperature rises above 40°C. When left untreated, heat stroke can quickly damage a victim's vital organs – sometimes permanently. Coupled with the fact that heat stroke victims are often elderly and that they have reduced sensitivity to higher temperatures and humidities, a preventative measure is needed to mitigate the risks of heat stroke occurring.

Carbon monoxide is an odourless and colourless gas that is emitted from cars. Mild exposure to CO can cause flu-like symptoms, while heavy exposure can lead to confusion, vomiting, loss of the ability to coordinate muscles, loss of consciousness, and death [2]. Small holes in vehicle floors can also cause noticeably increased CO levels in the vehicle. This can cause imperceptible impaired brain functions in passengers at best, and death at worst [3]. Because CO is practically invisible to human perceptual systems, there is a need for a monitoring system. Having a monitoring system inside the vehicle also allows the users to better understand when to take their vehicles for servicing, as prolonged increased levels in CO probably means that there is something wrong with the vehicle floor and/or exhaust system.

Vehicles are starting to be increasingly integrated with IoT systems. This includes systems such as self-driving systems, fleet management, condition monitoring systems for components, entertainment systems, and more [4]. These systems are all embedded into the automotives however, and non-embedded IoT systems for cars are much more novel, with the exception of one device – the dashcam.

1.2 Description of Project

This project aims to produce a portable IoT system to combat the aforementioned problems in the Background section. This is achieved through monitoring CO and temperature levels, and displaying the readings back to the user in the app through a helpful and digestible way. The user will also be alerted when the CO and/or temperature levels get dangerous. The system will comprise of two parts: the device, and the app. The device will be powered by a microcomputer/microcontroller, with the ability to read temperature and CO levels through its sensors. The device should be able to communicate this information to the app. The app would be able to represent this information in numerous ways (as line graphs, gauges, etc.). The app should also alert the user about the readings exceeding threshold values.

This device will need to have certain requirements to be feasible:

- Be portable and battery powered
- Take advantage of the car's accessory ports
- Be able to connect to the Internet while on the move so that it can communicate with the app
- As the system is forecasted to be mainly used by older people, it has to be easy to use
- Be able to work well in high heat

To ensure that the system works properly with the specifications above in mind, the literature and technology review sections looks through the following:

- causes and symptoms of heat stroke and carbon monoxide poisoning
- accessibility issues
- microcomputer/microcontroller literature
- legal issues that stem from processing data and from the system itself
- choice of hardware
- circuitry and sensors to be connected to the computer
- ways to power the computer
- operating systems
- mobile development paradigms
- communication methods between the device and the app

After drawing conclusions from the research from the previous section, the requirements and analysis section lays out concrete project objectives by splitting them into functional and non-functional requirements, which are prioritised through the MoSCoW technique.

The design of the system architecture and app interface, and the testing plan, along with the app's implementation will all be influenced by the functional and non-functional requirements that were dictated above.

After the implementation is finished, the system will be tested to see if it is reliable, accessible, and if it meets the project requirements. The report will culminate with a discussion of how complete the implementation of the system is, and how well the system works as a whole.

1.3 Relationship Between the Project and the Degree Programme

It is undeniable that the project benefits from the knowledge obtained from the BSc Computer Science course. This knowledge is more general than specialised however, as no specific modules were of use directly, besides the Web and Internet Technology module, the Introduction to Software Engineering module, and the Software Hut module, all of which were relevant as all of them involved building websites as their assessments. While helpful, they did not have material on building web apps.

The Internet of Things module was not as useful as one might expect. The module uses ESP32s and C++ as the programming language, while this project uses a Raspberry Pi 3B, with Python, Vanilla Javascript, and HTML as the languages of the system. While learning how to prototype would have been quite helpful to the dissertation, the design of the schematic has already been finalised over the Christmas break. The IFTTT example was somewhat helpful, as IFTTT also uses POST and GET requests, although Javascript demands a different way of doing HTTP requests through async functions. If the Internet of Things module took place in the first semester instead of the second, it would have been far more beneficial to the project.

Thomas David Plumpton, an assistant teaching technician in the lab was very helpful however, as he taught me how to do soldering correctly and helped me fix up the logic level converter that was soldered terribly on the first pass. I would not have had access to a soldering iron and came across Thomas if not for the fact that I had the Internet of Things module.

2. Literature Review

2.1 Overview

This chapter will provide some background knowledge on heat strokes and carbon monoxide poisoning as it will influence design choices in this project. Due to the nature of the application developed as part of this project, the average user will likely be a middle-aged to old-aged person. As such, this chapter will also discuss accessibility options seen in web design and/or software development.

Existing literature about microcomputer projects will also be reviewed to gather design choices relevant to the project, such as architectural design and hardware choices.

2.2 Heat Strokes

Heat strokes occur when the core temperature of a person rises above 40°C. In normal circumstances, the body is regulated at approximately 37°C [5]. Heat strokes are classified into two categories: non-exertional and exertional. Exertional heat strokes occur when individuals perform vigorous exercises/labour, while non-exertional heat strokes occur when individuals perform low-level physical activities [6].

Heat exhaustion is a less severe form of heat stroke, which is also caused by one's rising core temperature. Some of the symptoms of heat exhaustion are extreme thirst, dizziness, tiredness, weakness, and excessive sweating [7]. Heat exhaustion can usually be treated by going into a cooler place, rehydrating, and applying cooling methods such as using ice packs. Heat exhaustion can progress into heat stroke if left untreated.

Non-exertional heat strokes occur mostly to people that already have underlying issues, which causes increased susceptibility to heat strokes. Some of the factors include old age, cardiovascular disease, neurological dysfunctions, renal damage, and obesity [8].

Heat strokes can range from mild to life-threatening. In mild cases, a person can experience dizziness, faintness, and/or muscle pain. In severe cases, a person might experience seizures, delirium, disorientation, vomiting, and/or general impairment of consciousness. Life-threatening cases can cause liver, cardiovascular, and circulatory system failure [9]. As such, it is preferable to focus on the prevention of heat stroke rather than treating the symptoms.

It is important to note that heat strokes can "build up" over days. The elderly tend to be less sensitive to higher temperatures and humidity even though they are more susceptible to heat strokes, which causes them to neglect their hydration and to seek out cool places, causing their core temperature to slowly rise [9]. Temperatures between 32°C and 40°C can induce heat cramps and exhaustion. Temperatures between 40°C and 54°C make contracting heat exhaustion more likely. Temperatures beyond 54°C can easily lead to heat stroke. These temperature ranges would also be lower for older people [10]. As such, a user with a charge susceptible to heat strokes should watch out for car temperatures of about 35°C, and should perform basic check-ups on them if the car temperatures reach or exceed 35°C.

2.3 Carbon Monoxide Poisoning

As mentioned above, CO is an odourless and colourless gas, making it virtually impossible for humans to detect. CO poisoning kills 430 people and hospitalises around 50000 people every year in the U.S. alone [11]. CO is emitted from gasoline based machines (cars, generators, pressure washers, etc.), charcoal grills, and more. Prolonged exposure to CO levels of 70 parts per million (PPM) will induce headache, fatigue, nausea, and other symptoms of CO poisoning. Prolonged exposure to >150 ppm can cause loss of consciousness, confusion, and death [12]. It has been noted that normal CO levels in serviced cars vary from 12 to 60 ppm [13]. As such, it would be advisable to set the threshold level for the alarm at 70 ppm, a value that is above average in cars, but at the lower end of the range where exposure could be dangerous.

2.4 Accessibility

2.4.1 The Importance of Accessibility

It has been inferred that the age of the users will skew towards middle to old. This is because the elderly and children are the likeliest groups to suffer from heat stroke in cars. Their caretakers, of whom are the primary demographic of this app, will need to be able to use the app effortlessly to monitor their charges well. Carbon monoxide poisoning does not play a significant role in influencing the demographics of this system as it can occur in any age group.

The World Wide Web Consortium (W3C) argues that “the Web is fundamentally designed to work for all people, whatever their hardware, software, language, location, or ability. When the Web meets this goal, it is accessible to people with a diverse range of hearing, movement, sight, and cognitive ability” [14]. Making software more accessible is not only a point of equality, but it also helps to enhance the experience of the software as a whole. For example, people with computer vision syndrome might not be regarded as being traditionally disabled, but they will appreciate changes made to accommodate the visually impaired all the same.

Furthermore, it is necessary that all software/websites used in the UK must meet the Web Content Accessibility Guidelines (WCAG) 2.1 AA rated standards in accordance with the Equality Act 2010 [15]. Historically, violation of similar acts could result in hefty fines. The National Federation of the Blind sued Target Corporation, as the retailer’s website was not accessible enough for legally blind people. The case was settled in 2008, with Target paying out $6 million in class damages and $3 million in plaintiff legal fees, along with an undisclosed amount of defence legal fees. [16]

2.4.2 Accessibility Considerations

According to W3C, factors such as impaired vision, hearing, and physical ability tend to affect how older people use the web. As such, guidelines as set by W3C on visual and physical accessibility will be more heavily followed. The usage of sounds will not be necessary for this project. Some of the more relevant guidelines can be found in the list below: [17] [18]

- Provide sufficient contrast between foreground and background
- Include alternative text for images
- Use mark-up to convey meaning and structure
- Use headings and spacing to group related content

- Include image and media alternatives in your design
- Help users avoid and correct mistakes
- Reflect the reading order in the code order
- Write code that adapts to the user's technology back to section 5.2

It is important to note that the WCAG is meant to be a guideline for websites [19]. However, most of the principles can still be applied to mobile app development.

2.5 Microcomputer and Microprocessors Literature

2.5.1 Overview

This section will take a look at some microcomputer literature that uses an IoT approach to control and read sensors. The purpose of looking through pre-existing literature in this section is to gather information on the choice of hardware, architecture of their systems, and how information is transmitted within the system. This section will additionally look at the differences between microcomputers and microprocessors as they are both used in the reviewed literature, and because there are important distinctions between them.

2.5.2 Microcomputers

Microcomputers are essentially miniaturised PCs that are built on a single board. They use a single microprocessor as their central processing unit (CPU), and can have random access memory (RAM), input/output (I/O) ports, and a form of memory storage for their operating system and some software. An example of a microcomputer would be the Raspberry Pi 3B [20].

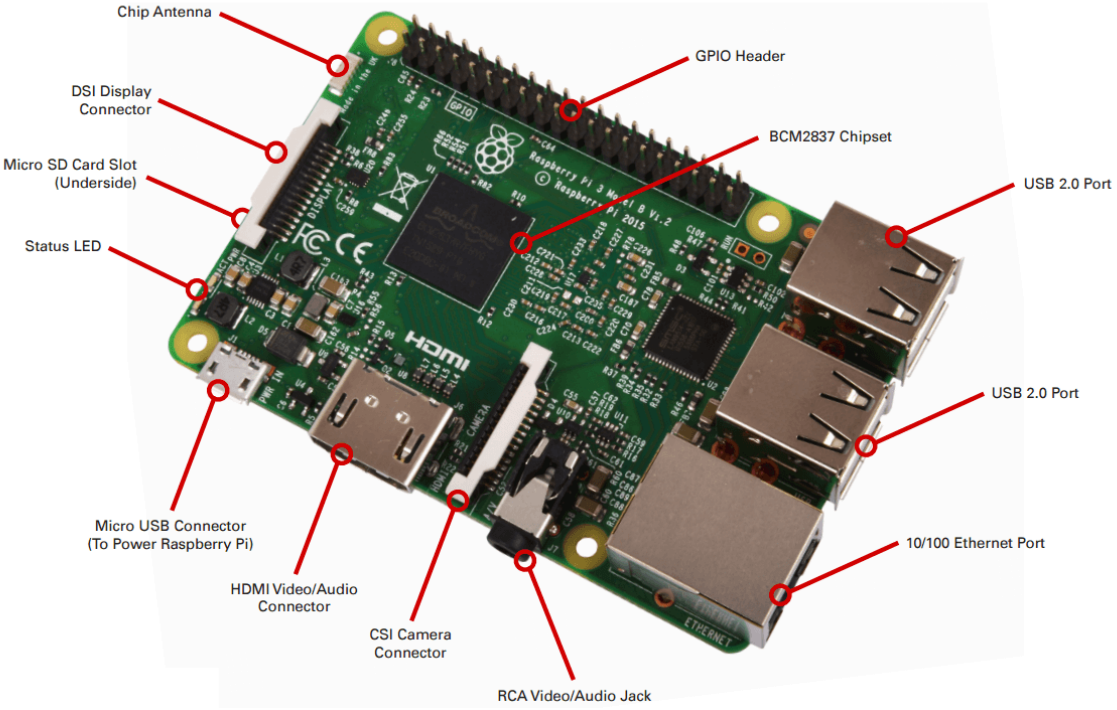


Figure 2.1 - A diagram of a Raspberry Pi 3B with its components [21]

2.5.2 Microcontrollers

Microcontrollers are also single board devices, and they typically have a processor, RAM, read only memory, and I/O ports. Unlike microcomputers, which are PCs, a microcontroller does not have an operating system. As such, it is usually used in embedded systems to govern specific operations. Because they are simpler, they also consume less power relative to microcomputers, but they are also restricted to doing simpler tasks [22].

Despite their simplicity, they are more flexible than microcomputers in some areas. Microcomputers, such as the ESP-8266, can handle boolean functions, are generally faster, and their bit handling functions are easier to understand [23].

2.5.3 Self-adaptive control system for Hydroponics

This system by Kelkar, Pange, and Sawlani [24] integrates a hydroponics system with the Internet of Things in order to operate pumps, valves, and to collect readings relevant to the system.

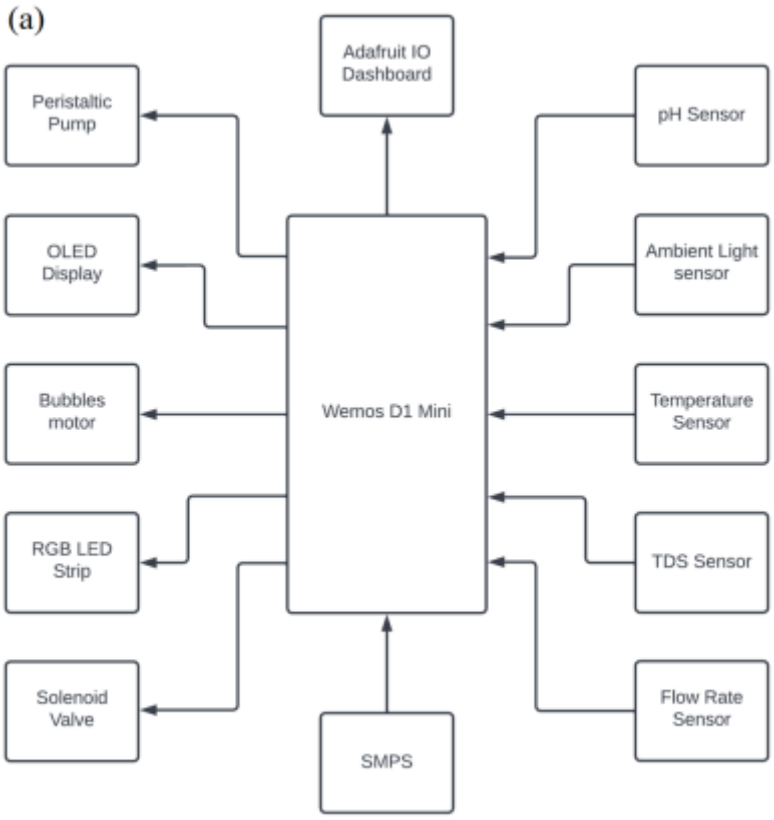


Figure 2.2 - The controller's block diagram

An ESP-8266 (Wernos D1 Mini) microcontroller is used as a control unit of the system. The ESP-8266 is connected to a solenoid valve, an OLED display, a peristaltic motor to pump fluids, and 4 different sensors. The ESP-8266 is written to be able to function autonomously. One of its autonomous capabilities was the ability to regulate pH levels of the water by pouring alkaline/acidic solutions into the water when it detects an abnormal pH value.

The ESP-8266 is also able to publish the readings from its sensors onto the Adafruit IO platform using MQTT over wifi. This allows a user to access the readings from any device as long as they have an internet connection.

2.5.4 Low-cost bus seating information technology system

This system by Murdan, Bucktowar, Oree, and Enoch [25] uses an IoT approach in order to capture operational data such as the number of passengers travelling on the bus, and the comings and goings of said passengers.

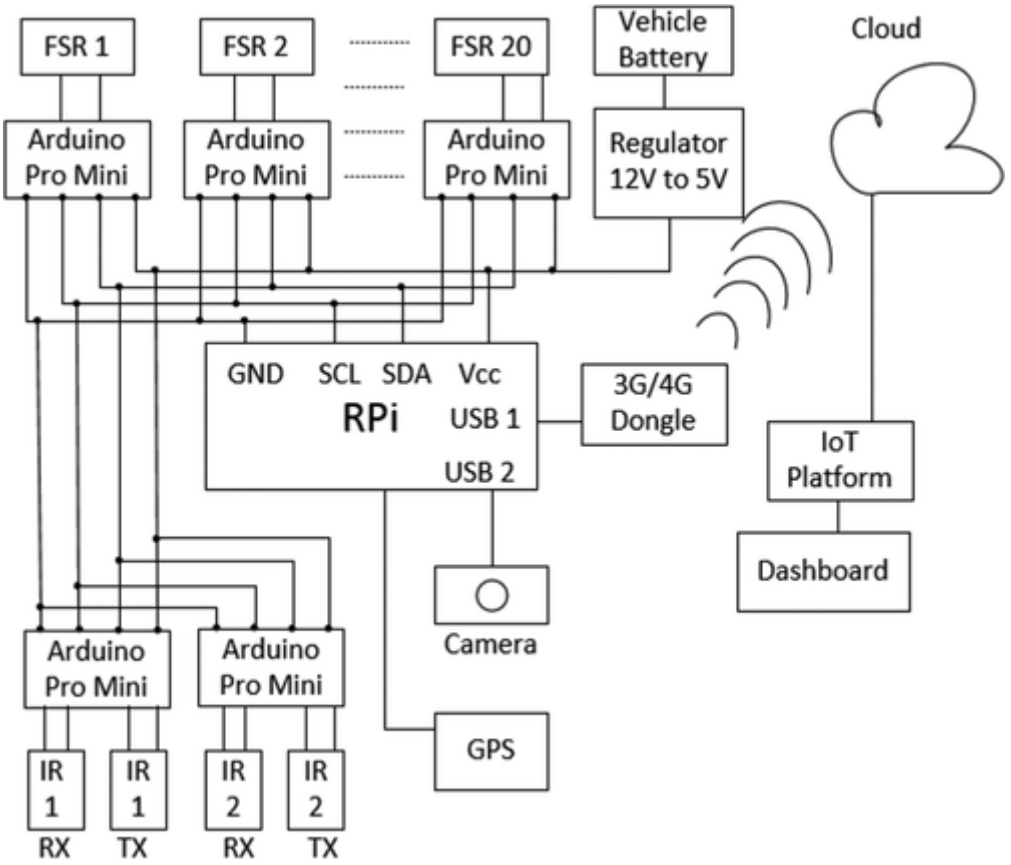


Figure 2.3 - Architecture of the bus seating system

The system uses a Raspberry Pi 3B connected to a GPS tracker, a camera, a 4G dongle, and Arduino Pro Minis that interfaces with IR sensors and weighted sensors (FSRs). The 4G dongle allowed the Raspberry Pi to receive internet through the usage of a SIM card while the bus was on the move.

Similarly to the hydroponics system, the microcomputer uploads its readings onto an IoT platform, thinger.io. An Android app developed by Murdan et al. in turn pulls the information from thinger.io, which then allows a user to access the readings in a more intuitive way. The system is also able to track live GPS output, which is then visualised on a map in thinger.io.

2.5.3 IoT based system for smart indoor hydroponic vertical farming system

This system by Yusuf, Sahrani, Sarker, et. al [26] is similar to the previous hydroponics project in 2.5.1, in the sense that this is also a hydroponic system which uses a microcontroller as a controller for actuators and sensors, and they also upload their readings onto Adafruit IO.

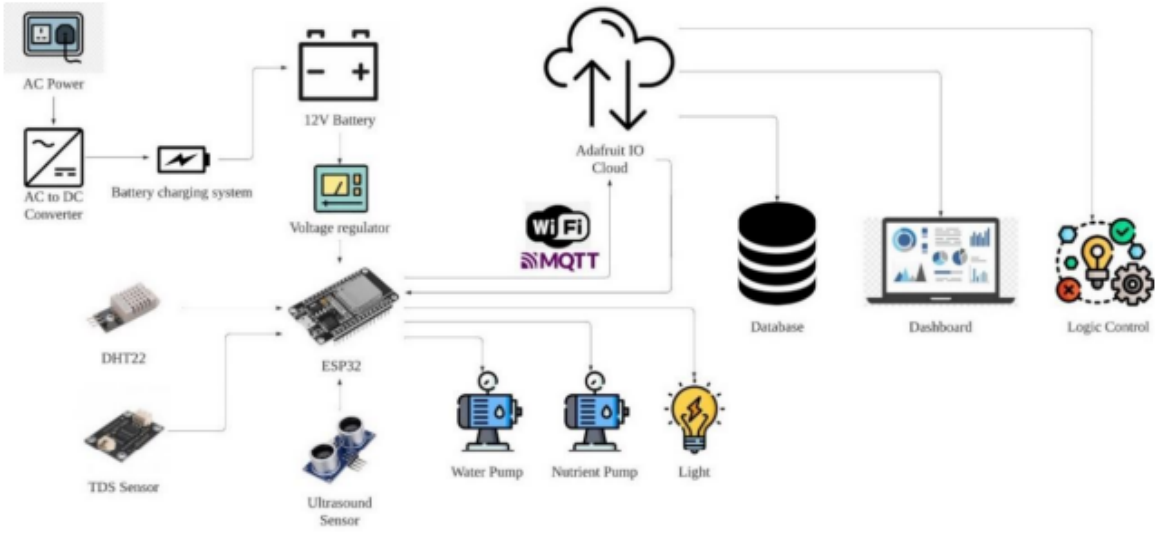


Figure 2.4 - Architecture of the hydroponic farming system

This system relies on a microcontroller, the ESP32, to control the system. The ESP32 is connected to four different sensors: the DHT22, which reads temperature levels, a TDS (Total Dissolved Solids) sensor, which measures the nutrient content in the system, and two ultrasonic sensors to measure water levels. The ESP32 also controls three actuator systems: a lighting system, a water pump, and a nutrient pump. Much like the previous hydroponics farming system, the ESP32 functions autonomously. The lights are scheduled to operate in certain time intervals, and both the water and the nutrient pumps will turn on automatically once the sensors read a certain value.

The ESP32 publishes all sensor data to Adafruit IO, which is their IoT platform of choice. This is done through MQTT with a pre-configured wifi point. However, instead of pre-programming their microcontroller to operate the actuator systems by monitoring sensor data locally, they have opted to use Adafruit IO to drive the actuator systems instead.

2.5.4 Summary

To summarise, the architectures of all reviewed literature were very similar. The choice of microcontroller/microcomputers found here were the ESP32, ESP8266, and the Raspberry Pi 3B, all of which were able to interface with sensors and upload readings onto an IoT platform, which were either Adafruit IO or thinger.io. These choices will be discussed more at length in the technology section.

Logical operations can seemingly be either carried out on the microcontroller/microcomputer itself, or on the IoT platform of choice. This can be useful for sending out notifications upon detecting a dangerous CO/temperature reading.

The Raspberry Pi in the bus seat system was also able to get internet connection with a 4G dongle, and the app that was built for the system could pull data from thinger.io through a HTTP API, although MQTT should also work with apps.

2.6 Ethical and Legal Issues

2.6.1 Data Protection

When processing data, one must be able to do so according to the clauses in Article 5.1-2 in the GDPR [27]:

1. Lawfulness, fairness, and transparency — Processing must be lawful, fair, and transparent to the data subject.
2. Purpose limitation — You must process data for the legitimate purposes specified explicitly to the data subject when you collected it.
3. Data minimization — You should collect and process only as much data as absolutely necessary for the purposes specified.
4. Accuracy — You must keep personal data accurate and up to date.
5. Storage limitation — You may only store personally identifying data for as long as necessary for the specified purpose.
6. Integrity and confidentiality — Processing must be done in such a way as to ensure appropriate security, integrity, and confidentiality (e.g. by using encryption).
7. Accountability — The data controller is responsible for being able to demonstrate GDPR compliance with all of these principles.

All information collected by the sensors will be used in a sensible manner within the project only. The only personal data collected in this project is the GPS data. This information will also be kept in Adafruit IO's cloud and not locally in any devices, so policies 5, 6, and 7 will only apply to Adafruit IO.

As such, it is necessary to check Adafruit IO's privacy policy for their GDPR compliance. Adafruit adheres to policy 6 by adopting "appropriate practices and security measures designed to protect against unauthorised access, alteration, disclosure, or destruction of personal information." [28] It is also possible to contact Adafruit IO to erase and/or rectify personal data, so they are in compliance with policies 4 and 5.

In order to evaluate this app with users and to collect data from them, it is also necessary to apply for ethics approval through the University's Ethics Application System.

2.6.2 Medical Device Classification

As this system is devised to help users avoid injuries and/or deaths from heat strokes and CO poisoning, there is ambiguity as to whether it is a medical device. Software being classified as a medical device [29] will be subject to the Medical Device Regulations 2002, which will impose more restrictions on the system.

To determine a software's classification, the MHRA (Medicines and Healthcare Regulatory Agency) has released a flowchart to aid in classification.

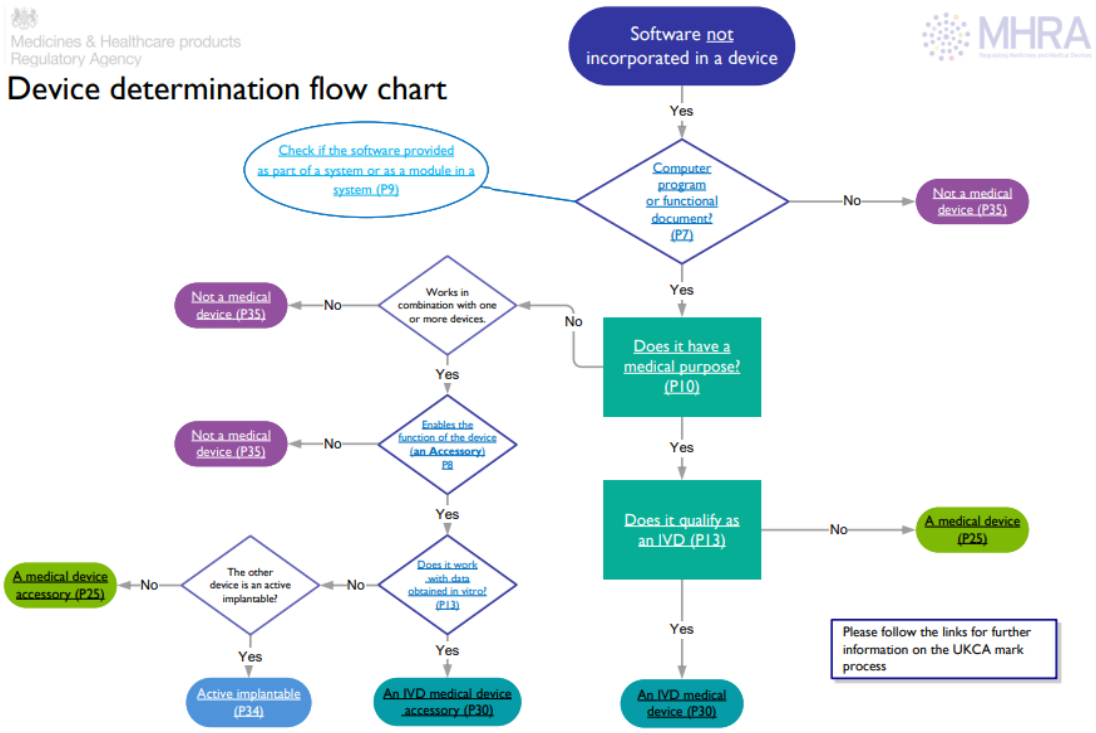


Figure 2.5 - Flowchart for determining the classification of software

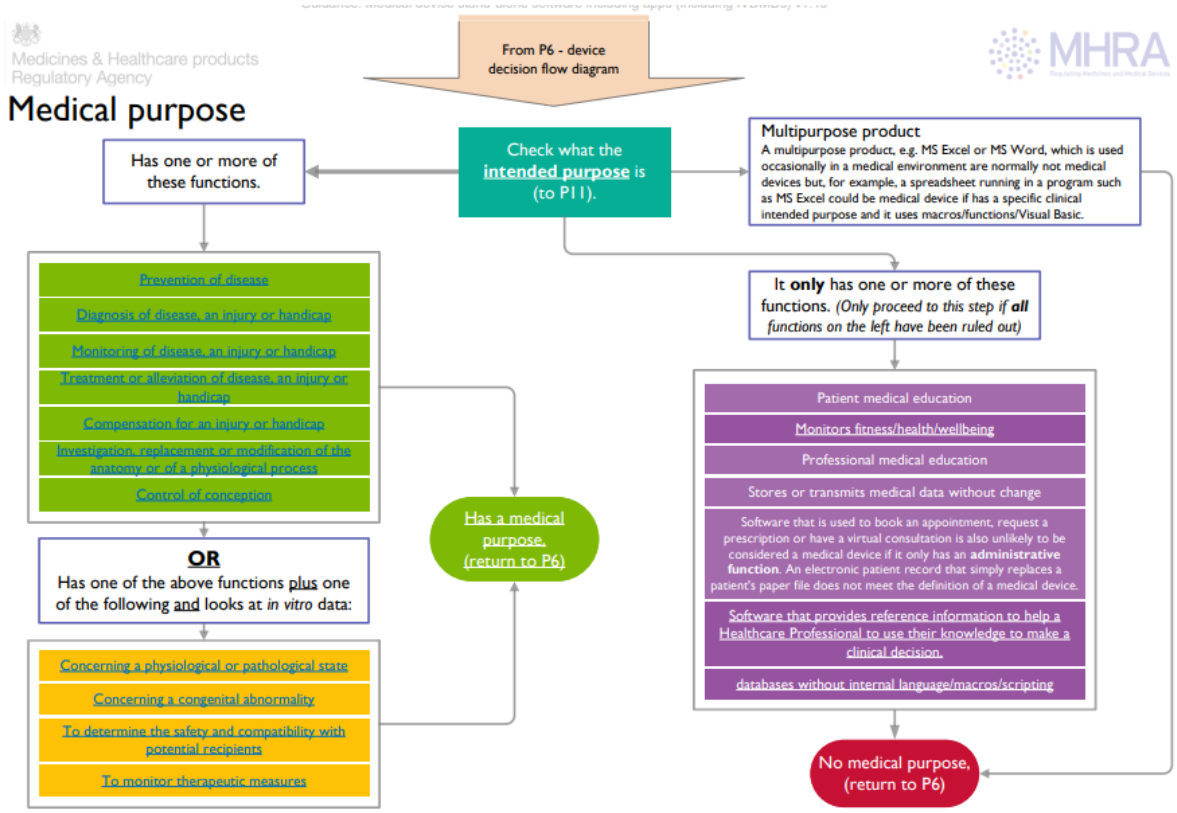


Figure 2.6 - Flowchart for determining medical purpose of software

Based on Fig. 2.3, it goes from P9, P7, and to P10. To determine if a software has a medical purpose, another flowchart will have to be followed, as seen in Fig 2.4. The dissertation's software will not have any of the functions mentioned in the green box, but it does loosely "monitor fitness/ health/

wellbeing". As such, it has no medical purpose. While it works in combination with one or more devices, it does not enable the function of a device that is a medical device (the Raspberry 3B+ being a device that only reads and sends temperature and CO levels), therefore the software would not be classified as a medical device, and so the system in its entirety is legally not a medical device.

3. Technology Review

3.1 Overview

This chapter will consolidate the knowledge gleaned from the literature review by comparing relevant design choices from literature against each other to evaluate their suitability for the project.

As such, this chapter will cover hardware (computers, circuitry, and sensors), methods to power the computer, as well as the computer's operating systems, development frameworks, and methods of facilitating communication between the computer and the app. The conclusion of this chapter will then summarise the chosen design choices and give details on the specifications of the system.

3.2 Hardware

3.2.1 Overview

Having sensible hardware is paramount to the success of this project. Usage of the wrong kinds of hardware can impede or downright halt progress on achieving the project goals. This section will discuss the consideration of different hardware, ranging from computers to low level circuitry components. Computers are considered in this section first as different computers will interact with the same sensors in their own ways.

3.2.2 Computer Choice

In order for this project to go smoothly, a computer must be carefully chosen. As such, the computers used in this project must have these characteristics:

- Decent processing power
- Able to support sensors needed for this project
- Enough input/output pins for said sensors
- Able to connect to the Internet while on the move
- (Optional) Good power efficiency
- (Optional) Easy to use

Two computers have been found to be the most suitable for this project, that being the ESP32 and the Raspberry Pi 3B. The following table of comparisons will evaluate their suitability:

Characteristic	**ESP32**	**Raspberry Pi 3B**
Input/output pins	34 [30]	26 [31]
Internet on the move	Yes, with external modules	Yes, with external modules
CPU speed	240 MHz [32]	1.2 GHz [33]
Power consumption	68 mA [34]	260 mA
Supports required sensors	Yes	Yes

Table 3.1 - A table of comparison between the ESP32 and the Raspberry Pi 3B

The number of input/output pins is fairly important as it dictates the amount of sensors and other components that can be used in the circuit. However, there will be a point of diminishing returns as the circuitry associated with this project is not meant to be complex. The Raspberry Pi 3B has an adequate amount of GPIOs, though the ESP32's 34 GPIOs allow for greater flexibility if there is a need to expand the project scope.

As the device is meant to be placed within a car, the computer must also be able to use the Internet while being driven around, so that it can communicate with the mobile app reliably. Both computers are able to do so using external modules. The ESP32 can use the SIM800L, while the Raspberry Pi 3B can use the SIM7600X 4G HAT. Both modules use a slotted sim card to provide internet connectivity on the go.

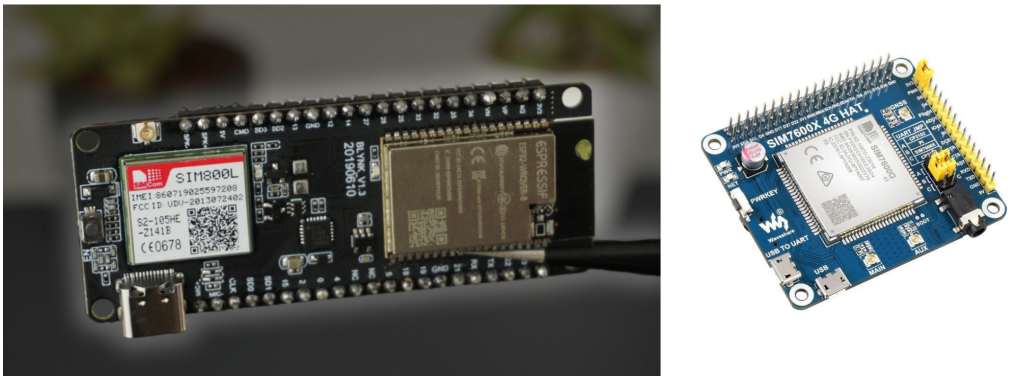


Figure 3.1 - The SIM800L (left) and the SIM7600X 4G Hat (right)

The CPU speed dictates the speed and the number of instructions a computer can process. The current project specifications will not be too taxing on the computer; it only requires the computer to send out sensor readings. However, adding on more functionalities (such as live GPS tracking) might be too taxing on the ESP32, which makes the Raspberry Pi 3B more flexible in this area.

The Raspberry Pi community is more mature than that of the ESP32's. This means that documentation for Raspberry Pis tends to be more up to date. They also benefit from a lower barrier of entry due to being more popular, which means more tutorials are available. Driver support and third party addons are more widely available on the Raspberry Pis as well, which makes them more flexible as a whole [34].

As such, this project will be using the Raspberry Pi 3B as the computer. While having better energy efficiency is a good attribute, it is ultimately not a highly important factor. These computers are meant to be plugged into the car's accessory port. A high capacity power bank can mitigate the Raspberry Pi 3B's increased power consumption when the car is inactive.

3.3 Circuitry and Sensors

3.3.1 Analog to Digital Converters

The Raspberry Pi's GPIOs are only able to control and read digital input and outputs. However, sensors typically output their readings in analogue, which prevents the Pi from reading them. To rectify this, an analog to digital converter (which does what it says on the tin) will be used in the circuit [36]. The MCP3008 has a 10 bit resolution and it has 8 input channels, which will be sufficient for the sensors used in this project, as they are relatively simple sensors.

3.3.2 Breadboards

Breadboards are commonly used to test out circuits and devices before they are soldered onto custom printed circuit boards. A breadboard contains several exposed holes that house horizontal terminal strips, which are conductive metal rails that allow current to flow through. A breadboard also has vertical power rails with positive and negative sides, and these rails provide power to the terminal strips. Breadboards also often have a "ravine" that separates the two sides of a breadboard. Specialised chips, such as the MCP3008, can be slotted into the ravine, thus allowing components from each side of the ravine to connect to the chip [36].

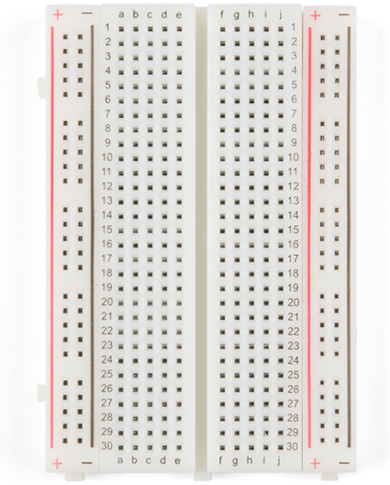


Figure 3.2 - A typical breadboard [37]

3.3.3 MQ-7 Sensor

The MQ-7 sensor is designed to detect and give a reading of carbon monoxide concentration levels. The sensor uses tin dioxide as a sensing element, which has many free electrons. When in the presence of clean air, the free electrons will bond with oxygen. As carbon monoxide reacts with oxygen, the bonds break, thus returning the electrons to their original positions and allowing them to conduct current. The more carbon monoxide there is, the more free electrons will be available, thus conducting more current. This signal can be then read by the analog to digital converter and passed onto the Pi, as the analog pin outputs a more precise reading compared to the digital pin [38].

Figure 3.3 - A MQ-7 Gas Sensor [39]

3.3.4 Logic Level Converter

Because the Pi's GPIO pins only support up to 3.3V signals, and the analog pin outputs a 5V signal, connecting the analog pin to the GPIO might damage the Pi. To remedy that, a logic level converter can be used to step down the 5V signal into a 3.3V signal. This 3.3V signal can then be passed onto the analog to digital converter. This component necessitates soldering as the pins usually come packaged separately.

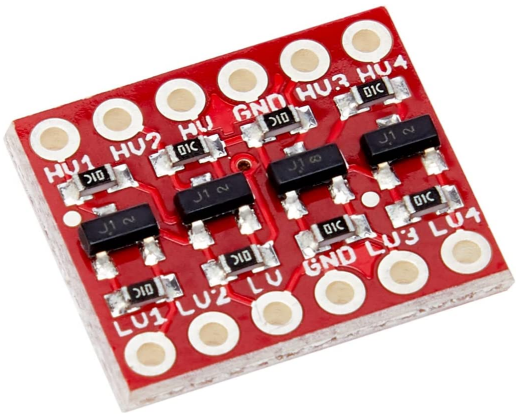


Figure 3.4 - A logic level converter without its pins [40]

3.3.5 AM2320 Temperature and Humidity Sensor

The AM2320 sensor can read humidity levels with an accuracy level of +-3%, with a resolution of 0.024%. It can also read temperature levels between -40 to 80°C, with an accuracy level of +-0.5°C, and a resolution of 0.01°C [41]. It takes a reading every 2 seconds. As it is a digital sensor, it can be directly connected to the Pi.

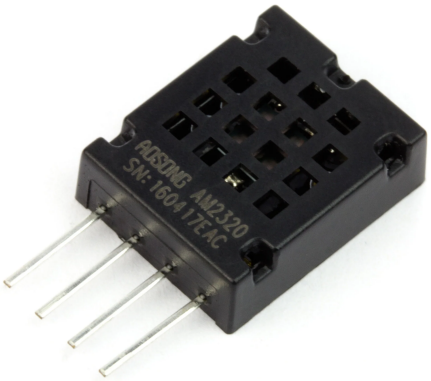


Figure 3.5 - An AM2320 sensor [42]

3.4 Power Sources

3.4.1 Overview

An external source of power is needed in order for the Pi to work when the car's ignition is switched off, as the accessory ports for most cars do not work in that state, and so the Pi cannot solely rely on a car to provide it with power.

The Pi can be powered through its microUSB port or through its 5 volt pin. The computer needs to be supplied with a voltage of 4.75 to 5.25 volts to work. If the computer is supplied with less voltage than required, then it will not power on. If the computer receives more than 5.25 volts through the microUSB port, a polyfuse will trip to protect it from drawing too much power [43].

It is important to note that the 5 volt pin does not come with some elements of protection that the microUSB port offers. The 5 volt pin does not come with the polyfuse, which means that using a >5.25 volt power source will brick the Pi. The microUSB also has a transient voltage suppression diode. This diode protects the sensitive circuits of the computer from power spikes, which happens a lot in electronics [44].

3.4.2 Batteries

The Pi can be powered by batteries through its 5 volt pin. The characteristics of a battery are affected by its chemical makeup; an alkaline battery of the shape AA will have a voltage of 1.5 volts, and will not be rechargeable. On the other hand, a nickel metal hydride (NiMH) battery or a nickel-cadmium (NiCd) battery will have a voltage of 1.2 volts, and are typically rechargeable. The capacity of a battery can also vary between the same types of batteries.

The voltage of batteries will also decrease as they are discharged. A fully charged lithium polymer battery will output ~4.23 volts, but can go down to ~2.7 volts with continued use [45].

In order for the Pi to work on batteries sustainably, processing power must be throttled, which restricts the scope of the project. There are also too many variables to consider in batteries, and a suitable battery may take quite a while to find. Furthermore, there are no single batteries that output 4.75-5.25 volts. This means that batteries have to be used in parallel to reach the computer's voltage range alongside voltage regulators so that they do not burn out the computer and/or the circuit. Even then, the batteries might go below 4.75 volts as they discharge, thus powering off the computer.

As such, batteries are not a good power source for the Pi, but they might be useful in prototyping due to their low cost and availability.

3.4.3 Power Banks

Power banks are compatible with the Pi as long as they have a microUSB cable. A typical power bank designed for Raspberry Pis outputs power at around 5 volts. Power banks have far greater capacity than batteries, and they can go up to 50000mAh, which can power a Pi for up to 30 hours before it completely drains [46].

Power banks will be chosen to power the Pi in this project as they charge the computer in a safer way, and they are able to power the computer for a longer period of time. They can also power the computer while being charged concurrently, which makes it convenient for the user, and greatly mitigates the risk of the device not operating without the user knowing.

3.5 Software

3.5.1 Operating Systems

Raspberry Pi 3Bs use a microSD card as storage. This card must be first flashed with an operating system (OS), which can be anything from the first party Raspberry Pi OS to the myriad of Linux distributions (Gentoo, Ubuntu, Debian, etc.). Both Raspberry Pi OS and Ubuntu come highly recommended to beginners to the Raspberry Pi ecosystem. The following table will lay out the core advantages and disadvantages of both operating systems:

Raspberry Pi OS	**Ubuntu**
+ First-party OS, so compatible with everything in Raspberry Pis + Fast performance + Stability due to slower updates + More community resources - User experience is lacking due to outdated UI - OS updates once every two years, which means software lags behind too	+ Modern, intuitive UI + OS updates twice a year + Best driver support for Raspberry Pis among other third-party OS - Performance is somewhat slower than the Raspberry Pi OS

Table 3.2 - A table of comparisons between Raspberry Pi OS and Ubuntu [47]

As the Pi is only used as a “headless computer”, the UI only plays a small role in the OS consideration. It is more important to have better performance and compatibility with drivers, especially when this project uses a lot of external modules for the Raspberry Pi 3B. The only significant advantage that Ubuntu has is that it has more frequent updates, so Ubuntu has Python 3.9 while Raspberry Pi OS has Python 3.7. With these considerations, the project is going to use Raspberry Pi OS.

3.6 Mobile Development

3.6.1 Overview

This section will explore the options available for enabling an app on a mobile phone to receive warning notifications for dangerous CO/temperature levels, to display current sensor readings on the Pi, and to visualise historical readings.

Two possible paradigms for apps were identified to be suitable for this project – that being native apps and progressive web apps.

3.6.2 Swift

Swift is an open source language used in iOS native app development. It was developed with the following quote as a core tenet: “make programming simple things easy, and difficult things possible.” [48] Apple Inc. has also built Swift with the community in mind, as community members can submit changes to be reviewed. An active and valuable member of the community can also be promoted to code owners – people who review contributions and approve them with the community’s feedback in mind. This aspect of Swift makes it a fast-evolving language that integrates sensible modern ideas continuously, thus making the language more versatile and accessible to newer developers.

3.6.3 Java and Kotlin

Both Java and Kotlin are heavily used in Android native app development. As Java has been in the market for 24 years, Java naturally boasts a higher usage rate compared to Kotlin.

Java is a popular language in Android as native Android tools and libraries already exist for Java. Java also has a large number of open source libraries, which makes development easier. However, there have been some problems with Java that prompted the creation of Kotlin [49].

Kotlin is a language developed by JetBrains that got its first stable release in 2016. Kotlin was created in order to save development time and enforce program safety. Its code is more concise than that of Java’s, and it can catch null pointer exceptions at compile time, which frees up debugging time. Kotlin is also fully compatible with Java, which means that Java code will work on Kotlin, so one can use the wealth of Android libraries developed for Java to expedite development [50].

3.6.4 Progressive Web App (PWA)

Progressive web apps are platform-agnostic applications built on the technologies that power websites [51]. They are therefore only restricted by the capabilities of individual browsers instead of the operating system of a device.

PWAs are able to closely emulate native apps by allowing users to download and/or save them as shortcuts to their devices. They utilise service workers to serve offline experiences – they are capable of caching resources for offline usage, and they allow the PWA to send push notifications [52].

PWAs need to adhere to certain criterias in order for browsers to deem them as a PWA (and thus becoming installable). They have to include a web app manifest, which allows them to launch in full screen and to set an icon for the start-up splash screen. They also have to include a valid service worker, be visited by a user twice, and be served over HTTPS [53].

As mentioned before, PWAs are built on web technologies. This makes them very flexible in the sense that they can be built from any languages used in web development. They are commonly developed with the myriad Javascript frameworks, such as Polymer, React, Virtual-DOM, and Angular.js.

3.6.5 Comparisons

Native apps and PWAs are both very suitable paradigms to use for the mobile app that needs to be developed as part of the system. The following table will summarise the pros and cons of both types of software:

Native Apps	**PWAs**
+ More powerful than PWAs, able to utilise more native functions that are inaccessible to PWAs + Can be optimised better - Facilitating usage on both iOS and Android means having different codebases	+ Accessible to any devices as long as their browsers support PWAs + More accessible, support for PWAs is based on browsers + Easier to distribute, no need to go through app stores

Table 3.3 - A table of comparisons between native apps and PWAs [54]

While native apps sound better on paper, the advantages they offer over PWAs are not too impactful compared to their trade-offs for the app that is going to be developed in this dissertation.

The app will be fairly lightweight in theory, which means that it is not a necessity to fully optimise the app. Native apps also offer better integration with their respective phones, allowing them to interact with other apps, access the phone's cameras and sensors, pay with NFC chips, and more. However, the planned app only requires push notifications (to warn the user about dangerous readings) and an internet connection (to read data from the Pi's sensors), which a PWA can accommodate.

On the other hand, PWAs enable the app to be used on any platforms that can run PWA-supporting browsers, which are used by 88% of web users [55]. In comparison, making a native app for both mobile platforms would dramatically increase development time, which will push a lot of

functionalities out of scope. PWAs are also easier to evaluate on an accessibility basis with automated tools such as Google Lighthouse, which checks websites for their degree of compliance with WCAG 2.1 accessibility guidelines. For these reasons, the app for the system will be developed as a PWA.

3.7 App to Raspberry Pi Communication

The data collected from the Pi's sensors will need to be accessible to the user from the app. To do this, the project will have to use an IoT cloud platform in order to read the data from the Pi and communicate the readings to the app. Based on the reviewed literature, there are two options for an IoT cloud platform: Adafruit IO and thinger.io.

Both platforms are cloud services that primarily store and retrieve data of IoT devices in real time. They allow users to view the data feed from their dashboard(s), which then can be represented in graphs and gauges. Both platforms also offer a HTTP API that will allow the app to gain access to the data feed.

The main difference between both platforms is the rate of writing data to the cloud for free accounts. Adafruit IO offers 30 data points (create, read, update, delete actions) per minute [55], while thinger.io limits its write rate to 1 minute and it has "limited data points" [56]. Because this system is driven by the output of the sensors on the Pi, having more data points can only be a good thing. In addition to that, having a better write rate allows for the app to notify the user earlier if excessive CO/temperature levels were detected, thus maximising the time window that the user can act on the emergency. Therefore, this system will be using Adafruit IO as its IoT cloud platform.

Figure 3.6 - An example of Adafruit IO's feed

3.8 Conclusion

Based on the research gathered from both the Literature Review and the Technology Review sections, a Raspberry Pi will act as the controller in the system. The AM2320 and MQ-7 sensors will be connected to the Pi, which is then powered by a power bank that can supply 5V to the Pi. The power bank will also be constantly connected to the car's accessory port to ensure that it will be fully charged, in case that the car housing the Pi is off for an extended period of time. A SIM7600X HAT (with a valid SIM card) will also be connected to the Pi to give it internet connection on the move so that it can upload its readings onto Adafruit IO.

The app will also be developed as a PWA in HTML and vanilla Javascript. This app will be able to display the Pi's sensor readings in a digestible manner, and would contain some form of data visualisation. These readings will be obtained from Adafruit IO through its HTTP API. The app will also be designed with accessibility issues in mind to accommodate older users.

4. Requirements and Analysis

4.1 Overview

The completeness of the project will be evaluated against a list of functional requirements that divides the project's goals into chunks, which is then prioritised with the MoSCoW system. The project can be considered as complete if all of the requirements with the "must have" priority are met, although the other requirements will also add value to the project and should be treated as a secondary objective.

The non-functional requirements are just as important to adhere to, as they will ensure that users will have a smooth experience and that the system will be robust. A generalised testing plan will also be detailed in section 4.3 to test both functional and non-functional requirements.

4.2 Functional Requirements

The MoSCoW method involves using 4 levels of prioritisation. These levels, in ascending order of importance, are as follows: "will not have", "could have", "should have", and "must have".

Requirements	Acceptance Criteria	MoSCoW
As a user, I should be able to connect the Raspberry Pi to my mobile phone	The user should be able to enter an Adafruit IO API key in order to link the Raspberry Pi and the phone	Must have
As a user, I should be able to see temperature and carbon monoxide readings from the app	The user should be able to see the current temperature and carbon monoxide values that the Raspberry Pi collects	Must have
As a user, I should be able to set threshold values for carbon monoxide and temperature levels	The user should be able to set a threshold value for both temperature and carbon monoxide levels through entering a numerical value. This will need input validation to check for sensible values.	Must have
As a user, I should get a notification warning me about temperature and/or carbon monoxide levels exceeding a threshold value	When the Raspberry Pi reads temperature and/or carbon monoxide readings that exceed the threshold value that the user set, a notification should be sent to their phone. The notification should say which reading exceeded the threshold value and display the current reading.	Must have
As a user, I should be able to download the app onto my phone	The app should follow the criterias set out in section 3.6.4 in order to be labelled as a PWA by browsers, which will then make it installable.	Must have
As a user, my Raspberry Pi 3B should send sensor data onto the cloud upon powering up without my intervention	When the Raspberry Pi boots up, the scripts that reads and sends sensor data to Adafruit IO should automatically boot up.	Must have

As a user, I should be able to see a history of temperature and carbon monoxide readings	The user should be able to see a line chart detailing readings (CO/temperature) that are averaged per day. The user should also be able to select a certain day to display on the chart.	Should have
As a user, I should be able to locate where the device is	The user should be able to see a menu that shows the last known location of the device through the 4G HAT's GPS function	Could have
As a passenger in the user's car, I should be able to hit a button on the Raspberry Pi to send a notification to the user's phone	The user should receive a notification stating that the emergency button on the Raspberry Pi has been triggered	Could have
As a user, I should be able to see a fact sheet on heat stroke symptoms	The user should be able to open a menu that contains information on symptoms related to the level of severity of heat strokes and advice to alleviate heat strokes	Could have
As a user, I should be able to see a daily breakdown of where the device was and the average and highest temperatures of those places	The user should be able to open a menu and see a list of locations where the Raspberry Pi was "stationary" at. This will only show locations where the Raspberry Pi has idled at for longer than 30 minutes. This positional data will be correlated with the average and highest temperatures that the Raspberry Pi collected when it was idling there.	Could have

4.3 Non-functional Requirements

Requirements	Justification
The Raspberry Pi should be able to have internet connection reliably	As the Raspberry Pi collects the temperature and carbon monoxide data, losing internet connection for an extended period of time can be dangerous to the health of a vulnerable passenger waiting in the car.
The Raspberry Pi should be robust in the face of intermittent internet connectivity	In some places where Internet infrastructure is less reliable, it might not be possible to have a persistent internet connection. The Raspberry Pi should still be able to upload its readings onto the cloud in the periods where it has internet connection.
The app should be accessible and easy to use	As stated in the above sections, the users' age group will tend towards the middle to old ages. As such, they tend to have less developed technical abilities, motor skills, and senses.
The Raspberry Pi should be able to withstand	The interior of cars can get very hot in summer

higher temperatures	time. The Raspberry Pi should continue to function normally in those conditions in order to be useful.
The Raspberry Pi should be able to operate for at least 12 hours when the car is switched off	Some people turn off their car's engine and roll down the window slightly when there's a passenger in the car. To operate normally, the Raspberry Pi will have to be powered through a power bank. Letting the Raspberry Pi operate for 12 hours will ensure that the Pi can still monitor readings overnight.
The sensors on the Raspberry Pi should be properly calibrated	As these sensors are cheap and mass produced, they are not calibrated well. The AM2320 does not need calibration, but the MQ-7 does, and it requires 48 hours of continuous operation. [57]
There should be an offline splash page	While this app is not meant to be used offline as the sensor data is pulled from the cloud, and none of the data is stored locally, the PWA checklist [58] recommends that there should be a custom offline page to make the PWA seem more like an offline page.

4.4 Testing Plan

This project will use a combination of white box testing, functional testing, and automated testing. As there are quite a lot of variables to account for in the system (especially with regards to the Pi), white box testing will be done in order to check how reliable the Pi is (how well does it hold up being active over a long period of time, how well can the Pi transmit the sensor's readings, etc.).

Functional testing will be used to check if the end user will be able to interact with the app in a bug-free way, and to test how well the app holds up to unintended interactions. Automated testing will be done with tools such as Google Lighthouse in order to evaluate how accessible and how good of a PWA the app is. This will be expanded upon in chapter 7.

5. Design

5.1 Schematic Design

The Pi schematic has gone through two iterations. The first iteration of the schematic was created with an automatic drag-and-drop circuit creator, circuito.io. The first schematic was done this way due to not having practical experience with circuits. This schematic had many problems – the MQ-7 sensor was not being read as no wires were connected to the AO (analog output) pin. It was also lacking a logic level converter to convert the MQ-7's 5V signal to 3.3V, which will damage the Pi if they are connected to each other.

The new schematic has been dramatically simplified. This schematic does not use resistors as all pins are either 3.3V or 5V, which means that there was no need to divide voltage (using the logic level converter was enough), and the Pi can handle the current fine, thus rendering resistors redundant.

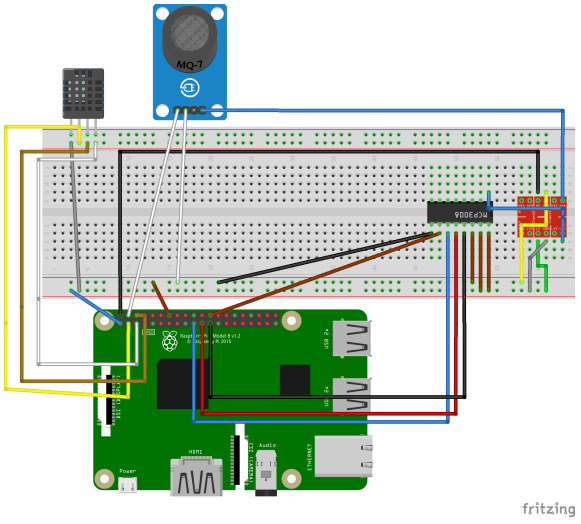


Fig 5.1 - The new system schematic

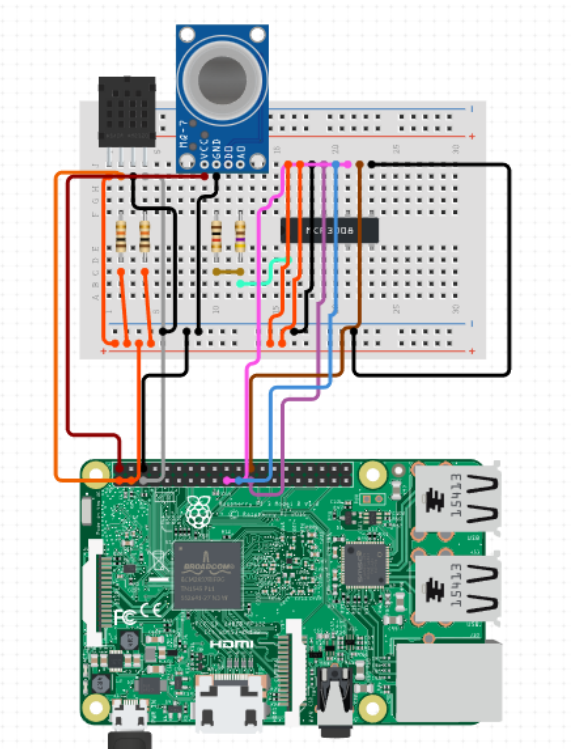


Fig 5.2 - The old system schematic

5.2 Accessible User Interface and HTML Design

Both UI and HTML designs will follow relevant WCAG 2.1 guidelines to ensure an accessible user experience in the app. The following UI mock-up was designed with the considerations highlighted in section 2.4.2.

Fig 5.3 - The dashboard

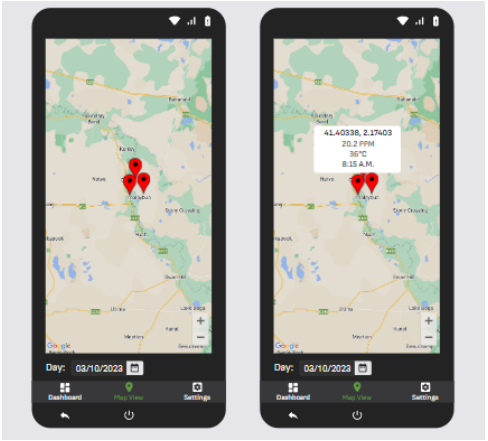


Fig 5.4 - The locational history page

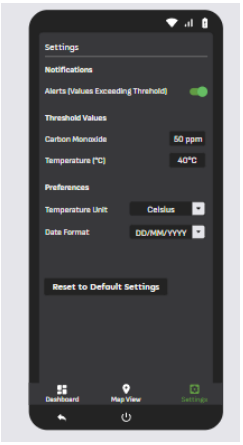


Fig 5.5 - The settings page

The black and green colour scheme is meant to make the app look modern while being an accessible theme with sharp colour contrasts so that visually impaired people can use the app better. Related elements are grouped together, and functionalities are separated by putting them in different menus. There are some notable challenges to this however – graphs are primarily a visual way to convey information, which is of course antithetical to accessibility. Some inputs (such as incorrect API keys) are not also able to be verified directly, so it is not possible to help users avoid and correct mistakes for certain inputs.

To make the HTML accessible, alternate text would be used in relevant areas, such as for links (both on the navigation bar and for external links) and images (if needed). HTML semantics and code order will also be considered when developing the app. More details about how the app's HTML is written to be accessible will be contained in the Implementation and Testing section.

5.3 System Architecture

A high level view of the system's architecture has been created to aid understanding of how the different components communicate with each other in the system.

Both of the sensors (MQ-7 and AM2320) are read continuously by the Raspberry Pi, and the readings with positioning data (GNSS) are then sent to Adafruit IO through the 4G connection provided by the SIM7600X.

The app will then pull data from Adafruit IO through its HTTP API and process the sensor data within the app itself.

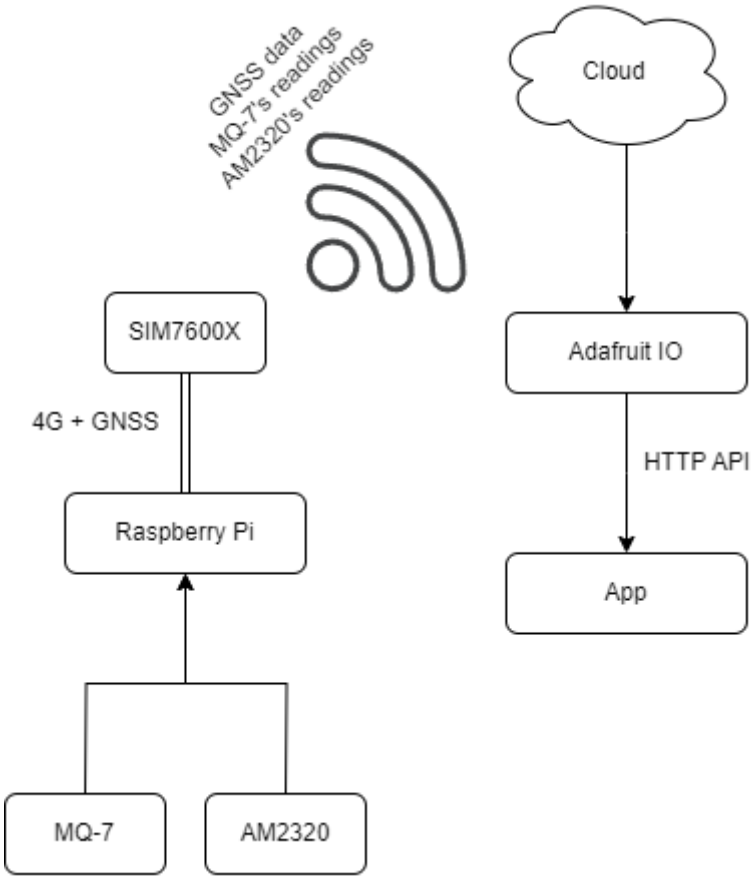


Fig 5.6 - High level view of the architecture

6. Implementation

6.1 Raspberry Pi Setup

6.1.1 Initialising the Pi

As the Pi does not have an onboard storage module, the first step to making it boot up is to flash a microSD card (32 GB in this case), which is then read to a laptop installed with the Raspberry Pi Imager. The setting "Enable SSH" was enabled in the settings menu, with the username being set to "zching1" and password set to "uniofsheffield". The Pi was slotted with the microSD card alongside a mouse, keyboard, and a monitor afterwards.

The Pi was then configured to allow remote access for ease of development. Raspberry Pi OS comes with one such program pre-installed: VNC Server. To enable VNC Server on the Pi, click on the menu in the taskbar, then Preferences > Raspberry Pi Configuration > Interfaces, and toggle the VNC option on. Restart the Pi afterwards and login to VNC Server on the Pi and log in with a VNC account. Install the VNC Viewer on the desktop and log in with the same VNC account, and remote access should now be possible.

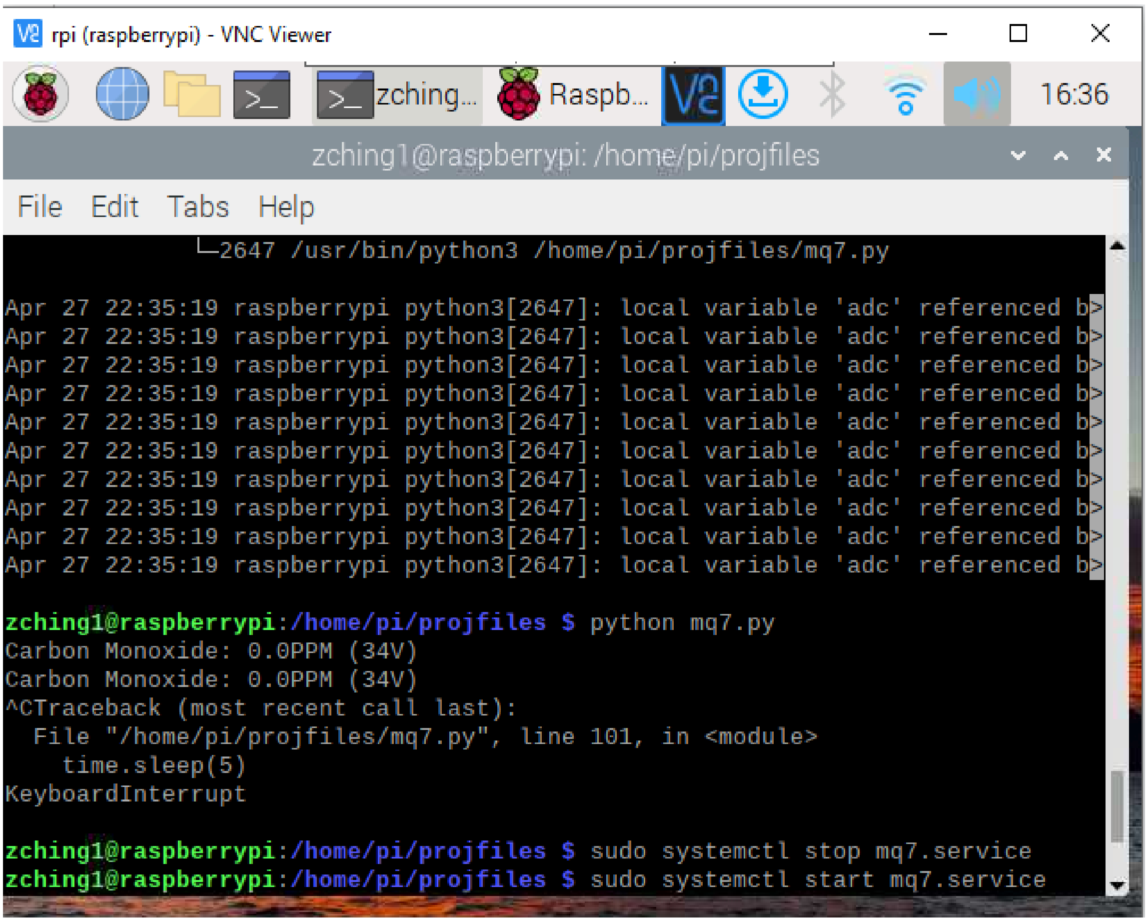


Fig 6.1 - The Pi accessed from a desktop

The Pi also uses two Python scripts to get readings from the sensors and to send their readings onto Adafruit IO. The scripts will also send a push notification to FCM whenever a dangerous reading

appears. These scripts are named after their respective sensors, and can be found at the directory /home/pi/projfiles/. The AM2320 script was based on Lotsengård's work [60], while the MQ-7 script was based on Luster's work [61].

These scripts are then launched by using the systemd service on Linux, which manages programs on start-up. In order to make the scripts work with systemd, a service file has to be written to /lib/systemd/system/. A service file indicates the directory of the file to launch, the username of the Pi, and the conditions that the Pi has to meet in order to launch it [62] The service files (am2320.service and mq7.service) has *network.target* as its condition, meaning that the scripts only start when the Pi has an internet connection. These services are then enabled with the command *sudo systemctl enable x*, where x is the name of the service files.

6.1.2 Initialising the SIM7600X HAT

A valid SIM card must be placed into the HAT's SIM card slot. The SIM card should ideally be a Standard SIM, but Micro SIMs were also found to be compatible with the HAT, although it was quite tricky to get the Micro SIM placed just right for the HAT to work.

After placing the SIM card correctly, the HAT should be connected to the Pi in both ways – through the Pi's pins, and through the USB port of the Pi. Minicom (a serial communication program) can be installed and used to verify that the SIM card is being read by the HAT with the command *AT+CPIN?*, which will return "+CPIN: READY" if the SIM card is able to be read.

The HAT will use PPP in this dissertation, which is a TCP/IP protocol that can be used for communication over a telephone network or the internet [63]. The PPP package must be first downloaded by typing *apt-get install ppp* into a terminal. After that, two files will have to be created in */etc/ppp/peers/rnet* and */etc/chatscripts/gprs* in order for the HAT to be configured to run over the PPP protocol. The configuration files used in the dissertation were based on Anduino System's example [65].

After setting up the configuration files, enter *sudo pon rnet,* and then *ping 8.8.8.8* into a terminal in order to test the internet connection provided by the HAT. If everything went right in the setup phase, the Pi should be able to both transmit and receive packets.

This module was terribly painful to set up, and took 2 full days of trial and error to do so. The SIM7600X could read the SIM card relatively early on in the setup phase, but could not establish a 4G connection. As the SIM7600X can use a variety of protocols to establish a connection (RNDIS, NDIS, PPP), it was suspected that some protocols were outdated, and so the HAT was configured to run each of those protocols with first party documentation [66], to no avail, and forums were of no help either (this post on the Raspberry Pi Forums had 150 posts about the problems on the HAT before it was locked down).

The issue turned out to be a faulty USB cable connected through the Pi. As the HAT is connected both with the pins and the USB cable, the terminal showed that the device was registered on the Pi, which meant that there were no indications that the USB cable was faulty. It was only replaced on a whim after every possible fix that could be found was exhausted.

6.2 Setting Up Adafruit IO and Firebase Cloud Messaging

To set up Adafruit IO to work with existing code, an Adafruit IO account will need to have 5 feeds created (essentially a method to group data in Adafruit IO). The feeds will have to be named co, codanger, temp, tempdanger, and tokens. The purpose of these different feeds will be elaborated upon in the later sections. The API key (accessible through the Adafruit IO dashboard) will also have to be noted down for later usage.

To set up Firebase Cloud Messaging (FCM), sign-up for a Firebase account, then navigate to the console page. Create a new project, then go to Project Settings[1]. Navigate to the Cloud Messaging tab, then click on the vertical ellipses in the Cloud Messaging API section[2], then on "Manage API in Google Cloud Console". Press "Enable" on the new window that pops up. Go back to the General tab, then scroll down to the "Your apps" section. Make a note of the parameters in firebaseConfig[3].

6.3 Showing Data

6.3.1 Dealing with Live Data

The app will have to pull data from Adafruit IO's HTTP API in order to show live data. The modern way to make HTTP requests in Javascript is to make use of asynchronous functions and the Fetch API. [66]

Asynchronous functions in Javascript allow for other code to run while waiting on Promises to resolve. Promises have no values initially, but "promises" to return a value in the future. Promises can take on the states of being pending (still resolving, and is neither fulfilled nor rejected), fulfilled (promise takes on a value), or rejected (promise returns error). A promise is said to be resolved if it's *fulfilled* or *rejected*. [67]

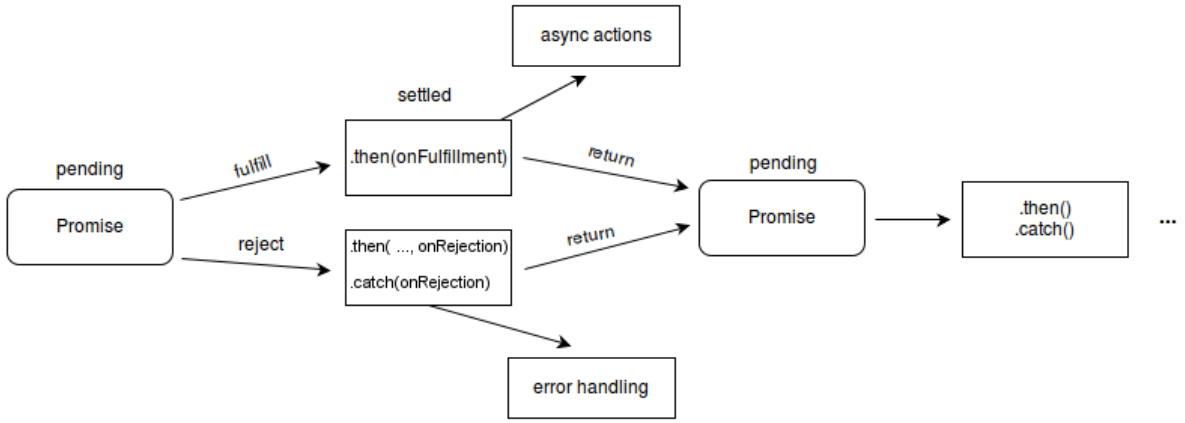


Fig 6.2 - A flowchart showing how Promises function [67]

In the context of fetching data from Adafruit IO's HTTP API, a Promise is first created, and the whole function will *await* (JavaScript essentially halts at the line with the *await* keyword) until the Promise is resolved by taking on a value as a response code resulting from the HTTP API call.

The promise is then *chained* with .then() functions (which trigger after a promise is resolved) to process the data that was pulled from the API. The first *then* checks if the response is a 200 OK code,

and if so, the second *then* processes the JSON object and displays the obtained sensor reading in the dashboard of the app, and the function then calls back to itself to loop after a timeout of 5 seconds.

There are two sections of live data, that being the Latest Alerts section and the individual readings sections. The Latest Alerts section pulls data from the *codanger* and the *tempdanger* feeds (the danger feeds), and the individual readings sections pulls data from the *co* or the *temp* feeds (the nondanger feeds). The only difference between them is that every reading gets sent to the non-danger feeds, while only readings that exceed the threshold value are sent to the danger feeds. While this might seem somewhat redundant, the Adafruit IO HTTP API does not actually include a way to let users filter results by value.

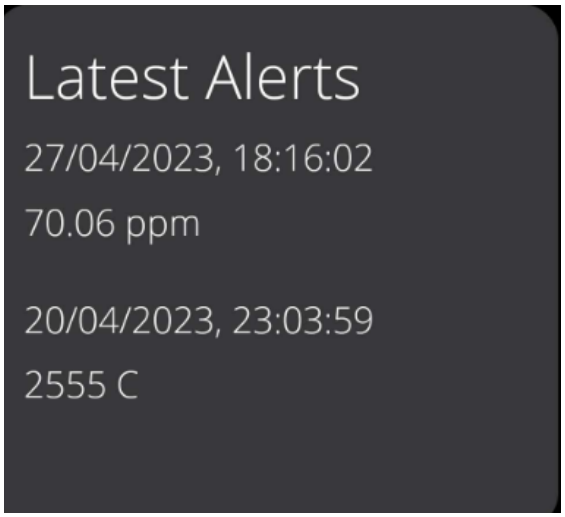


Fig 6.3 - Latest Alerts section

Fig 6.4 - Individual readings section

There were two ways to get the latest value that exceeded the threshold. The first way was to filter the non-danger feeds' arrays to get said value. This will get more computationally expensive as the arrays grow (and they grow fast, at a rate of up to 17280 data points in a single day). The API also paginates its data, meaning that the app can only pull up to 1000 data points at a time, thus making data retrieval more convoluted. [68]

In contrast, the implemented option ignores all of those problems with the trade-off being that data charges might be marginally higher due to having to update and read redundant data.

6.3.2 Visualising Historical Data

The app uses Chart.js to let users select and view a day's worth of sensor readings in the form of a line graph. The Adafruit IO API has a convenient Chart Feed Data function that can average readings through a time-slice parameter, and the returned data points are represented by [*time*, *value*].

While that is all the information Chart.js needs to make a plot, Chart.js requires data points to be in the format of {x: *time*, y: *value*} instead. After converting all data points from the API's format to that of Chart.js', a fresh line chart is ready to be made by plugging in said data points into a Chart.js config along with some customisation options. The rendered line chart is also interactive, as one can click on the individual data points to see the averaged values and its associated time-slice.

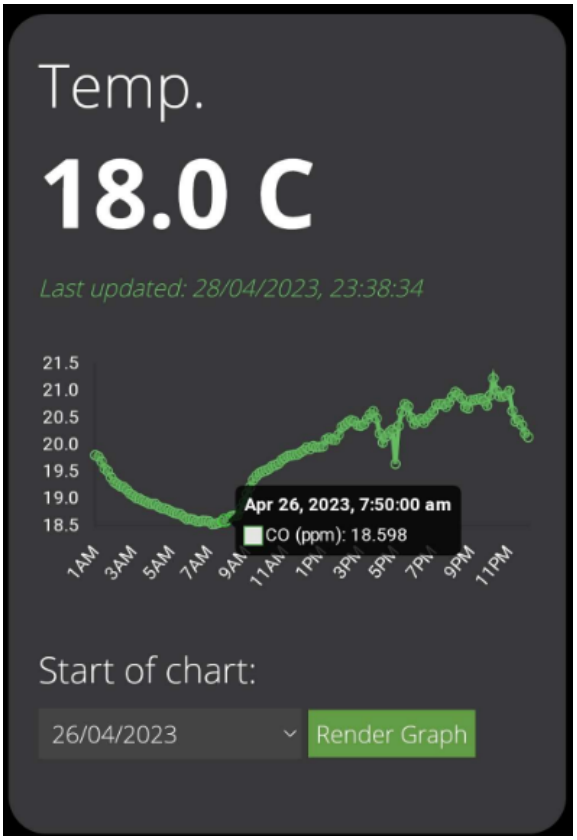


Fig 6.5 - A line chart representing temperature readings from April 26th, 2023

6.4 Storing Data

The data that is stored in the app are the parameters offered in the Settings menu, which entails storing credentials and API keys from both Adafruit IO and Firebase Cloud Messaging. This is not a lot of data, and therefore storage space will not be a consideration here.

Data from the cloud will not be stored locally as the amount of data will balloon significantly over a long period of usage, and the storage options identified here can only store up to 250 MB of data (in the case of IndexedDB) [69]. Furthermore, storing sensor data does not offer a lot of benefits, even when the app is offline. While it allows users to access historical data, they will not be able to receive alerts, which defeats the purpose of the system.

Javascript offers a lot of ways to store data locally, and three of these methods were identified to be suitable for the app: local storage, cookies, and IndexedDB. [70]

Local storage is a type of key-value storage where values are stored as strings. The values stored within are persistent, and only accessible client-side.

Cookies function exactly the same way as local storage does, except that they are non-persistent (having a max life of 365 days), but they are also accessible server-side.

IndexedDB is also a type of key-value storage, but values here can be stored as objects as well. IndexedDB is also asynchronous, unlike the other storage options offered by Javascript.

The app will ultimately use cookies just because cookies are more secure than local storage [71], and cookies are more straightforward to implement comparatively. To work around cookies having a limited lifespan, the app will refresh the expiration date every time the user loads a page, meaning that the cookies will only expire after the user has stopped using the app for a whole year.

6.5 Accessible HTML and CSS designs

This section is written as a continuation of section 5.2, and will discuss how HTML is made to be accessible in the app. This section will also make a lot of references to WCAG 2.1's section headings.

The app does not contain any time-based media, so 1.2 Time-based Media does not apply.

To adhere to 1.3 Adaptable, the app's HTML is structured to follow normal text flow: h1 tags will always be the first to appear in a text section, and p tags will be the last, with any intermediate header tags in between following the same order (1.3.1 & 1.3.2). All content in the app relies on text to convey meaning (1.3.3). The app is responsive, and thus it will work in any orientation (1.3.4). All inputs are identified with a label on the top of them (1.3.5).

To adhere to 1.4 Distinguishable, the app does not use colour to convey meaning. (1.4.1). The contrast of colours should be pretty distinguishable in the app (1.4.3). No images of text are used, and text can be resized without problem (1.4.4 & 1.4.5).

2.1 Keyboard Accessible is passed as the app provides full functionality even with just a keyboard, and there are no keyboard traps.

2.2 Enough Time is not applicable to the app because there is no timed content in the app. 2.3 Seizures and Physical Reactions are also not applicable because the app does not use animations.

To adhere to 2.4 Navigable, pages are titled accordingly (2.4.2), links provide more information on hover to provide context (2.4.4), headings/labels describe purpose (2.6), and section headings are used to organise information. (2.4.10).

The app adheres to 3.1 Readable by using simple language without any unusual abbreviations. 3.2 Predictable is also met by having consistent navigation, and not having any changes of contexts.

Some notable failings are that there are no text alternatives to the line graph, so 1.1 Text Alternatives is not met. While a text alternative can be done by showing a table instead of a line graph, the line graph is interactive and does show textual numeric values upon interacting with data points. There is

also no error identification for the API keys being wrong, which violates 3.3.1, but not having working readings in the dashboard should somewhat imply that.

6.5 Making a PWA

As mentioned in Section 3.6.4, PWAs need to be served over HTTPS, be visited twice with at least 5 minutes in between, have a valid web app manifest, and have a valid service worker so that browsers will make it downloadable to users.

A web app manifest is basically a document that provides information about how a web application should behave when being rendered by a browser. A valid web manifest for PWAs will need to include four parameters within it – *name*, *icons*, *start_url*, and *display* [72]. All of these parameters are quite self-explanatory, so discussion about them will be omitted from this section. FCM requires manifests to have the parameters *gcm_user_visible_only* to be set to true, and *gcm_sender_id* to be set to the appropriate value obtained from Firebase's project settings menu [73].

Service workers are proxy servers that are intended to be used to cache resources in order to improve loading times and to provide offline experiences. They are also fully asynchronous, as they are intended to run alongside the main Javascript thread. Service workers also enable the usage of push notifications, which will be elaborated upon later. Service workers will also "live" in the background for a moment of time as dictated by the host browser, even with the PWA closed [52].

In the context of the app, only the offline page and the relevant CSS stylesheet is cached. If the user is offline (identified through listening for navigation requests and non-valid HTTP requests), the service worker will intercept the request and redirect the user to the offline page instead. Github Pages will be used to host and serve the PWA over HTTPS at https://ziyuancee.github.io.

6.6 Sending Notifications

The PWA uses two APIs for sending notifications: the Notifications API and the Push API. While they serve similar purposes, they function in remarkably different ways. The Notifications API only deals with sending notifications "locally" (i.e. based on events processed by JavaScript) [74]. On the other hand, the Push API is integrated with the service worker, and it sends notifications whenever a properly set up server pushes a message to the service worker [75].

The Push API is used in the app in conjunction with FCM. Whenever a dangerous value is read by the Pi, a HTTP POST request specifying the type and value of the reading will be sent out to a token, which is how FCM distinguishes clients. In turn, FCM will push the request to the said client and thus trigger a push notification, as long as the service worker is still alive.

A client token will eventually expire for reasons unknown after a long period of time. Users can update their client token by clicking on the "Update Token" button in the settings to update the tokens stored on the Pi. This new token will be pushed to the Adafruit IO feed "tokens", and the scripts on the Pi constantly poll the feed for a new token update. The push notifications for this app will not work if these tokens are not updated.

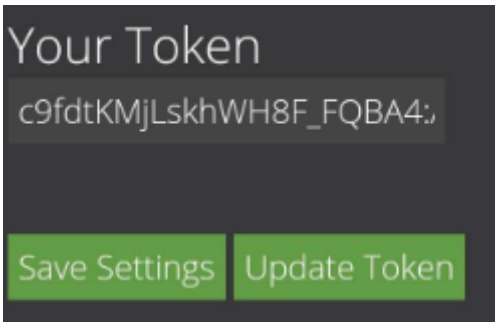


Fig 6.5 - A screenshot of the token and the prompt to update it in the Settings menu

FCM dictates that the script that houses the service worker should be named "firebase-messaging-sw.js". The service worker should also include the messagingSenderId parameter obtained from FCM's project settings page, which causes incompatibility with features that were implemented earlier [76] The messagingSenderId is stored in a cookie, but the Cookies API is synchronous. As stated in the section above, service workers are fully asynchronous, which means that they are not able to access the messagingSenderId cookie. This means that a user must change the hard-coded messagingSenderId value in the service worker whenever they want to run their own version of the app.

The Notification API is used in the app as a fallback option for users experiencing spotty internet connection, as the push notification from FCM would be lost whenever the POST request is sent out but not received by the user's service worker. Whenever the app is able to successfully pull a value from the danger feeds that dates to at most 20 minutes ago, and when the user has the app in the foreground, it will send a notification to the user every 10 minutes, although the timer would be reset on refresh.

A huge amount of time was wasted in trying to send notifications in the background through the periodic background sync API, which wakes up the service worker every once in a while to invoke some function [77]. This would have been useful to the app by letting it poll the danger feeds regularly and send notifications accordingly. However, Microsoft's documentation on the periodic background sync API did not mention that a service worker can only be woken up twice a day in the best scenarios, making this API useless for the purposes of the app. It took 3 days worth of work and research to discover that fact, which was rather frustrating.

The architecture of the system has therefore been updated in order to reflect the usage of FCM to send push notifications.

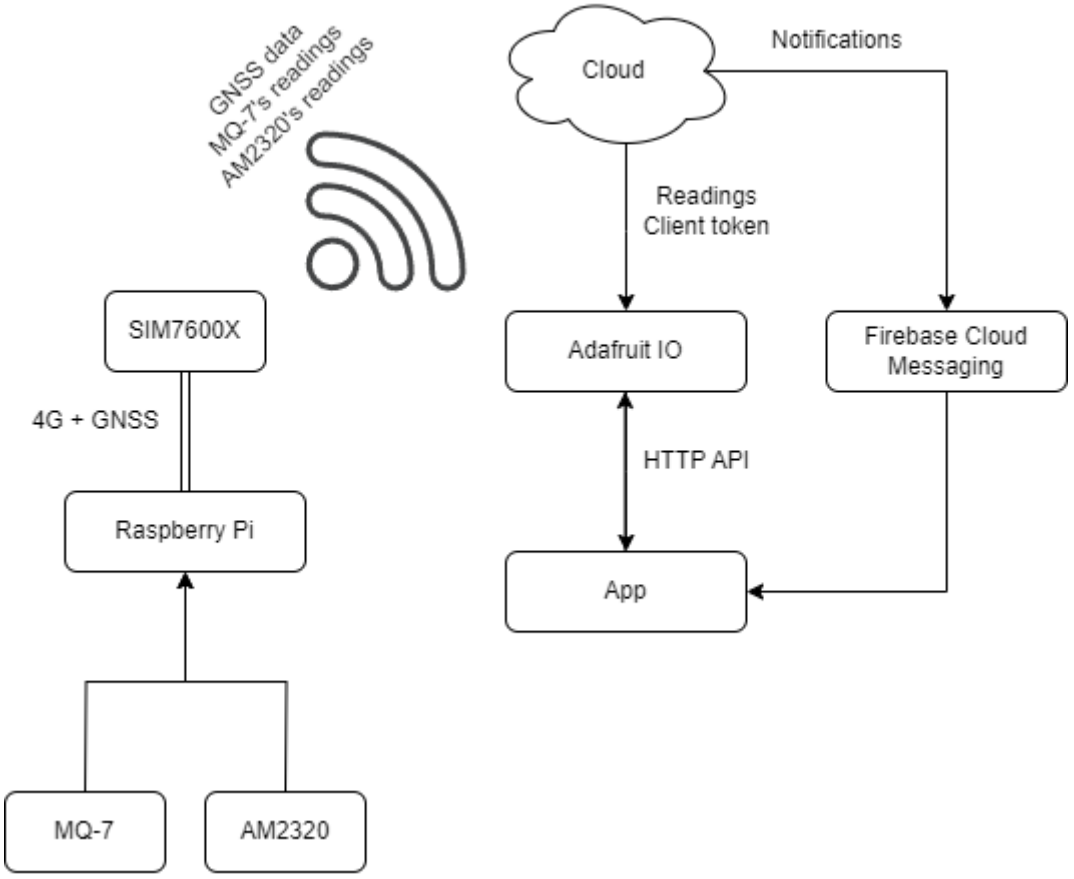


Fig 6.6 - Updated system architecture diagram

7. Testing

7.1 Overview

This section will cover three types of testing: white box testing, functional testing, and automated testing. As said before in section 4.3, white box testing is used to mainly test the reliability of the Pi. Functional testing will be done to ensure that the user experience is bug-free and consistent. Automated testing will be used to ensure that the site is accessible.

7.2 White Box Testing Table

Test Cases	Test Steps	Expected	Actual	Result
Sensor values in the app's dashboard should match up with the sensor values that the Pi read in conditions with good Internet	1. Run the sensor reading scripts on the Pi and print out their values + timestamp in the terminal 2. Verify that the latest reading on the app matches up with all readings	All readings will match up on both the app and the Pi.	All readings did match up on both the app and the Pi.	Pass
The Pi should be able to be work continuously over a long time and reliably upload its readings onto the cloud assuming it has a good connection	1. The Pi will be continuously powered on to collect a week's worth of sensor data while being connected to the SIM card 2. After the one week period, check Adafruit IO to see how many data points were uploaded. In a perfect scenario, there will be 120960 data points uploaded to a feed after a week.	It is probably not possible for the sensors to get the full 120960 data points (one reading per 5 seconds), but they ideally should be able to achieve around 80% of that value (one reading per 6 seconds).	Between 04/16/2023 4:49 P.M. to 04/23/2023 4:49 P.M, cofeed collected 104143 data points (one reading every 5.6 seconds), while tempfeed collected 101189 data points (one reading every 5.8 seconds).	Pass
When the Pi boots up, it should automatically try to connect to the Internet and start up the scripts that reads and uploads the readings from the sensor	1. The Pi will be plugged into power, upon which the Pi will power on by itself 2. The feeds on Adafruit IO will be monitored to see if new	After a short wait, new data points should appear on the Adafruit IO feeds.	New data points were created in the Adafruit IO feeds approximately 10 seconds after the Pi was powered.	Pass

	readings are being sent by the Pi			
The Pi should be able to transmit data for up to 12 hours when the power bank that it's connected to is not being charged	1. The Pi will be powered on by plugging it into the fully charged power bank, and note down the time of this action 2. The feeds on Adafruit IO will be checked for the last reading sent out after the power bank runs out of batteries to determine how long it lasted	Ideally the Pi would last around 12 hours or so with the power bank being fully charged, so that the Pi can still gather data long after the car has been powered off. A battery that lasts for 3 hours should be enough as it is highly unlikely that someone would be left in the car for longer than that period.	The Pi was initially powered on with the fully charged power bank at 04/30/2023 at 11:00 P.M. The last reading sent to Adafruit IO was at 05/01/2023 at 11:45 P.M.	Pass
The Pi's sensors should be reasonably accurate	1. The Pi will be placed on a windowsill with the window open for a few hours 2. The feeds on Adafruit IO will then show hourly averages. This will be compared against the Met Office's weather data	The temperature sensor will roughly follow the trend of the Met Office's data. Some factors will affect the comparison, such as: heater inside the room radiating to the window, and the Met Office data was being collected from a different place that was closest to West Street (from Watnall, Nottinghamshire, a place 27.5 miles away from West Street, and 33 m higher) [78]. The MQ-7 cannot be tested in this manner due to the lack of a supply of CO with a known concentration.	The Pi was placed on the windowsill for 5 hours. It broadly followed the Met Office's data. On May 1st 2023, the Pi collected average readings of 16.97°C at 7 p.m., 16.05°C at 8 p.m., 14.73°C at 9 p.m., 13.23°C at 10 p.m., and 12.22°C at 11 p.m. In comparison, the Met Office reported 15°C, 13°C, 12°C, 12°C, and 10°C for the same times respectively, and so it was reasonably accurate [78].	Pass

7.3 Functional Testing Table

Test Cases	Test Steps	Expected	Actual	Result
The app should be installable on modern mobile phones	1. Visit https://ziyuancee.github.io/ 2. On Android Google Chrome, click on three dots button (overflow menu) and click on install app	The app will appear in the home screen of the phone	The app appeared in the home screen of the phone	Pass
The app should be able to work with the API keys entered correctly into the Settings menu	1. Uninstall the app on the phone and empty cache for the website to wipe cookies 2. Reinstall the app 3. Enter all of the API keys needed and click on “Save Settings” 4. Click on the “Refresh Token” button that pops up	After going the test steps, readings should pop up in the dashboard instead of them having a value of “N/A”.	Readings popped up in the dashboard. Was also able to receive notifications, although the PWA must be in the background and not truly “closed”. This is a limitation of service workers having a limited lifespan.	Pass
The app should be able to receive notifications whenever a dangerous value is detected	1. Light a candle and deprive it of oxygen by covering it with a jar, which will cause an incomplete combustion and generate a lot of CO 2. Place the CO sensor within the jar after the candle is extinguished from the lack of oxygen 3. Wait for a notification to pop	After the CO sensor is placed within the jar, it should send a notification to the phone as the CO concentration within the jar should exceed 70 ppm. While the AM2320 can not be tested in the same way due to a lack of resources, it should function the same as the code for sending notifications is the same for both of the scripts.	A notification was sent to the phone with the header “Dangerous CO levels!” and with the body text “70.06 ppm”.	Pass

	up			
When the user is offline and the user can not receive push notifications, they should still be able to receive notifications about dangerous readings from up to 20 minutes ago whenever the app is launched with an internet connection	1. Put device in offline mode 2. Repeat above experiment 3. After taking the CO sensor out and verifying that it is not picking up >70 ppm readings on Adafruit IO, connect phone to wifi again and launch the app	A notification will be received by the user conveying that dangerous CO levels have been detected.	The notification is received when the app is used on the desktop. However, it does not work on mobile.	Fail
The charts should display a day's worth of data as long as the selected day has associated data points stored on Adafruit IO	1. An arbitrary day with a full day's worth of readings will be chosen 2. The graphs on Adafruit IO's feeds will be compared to the app's graphs to evaluate how accurate the graphs are on the app	The graphs on the app should match closely with Adafruit IO's, if not identical, as both of them use the same data to plot the graph (although the time slices might be different)	The graphs on the app have the same contours as that of Adafruit IO. Graphs were based on data from 17/04/2023. See Appendix 4 for comparisons.	Pass
The charts should be able to handle displaying a day's graph that does not have data in it	1. An arbitrary day with a full day's without readings on Adafruit IO will be chosen 2. Check that the website is able to handle graphing that without errors	Attempting to render a graph with a day that has no data associated with it should not cause an error and/or bugs in the app.	The graphs were rendered with the x and y axis, although no data points appeared. The app was still able to function normally after that.	Pass
The app should show a custom offline page whenever the user is not connected to the Internet	1. Put device in offline mode 2. Load up the app	The dashboard of the app should say that the user is not connected to the internet	The dashboard of the app said that the user is not connected to the internet	Pass

7.4 Automated Testing

Lighthouse is an automated auditing tool that can be used to audit PWAs, accessibility, performance, etc. PageSpeed Insights is also another tool that will be used here, although it is a functionally identical tool that also uses Lighthouse to do the auditing [79].

The app was able to score 100/100 in accessibility according to PageSpeed Insights. The app also passed all of the tests that Lighthouse used in order to determine if a PWA was optimised.

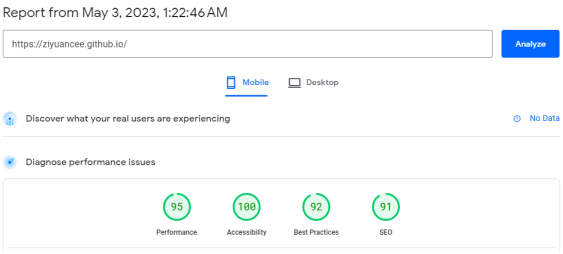


Fig 7.1 - Summary of PageSpeed Insight's report on the app

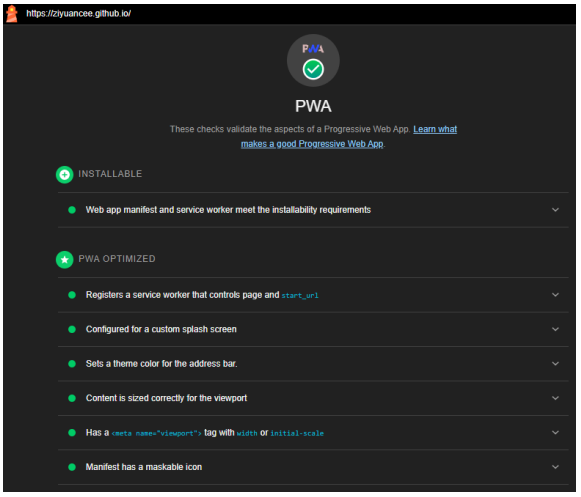


Fig 7.2 - Lighthouse's report on the app

8. Results and Discussion

8.1 Overview

This chapter will evaluate the completeness of the system by considering its usability, accessibility, and reliability, which are the most important aspects of the system by far. The system will be compared against the list of functional requirements established in section 4.2 in order to determine the extent of the objectives that were achieved in this project. Potential extensions to the system will also be discussed.

8.2 Usability and Accessibility

The PWA developed for the system is quite easy to use and has been made to be as accessible as it can be in the current project scope. The app is quite capable in the sense that the current implementation has achieved the fundamental goals of the dissertation – it can read live data, alert the user to dangerous CO and/or temperature levels, and it allows the user to visualise day-to-day readings.

However, the PWA also feels too simplistic in its current iteration – it does not offer much functionality outside of the core functionalities that the system requires. Some of the proposed features, such as GPS tracking and its associated “heatmap” were not feasible to implement, as there was not enough documentation on operating the SIM7600X’s GNSS module. The first-party guide by Waveshare was not enough to make the module work with the current Pi setup either.

Letting the user set up threshold values would have also been a great addition to the app, but it was not implemented due to time constraints. The architecture to make that work has already been implemented however, in the form of tokens. The token’s value is pushed into an Adafruit IO feed whenever the user clicks on the “Update Token” button. The Raspberry Pi then polls the tokens feed every once in a while to update its own token value so that it knows which device to send the FCM notifications to. This process can definitely be applied to the threshold value settings function, as the threshold values are stored on the Pi, much like the tokens.

The PWA would also benefit from using IndexedDB instead of cookies to store data, as cookies cannot be read by the service worker, thus necessitating that the “messagingSenderId” parameter be hard-coded into the service worker in order for FCM to function properly.

The incomplete functions mentioned above are some of the extensions that can be applied to the PWA. The PWA can also benefit from being more accessible by showing the chart as a table instead so that screen readers can pick it up. More ways of visualising the readings can be valuable too, such as being able to visualise data over a week instead of a single day, and so on. Additionally, the problem where the “backup” notifications were able to show up on a desktop but not on a mobile is worth looking into as well. However, that would not be a problem to attentive users as the Latest Alerts section does show the timestamp of when the latest dangerous CO/temperature levels were observed.

Despite how accessible the system as a whole is, the setup required to build the project from scratch is not at all accessible to the layman due to how technical setting up the system could get. This is a fundamental issue with the project, as the setup requires a certain amount of knowledge for wiring,

programming, Linux commands, and setting up both Adafruit IO and FCM. There might not be a solution with the current architecture of the system, and as such, the middle-to-old aged people should ask a more technically minded person to help them with setting up the system.

8.3 The Pi's Reliability and Features

The Raspberry Pi is actually fairly reliable as a controller for the system. It is able to fully initialise its scripts on startup, it can run continuously for at least up to a week while still being able to upload most of its data points onto Adafruit IO, it can run for a little over a day with its power bank fully charged, and the sensors seem to generally work.

The sensors being able to "generally work" is not the same as being accurate, however. While it seems like the AM2320 is fairly accurate, being able to match the Met Office's data on Watnall, Nottinghamshire to +2°C, it is unknown how accurate the MQ-7 is without access to a laboratory. The MQ-7 does pick up on the CO produced in the incomplete combustion test, and it does give an output that implies that it is a dangerous concentration of CO (70.06 PPM), which sounds right intuitively. Regardless, the CO's actual concentration in the jar was unknown, so further testing (and calibration) of these sensors in laboratory conditions will be very helpful in giving better results.

As with the PWA, the incomplete requirements can be added to further improve the system in the future. The requirement "As a passenger in the user's car, I should be able to hit a button on the Raspberry Pi to send a notification to the user's phone" was not completed due to time constraints. However, this should be a fairly easy implementation as well. A button can be connected to one of the GPIO pins of the Pi, a library (RPi.GPIO) can then be used to read the button press, and the FCM code to send push notifications can also be reused from the AM2320/MQ-7 scripts.

The system is also very modular, so expanding the Pi's functionality is quite easy. More sensors could be added in order to detect different pollutants, for example. The system could also be kept to a lower cost, and maybe even developed on a different computer (the Raspberry Pi 3B with the SIM7600X HAT cost 150 pounds alone). The design of the board could also be transferred to that of a PCB, which will make storage of the device easier and less fragile, as the exposed wires on the breadboard can detach easily if something knocks into it.

8.4 Comparison against Functional Requirements

Requirements	Result and Discussion
As a user, I should be able to connect the Raspberry Pi to my mobile phone	**Partially complete**. The messagingSenderId is hard-coded into the service worker due to unforeseen technical limitations.
As a user, I should be able to see temperature and carbon monoxide readings from the app	**Fully complete.**
As a user, I should be able to set threshold values for carbon	**Partially complete**. This is possible, but the user has to change the code on the scripts that read the sensors to accommodate for

monoxide and temperature levels	this requirement.
As a user, I should get a notification warning me about temperature and/or carbon monoxide levels exceeding a threshold value	**Fully complete.**
As a user, I should be able to download the app onto my phone	**Fully complete.**
As a user, my Raspberry Pi 3B should send sensor data onto the cloud upon powering up without my intervention	**Fully complete.**
As a user, I should be able to see a history of temperature and carbon monoxide readings	**Fully complete.**
As a user, I should be able to locate where the device is	**Incomplete**. Lack of documentation on the SIM7600X's GPS module made this requirement hard to implement.
As a passenger in the user's car, I should be able to hit a button on the Raspberry Pi to send a notification to the user's phone	**Incomplete** due to time constraints.
As a user, I should be able to see a fact sheet on heat stroke symptoms	**Fully complete** although it was a link to the NHS' factsheet instead, which was a better alternative as it would be regularly updated.
As a user, I should be able to see a daily breakdown of where the device was and the average and highest temperatures of those places	**Incomplete**. Lack of documentation on the SIM7600X's GPS module made this requirement hard to implement.

Based on the table above, the system is fairly complete, as the "must have" requirements are all complete to a degree. The messagingSenderId and threshold values can be changed in the code when a person is trying to set this up, and a person that is able to set up this system is most likely not going to have any issues with changing a few lines of code to suit their setup. The one "should have" requirement is also complete, but only 1/4 "could have" requirements were complete, meaning that the system would not have as many optional features as it could have.

Still, the completed requirements prove that such an IoT system is feasible to build, and the current implementation of the system can serve as a nice springboard for people to expand upon.

9. Conclusion

While heat strokes and CO poisonings can easily be mitigated with vigilance, many people still sustain injuries or die from said problems. An IoT based monitoring system involving a sensing device placed in a car and interfaced with an app was developed in order to combat said risks.

An extensive literature and technology review was then conducted in order to find suitable frameworks for the app, which was vanilla Javascript with HTML, and to find suitable hardware to use in tandem with each other to act as the sensing device, which was a Raspberry Pi 3B connected to an AM2320, a MQ-7, and the SIM7600X HAT.

The literature and technology review also allowed for a list of functional and non-functional requirements to be created that takes into account specific use cases, accessibility requirements, and how the app and system as a whole should function. This list will then guide the design and development of the system.

The app was then tested to see if the existing functionalities were implemented properly and if the Pi was reliable enough as a controller. The results indicate that the Pi was indeed reliable enough, and that the app works mostly as intended, with only one test failing. Google Lighthouse, which is a tool that can automatically evaluate accessibility and optimisation of a PWA, gave the app a 100/100 in accessibility, and determined that the PWA was optimised. The app also seems to be quite snappy and responsive in practice.

This project could be considered as a success overall as the system has its core functionalities implemented with a strong foundation built for it. While optional functionalities are somewhat lacking from the system, the architecture for the system is already there, along with an established way for the app to communicate with the Pi, and vice versa. Integrating the GPS module with the system is an obvious choice that will add a lot of value to the system, for example. As the system collects a lot of data, a machine learning model can also be employed to perform analytics to grant users more insight on their surroundings.

The nature of it being a microcomputer project only makes the system more modular, which should make implementing new functionalities far more easier.

References

1. Kim, E. (2022). “Europe Heatwaves Disastrous for Older People, People with Disabilities.” [online] *Human Rights Watch*. Available: https://www.hrw.org/news/2022/08/12/europe-heatwaves-disastrous-older-people-people-disabilities. [Accessed 9 Nov. 2022].
2. NHS. (2019). “Carbon monoxide poisoning.” [online] Available: https://www.nhs.uk/conditions/carbon-monoxide-poisoning/. [Accessed 18 Nov. 2022].
3. T. H. Greiner (n.d). “Carbon Monoxide Poisoning: Vehicles (AEN-208),” [online] *Department of Agricultural and Biosystems Engineering, University of Iowa*. Available:https://www.abe.iastate.edu/extension-and-outreach/carbon-monoxide-poisoning-vehicles-aen-208/#:~:text=An%20exhaust%20leak%20can%20allow,the%20end%20of%20the%20tailpipe. [Accessed 6 May 2023].
4. B. Eshghi (n.d.). “5 Use Cases of IoT in Automotive in 2023.” [online] *AIMultiple*. Available: https://research.aimultiple.com/iot-automotive/. [Accessed 6 May 2023].
5. A. Bouchama, “Heat Stroke,” *New England Journal of Medicine*, July, 2022. [Online serial]. Available: http://medicalux.fr/www/nicbook/Biblio/Insolation/heat_stroke_nejm_6-02.pdf. [Accessed 9 Nov. 2022].
6. Mayo Clinic. (2017). “Heatstroke - Symptoms and causes.” [online] Available: https://www.mayoclinic.org/diseases-conditions/heat-stroke/symptoms-causes/syc-20353581.[Accessed 9 Nov. 2022].
7. nidirect Government Services. (2017). “Heat exhaustion and heatstroke | nidirect. [online] Available at: https://www.nidirect.gov.uk/conditions/heat-exhaustion-and-heatstroke. [Accessed 9 Nov. 2022].
8. T. Hifumi, Y. Kondo, K. Shimizu, Y. Miyake, “Heat stroke”, *Journal of Intensive Care,* vol 6, no. 30, May 2008. [Online serial]. Available: https://jintensivecare.biomedcentral.com/articles/10.1186/s40560-018-0298-4. [Accessed 9 Nov. 2022].
9. Y. Miyake, “Pathophysiology of heat illness: Thermoregulation, risk factors, and indicators of aggravation.”, *Japan Medical Association Journal,* vol 56, no. 3, p. 167-173. May 2013. [Online serial]. Available: https://www1.med.or.jp/english/journal/pdf/2013_03/167_173.pdf. [Accessed 9 Nov. 2022].
10. The Healthline Editorial Group (2018). “Hot and Cold: Extreme Temperature Safety.” [online]. Available: https://www.healthline.com/health/extreme-temperature-safety#extreme-heat-temperatures. [Accessed 29 Nov. 2022].
11. CDC (2022). “Disease of the Week - CO Poisoning.” [online] Centers for Disease Control and Prevention. Available: https://www.cdc.gov/dotw/carbonmonoxide/index.html#:~:text=Every%20year%2C%20at%20least%20430. [Accessed 29 Nov. 2022].
12. D. M. Colwill, A. J. Hickman, “Exposure of Drivers to Carbon Monoxide.”, *Journal of the Air Pollution Control Association,* vol 30, no. 12, p. 1316-1319. 1980. [Online serial]. Available: https://www1.med.or.jp/english/journal/pdf/2013_03/167_173.pdf. [Accessed 1 Dec. 2022].
13. United State Consumer Product Safety Commission (n.d.). “Carbon-Monoxide-Questions-and-Answers.” [online] cpsc.gov. Available:

https://www.cdc.gov/dotw/carbonmonoxide/index.html#:~:text=Every%20year%2C%20at%20least%20430. [Accessed 1 Dec. 2022].
14. W3C (n.d.). "Accessibility." [online] W3.org. Available: https://www.w3.org/standards/webdesign/accessibility. [Accessed 10 Nov. 2022]
15. Central Digital and Data Office (2018). "Understanding accessibility requirements for public sector bodies." [online] GOV.UK. Available: https://www.gov.uk/guidance/accessibility-requirements-for-public-sector-websites-and-apps.
16. Disability Rights Advocates (2007). "NATIONAL FEDERATION OF THE BLIND V. TARGET." [online] Available: https://web.archive.org/web/20070705103922/http://www.dralegal.org/cases/private_business/nfb_v_target.php [Accessed 10 Nov. 2022].
17. W3C (n.d.). "Tips for Getting Started Designing for Web Accessibility." [online] W3.org. Available: https://www.w3.org/WAI/tips/developing/. [Accessed 10 Nov. 2022].
18. W3C (n.d.). "W3C Accessibility Standards Overview." [online] W3.org. Available: https://www.w3.org/WAI/standards-guidelines/#wcag2. [Accessed 10 Nov. 2022].
19. W3C (n.d.). "Designing for Web Accessibility." [online] W3.org. Available: https://www.w3.org/WAI/tips/designing/. [Accessed 10 Nov. 2022].
20. Shea, S. (2018). "microcomputer." [online] Tech Target. Available: https://www.techtarget.com/iotagenda/definition/microcomputer. [Accessed 11 Nov. 2022].
21. Dave, P. (2016). "Raspberry Pi 3 out, what's the difference? A simple comparison chart and some references." [online] Peter Dave Hello's Blog. Availabl: https://www.peterdavehello.org/2016/02/raspberry-pi-3-out-what-is-the-difference-a-simple-comparison-chart-and-some-references/. [Accessed 11 Nov. 2022].
22. Lutkevich, B. (2019). "microcontroller (MCU)." [online] Tech Target. Available: https://www.techtarget.com/iotagenda/definition/microcontroller. [Accessed 11 Nov. 2022].
23. Vysakh (2018). "Basics of Microcontrollers. [online] Electronic Circuits and Diagrams-Electronic Projects and Design." Available: http://www.circuitstoday.com/basics-of-microcontrollers. [Accessed 11 Nov. 2022].
24. V. Kelkar, M. Pange, K. Sawlani. "Exposure of Drivers to Carbon Monoxide.", [online] *SSRN*. Available: https://www1.med.or.jp/english/journal/pdf/2013_03/167_173.pdf. [Accessed 7 May 2023].
25. A. P. Murdan, V. Bucktowar, V. Oree, M. P. Enoch, "Low-cost bus seating information technology system.", *IET Intelligent Transport Systems,* vol 14, no. 10, p. 1303-1310. September 2020. [Online serial]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-its.2019.0529. [Accessed 7 May 2023].
26. M. M. Yusuf, S. Sahrani, M. H. M. Saad, M. Sarker, M. Z. A. Samah, "DESIGN AND DEVELOPMENT OF AN INTERNET OF THINGS (IOT) BASED REAL TIME MONITORING AND CONTROL SYSTEM FOR SMART INDOOR HYDROPONIC VERTICAL FARMING SYSTEM WITH ESP32 AND ADAFRUIT IO.", *Journal of Information System and Technology Management,* vol 7, no. 28, p. 155-163. November 2022. [Online serial]. Available: https://www.researchgate.net/profile/Mohamad-Hanif-Md-Saad/publication/368424343_Design_and_Development_of_An_Internet_of_Things_IoT_Based_Real_Time_Monitoring_and_Control_System_for_Smart_Indoor_Hydroponic_Vertical_Farming_System_With_ESP32_and_Adafruit_IO/links/63e71334dea61217579de9c9/Design-and-Development-of-An-Internet-of-Things-IoT-Based-Real-Time-Monitoring-and-Control-System-for-Smart-Indoor-Hydroponic-Vertical-Farming-System-With-ESP32-and-Adafruit-IO.pdf. [Accessed 7 May 2023].

27. Wolford, B. (n.d.). “What Is GDPR, the EU’s New Data Protection law?” [online] GDPR.eu. Available: https://gdpr.eu/what-is-gdpr/. [Accessed 17 Nov. 2022].
28. Adafruit (2021). “Adafruit Privacy Policy.” [online] www.adafruit.com. Available at: https://www.adafruit.com/privacy [Accessed 17 Nov. 2022].
29. Medicine & Healthcare products Regulatory Agency. “Guidance: Medical device stand-alone software including apps (including IVDMDs).” MHRA. Available: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1105233/Medical_device_stand-alone_software_including_apps.pdf. [Accessed 7 May 2023].
30. Espressif (n.d.). “GPIO & RTC GPIO.” [online] ESP-IDF Programming Guide. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/gpio.html. [Accessed 11 Nov. 2022].
31. ElectronicWings (n.d.). Raspberry Pi GPIO Access | Raspberry Pi. [online] Available: https://www.electronicwings.com/raspberry-pi/raspberry-pi-gpio-access. [Accessed 11 Nov. 2022].
32. Espressif (n.d.). “Power Management.” [online] ESP-IDF Programming Guide. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/power_management.html. [Accessed 11 Nov. 2022].
33. Hildenbrand, J. (2019). “Raspberry Pi 3 Model B vs. 3 B+: Which should you buy?” [online] Android Central. Available: https://www.androidcentral.com/raspberry-pi-3-model-b-vs-3-b. [Accessed 11 Nov. 2022].
34. Shovon, S. (2018). “Raspberry Pi 3 Power Requirements.” [online] Linux Hint. Available: https://linuxhint.com/raspberry_pi_3_power_requirements/#:~:text=Raspberry%20Pi%203%20Model%20B%20consumes%20about%20260%20mA%20of.
35. J. W. Jolles, “Broad-scale applications of the Raspberry Pi: A review and guide for biologists.”, *Methods in Ecology and Evolution,* vol 12, p. 1562-1579. June 2021. [Online serial]. Available: https://besjournals.onlinelibrary.wiley.com/doi/full/10.1111/2041-210X.13652. [Accessed 14 Nov. 2022].
36. DiCola, T. (2016). “Raspberry Pi Analog to Digital Converters.” [online] Adafruit Learning System. Available: https://learn.adafruit.com/raspberry-pi-analog-to-digital-converters/mcp3008. [Accessed 14 Nov. 2022].
37. Short, M. and Joel, E. B. (n.d.). “How to Use a Breadboard.” [online] learn.sparkfun.com. Available: https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard/all#providing-power-to-a-breadboard. [Accessed 14 Nov. 2022].
38. Components101 (2020). “What is a Gas Sensor? Construction, Types & Working of Gas Sensors.” [online] Available: https://components101.com/articles/introduction-to-gas-sensors-types-working-and-applications. [Accessed 14 Nov. 2022].
39. The Pi Hut (n.d.). “MQ-7 Gas Sensor (Carbon Monoxide).” [online] Available: https://thepihut.com/products/mq-7-gas-sensor. [Accessed 15 Nov. 2022].
40. Amazon (n.d.). “Logic Level Converter Bi-Directional.” [online] Available: https://www.amazon.co.uk/SparkFun-BOB-12009-Logic-Converter-Bi-Directional/dp/B00M7U5DV2. [Accessed 7 May 2023].
41. Last Minute Engineers (n.d.). “Interfacing AM2320 Temperature & Humidity Sensor with Arduino.” [online] Available:

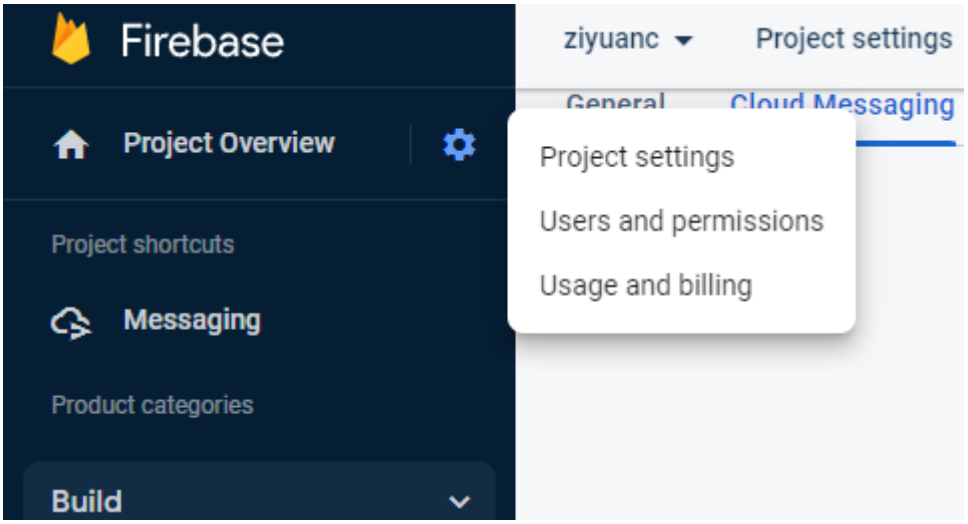
https://lastminuteengineers.com/am2320-temperature-humidity-sensor-arduino-tutorial/. [Accessed 15 Nov. 2022].
42. Pimoroni (n.d.). “AM2320 Digital Temperature and Humidity Sensor.” [online] Available: https://shop.pimoroni.com/products/digital-temperature-and-humidity-sensor. [Accessed 7 May 2023].
43. Thurium, T. (2019). “Three Easy Ways To Brick A Raspberry Pi. [online] Twilio Blog.” Available: https://www.twilio.com/blog/3-ways-brick-raspberry-pi#:~:text=The%20Pi%20is%20engineered%20to. [Accessed 15 Nov. 2022].
44. Vis, P.J. (n.d.). “Raspberry Pi Power Supply.” [online] www.petervis.com. Availabl: https://www.petervis.com/Raspberry_PI/Raspberry_PI_Power_Supply/Raspberry_PI_Power_Supply.html. [Accessed 16 Nov. 2022].
45. Seidle, N. (n.d.). “Battery Technologies.” [online] Sparkfun.com. Available: https://learn.sparkfun.com/tutorials/battery-technologies/all. [Accessed 16 Nov. 2022].
46. Vrabie, R. (2022). “How to choose the best Raspberry Pi power bank.” [online] Power Bank Expert. Available: https://www.powerbankexpert.com/best-raspberry-pi-power-bank/. [Accessed 16 Nov. 2022].
47. Fromaget, P. (n.d.). “Raspberry Pi OS vs Ubuntu: What’s the Best for Desktop Usage?” [online] Raspberry Tips. Available: https://raspberrytips.com/raspberry-pi-os-vs-ubuntu/. [Accessed 16 Nov. 2022].
48. Apple Inc. (n.d.). “Welcome to Swift.org.” [online] www.swift.org. Available: https://www.swift.org/. [Accessed 7 May 2023].
49. BLAKIT IT Solutions (2018). “Why developers prefer Kotlin. The key Kotlin features & updates.” [online] DEV Community. Available: https://dev.to/blak_it/why-developers-prefer-kotlin-the-key-kotlin-features--updates-10hp. [Accessed 16 Nov. 2022].
50. Codecademy Team (2021). “What Is Kotlin Used For?” [online] Codecademy. Available: https://www.codecademy.com/resources/blog/what-is-kotlin-used-for/. [Accessed 16 Nov. 2022].
51. S. Richard, P. Lapage. (2020). “What are Progressive Web Apps?” [online] web.dev. Available: https://web.dev/what-are-pwas/. [Accessed 7 May 2023].
52. Mozilla (n.d.). “Service Worker API” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. [Accessed 7 May 2023].
53. A. Osmani (2020). “Getting started with Progressive Web Apps.” [online] Chrome Developers. Available: https://developer.chrome.com/blog/getting-started-pwa/#add-to-home-screen-banner. [Accessed 7 May 2023].
54. J. Tran (2022). “PWA vs Native App and how to choose between them.” [online] Magestore. Available: https://www.magestore.com/blog/pwa-vs-native-app-and-how-to-choose-between-them/. [Accessed 7 May 2023].
55. Chrome Developers (n.d.). “Progressive Web Apps.” [online] web.dev. Available: https://web.dev/learn/pwa/progressive-web-apps/#desktop-and-laptops. [Accessed 7 May 2023].
56. Adafruit IO (n.d.). “Adafruit IO HTTP API.” [online] www.io.adafruit.com. Available: https://io.adafruit.com/api/docs/#about-the-api-docs. [Accessed 7 May 2023].

57. thinger.io (n.d.). “Pricing.” [online] www.thinger.io. Available: https://thinger.io/pricing/#community. [Accessed 7 May 2023].
58. Hanwei Electronics Co. Ltd. (n.d.) “Technical Data MQ-7 Gas Sensor.” [online] Sparkfun. Available: https://www.sparkfun.com/datasheets/Sensors/Biometric/MQ-7.pdf. [Accessed 7 May 2023].
59. S. Richard, P. Lapage. (2022). “What makes a good Progressive Web App?” [online] web.dev. Available: https://web.dev/pwa-checklist/. [Accessed 7 May 2023].
60. J. Lotsengård. (2021). “am2320.” [online] www.github.com. Available: https://github.com/Gozem/am2320. [Accessed 7 May 2023].
61. J. Luster. (2015). “GoodAir.” [online] www.github.com. Available: https://github.com/jonesluster/GoodAir/tree/master. [Accessed 7 May 2023].
62. Y. Wate. (2021). “How to Use systemd to Launch Programs at Startup on Raspberry Pi.” [online] Make Use Of. Available: https://www.makeuseof.com/what-is-systemd-launch-programs-raspberry-pi/. [Accessed 7 May 2023].
63. IBM. (2021). “What is PPP.” [online] www.ibm.com. Available: https://www.ibm.com/docs/en/i/7.2?topic=concepts-what-is-ppp. [Accessed 7 May 2023].
64. Andino Systems. “SIM7600E: Setup via ppp (4G/LTE Modem).” [online] Clear Systems GmbH. Available: https://andino.systems/extensions/4g-modem-sim7600/ppp. [Accessed 7 May 2023].
65. Waveshare. “SIM7600E-H 4G HAT.” [online] www.waveshare.com. Available: https://www.waveshare.com/wiki/SIM7600E-H_4G_HAT. [Accessed 7 May 2023].
66. Mozilla (n.d.). “Using the Fetch API.” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. [Accessed 7 May 2023].
67. Mozilla (n.d.). “Promise.” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed 7 May 2023].
68. Adafruit IO (n.d.). “Adafruit IO HTTP API.” [online] www.io.adafruit.com. Available: https://io.adafruit.com/api/docs/#adafruit-io-http-api. [Accessed 7 May 2023].
69. WebPlatform (n.d.). “Indexed Database API.” [online] www.webplatform.github.io. Available: https://webplatform.github.io/docs/apis/indexeddb/. [Accessed 7 May 2023].
70. J. Liao (2020). “Browser storage: Local Storage, Session Storage, Cookie, IndexedDB and WebSQL.” [online] Medium. Available: https://medium.com/@lancelyao/browser-storage-local-storage-session-storage-cookie-indexeddb-and-websql-be6721ebe32a. [Accessed 7 May 2023].
71. Pivot Point Security (2023). “Local Storage Versus Cookies: Which to Use to Securely Store Session Tokens.” [online] www.pivotpointsecurity.com. Available: https://www.pivotpointsecurity.com/local-storage-versus-cookies-which-to-use-to-securely-store-session-tokens/. [Accessed 7 May 2023].
72. Mozilla (n.d.). “Web app manifests.” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/Manifest. [Accessed 7 May 2023].
73. Google (n.d.). “Set up a JavaScript Firebase Cloud Messaging client app.” [online] Firebase. Available: https://web.dev/learn/pwa/progressive-web-apps/#desktop-and-laptops. [Accessed 7 May 2023].
74. Mozilla (n.d.). “Notification.” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/notification. [Accessed 7 May 2023].

75. Mozilla (n.d.). “Push API.” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/Push_API. [Accessed 7 May 2023].
76. Google (n.d.). “Set up a JavaScript Firebase Cloud Messaging client app.” [online] Firebase. Available: https://web.dev/learn/pwa/progressive-web-apps/#desktop-and-laptops. [Accessed 7 May 2023].
77. Mozilla (n.d.). “ServiceWorkerRegistration: periodicSync property.” [online] Mozilla Developer Network. Available: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration/periodicSync. [Accessed 7 May 2023].
78. Met Office. “Sheffield (South Yorkshire) last 24 hours weather”. [online] www.metoffice.gov.uk. Available: https://www.metoffice.gov.uk/weather/observations/gcqzwtdw7. [Accessed 7 May 2023].
79. Chrome Developers (2022). “Lighthouse overview.” [online] developer.chrome.com. Available: https://developer.chrome.com/docs/lighthouse/overview/. [Accessed 7 May 2023].

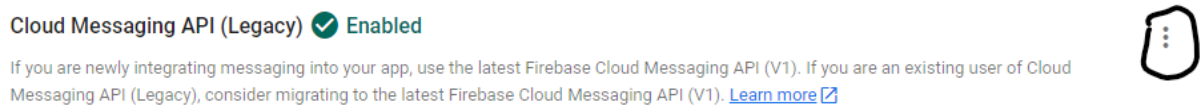
Appendices

Appendix 1 - Project Settings



Back to content

Appendix 2 - Cloud Messaging API (Legacy)



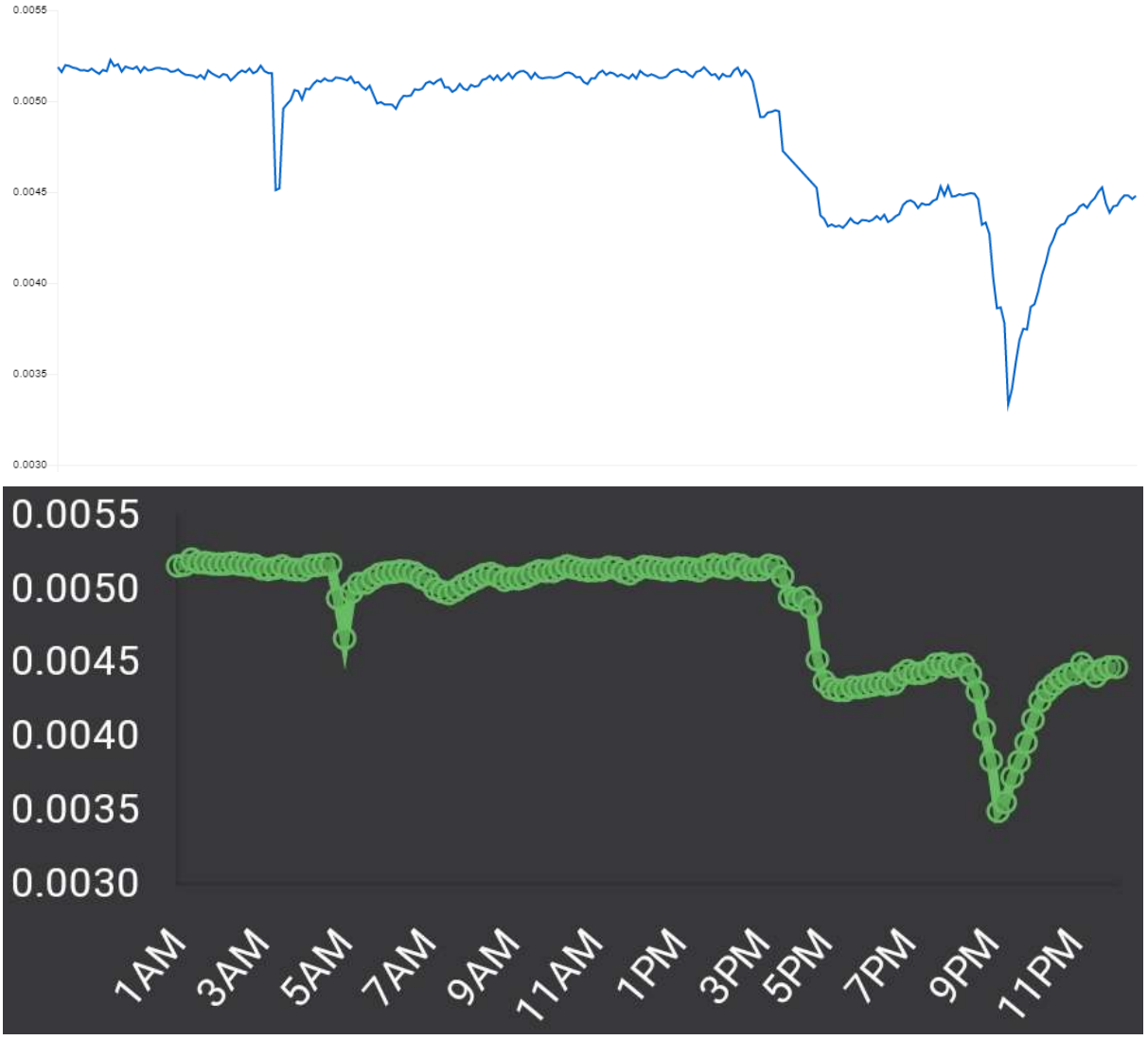
Back to content

Appendix 3 - firebaseConfig parameters

```
const firebaseConfig = {
  apiKey: [redacted],
  authDomain: [redacted],
  projectId: [redacted],
  storageBucket: [redacted],
  messagingSenderId: [redacted],
  appId: [redacted],
  measurementId: [redacted]
};
```

Back to content

Appendix 4 - CO graphs comparison



Back to content