GNG5140

Design Project User and Product Manual

CITY POLLUTION MONITORING

Submitted by:

Adrian Iannantuono, 300071774

Drishya Suresh, 300322663

Max Shui, 300259431

Myron Soares, 300346181

Tarin Sultana, 300393224

20/12/2023

University of Ottawa

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
CO2	Carbon Dioxide
LED	Light Emitting Diode
LCD	Liquid Crystal Display
PM2.5	Fine particulate matter (particles 2.5μm or less in diameter)
GPS	Global Positioning System

Table 1: Acronyms

1 Introduction

This User and Product Manual (UPM) provides the information necessary for users, technicians, and engineers to effectively use the Pollution Monitoring Module, and for prototype documentation. The intent of this manual is to ensure the collection of accurate and reliable air pollutant data that can be used for further research by the City of Ottawa or any interested parties.

In the development of this manual, we assume a basic familiarity with technological devices, adherence to safety guidelines, and an interest in utilizing personal monitoring technology for research or for personal health and well-being. We further assume that users will comply with all applicable laws and regulations concerning the use of such devices.
An overview of the topics covered in this manual is given below:

- Section 2 gives a basic introduction to the problem and the solution our product offers. It also sets the criteria that distinguishes our product from similar items on the market.
- Section 3 describes how to set up the module in non-technical terms for users unfamiliar with the product. It describes all functions that can be performed by the module.
- Section 4 gives an overview of the common error messages that the product can show and steps to resolve the same.
- Section 5 is a technical walkthrough of the final prototype developed. It describes the components and tools used, as well as steps to recreate the product. It also gives an overview of the testing performed on the system.
- Section 6 covers the final comments and future work recommendations for our product.

2 Overview

Rising air pollution levels pose a significant hazard to the general population. Due to rapid urbanization and an increasing number of fossil fuel-powered vehicles, urban areas are particularly susceptible to the detrimental effects of air pollution. In response to this environmental challenge, transportation companies are gradually transitioning to electric vehicles with the aim of mitigating pollutant and greenhouse gas emissions. Governments worldwide are also incentivizing the adoption of sustainable energy practices and electric vehicles, and imposing other restrictions with the aim of accelerating the switch from fossil fuel powered transportation options.

The City of Ottawa is funding the development of a portable module that can collect data on varying pollution levels throughout different times of the day within the city. They proposed a community-based data collection program where volunteers could carry around a portable monitoring system during their daily commute and activities, and record the fluctuating pollutant levels around them throughout the day. Their aim is to use this data to study the correlation of the concentration of electric vehicles within a neighborhood with the concentration of pollutants in the area. The device is also intended for use in monitoring and potentially predicting pollution levels in an area over time.

Our prototype draws inspiration from an existing e-bike mounted module, with enhancements tailored to meet the requirements set by the City of Ottawa. Notable adjustments include scaling down of the module to a size conducive for volunteer users to effortlessly carry the device around with minimum obstruction to their day-to-day activities, i.e.; small enough to clip onto a backpack, a bike, or a mobile phone. This design ensures that a wider area can be surveyed for pollutants, as volunteers are not restricted to e-bike users who may or may not follow the same paths for their daily activities. The module's ingress protection has been upgraded to ensure functionality in all weather conditions, for reliable data collection. A power bank feature has also been incorporated in our design to augment functionality and serve as an incentive for volunteers to carry it regularly.

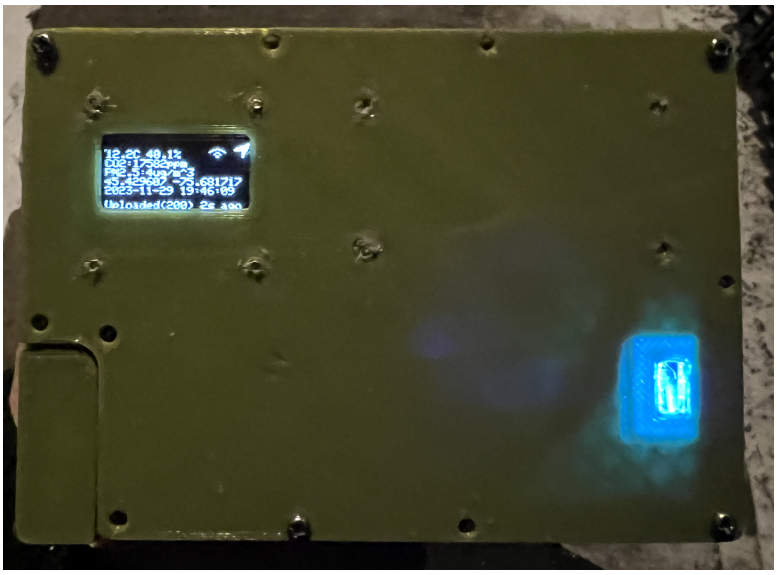


Fig 1: Pollution Monitoring Module

Our product is a microcontroller-based system that uses temperature, humidity, CO2, and particulate matter (PM2.5) sensors to monitor the air quality of the surrounding environment, and sends the collected data over WiFi to our web server to be saved into a database. The product only needs to be switched on by pressing a button on the side of the module and has to be connected to WiFi, for recording of the data. If the user does not want to connect it to WiFi, the module can also serve as a personal air quality monitoring device, where the user can see the readouts on the module's LCD screen, but the collected data will not be stored on the server. It can also be used as a power bank. The LED on the module indicates the battery level. The product also comes with a web application that can be accessed here and can be used to view the latest readings collected, along with a heatmap for CO2 pollution and variations in the pollutant levels over the last day on which data was collected.

We believe our product stands out from other similar devices on the market due to the following factors:

1. **Visibility of readings**
   The system has an LCD screen mounted on the top of the device. This allows the user to read the real-time sensor readouts for the level of pollutants in that area. A window was added to indicate the battery charge percentage of the device as well.

2. **User control**
   The product is designed such that there is very little user engagement. With the single click of a button, the device and components are powered on and run efficiently with real-time readings. The web app also requires minimal user input to display the data visualizations.

3. **Charging capability**
   To incentivize the use of this product the device comes with an inbuilt power bank capability, which allows the user to charge their electronic device while simultaneously using the device to measure pollutant levels.

4. **Form factor**
   The device was designed while keeping target specifications comparable to a typical mobile device such as an iPhone in footprint. The client requested a compact device to improve ease of use and to make it easy for users to carry around when they commute.

5. **Minimalist design**
   The product was designed to showcase very few electrical components and only the necessary components like the LCD screen, power button, and charging ports are visible to the end user.

6. **Accuracy of readings**
   The product was designed to get as accurate readings as possible. This was done by incorporating a fan to ensure consistent airflow over the sensors for better sensor readouts, and through the selection of self-correcting true sensors.

7. **Ease of Use**
   Simple 1-button operation allows for anyone to pick up and immediately use the unit. By implementing a standard VESA hole pattern on the back of the device, it allows for the attachment of a variety of mounts. This allows the module to be attached to a vehicle or backpack to make it easier to carry around.

8. **Ingress protection**
   O-rings have been used between the housing and lid for optimal protection. The use of a waterproof fan, sealant in strategic locations, and O-rings was to protect the internal electronic components from moisture and debris ingress. The use of these components increased ingress protection and the ability to operate in all expected environmental conditions.

3 Getting started

The system is simple to set up, and only needs to be turned on by pressing the button on the side of the module. When the device is ON, it will be indicated by the button lighting up as shown in the image below.

Fig 2: Power button ON

An LED on top of the module indicates the amount of power left in the power bank. There is also an LCD screen that shows the pollutant levels, WiFi, and GPS status, as well as the current date and time. Text at the bottom of the screen also shows the user how long ago the data was uploaded to the server. When connected to WiFi, data is uploaded to the server every 10 seconds. Figure 3 shows the display when connected to WiFi and GPS.

Fig 3: Device connected to Internet and GPS

The module is only able to connect to GPS when there is a clear view of the sky; this means that it cannot connect to GPS while indoors. Figure 4 shows the display when the module is not connected to WiFi or GPS. The data can only be sent to the server while the module is connected to WiFi.

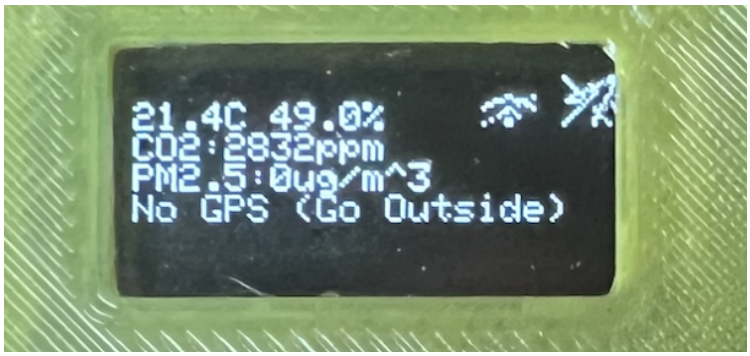


Fig 4: Device not connected to WiFi or GPS

In order to use the device as a power bank, simply plug in the USB charger to the charging port on the side of the device as shown in the picture below.

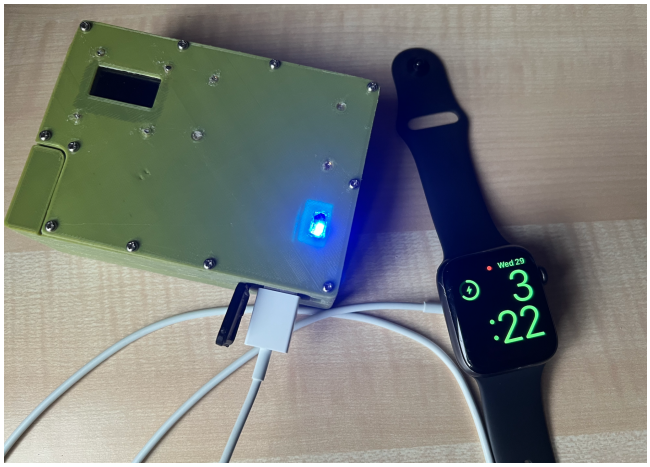


Fig 5: Device charging an Apple watch

To charge the device, connect a USB-type cable to the power outlet port on the side of the device as shown below. The power outlet supports up to 5V/2A of power output. The battery indicator lights will flash while charging.

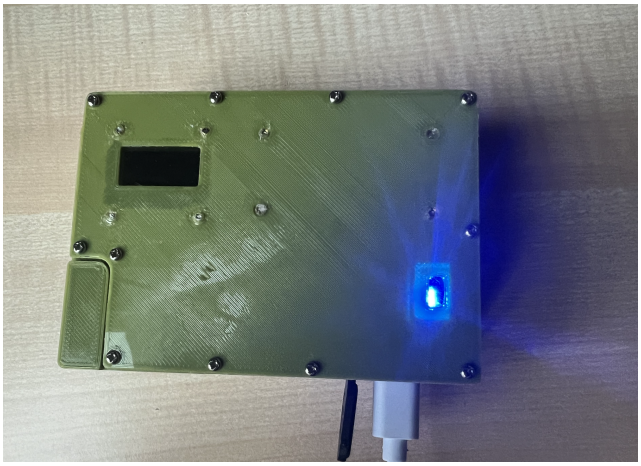


Fig 6: Charging the device

To turn the device off, simply press the power button again.

3.1 System Organization & Navigation

The web application has three main pages - the Homepage, Pollution History, and Heatmap. It can be accessed at the following link:

https://city-pollution-app-a4a59953b054.herokuapp.com

3.1.1 Homepage

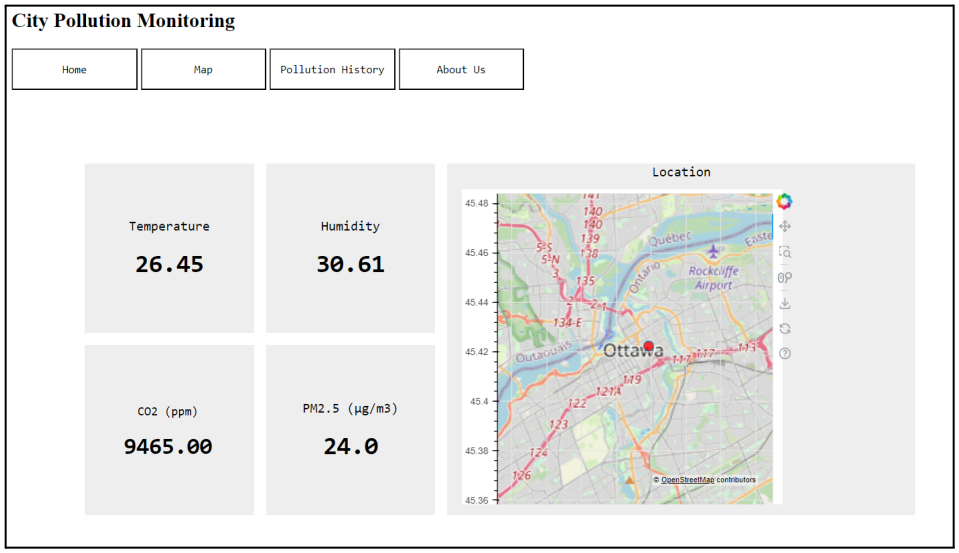


Fig 7: Homepage

The homepage shows the most recent temperature, humidity, PM2.5, and CO2 values. The last pinged location is also indicated by a red dot on the map on the right side of the page. The map can be zoomed into and moved around using the widgets next to it.

3.1.2 Pollution History

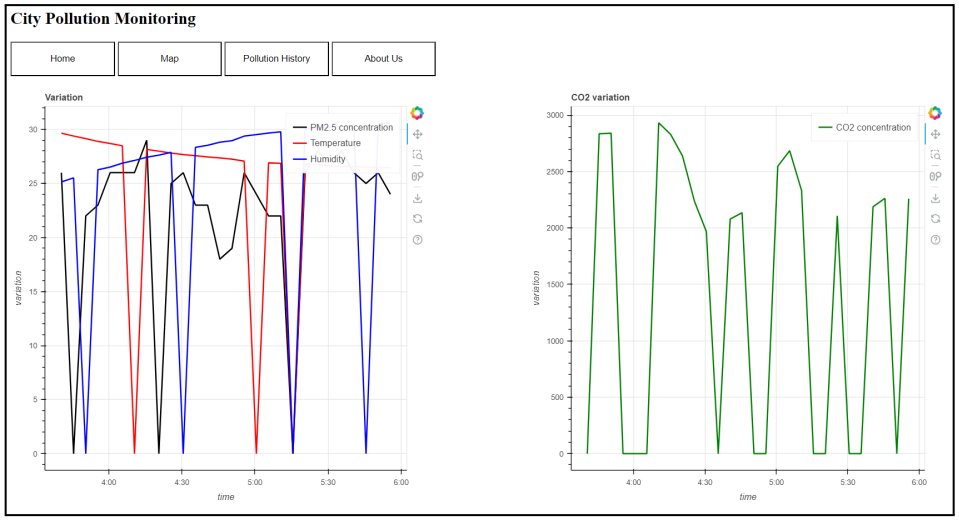


Fig 8: Pollution History

The pollution history page shows two graphs. The first graph displays the variation of temperature, humidity, and PM2.5 concentration with time, as measured on the last day on which data was collected. The second graph displays the variation of CO2 concentration over the same time period.

3.1.3 Heatmap

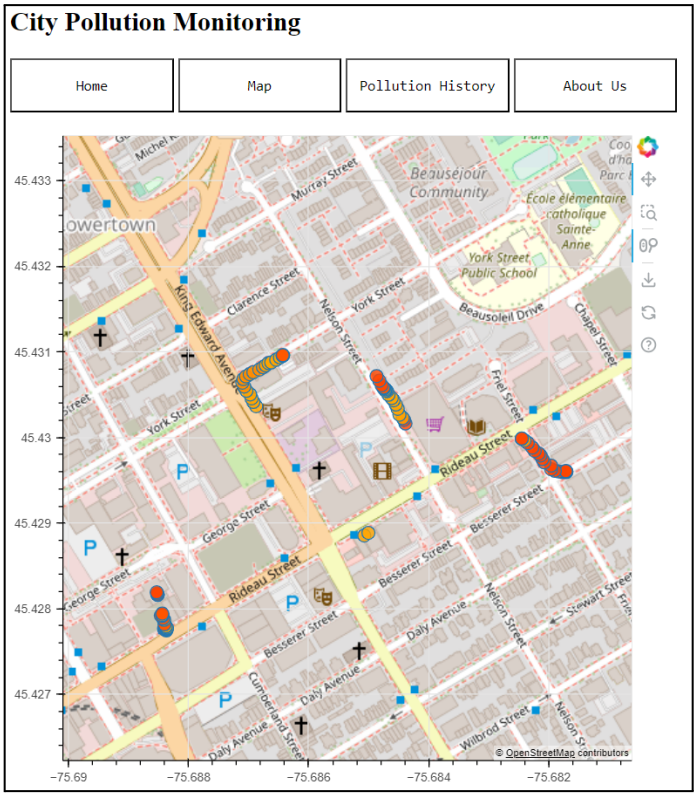


Fig 9: Heatmap

The ‘Map’ tab takes you to a page displaying a heatmap of CO2 concentration as measured over the last day on which the data was collected. The colors on the heatmap indicate the following concentration levels:

- Green - Less than 1000 ppm
- Dark orange - Between 1000 to 2000 ppm
- Orange - Between 2000 to 5000 ppm
- OrangeRed - Between 5000 to 40000 ppm
- Red - Higher than 40000 ppm

4 Troubleshooting & Support

This section lists all the error messages that can be displayed on the module or the web application during device operation.

4.1 Error Messages or Behaviors

- **WiFi not connected/no data sent to the server**: This is displayed as an icon on the top right of the LCD screen on the device. Connect the device to a WiFi network or mobile hotspot.
- **GPS not connected (go outside)**: This error is displayed when the device does not have a clear view of the sky. Pollution data is still collected and sent to the server when this occurs, however, the location data will default to (0,0). This will cause the location on the web page to not be displayed.

4.2 Maintenance

The device is designed to operate automatically and does not require any external maintenance from the user side. Concerning firmware updates for the device, connecting to wifi or a mobile device would allow the web application to automatically upload required firmware updates to the device.

4.3 Support

As creators of the device, we would be happy to provide product support on the following chosen topics. Please send emails with a brief description of the issue experienced or support requested.

- Mechanical Support
  - Max Shui mshui036@uottawa.ca
  - Myron Soares msoar086@uottawa.ca
- Electrical Support
  - Adrian Iannantuono aiann079@uottawa.ca
  - Max Shui mshui036@uottawa.ca
- Software Support
  - Adrian Iannantuono aiann079@uottawa.ca
  - Drishya Suresh dsure069@uottawa.ca
- Administrative Support
  - Tarin Sultana tsult029@uottawa.ca

5 Product Documentation

This section deals with in-depth technical aspects of the product, including design considerations and materials used.

5.1 Mechanical design

The ultimate goal for the mechanical design of our system was to create the most functional, compact, and rugged housing possible to package our selected hardware. Our housings were 3D printed from PLA plastic for economical and fast prototyping. The basic concept of the enclosure revolves around the use of a sealed internal air channel for gathering sensor data. This air channel is entirely isolated from the rest of the internal volume of the housing, ensuring ingress protection from the elements. Only sensor inputs pull air from this channel, which is ventilated by a waterproof fan on one side to generate positive pressure, ensuring consistent airflow regardless of exterior atmospheric conditions. All sensor, screen, button, and wire passthrough openings are sealed, and both the top lid assembly and charging port lid are protected by an O-ring. The development of the overall assembly has been an iterative process, with improvements based on lessons learned from assembling and testing the previous prototype models. This final refinement represents a cumulation of knowledge from our entire course of development, learning which would not have been possible without hands-on experience.

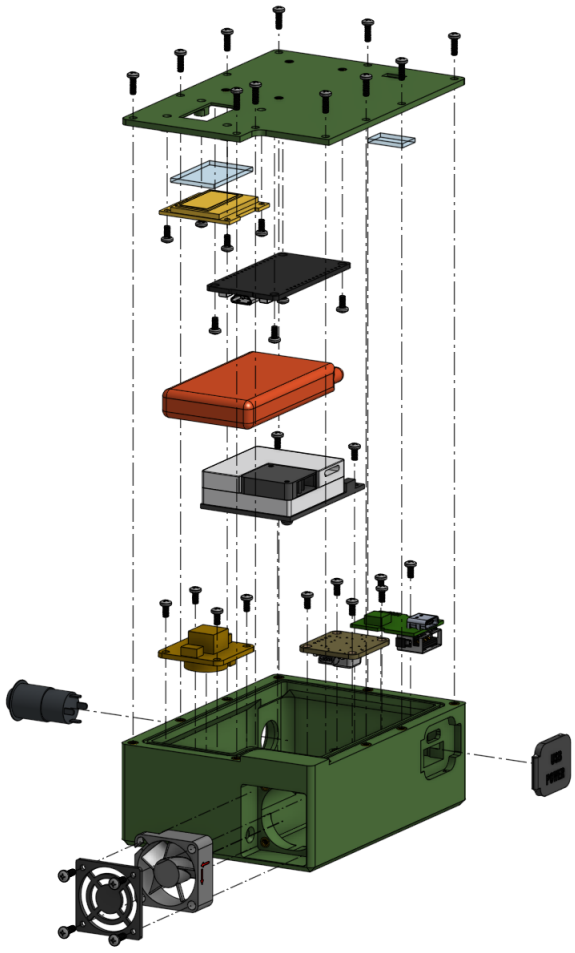


Fig 10: Exploded view of Prototype II

There were many improvements made for ease of assembly. The GPS antenna was initially oriented to face the y-axis while in operation. Mounting it facing the z-axis of orientation improved ease of assembly and demonstrated better connectivity while retaining line of sight for signal acquisition. The location and orientation of the microprocessor was adjusted for better packaging as well as easier USB access for software updates and maintenance. General component tolerances were evolved to allow for easier fitting and more error on 3D-printed parts.

To further improve ingress protection, an acrylic lens was installed in the lid to preclude any issues from sealing the screen module directly to the lid. An additional smaller acrylic lens was implemented to allow viewing of the battery charge indicator on the power board.

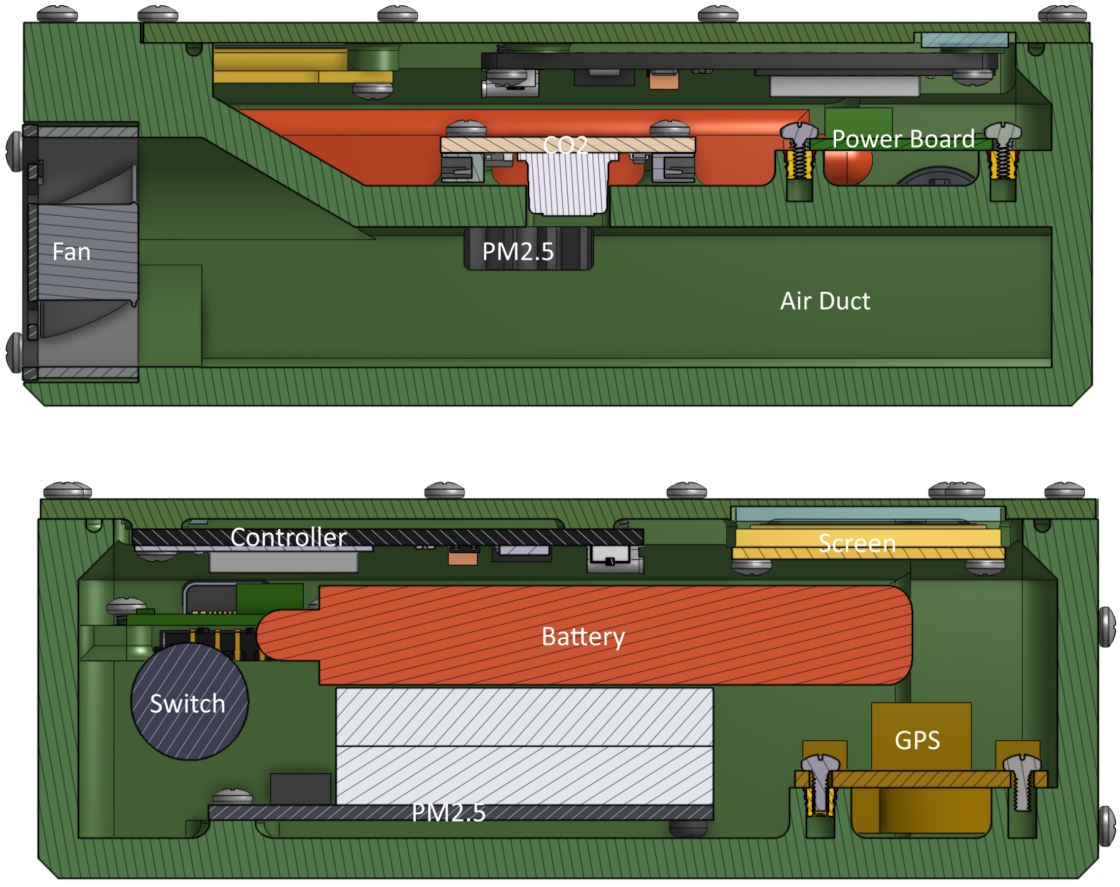


Fig 11: Cross-section views showing component placement

With consideration to user ergonomics, a standard VESA (FDMI-MIS-B) hole pattern was integrated into the housing. Any VESA standard attachment may be mounted for the unit. This includes a variety of off-the-shelf adjustable ball-type mounts, typical for vehicle mounting. We have also designed two original mounting solutions for demonstration. These include a clamp-on mount created for mounting to 1” diameter bicycle handlebars and a clip-on mount for mounting to a backpack strap or standard MOLLE.

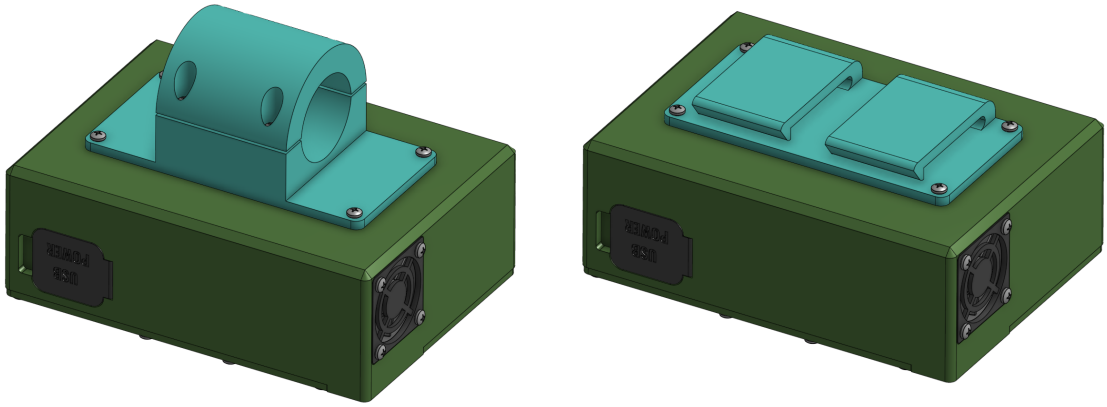


Fig 12: VESA mounting solutions

Overall, these modifications over the course of development have resulted in a minor increase in our product footprint from 4”x2.9”x1.5” to 4.2”x3”x1.5”, which is an 8.6% increase in volume size from the initial prototype. We do not foresee this increase in size having any noticeable impact on the end user's ergonomics and experience with our unit.

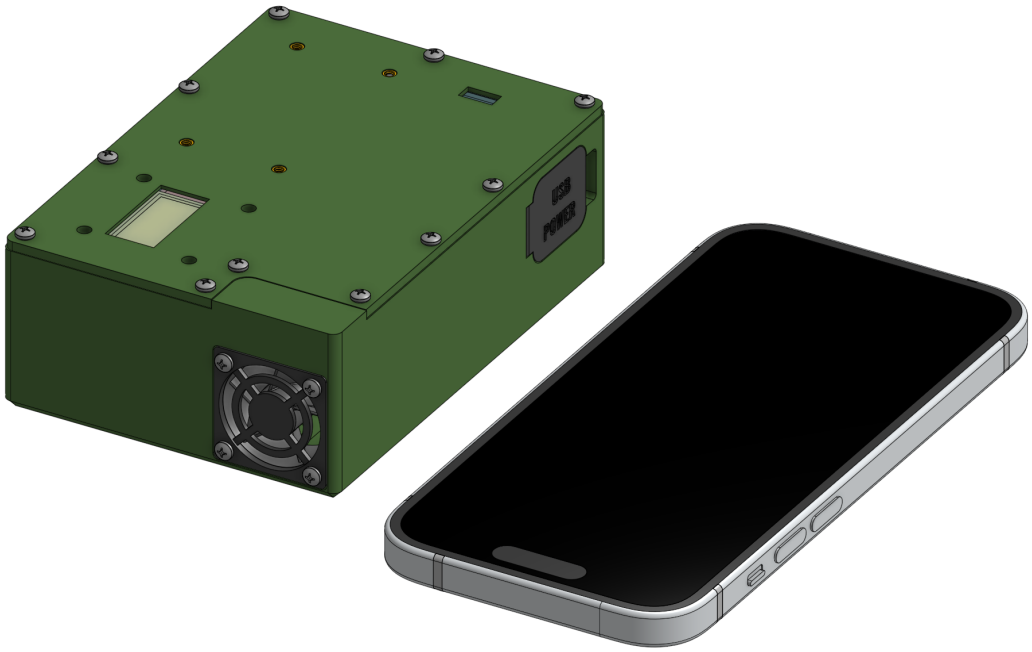


Fig 13: Comparison of Prototype II size with Iphone™ 14

5.2 Electrical design

The device consists of 7 main hardware components. The components are an ESP8266 development board with built-in WiFi, a temperature, humidity, and CO_2 sensor, an OLED screen, a GPS sensor, a PM2.5 sensor, a battery, and a battery management module. The exact models of these components are mentioned in the bill of materials in Section 5.4. These components were all tested on a breadboard in previous prototypes and wired according to their specifications. The components are connected to each other and the Arduino board via an I^2C data bus. This wired communication protocol allows for the connection of multiple devices with shared data and clock lines for communication and data acquisition with the microcontroller. After they were placed on a breadboard and confirmed to be working we transferred the electronics into the mechanical enclosure to ensure correct fit.

The following figures show the schematic used in our device and the final assembly using the schematic.

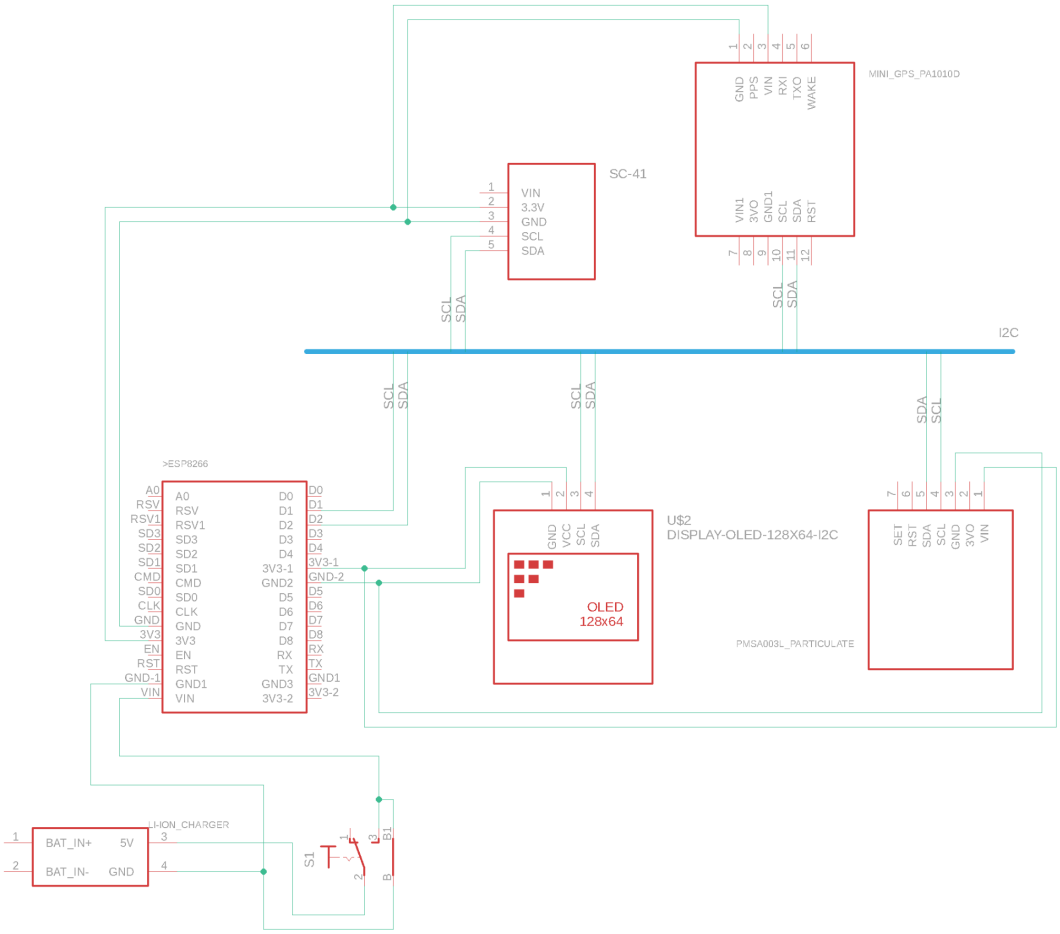


Fig 14: Final schematic

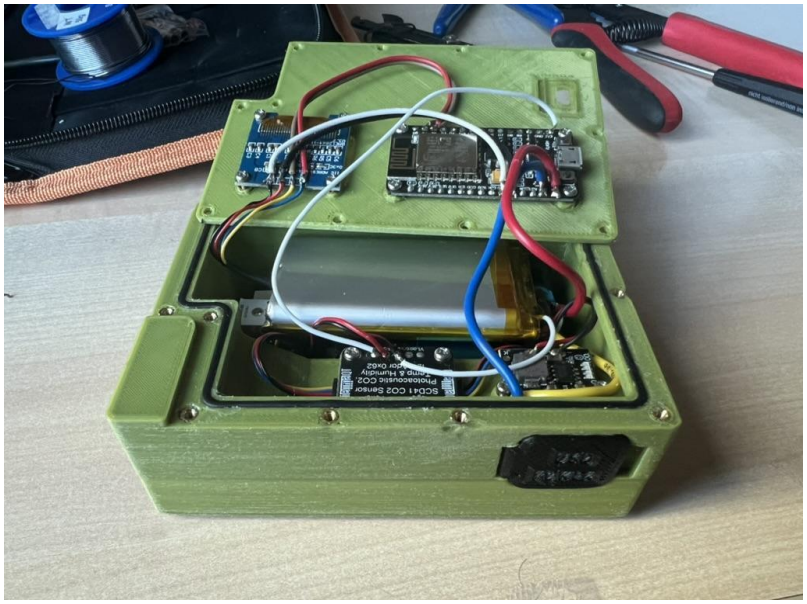


Fig 15: Final prototype assembly

5.3 Software design

5.3.1 Arduino Code

To develop the device we used the Arduino bootloader running on the ESP8266 WiFi development board. This allows us to utilize existing modules for connecting to WiFi and servers as well as collect data from the sensors using the Arduino interface.

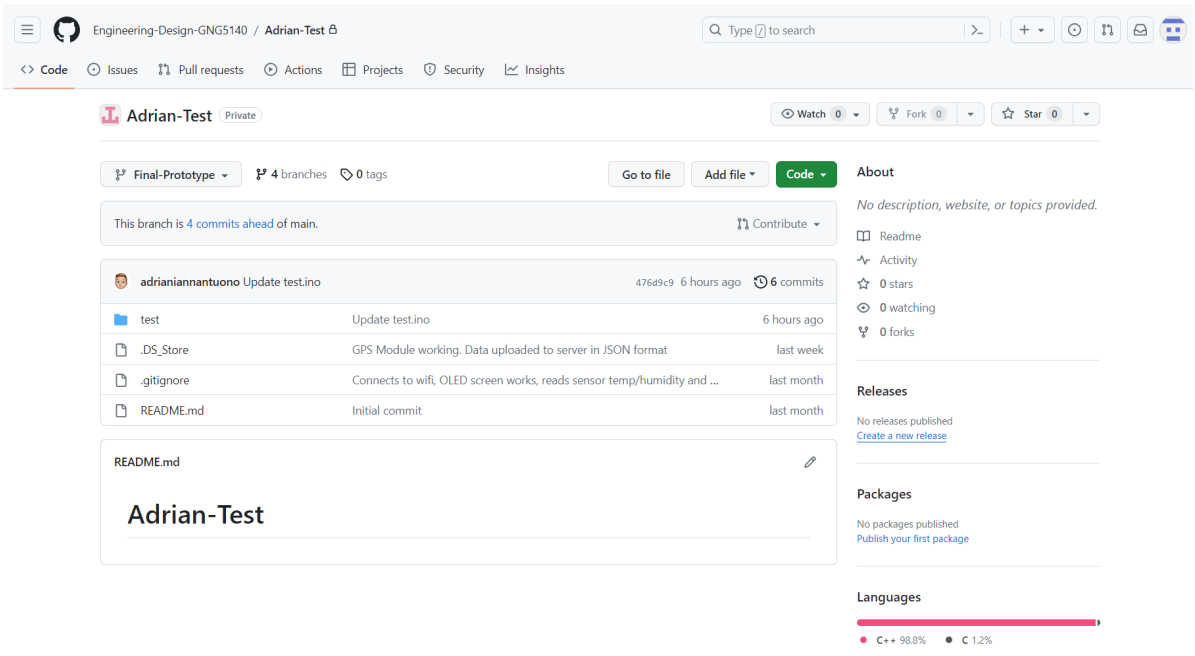


Fig 16: GitHub branch created for final prototype

The code written for the Arduino can be found in the Appendix, under the heading “Arduino code”.

The two constants ‘WIFI_NOT_CONNECTED’ and ‘WIFI_CONNECTED’ are used to display connection symbols on the OLED screen. ‘GPS_NOT_CONNECTED’ and ‘GPS_CONNECTED’ constants have been added to display symbols for the status of the GPS sensor. A separate file ‘secrets.h’ is maintained to store the SSID and password required to connect to the WiFi network. The ‘remote_host’ variable stores the url for our web server, which is pinged every 10 seconds with a POST request containing the collected data in JSON format.

In the setup loop, we first initialize the sensor pins and define the parameters required for the display. Since the sensors read analog values, we read the values through Serial communication. We also check if the display is working and connect to the WiFi network.

Separate functions are written to read the data from the CO2 sensor, the PM2.5 sensor, as well as the GPS module. Finally, the getSensorData() function is used to read and store values from all the sensors in one line of code. The updateDisplay() function initializes the display and configures it to display the connection status for WiFi and GPS sensor, battery percentage, the temperature in Celsius, humidity in percentage, CO_2 concentration (ppm), PM2.5 concentration ($\mu g/m^3$), and last server response time in milliseconds.

The sendData() function has been updated to serialize the collected data - date, time, GPS coordinates, temperature, humidity, CO2, and PM2.5 concentration levels - into a JSON object, which is then sent as a HTTP POST request to the ‘save_data’ endpoint of our web server.

5.3.2 Web Server

Our device is integrated with a web server that is used to save the collected data into a database, as well as display the collected data on a simple web interface. The web server was created using Python’s flask library which is a light web framework which is used for creating simple web applications. The database used is MariaDB, which is an open source database that is generally used in place of MySQL on the Heroku platform.

Figure 17 shows the block diagram for the integration of Arduino with the web server.

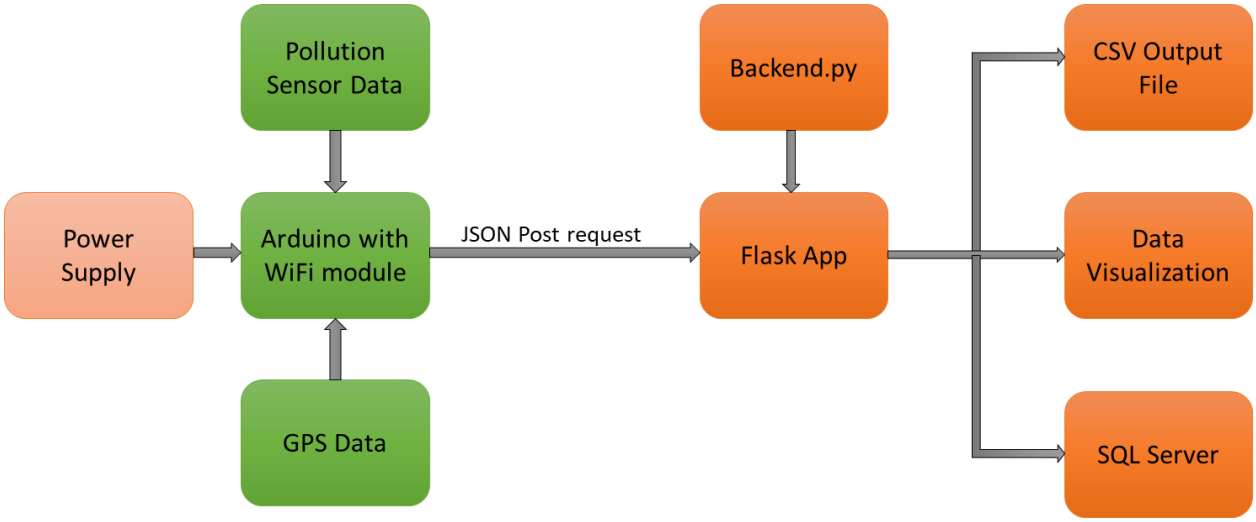


Fig 17: Integration of Arduino with web server

The web server is hosted on Heroku, which is a cloud Platform-as-a-Service (PaaS) that is commonly used to host apps by packaging them in virtual containers called 'Dynos'. We are hosting our app using Heroku's free tier option. Heroku also allows us to set environment variables so that the database connection strings are not exposed via the code.

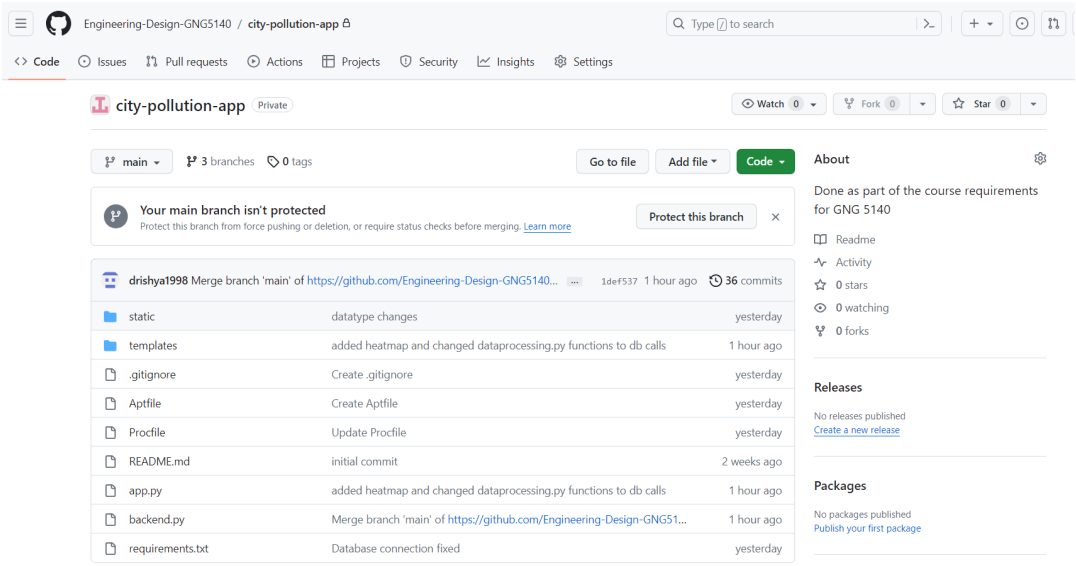


Fig 18: Github repository created for the web server

The code for the application can be found in the appendix under the heading "Code for web application". The database connectivity and database calls are done in the backend.py file. The database connection is initialized using the initialize_conn() function and closed using the close_conn() function. The insert_data() function is used to insert values into the database whenever the Arduino pings the server. This function also saves the collected data into a csv file for further use.

The get_latest() function is used to retrieve the latest value in the database whenever the web application is initialized. These values are then displayed on the home screen as shown in figure 19.

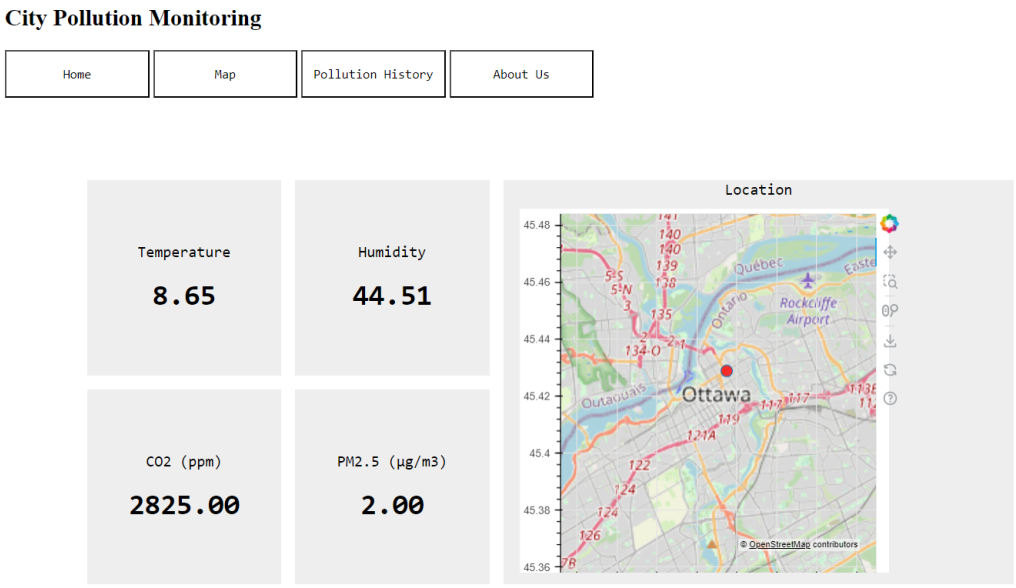


Fig 19: Home screen of web application

The get_heatmap_points() function makes a database call to retrieve the values of longitude, latitude, and CO2 concentration over the last day. This data is then used to display a heatmap on the "Map" tab of the web application. The colors vary from green to orange to red depending on the concentration of CO2 in that location. The following logic was used to assign the colors on the heatmap:

- Green - Less than 1000 ppm
- Dark orange - Between 1000 to 2000 ppm
- Orange - Between 2000 to 5000 ppm
- OrangeRed - Between 5000 to 40000 ppm
- Red - Higher than 40000 ppm

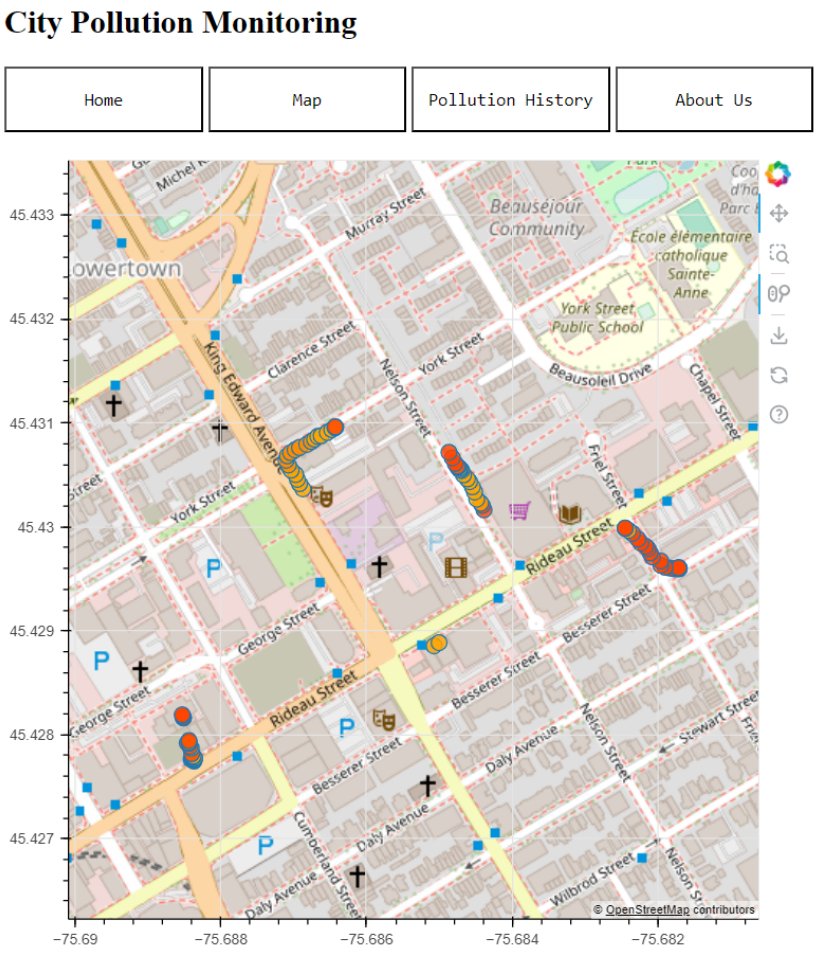


Fig 20: Heatmap showing CO2 concentration across Downtown Ottawa

The get_variation() function is used to retrieve the CO2, PM2.5, temperature, and humidity values over the past day to display it as a graph on the web application on the “Pollution History” tab.

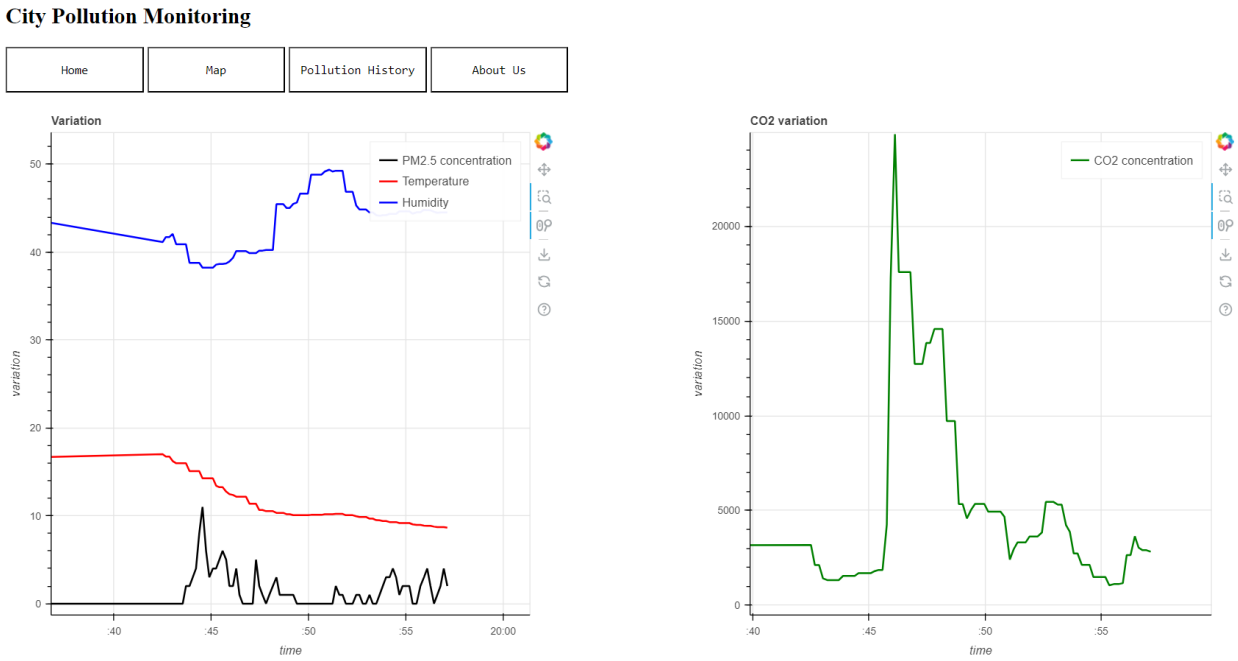


Fig 21: Variation of pollutants, temperature, and humidity

modify_coordinates() is a helper function that is used to format the data received from the Arduino. The GPS module sends the latitude and longitude coordinates by marking them with a N/S/E/W suffix, while the general format for latitude and longitude is by using a +/- prefix.

The code in app.py is used to set up the Flask application. There are two endpoints - one for the web application, and one for the webserver to store the data from Arduino into the database.

The UI for the web application has been kept very simple so that the user needs to only have minimal interaction with the module and application to view or collect data.

5.4 BOM (Bill of Materials)

The bill of materials for all components used in our device is given below:

Item	Description	Qty	Price	Subtotal	URL
Battery, 103665	3.7V 3000mAh LiPo	1	$29.00	$29.00	https://www.amazon.ca/gp/product/B0BP99H5MH
Button	1/2" Mount, Latching, Illuminated, 3A, IP67	1	$8.00	$8.00	https://www.amazon.ca/gp/product/B0BNQ6FNTV
Cable, Hookup Wire	26 gauge, per inch	40	$0.07	$2.80	
Cable, Stemma QT	4-pin JST	3	$1.31	$3.93	https://www.adafruit.com/product/4210
Charging Module	Daoki 5V, 2A, USB-A/C	1	$3.74	$3.74	https://www.amazon.ca/gp/product/B091TPZH13
Display, OLED, 0.96"	GME12864-11	1	$24.15	$24.15	https://www.adafruit.com/product/326
Fan, 25mm	Ball bearing, 2.45 cfm, 5V, 0.15A	1	$6.26	$6.26	https://www.amazon.ca/gp/product/B09KFV9K23
Housing, Base	3D Printed, PLA, per gram	101	$0.03	$3.03	
Housing, Fan Grille	3D Printed, PLA, per gram	1	$0.03	$0.03	
Housing, Lid	3D Printed, PLA, per gram	15	$0.03	$0.45	
Housing, Port Cover	3D Printed, PLA, per gram	1	$0.03	$0.03	
Inserts, Heat Set, M2 x 3mm	Brass	37	$0.10	$3.70	https://www.amazon.ca/gp/product/B07NYYLQBJ
Lens, Battery Indicator	Acrylic, per sq in	0.19	$0.50	$0.10	
Lens, Screen	Acrylic, per sq in	0.84	$0.50	$0.42	
Microcontroller, NodeMCU Amica	With WiFi	1	$7.67	$7.67	https://www.pcboard.ca/nodemcu-v3-ch340
Oring Strip, 1/16"	Buna-N, 70A Durometer, per inch	15	$0.09	$1.31	https://www.amazon.ca/gp/product/B00QVB6NE4
Screw, Pan Head, M2 x 16mm	Pan Head, Stainless	4	$0.05	$0.21	https://www.amazon.ca/gp/product/B07GRMS6WM/
Screw, Pan Head, M2 x 4mm	Pan Head, Stainless	29	$0.05	$1.54	https://www.amazon.ca/gp/product/B07GRMS6WM/
Sensor, CO2	Adafruit SCD-41, True CO2, Temperature, Humidity	1	$68.93	$68.93	https://www.adafruit.com/product/5190
Sensor, GPS	Adafruit PA1010D, GPS/GNSS, I2C and UART	1	$41.33	$41.33	https://www.adafruit.com/product/4415
Sensor, PM2.5	Adafruit PMSA003I, Particulate	1	$62.03	$62.03	https://www.adafruit.com/product/4632
			Total (CAD)	$268.66	

Table 2: Bill of Materials (BOM)

5.4.1 Equipment list

Physical Tools:

- 3D Printers - Ultimaker 2+, FLSUN V400
- Screwdriver
- Needlenose Pliers
- Wire Strippers
- Soldering Iron
- Heat Gun
- Epoxy Adhesive
- Rubber Adhesive

Software Tools:

- OnShape
- Cura
- Arduino IDE
- Visual Studio Code/any python editor
- Heroku platform

5.5 Testing & Validation

5.5.1 Physical Prototype

The final prototype of the pollution monitoring device collects data from the temperature/humidity/CO_2, PM2.5 and GPS sensors and displays that information in real-time on the device's OLED screen. The device includes a 3000 mAh LiPo battery which allows for the device to operate for a minimum of 6 hours without being connected to external power. This allows the device to be extremely portable. The device can also output power through its USB-A port which allows it to charge devices and operate as an external battery bank. Operation of the device is as simple as pushing the power button to turn the device on. The power button lights up to show that the device is running as seen in Figure 22. When the device is off this button does not light up to save power.

Fig 22: Power button ON

Once the device is turned on it will try to acquire a satellite signal from the GPS module as well as connect to WiFi. Once the device connects to WiFi, it will send current sensor readout data every 10s to the external server which stores the data for the front-end application to use. It does this by sending a POST request to an endpoint on the server which then processes the request and stores the data. The device also displays the WiFi and GPS status in the top right corner. The device displays and stores the current time and date once it connects to WiFi so that each measurement can be timestamped. The device also displays the status of the POST request along with the time elapsed from the last time the device sent data to the server. The following endpoint is used on the server:

http://city-pollution-app-a4a59953b054.herokuapp.com/save_data

A 200 response means that the server received the data and stored it in the database. The server and SQL database are running on the Heroku web hosting platform as seen in Figure 23 below.

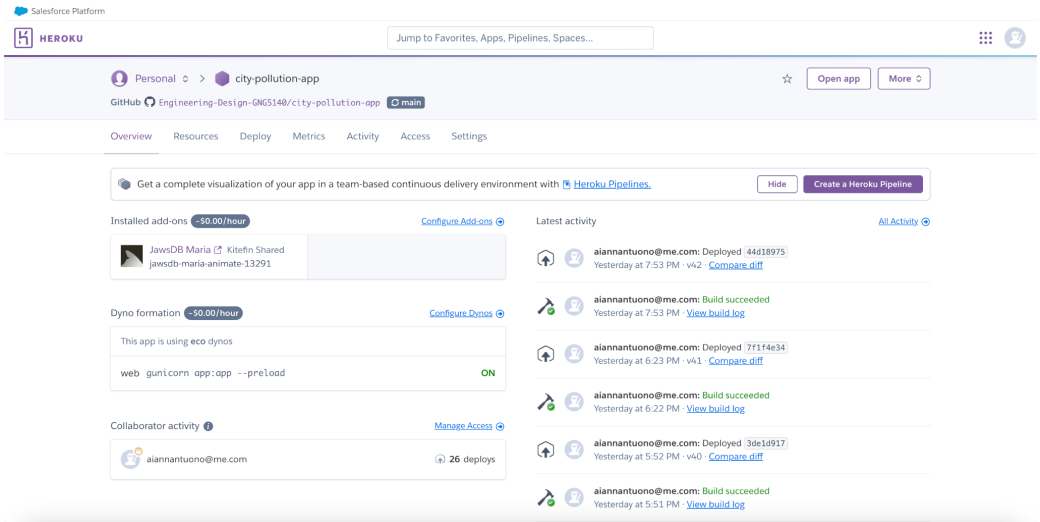


Fig 23: Heroku web app hosting

Fig 24: Device operating outside

Charging the device is done through the USB-C charging port on the side of the device. The device can also provide output power (max 2A of current) via a separate USB-A port pictured in Figure 25 below.

Fig 25: Device I/O

The battery charging module also includes LEDs that show the current battery percentage as well as blink to indicate charging or discharging through its battery bank feature. The device has a battery capacity of 3000 mAh so it can operate as a 3000 mAh external battery bank as seen in Figure 26 below. The device can output power while it is on or off so the user can choose when they would like to collect pollution data or just use it as a battery bank.

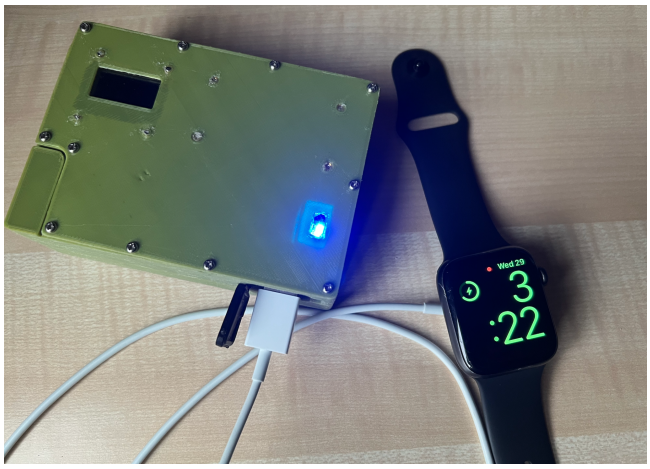


Fig 26: Device charging an Apple Watch

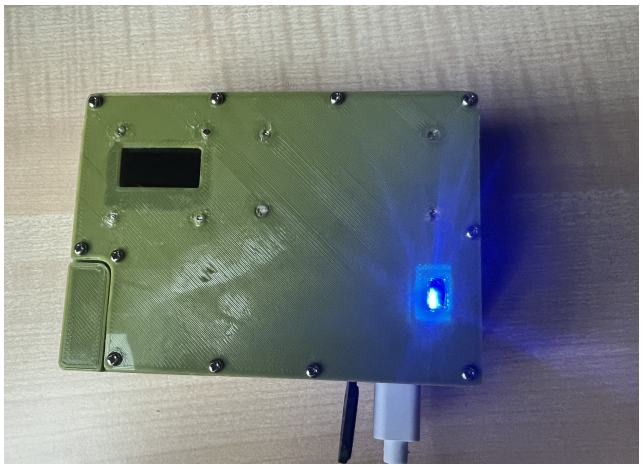


Fig 27: Device internal battery being charged

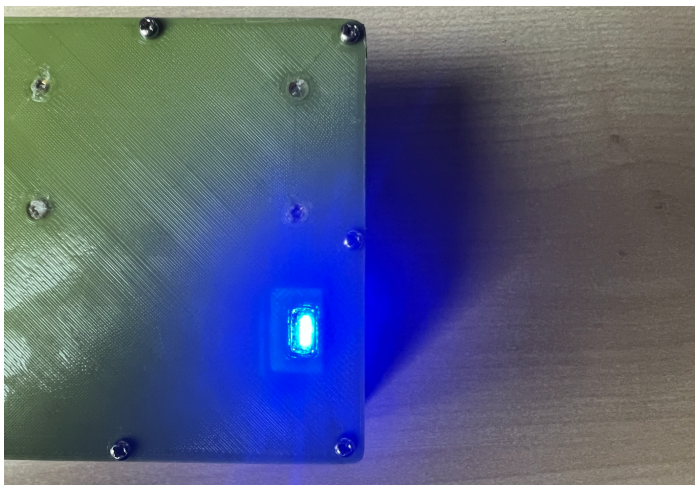


Fig 28: Battery Level/Charge/Discharge Indicator

5.5.2 Testing Process

Testing of our device was done by sending POST requests from the device to the server using JSON format. The server is hosted on Heroku and accepts POST requests at the following endpoint:

http://city-pollution-app-a4a59953b054.herokuapp.com/save_data

Figure 29 shows the POST request returning with a 200 OK code meaning that the request was processed and the data is stored in the SQL server. There is also a message returned indicating successful save of the data.

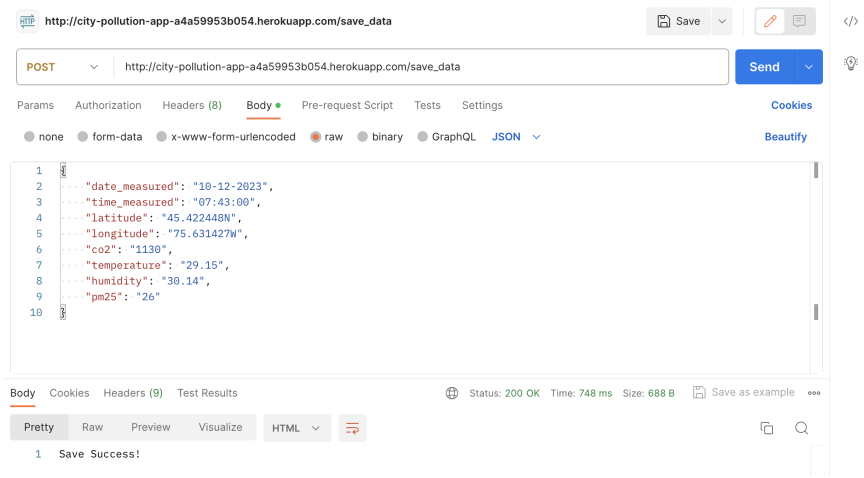
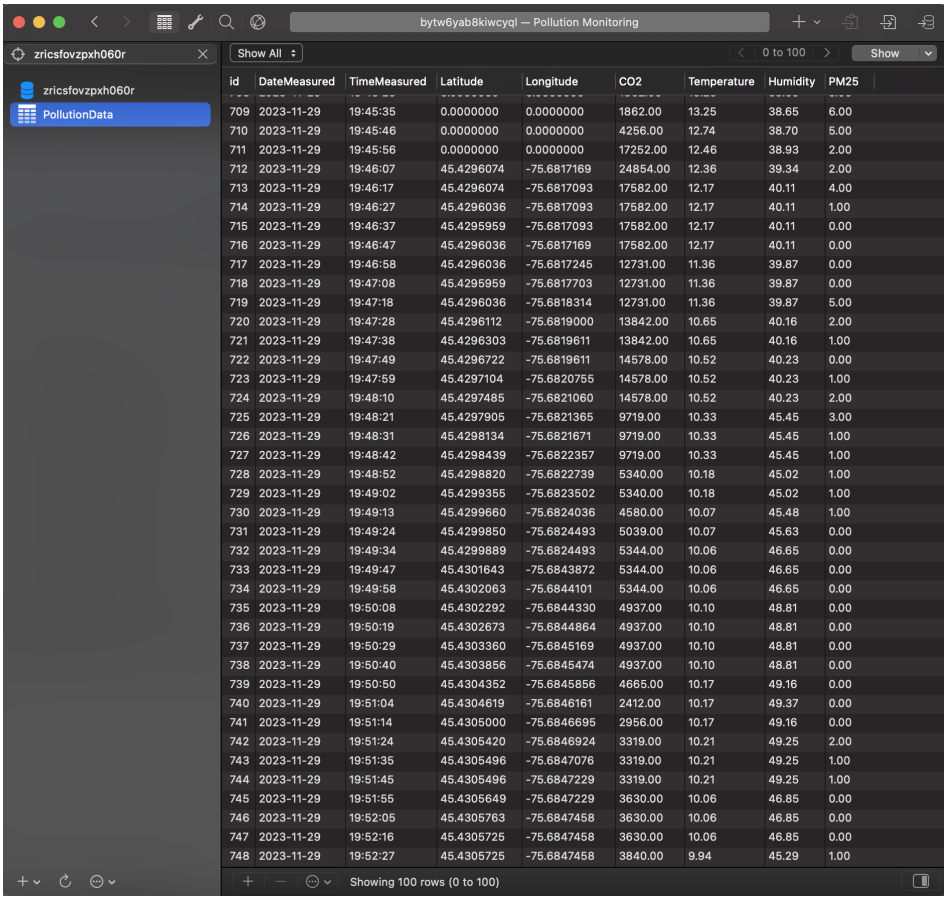


Fig 29: Example POST request sent to server

Figure 30 below shows the data contained in the SQL server after testing the device by walking around the city.



id	DateMeasured	TimeMeasured	Latitude	Longitude	CO2	Temperature	Humidity	PM25
709	2023-11-29	19:45:35	0.0000000	0.0000000	1862.00	13.25	38.65	6.00
710	2023-11-29	19:45:46	0.0000000	0.0000000	4256.00	12.74	38.70	5.00
711	2023-11-29	19:45:56	0.0000000	0.0000000	17252.00	12.46	38.93	2.00
712	2023-11-29	19:46:07	45.4296074	-75.6817169	24854.00	12.36	39.34	2.00
713	2023-11-29	19:46:17	45.4296074	-75.6817093	17582.00	12.17	40.11	4.00
714	2023-11-29	19:46:27	45.4296036	-75.6817093	17582.00	12.17	40.11	1.00
715	2023-11-29	19:46:37	45.4295959	-75.6817093	17582.00	12.17	40.11	0.00
716	2023-11-29	19:46:47	45.4296036	-75.6817169	17582.00	12.17	40.11	0.00
717	2023-11-29	19:46:58	45.4296036	-75.6817245	12731.00	11.36	39.87	0.00
718	2023-11-29	19:47:08	45.4295959	-75.6817703	12731.00	11.36	39.87	0.00
719	2023-11-29	19:47:18	45.4296036	-75.6818314	12731.00	11.36	39.87	5.00
720	2023-11-29	19:47:28	45.4296112	-75.6819000	13842.00	10.65	40.16	2.00
721	2023-11-29	19:47:38	45.4296303	-75.6819611	13842.00	10.65	40.16	1.00
722	2023-11-29	19:47:49	45.4296722	-75.6819611	14578.00	10.52	40.23	0.00
723	2023-11-29	19:47:59	45.4297104	-75.6820755	14578.00	10.52	40.23	1.00
724	2023-11-29	19:48:10	45.4297485	-75.6821060	14578.00	10.52	40.23	2.00
725	2023-11-29	19:48:21	45.4297905	-75.6821365	9719.00	10.33	45.45	3.00
726	2023-11-29	19:48:31	45.4298134	-75.6821671	9719.00	10.33	45.45	1.00
727	2023-11-29	19:48:42	45.4298439	-75.6822357	9719.00	10.33	45.45	1.00
728	2023-11-29	19:48:52	45.4298820	-75.6822739	5340.00	10.18	45.02	1.00
729	2023-11-29	19:49:02	45.4299355	-75.6823502	5340.00	10.18	45.02	1.00
730	2023-11-29	19:49:13	45.4299660	-75.6824036	4580.00	10.07	45.48	1.00
731	2023-11-29	19:49:24	45.4299850	-75.6824493	5039.00	10.07	45.63	0.00
732	2023-11-29	19:49:34	45.4299889	-75.6824493	5344.00	10.06	46.65	0.00
733	2023-11-29	19:49:47	45.4301643	-75.6843872	5344.00	10.06	46.65	0.00
734	2023-11-29	19:49:58	45.4302063	-75.6844101	5344.00	10.06	46.65	0.00
735	2023-11-29	19:50:08	45.4302292	-75.6844330	4937.00	10.10	48.81	0.00
736	2023-11-29	19:50:19	45.4302673	-75.6844864	4937.00	10.10	48.81	0.00
737	2023-11-29	19:50:29	45.4303360	-75.6845169	4937.00	10.10	48.81	0.00
738	2023-11-29	19:50:40	45.4303856	-75.6845474	4937.00	10.10	48.81	0.00
739	2023-11-29	19:50:50	45.4304352	-75.6845856	4665.00	10.17	49.16	0.00
740	2023-11-29	19:51:04	45.4304619	-75.6846161	2412.00	10.17	49.37	0.00
741	2023-11-29	19:51:14	45.4305000	-75.6846695	2956.00	10.17	49.16	0.00
742	2023-11-29	19:51:24	45.4305420	-75.6846924	3319.00	10.21	49.25	2.00
743	2023-11-29	19:51:35	45.4305496	-75.6847076	3319.00	10.21	49.25	1.00
744	2023-11-29	19:51:45	45.4305496	-75.6847229	3319.00	10.21	49.25	1.00
745	2023-11-29	19:51:55	45.4305649	-75.6847229	3630.00	10.06	46.85	0.00
746	2023-11-29	19:52:05	45.4305763	-75.6847458	3630.00	10.06	46.85	0.00
747	2023-11-29	19:52:16	45.4305725	-75.6847458	3630.00	10.06	46.85	0.00
748	2023-11-29	19:52:27	45.4305725	-75.6847458	3840.00	9.94	45.29	1.00

Fig 30: SQL server containing data

The following images show the device operating under different test scenarios.

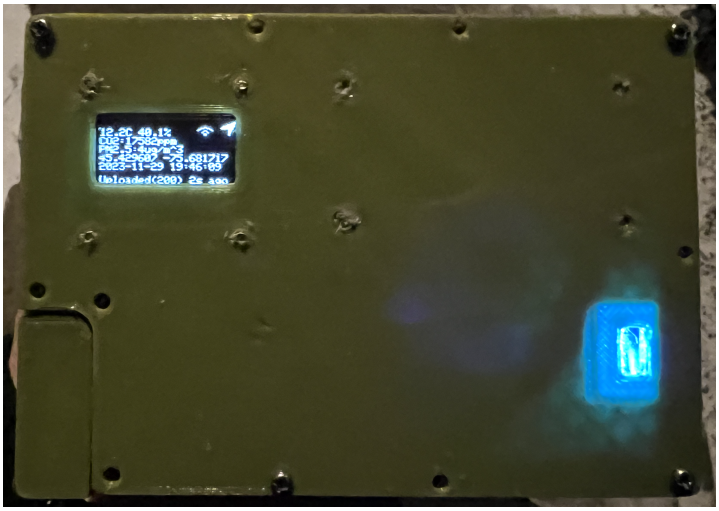


Fig 31: Device Connected to Internet and GPS

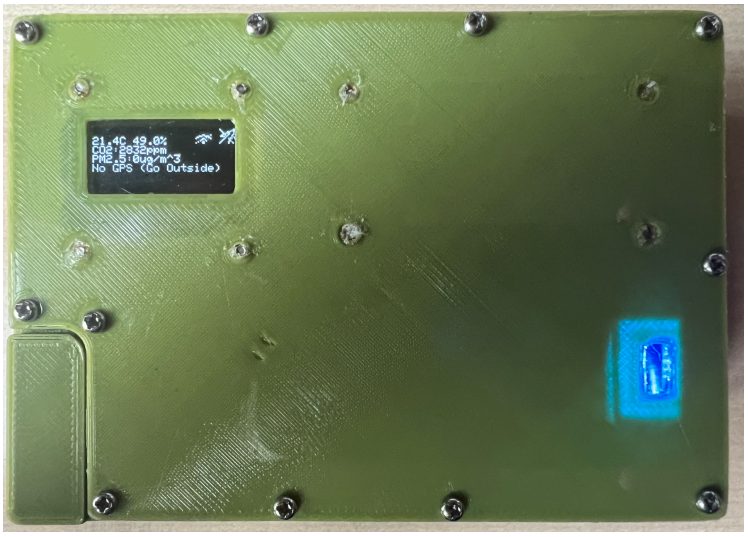


Fig 32: Device Not Connected to Internet and GPS

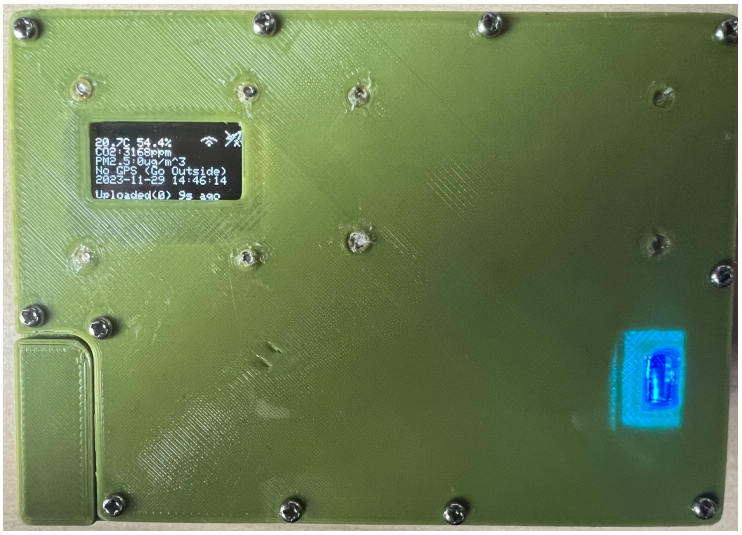


Fig 33: Device connected to the Internet but not GPS

In addition to these tests, we also performed an ingress protection test by holding the device under a running tap. It was observed that the device was fully functional despite being underwater.

Once we completed the electrical connection we placed a multimeter in series with the battery module to see the amount of current that the device draws when it is on. From our tests, it can be seen that the device uses around 0.5A of power but that number fluctuates a lot due to the amount of sensors and the fan connected to the device. Using 0.5A as a worst-case current draw and our battery of 3000 mAh, it can be calculated that the device will last a minimum of 6 hours collecting pollution data with no external power.

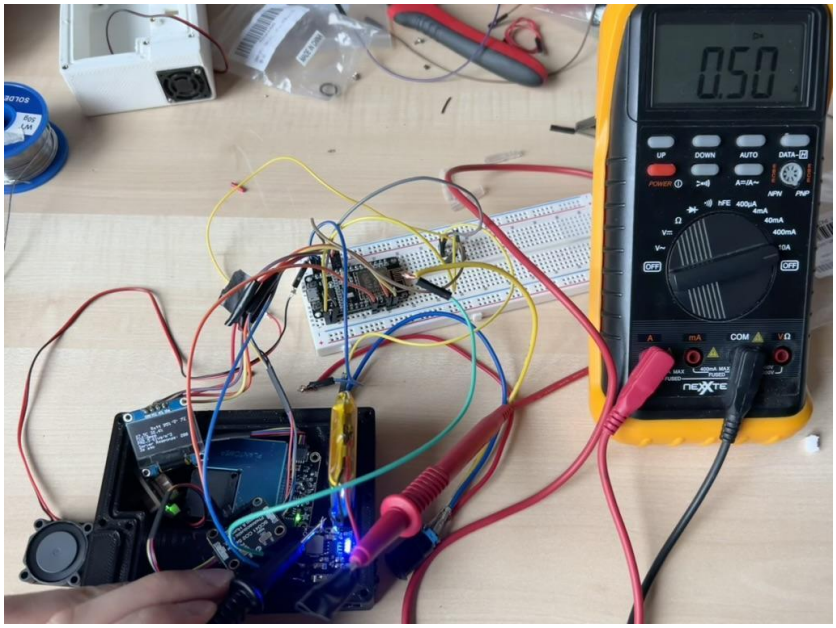


Fig 34: Device power consumption test

The web application was tested by sending POST requests using the Postman application and checking the response, as shown previously in Figure 29. It was also observed that the pollutant and location values on the home screen are updated every 10 seconds with the latest values. This refresh can also be observed by viewing the application logs on Heroku as seen in Figure 35.

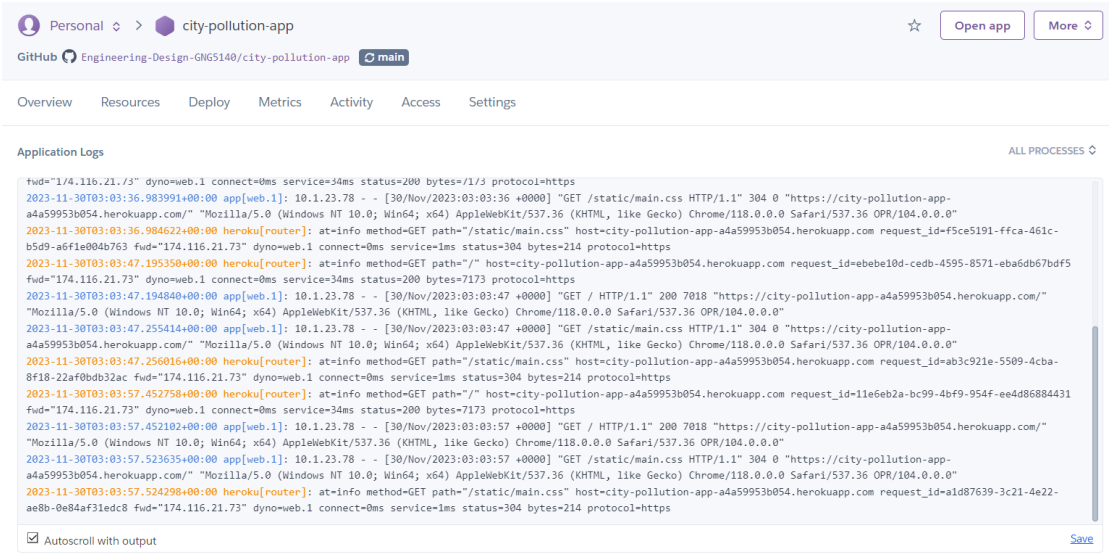


Fig 35: Application logs on Heroku

5.5.3 Target Specs vs Actual Specs

The following table shows the list of target specifications vs actual measured specifications of our prototype along with test values and comments.

Criteria	Target Specification	Actual Specifications	Tested Value and Comments
1. Size	3.5'' x 2'' x 1.25''	4.2" x 3.0" x 1.5"	Final dimensions after all adjustments.
2. Weight	100-120g	140g	Final weight with hardware.
3. Power Source	5v Micro USB	3.7V 3000 mAh Li-ion, 5V USB Type-C	Determined through necessary capacity and form factor.
4. Sensors	PM2.5, Nitrous Oxide, Ozone, Temperature, Humidity	PM 2.5 Air Quality, CO2, Temperature, and Humidity	Sensors have been tested and verified to be working.
5. Sensor accuracy	Up to 95%	Approx. 96.5%	Approximate average efficiency of sensors based on specifications.

Table 3: Results Comparison

6 Conclusions and Recommendations for Future Work

In the course of this project, we have successfully achieved key milestones within the set timelines and fulfilled the requirements set by the client. We have verified all intended functionality; namely the Arduino, OLED screen, pollutant sensors, GPS sensor, as well as the WiFi connectivity and web application. The device was designed to minimize form factor making it as compact in size as possible without any compromises. This was done by the sourcing of miniature components and an efficient packaging design which saved space and allowed for a great reduction in size.

While we are happy to have met all the development and design goals, there are many modifications and improvements that could be made to increase the functionality and usability of the module in future iterations. Some of our suggestions are listed below:

1. **Addition of other pollutant sensors**: Due to budgetary constraints, we restricted the pollutant sensors used to just PM2.5 and CO2 sensors. Future iterations of the module can be integrated with more pollutant sensors such as nitrous oxide, ozone, VOC, etc.

2. **Reduction of module size**: When we consider functionality, the device components could be revised by consolidating the sensor modules, the GPS, and the microprocessor into a single custom PCB. This would improve functionality in several aspects. It would reduce wiring between the various components and improve connectivity and manufacturability. The addition of a custom PCB would not only optimize space in the product but also improve the product's performance and reliability. Using a surface-mounted microcontroller with custom embedded C code can lower power consumption as well.

3. **Addition of features on web application**: Currently the web application only shows variation of pollutant levels over the past day, and a heatmap of CO2 levels over the past day. More customization options can be added where the user can choose the date and time over which the variation and heatmap can be observed, as well as adding a heatmap option for PM2.5 concentrations as well.

4. **Sustainable Manufacture**: The prototype design is currently being manufactured using an additive manufacturing technique. The process is FDM 3D printing and utilizes PLA material. This manufacturing technique is ideal for producing small runs of complex designs. It is a fast process with great time efficiency for rapid prototyping along with being easily customizable. However, it is less desirable for mass manufacturing. The housing could be redesigned to utilize injection molding. In this manufacturing technique, molten plastic is injected into a mold cavity which then sets and is ejected to deliver the complete part. This manufacturing technique allows for a higher production rate at quantity with a greatly reduced cost with high accuracy.

7 Bibliography

1. https://en.wikipedia.org/wiki/Heroku
2. https://elements.heroku.com/addons/jawsdb-maria
3. https://www.digitalocean.com/community/tutorials/how-to-create-your-first-web-application-using-flask-and-python-3

APPENDICES

8 APPENDIX I: Design Files

This project was redesigned from two previously designed projects as attached in below MakerRepo files-

1. https://makerepo.com/MohitShah/1649.creating-a-pollution-measuring-sensor-network-for-ebikes-with-realtime-data-collection-and-data-visualization
2. https://makerepo.com/Daniyal/1667.personal-tracking-ecosystem-group-b

The below link directs you to our project details and design files-

https://makerepo.com/myronsoares/1922.city-pollution-monitoring-gng-5140

9 APPENDIX II: Other Appendices

Arduino Code

```
#include "SparkFun_SCD4x_Arduino_Library.h"
#include <Adafruit_GPS.h>
#include <Wire.h>
#include <ArduinoJson.h>
#include <NTPClient.h>
#include <WiFiUdp.h>

#include "secrets.h"

const unsigned char WIFI_NOT_CONNECTED [] PROGMEM = {
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000001, B10000010,
  B00001110, B01111000,
  B00110000, B00101100,
  B01000001, B10000010,
  B00000011, B01000000,
  B00000100, B00100000,
  B00010001, B10000000,
  B01000001, B10000000 };

const unsigned char WIFI_CONNECTED [] PROGMEM = {
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000000, B00000000,
  B00000001, B10000000,
  B00001110, B01110000,
  B00110000, B00001100,
  B01000001, B10000010,
  B00000010, B01000000,
  B00000100, B00100000,
  B00000001, B10000000,
  B00000001, B10000000 };
```

```
53  const unsigned char GPS_NOT_CONNECTED [] PROGMEM = {
54    B00000000, B00000000,
55    B00000000, B00000000,
56    B00000000, B00000000,
57    B10000000, B00001100,
58    B01000000, B00011100,
59    B00100000, B01100100,
60    B00010011, B11001000,
61    B00011100, B10001000,
62    B00100101, B00010000,
63    B01111010, B00010000,
64    B00000100, B00100000,
65    B00001000, B10100000,
66    B00010000, B01000000,
67    B00100000, B01100000,
68    B01000000, B10010000,
69    B10000000, B10001000 };
70
71  const unsigned char GPS_CONNECTED [] PROGMEM = {
72    B00000000, B00000000,
73    B00000000, B00000000,
74    B00000000, B00000000,
75    B00000000, B00000100,
76    B00000000, B00011100,
77    B00000000, B01111100,
78    B00000011, B11111000,
79    B00011111, B11111000,
80    B00111111, B11110000,
81    B01111111, B11110000,
82    B00000000, B11100000,
83    B00000000, B11100000,
84    B00000000, B11000000,
85    B00000000, B11000000,
86    B00000000, B10000000,
87    B00000000, B10000000 };
88
89  #define SCREEN_WIDTH 128 // OLED display width, in pixels
90  #define SCREEN_HEIGHT 64 // OLED display height, in pixels
```

```
// Declaration for SSD1306 display connected using I2C
#define OLED_RESET     -1 // Reset pin
#define SCREEN_ADDRESS 0x3C
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

Adafruit_GPS GPS(&Wire);

Adafruit_PM25AQI aqi = Adafruit_PM25AQI();

SCD4x co2TempHumSensor;

// Network credentials
const char* ssid = SECRET_SSID;
const char* password = SECRET_PASS;

String remote_host = "http://city-pollution-app-a4a59953b054.herokuapp.com/save_data";

unsigned long previousMillis = 0;
unsigned long interval = 3000;

unsigned long lastTime = 0;
unsigned long lastPrintTime = 0;
unsigned long lastSaveTime = 0;

int httpResponseCode;

// Measurements
String date_measured = "0000-00-00"; // YEAR[00]MONTH[00]DAY[00]
String time_measured = "00:00:00"; // HOUR[00]MINUTE[00]SECOND[00]
String latitude = "00.000000";
String longitude = "00.000000";
float co2 = 0;
float temperature = 0;
float humidity = 0;
float pm25 = 0;

const long utcOffsetInSeconds = -18000;
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, "pool.ntp.org", utcOffsetInSeconds);

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

```
136       // initialize the OLED object
137       if (!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
138         //Serial.println(F("SSD1306 allocation failed"));
139         for (;;); // Don't proceed, loop forever
140       }
141
142       display.clearDisplay();
143       display.setTextSize(2);
144       display.setTextColor(WHITE);
145       display.println("Pollution");
146       display.println("Monitoring");
147       display.println("Device");
148       display.setTextSize(1);
149       display.println("Initializing Sensors");
150       display.display();
151
152       //Serial.println();
153       //Serial.print("Connecting to ");
154       //Serial.println(ssid);
155       WiFi.mode(WIFI_STA);
156       //Serial.printf("HERE: %d\n", WiFi.localIP());
157       if (!WiFi.localIP()) {
158         WiFi.begin(ssid, password);
159       }
160
161       if (co2TempHumSensor.begin() == false) {
162         //Serial.println(F("CO2 Sensor not detected. Please check wiring. Freezing..."));
163         while (true);
164       }
165
166       if (!aqi.begin_I2C()) {
167         //Serial.println("Could not find PM 2.5 sensor! Freezing...");
168         while (true);
169       }
170
171       GPS.begin(0x10);  // The I2C address to use is 0x10
172       GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);
173       GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ); // 1 Hz update rate
174
175       timeClient.begin();
176
177       delay(3000);
178     }
```

```
180  void getCo2TempHumData() {
181    if (co2TempHumSensor.readMeasurement()) // readMeasurement will return true when fresh data is available
182    {
183      //Serial.println();
184
185      //Serial.print(F("CO2(ppm):"));
186      if (co2TempHumSensor.getCO2() == 0)  {
187        // invalid sample
188        return;
189      }
190      co2 = co2TempHumSensor.getCO2();
191      //Serial.print(co2);
192
193      //Serial.print(F("\tTemperature(C):"));
194      temperature = co2TempHumSensor.getTemperature();
195      //Serial.print(temperature, 1);
196
197      //Serial.print(F("\tHumidity(%RH):"));
198      humidity = co2TempHumSensor.getHumidity();
199      //Serial.print(humidity, 1);
200
201      //Serial.println();
202    } else {
203      //Serial.print(".");
204    }
205  }
206
207  void getAQIData() {
208      PM25_AQI_Data data;
209
210      if (!aqi.read(&data)) {
211        return;
212      }
213      /*Serial.print(F("\t\tPM 2.5: "));
214      Serial.print(data.pm25_standard);
215      Serial.println("");*/
216      pm25 = data.pm25_standard;
```

```
240   void getGPSData() {
241      // read data from the GPS in the 'main loop'
242     char c = GPS.read();
243     if (GPS.newNMEAreceived()) {
244         if (!GPS.parse(GPS.lastNMEA())) // this also sets the newNMEAreceived() flag to false
245           return; // we can fail to parse a sentence in which case we should just wait for another
246       }
247
248     if (GPS.fix) {
249
250       } else {
251         latitude = "00.000000";
252         longitude = "00.000000";
253         //Serial.print("No Satelites");
254       }
255   }
256
257   void getTimeAndDate() {
258     time_t epochTime = timeClient.getEpochTime();
259     struct tm *ptm = gmtime ((time_t *)&epochTime);
260     int currentMonth = ptm->tm_mon+1;
261     int monthDay = ptm->tm_mday;
262     int currentYear = ptm->tm_year+1900;
263     date_measured = String(currentYear) + "-" + String(currentMonth) + "-" + String(monthDay);
264     time_measured = timeClient.getFormattedTime();
265   }
266
267   void getAllSensorData() {
268     getCo2TempHumData();
269     getAQIData();
270     getGPSData();
271   }
272
273   void updateDisplay() {
274     // Clear the buffer.
275     display.clearDisplay();
```

```
294       // Display Text
295       display.setTextSize(1);
296       display.setTextColor(WHITE);
297       if (!GPS.fix) {
298         display.drawBitmap(112, 0, GPS_NOT_CONNECTED, 16, 16, WHITE);
299       } else {
300         display.drawBitmap(112, 0, GPS_CONNECTED, 16, 16, WHITE);
301       }
302       switch (WiFi.status()) {
303         case WL_CONNECTED:
304           display.drawBitmap(90, 0, WIFI_CONNECTED, 16, 16, WHITE);
305           display.setCursor(0, 55);
306           display.print("Uploaded(");
307           display.print(httpResponseCode);
308           display.print(") ");
309           display.print((millis() - lastTime)/1000, 1);
310           display.print("s ago");
311           /*display.printf("Signal Strength (RSSI): %d\n", WiFi.RSSI());*/
312           break;
313         default:
314           display.drawBitmap(90, 0, WIFI_NOT_CONNECTED, 16, 16, WHITE);
315           break;
316       }
317
318       display.setCursor(0, 10);
319       display.print(temperature, 1);
320       display.print("C ");
321       display.print(humidity, 1);
322       display.print("%");
323       display.println();
324       display.print("CO2:");
325       display.print(co2, 0);
326       display.print("ppm");
327       display.println("");
328       display.print("PM2.5:");
329       display.print(pm25, 0);
330       display.print("ug/m^3");
331       display.println("");
332       if (!GPS.fix) {
333         display.print("No GPS (Go Outside)");
334       } else {
```

```
    } else {
      display.print(String(GPS.latitude/100, 6) + GPS.lat);
      display.print(" ");
      display.print(String(GPS.longitude/100, 6) + GPS.lon);
    }
    if (WiFi.status() == WL_CONNECTED) {
      display.println("");
      display.print(date_measured);
      display.print(" ");
      display.print(time_measured);
    }


    display.println("");


    display.display();
  }

  void sendData() {
    // Send an HTTP GET request
    if ((millis() - lastTime) > 10000) {
      //Check WiFi connection status
      if(WiFi.status()== WL_CONNECTED){
        WiFiClient client;
        HTTPClient http;

        String serverPath = remote_host + "";

        http.begin(client, serverPath.c_str());

        // Send HTTP POST request
        http.addHeader("Content-Type", "application/json");

        String s;

        // Serialize the JSON data as a string
        DynamicJsonDocument doc(1024);
```

```
372           doc["date_measured"] = date_measured;
373           doc["time_measured"] = time_measured;
374           doc["latitude"] = (String(GPS.latitude/100, 6) + GPS.lat);
375           doc["longitude"] = String(GPS.longitude/100, 6) + GPS.lon;
376           doc["co2"] = String(co2);
377           doc["temperature"] = String(temperature, 2);
378           doc["humidity"] = String(humidity, 2);
379           doc["pm25"] = String(pm25);
380           serializeJson(doc, s);
381
382
383           // Send JSON data as string in a POST request
384           httpResponseCode = http.POST(s);
385
386           if (httpResponseCode>0) {
387             Serial.print("HTTP Response code: ");
388             Serial.println(httpResponseCode);
389             /*String payload = http.getString();
390             Serial.println(payload);*/
391           }
392           else {
393             //Serial.print("Error code: ");
394             //Serial.println(httpResponseCode);
395           }
396           http.end();
397         }
398         else {
399           //Serial.println("WiFi Disconnected");
400         }
401         lastTime = millis();
402       }
403     }
404
405   void serialPrintData() {
406       //Serial.printf("Connection status: %d\n", WiFi.status());
407       if ((millis() - lastPrintTime) > 5000) {
408         Serial.print(date_measured);
409         Serial.print(",");
410         Serial.print(time_measured);
411         Serial.print(",");
412         Serial.print(latitude);
413         Serial.print(",");
414         Serial.print(longitude);
```

```
414         Serial.print(longitude);
415         Serial.print(",");
416         Serial.print(co2);
417         Serial.print(",");
418         Serial.print(temperature);
419         Serial.print(",");
420         Serial.print(humidity);
421         Serial.print(",");
422         Serial.print(pm25);
423         Serial.println("");
424         lastPrintTime = millis();
425       }
426     }
427
428   void loop() {
429         timeClient.update();
430         getAllSensorData();
431         getTimeAndDate();
432         sendData();
433         updateDisplay();
434         serialPrintData();
435     }
```

Web Application Code

1. backend.py

```python
import mariadb
import csv
import os

def initialize_conn():
        #Establish the connection
        try:
                conn = mariadb.connect(
                        user=os.getenv('DATABASE_USERNAME'),
                        password=os.getenv('DATABASE_PASSWORD'),
                        host=os.getenv('DATABASE_HOST'),
                        port=3306,
                        database=os.getenv('DATABASE_NAME')
                )
        except mariadb.Error as e:
                raise Exception(e)

        cursor = conn.cursor()
        return conn, cursor



def close_conn(conn):
    conn.close()

def insert_data(conn, cursor, data):
    #expecting data will be a JSON object
    table_name = 'PollutionData'

    #GPS module outputs have N/S/E/W instead of +/- lat and long
    row = modify_coordinates(data)

    date=row['date_measured']
    time=row['time_measured']
    co2=row['co2']
    temp=row['temperature']
    humidity=row['humidity']
    pm25=row['pm25']
    lat = row['latitude']
```

```
        pm25=row['pm25']
        lat = row['latitude']
        long=row['longitude']



        try:
            cursor.execute("INSERT INTO PollutionData (DateMeasured, TimeMeasured, Latitude, Longitude, CO2, Temperature,
            Humidity, PM25) VALUES (?, ?, ?, ?, ?, ?, ?, ?)", (date, time, lat, long, co2, temp, humidity, pm25))
        except mariadb.Error as e:
            print(f"Error: {e}")

        conn.commit()

        #Also save the data into a csv file
        csv_file_path = 'static/output_data.csv'
        file_exists = os.path.exists(csv_file_path)
        with open(csv_file_path, mode="a", newline='') as csv_file:
            fieldnames=['date_measured','time_measured','latitude','longitude','co2','temperature','humidity','pm25']
            writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

            #header
            if not file_exi
                writer.writeheader()

            #write data
            writer.writerow(row)

    def get_variation(cursor):
            query = "SELECT TimeMeasured, CO2, PM25, Temperature, Humidity FROM PollutionData WHERE DateMeasured IN (SELECT max
            (DateMeasured) FROM PollutionData)"
            time = []
            co2 = []
            temp = []
            humidity = []
            pm25 = []
```

(method) def writeheader() -> Any

See Real World Examples From GitHub

```
            try:
                cursor.execute(query)
                for item in cursor:
                    time.append(item[0])
                    co2.append(item[1])
                    pm25.append(item[2])
                    temp.append(item[3])
                    humidity.append(item[4])
                return time, co2, pm25, temp, humidity
            except mariadb.Error as e:
                print(f"Error: {e}")

    def get_latest(cursor):
            query = "SELECT CO2, PM25, Temperature, Humidity, Latitude, Longitude FROM PollutionData WHERE DateMeasured IN (SELECT
            max(DateMeasured) FROM PollutionData) AND TimeMeasured IN (SELECT MAX(TimeMeasured) FROM PollutionData WHERE
            DateMeasured IN (SELECT MAX(DateMeasured) FROM PollutionData))"

            co2 = "0"
            temp = "0"
            humidity = "0"
            pm25 = "0.0"
            lat = "0.0"
            longi = "0.0"

            try:
                cursor.execute(query)
                for item in cursor:
                    co2 = item[0]
                    pm25 = item[1]
                    temp = item[2]
                    humidity = item[3]
                    lat = item[4]
                    longi = item[5]
                return co2, pm25, temp, humidity, lat, longi
            except mariadb.Error as e:
                print(f"Error: {e}")
```

```
108 def get_heatmap_points(cursor):
109     query = "SELECT DISTINCT(CONCAT(Latitude, ' ', Longitude)) AS cc, CO2 FROM PollutionData pd WHERE DateMeasured IN
    (SELECT max(DateMeasured) FROM PollutionData) AND pd.Latitude NOT IN ('-0.000000','0.0000000') GROUP BY cc"
110
111     lat_points = []
112     long_points = []
113     co2_conc = []
114
115     try:
116         cursor.execute(query)
117         for item in cursor:
118             lat = item[0].split(" ")[0]
119             long = item[0].split(" ")[1]
120             lat_points.append(float(lat))
121             long_points.append(float(long))
122             co2_conc.append(item[1])
123         return lat_points, long_points, co2_conc
124     except mariadb.Error as e:
125         print(f"Error: {e}")
126
127 def modify_coordinates(row_data):
128     #N/E are +ve coordinates, S/W are -ve coordinates
129     lat_string=row_data['latitude']
130     long_string=row_data['longitude']
131     #if lat_string != "":
132     #     lat=lat_string[:-1] if lat_string[-1]=="N" else "-"+lat_string[:-1]
133     #else:
134     #     lat = lat_string
135     #if long_string != "":
136     #     long=long_string[:-1] if long_string[-1]=="E" else "-"+long_string[:-1]
137     #else:
138     #     long = long_string
139     row_data['latitude']=lat_string
140     row_data['longitude']=long_string
141     return row_data
```

2. app.py

app.py > ...

```python
from flask import Flask, render_template, request, jsonify
from bokeh.plotting import figure, show
from bokeh.embed import components
from bokeh.resources import CDN
from bokeh.layouts import row
import xyzservices.providers as xyz
import backend
import numpy as np
import pandas as pd

app = Flask(__name__)

#get initial values to display on homescreen
conn, cursor = backend.initialize_conn()
latest_co2, latest_pm25, latest_temp, latest_humidity, latest_lat, latest_long = backend.get_latest(cursor)
backend.close_conn(conn)


@app.route('/', methods=['GET', 'POST'])
def index():

    #get latest values for homescreen
    global latest_temp, latest_humidity, latest_pm25, latest_co2, latest_lat, latest_long

    #plot latest location for homescreen
    long_merc, lat_merc = wgs84_to_web_mercator(latest_lat, latest_long)
    map_plot = figure(x_range=(-8425584.61, -8425784.61), y_range=(5678025.27, 5698025.27),
                      x_axis_type="mercator", y_axis_type="mercator")
    map_plot.add_tile(xyz.OpenStreetMap.Mapnik)
    map_plot.circle(x=long_merc,y=lat_merc, size=12, fill_color="red", fill_alpha=0.8)
    map_script, map_div = components(map_plot)


    if request.method == 'POST':    #homescreen

        cdn_js = CDN.js_files[0]
```

```python
        if request.form.get('history') == 'Pollution History':

            conn, cursor = backend.initialize_conn()
            x,y1, y2, y3, y4 = backend.get_variation(cursor)    #get variation values for the latest day from db
            backend.close_conn(conn)

            #convert time from string to time format for plotting
            x_formatted = pd.to_datetime(pd.Series(x), format='%H:%M:%S').dt.time

            #y1 - co2, y2 - PM2.5,  y3 - temp, y4 - humidity
            p1 = figure(x_axis_label="time",
                        y_axis_label="variation",
                        x_axis_type='datetime',
                        title="Variation")
            p2 = figure(x_axis_label="time",
                        y_axis_label="variation",
                        x_axis_type='datetime',
                        title="CO2 variation",
                        y_range=(500, 20000))

            p1.line(x_formatted, y2, line_width=2,
                    legend_label ="PM2.5 concentration", color="black")
            p1.line(x_formatted, y3, line_width=2,
                    legend_label ="Temperature", color="red")
            p1.line(x_formatted, y4, line_width=2,
                    legend_label ="Humidity", color="blue")

            p2.line(x_formatted, y1, line_width=2,
                     legend_label ="CO2 concentration", color="green")


            script_p1, div_p1 = components(p1)  #PM25, temp, humidity plot
            script_p2, div_p2 = components(p2)  #CO2 plot
```

```
            return render_template("plot.html",
                                   script1=script_p1,
                                   div1=div_p1,
                                   script2=script_p2,
                                   div2=div_p2,
                                   cdn_js=cdn_js)



        if request.form.get('about') == 'About Us':
            return render_template("about_us.html")


        if request.form.get('map') == 'Map':
            conn, cursor = backend.initialize_conn()
            lat_points, long_points, co2_conc = backend.get_heatmap_points(cursor)
            backend.close_conn(conn)

            #find the area on the map to be plotted
            if len(lat_points)>0:
                lat_avg = sum(lat_points)/len(lat_points)
                long_avg = sum(long_points)/len(long_points)
                long_avg_merc, lat_avg_merc = wgs84_to_web_mercator(lat_avg, long_avg)
                x1 = long_avg_merc-700.693851693
                x2 = long_avg_merc+700.693851693
                y1 = lat_avg_merc-1000.14474785
                y2 = lat_avg_merc+1000.14474785
            else:
                x1, x2, y1, y2 = -8425584.61, -8425784.61, 5678025.27, 5698025.27   #plot general ottawa area


            heatmap_plot = figure(x_range=(x1,x2), y_range=(y1,y2),
                                  x_axis_type="mercator", y_axis_type="mercator")
            heatmap_plot.add_tile(xyz.OpenStreetMap.Mapnik)
```

```
            #create dict with x,y,colour to act as a data source for plotting
            data_src = {'x':[],'y':[],'color':[]}
            for i in range(0,len(lat_points)):
                x_pt, y_pt = wgs84_to_web_mercator(lat_points[i],long_points[i])
                color = "green"
                co2_level = float(co2_conc[i])
                if co2_level >= 1000 and co2_level <= 2000:
                    color = "darkorange"
                elif co2_level > 2000 and co2_level <= 5000:
                    color = "orange"
                elif co2_level > 5000 and co2_level <= 40000:
                    color = "orangered"
                elif co2_level > 40000:
                    color = "red"
                data_src['x'].append(x_pt)
                data_src['y'].append(y_pt)
                data_src['color'].append(color)

            heatmap_plot.circle(x='x',y='y', size=12, fill_color='color', fill_alpha=0.8, source = data_src)


            heatmap_script, heatmap_div = components(heatmap_plot)
            return render_template("map.html",
                                   heatmap_script=heatmap_script,
                                   heatmap_div=heatmap_div,
                                   cdn_js=cdn_js
                                   )
```

```
    elif request.method == 'GET':
        return render_template("main.html",
                               temp=latest_temp,
                               humidity=latest_humidity,
                               CO2=latest_co2,
                               PM25=latest_pm25,
                               map_script=map_script,
                               map_div=map_div)

    return render_template("main.html",
                           temp=latest_temp,
                           humidity=latest_humidity,
                           CO2=latest_co2,
                           PM25=latest_pm25,
                           map_script=map_script,
                           map_div=map_div)



#Save data sent by Arduino
@app.route('/save_data', methods=['POST'])
def save_data():
    try:
        data = request.get_json()
        conn, cursor = backend.initialize_conn()
        backend.insert_data(conn, cursor, data)
        backend.close_conn(conn)

        #Update the latest values to be displayed on homescreen
        update_latest(data)
        return "Save Success!"
    except Exception as e:
        return jsonify({"error":str(e)}), 500
```

```
#Function to update the latest values to be displayed on homescreen
def update_latest(data):

    global latest_co2, latest_pm25, latest_temp, latest_humidity, latest_lat, latest_long

    latest_co2=data['co2']
    latest_temp=data['temperature']
    latest_humidity=data['humidity']
    latest_pm25=data['pm25']
    latest_lat = data['latitude']
    latest_long=data['longitude']


#Function to convert latitude and longitude to web mercator coordinate format
def wgs84_to_web_mercator(latitude, longitude):
      k = 6378137
      x = float(longitude) * (k * np.pi/180.0)
      y = np.log(np.tan((90 + float(latitude)) * np.pi/360.0)) * k
      return x,y
```

app.py > ...

```python
from flask import Flask, render_template, request, jsonify
from bokeh.plotting import figure, show
from bokeh.embed import components
from bokeh.resources import CDN
from bokeh.layouts import row
import xyzservices.providers as xyz
import backend
import numpy as np
import pandas as pd

app = Flask(__name__)

#get initial values to display on homescreen
conn, cursor = backend.initialize_conn()
latest_co2, latest_pm25, latest_temp, latest_humidity, latest_lat, latest_long = backend.get_latest(cursor)
backend.close_conn(conn)


@app.route('/', methods=['GET', 'POST'])
def index():

    #get latest values for homescreen
    global latest_temp, latest_humidity, latest_pm25, latest_co2, latest_lat, latest_long

    #plot latest location for homescreen
    long_merc, lat_merc = wgs84_to_web_mercator(latest_lat, latest_long)
    map_plot = figure(x_range=(-8425584.61, -8425784.61), y_range=(5678025.27, 5698025.27),
                      x_axis_type="mercator", y_axis_type="mercator")
    map_plot.add_tile(xyz.OpenStreetMap.Mapnik)
    map_plot.circle(x=long_merc,y=lat_merc, size=12, fill_color="red", fill_alpha=0.8)
    map_script, map_div = components(map_plot)


    if request.method == 'POST':    #homescreen

        cdn_js = CDN.js_files[0]
```

```
    if request.form.get('history') == 'Pollution History':

        conn, cursor = backend.initialize_conn()
        x,y1, y2, y3, y4 = backend.get_variation(cursor)    #get variation values for the latest day from db
        backend.close_conn(conn)

        #convert time from string to time format for plotting
        x_formatted = pd.to_datetime(pd.Series(x), format='%H:%M:%S').dt.time

        #y1 - co2, y2 - PM2.5,  y3 - temp, y4 - humidity
        p1 = figure(x_axis_label="time",
                    y_axis_label="variation",
                    x_axis_type='datetime',
                    title="Variation")
        p2 = figure(x_axis_label="time",
                    y_axis_label="variation",
                    x_axis_type='datetime',
                    title="CO2 variation",
                    y_range=(500, 20000))

        p1.line(x_formatted, y2, line_width=2,
               legend_label ="PM2.5 concentration", color="black")
        p1.line(x_formatted, y3, line_width=2,
               legend_label ="Temperature", color="red")
        p1.line(x_formatted, y4, line_width=2,
               legend_label ="Humidity", color="blue")

        p2.line(x_formatted, y1, line_width=2,
                legend_label ="CO2 concentration", color="green")


        script_p1, div_p1 = components(p1)  #PM25, temp, humidity plot
        script_p2, div_p2 = components(p2)  #CO2 plot
```

```
        return render_template("plot.html",
                               script1=script_p1,
                               div1=div_p1,
                               script2=script_p2,
                               div2=div_p2,
                               cdn_js=cdn_js)



    if request.form.get('about') == 'About Us':
        return render_template("about_us.html")


    if request.form.get('map') == 'Map':
        conn, cursor = backend.initialize_conn()
        lat_points, long_points, co2_conc = backend.get_heatmap_points(cursor)
        backend.close_conn(conn)

        #find the area on the map to be plotted
        if len(lat_points)>0:
            lat_avg = sum(lat_points)/len(lat_points)
            long_avg = sum(long_points)/len(long_points)
            long_avg_merc, lat_avg_merc = wgs84_to_web_mercator(lat_avg, long_avg)
            x1 = long_avg_merc-700.693851693
            x2 = long_avg_merc+700.693851693
            y1 = lat_avg_merc-1000.14474785
            y2 = lat_avg_merc+1000.14474785
        else:
            x1, x2, y1, y2 = -8425584.61, -8425784.61, 5678025.27, 5698025.27   #plot general ottawa area


        heatmap_plot = figure(x_range=(x1,x2), y_range=(y1,y2),
                              x_axis_type="mercator", y_axis_type="mercator")
        heatmap_plot.add_tile(xyz.OpenStreetMap.Mapnik)
```

```
        #create dict with x,y,colour to act as a data source for plotting
        data_src = {'x':[],'y':[],'color':[]}
        for i in range(0,len(lat_points)):
            x_pt, y_pt = wgs84_to_web_mercator(lat_points[i],long_points[i])
            color = "green"
            co2_level = float(co2_conc[i])
            if co2_level >= 1000 and co2_level <= 2000:
                color = "darkorange"
            elif co2_level > 2000 and co2_level <= 5000:
                color = "orange"
            elif co2_level > 5000 and co2_level <= 40000:
                color = "orangered"
            elif co2_level > 40000:
                color = "red"
            data_src['x'].append(x_pt)
            data_src['y'].append(y_pt)
            data_src['color'].append(color)

        heatmap_plot.circle(x='x',y='y', size=12, fill_color='color', fill_alpha=0.8, source = data_src)


        heatmap_script, heatmap_div = components(heatmap_plot)
        return render_template("map.html",
                               heatmap_script=heatmap_script,
                               heatmap_div=heatmap_div,
                               cdn_js=cdn_js
                               )
```

```
    elif request.method == 'GET':
        return render_template("main.html",
                               temp=latest_temp,
                               humidity=latest_humidity,
                               CO2=latest_co2,
                               PM25=latest_pm25,
                               map_script=map_script,
                               map_div=map_div)

    return render_template("main.html",
                           temp=latest_temp,
                           humidity=latest_humidity,
                           CO2=latest_co2,
                           PM25=latest_pm25,
                           map_script=map_script,
                           map_div=map_div)



#Save data sent by Arduino
@app.route('/save_data', methods=['POST'])
def save_data():
    try:
        data = request.get_json()
        conn, cursor = backend.initialize_conn()
        backend.insert_data(conn, cursor, data)
        backend.close_conn(conn)

        #Update the latest values to be displayed on homescreen
        update_latest(data)
        return "Save Success!"
    except Exception as e:
        return jsonify({"error":str(e)}), 500
```

```
#Function to update the latest values to be displayed on homescreen
def update_latest(data):

    global latest_co2, latest_pm25, latest_temp, latest_humidity, latest_lat, latest_long

    latest_co2=data['co2']
    latest_temp=data['temperature']
    latest_humidity=data['humidity']
    latest_pm25=data['pm25']
    latest_lat = data['latitude']
    latest_long=data['longitude']


#Function to convert latitude and longitude to web mercator coordinate format
def wgs84_to_web_mercator(latitude, longitude):
      k = 6378137
      x = float(longitude) * (k * np.pi/180.0)
      y = np.log(np.tan((90 + float(latitude)) * np.pi/360.0)) * k
      return x,y
```