MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Electronic Systems Technology Project 2024

05/Dec/2024

Alan O'Connell

Cathal O'Regan

Jamie O'Connor

I acknowledge, by submitting this report, that I am aware of the provisions in MTU's Student Regulations which specifically cover cheating where any attempt is made to gain unfair advantage, whether in coursework, assessments or examinations.

I further acknowledge that I understand the Policy and associated Procedures together with the Supplementary Policy and Supplementary Procedure which have been approved for this period

Abstract

This project focuses on the development and implementation of an environmental monitoring system designed to collect, log, and analyse data from multiple sensors. The system integrates an Arduino and Raspberry Pi to read environmental parameters, such as temperature, humidity, and air quality, using sensors like the DHT11 and MQ135. The system then logs the collected data into a CSV file, which is automatically uploaded to Google Drive for easy access and storage. Python-based scripts automate the data collection, processing, and cloud upload, ensuring real-time monitoring and remote access.

The primary objective of this project was to create a reliable, automated system that continuously collects environmental data, stores it securely on Google Drive, and allows for future data analysis or predictive modelling. The system was designed to run autonomously, minimizing manual intervention, and offering a scalable solution for long-term environmental monitoring.

The project makes use of technologies such as Python, Google Drive API, and serial communication via the Arduino. Key challenges included ensuring stable data logging and handling real-time data uploads. The final system successfully met its set objectives, offering a robust solution for environmental data monitoring and storage. Future work may include enhancing the data analysis capabilities and exploring ways to optimize the system's performance for large-scale deployments.

See the full report below for a more thorough walkthrough of the project.

Contents

Table of Figures

1. Introduction

The main focus of this project was to develop an automated environmental monitoring system to collect real-time data from sensors utilizing an Arduino in addition with a Raspberry Pi to record key environmental parameters, such as temperature, humidity, and air quality with the data being logged continuously for further analysis.

The system was designed to operate autonomously without need for manual intervention by automatically logging sensor readings to a CSV file, stored locally on the Raspberry Pi. The data is then uploaded to a Google Drive using Python scripts for automated cloud storage. This ensures that all the collected data can be easily accessible and can be analysed remotely.

This report provides a breakdown of the project, including a detailed description of the hardware and software components used, the process followed to complete the system, and an analysis of its performance. By using both the Arduino, Raspberry Pi, and cloud-based storage with the Google Drive API, this system aims to provide a reliable and scalable solution for environmental data collection and storage.

2. Hardware and Software Considerations

This section outlines the various hardware and software options that were evaluated during the development of the project. It discusses the decision making behind the selection of the components, including sensors, cloud storage solutions, and also explains why these choices were made based on the project requirements. It also highlights the software tools and libraries considered for the project that were chosen to ensure efficient data collection, processing, and storage.

2.1 Hardware

Given the project requirements which required the use of a Raspberry Pi and Arduino, the hardware selection was driven by the need to monitor key environmental parameters, including the chosen parameters of temperature, humidity, and air quality. The following were chosen based on their compatibility with the Arduino and Raspberry Pi, their ease of integration, and the specific environmental data they could provide to the project.

2.1.1 DHT11 Sensor (Temperature and Humidity): The DHT11 sensor is a low-cost digital sensor which measures both the temperature and humidity levels in the surrounding environment. The DHT11 was selected due to its simplicity and ease of use with the Arduino, by providing straightforward digital outputs for temperature and humidity readings. It offers sufficient accuracy for basic environmental monitoring, with a temperature range of 0°C to 50°C and humidity measurements between 20% and 80%. The sensor operates on a low voltage (3-5V), which makes it compatible with both the Raspberry Pi and Arduino. [1]

2.1.2 MQ135 Sensor (Air Quality): The MQ135 gas sensor is capable of detecting a wide range of gases, including ammonia, carbon dioxide, alcohol, benzene, and other toxic gases. This sensor allowed the project to focus on general air quality monitoring. The sensor's basic power connection allowed it to operate and generate a general level of detection, which is read by the Arduino and sent to the Raspberry Pi for processing and logging. [2]

2.1.3 Photoresistor (LDR): The photoresistor (LDR) was used in this project to measure the ambient light levels in the environment. A photoresistor is a light sensor that changes its resistance depending on the intensity of light it detects. In bright light, its resistance is low,

while in darkness, its resistance is high. It was connected to an analog input pin on the Arduino to read the light level. The Arduino then interprets the analog signal, converting it into a digital value that can be logged.

2.1.4 DS1307 Real-Time Clock (RTC): The DS1307 RTC was utilized in this project to add a timestamp to the sensor data ensuring that each environmental reading is logged with an accurate and reliable timestamp. The RTC keeps time using its internal oscillator, which is powered by a small coin cell battery. This allows it to retain the current date and time even when the main power supply is disconnected. The time is communicated to the Arduino via I2C requiring only two data lines (SDA and SCL) for operation. [3]

2.1.5 LEDs for Visual Indicators: LEDs (Light Emitting Diodes) were used to provide visual feedback about the system's operational status. These LEDs served as a simple yet effective indicators to show when data was being collected from the sensors, offering real-time feedback to the user.

In summary, the hardware components chosen for this project were selected based on their compatibility with the Raspberry Pi and Arduino, as well as their ability to accurately monitor the required environmental parameters. The DHT11, MQ135, photoresistor (LDR), DS1307 RTC, and LEDs were integrated into the system to ensure reliable data collection, real-time monitoring, and timestamping. These components met the project's requirements while maintaining simplicity, cost-effectiveness, and ease of integration, resulting in a robust and scalable project.

2.2 Alternative Hardware Considerations (SDS011)

While developing the environmental monitoring system, the SDS011 particulate matter sensor was also initially considered as an alternative to the MQ135 sensor for monitoring the air quality. The SDS011 is capable of measuring particulate matter (PM2.5 and PM10) levels, providing a more specific indicator of air pollution. It operates based on laser scattering technology and is highly accurate for detecting fine dust particles within the air, which could have been a valuable addition to the project.

However, during testing it was found that the power requirements of the SDS011 exceeded the capabilities of the Arduino’s power supply. The sensor required a stable 5V power source and a high current draw, which the Arduino's onboard voltage regulator struggled to maintain especially when combined with the other components in the system. This resulted in voltage fluctuations, intern leading to inconsistent data readings and making it difficult to rely on the sensor for accurate environmental measurements over a long period of time. [4]

Due to these power limitations and the inconsistent data readings, the team decided that the SDS011 was ultimately not going to be implemented in the final system.

2.3 Software

This section covers the software tools, and libraries chosen to implement the environmental monitoring system. The software was designed to automate the collection, processing, and storage of the environmental data while ensuring easy accessibility and future scalability. The choice of software was influenced by factors such as ease of integration with hardware, efficiency, and compatibility with the Raspberry Pi and Arduino platforms.

2.3.1 Programming Languages

The primary programming language used for this project was Python, due to its simplicity, versatility, and the extensive range of libraries suitable for the hardware interfacing, data processing, and cloud integration.

The Arduino IDE was also used to write and upload code to the Arduino. This was necessary to interface with the sensors and perform the required sensor readings. The Arduino IDE supports C++ based programming which provided a straightforward and efficient way to interact with the hardware components.

2.3.2 Arduino Libraries

As previously mentioned, the Arduino platform was primarily responsible for interacting with the sensors, collecting raw environmental data, and passing it on to the Raspberry Pi for further processing. The following Arduino libraries were used in the project:

- **DHT.h:** This library is used to interface with the DHT11 sensor. It simplifies the process of obtaining readings from the sensor, allowing for easy data retrieval by handling sensor communication and providing temperature and humidity values in a readable format for use in the project.
- **Wire.h:** The Wire library is essential for the I2C communication, which is used to connect the Arduino and any device that uses the I2C protocol, such as the DS1307 RTC, allowing the Arduino to retrieve accurate time and date data.
- **RTClib.h:** The RTClib library supports the DS1307 RTC module, which allows for configuring the time settings of the RTC. This ensures that every sensor reading is logged with an accurate timestamp.

2.3.3 Python Libraries

The Python libraries used in this project were essential for managing sensor data, handling cloud storage, and providing real-time visualization through a dashboard. Below is a breakdown of the libraries used in the project based on their role in the system.

2.3.3.1 Data Storage and File Management

These libraries were used for managing the CSV file, storing the sensor data, and ensuring data persistence.

- **csv**: Handles the reading and writing of CSV files which store the collected sensor data from the Arduino. This library allows the program to log the sensor data along with timestamps into a CSV file, ensuring that the data is saved in an accessible format.
- **os:** This provides access to operating system functions, such as checking file paths and directories. This library helps ensure that the necessary files and directories for storing the sensor data are created and managed correctly.

2.3.3.2 Cloud Storage (Google Drive)

These libraries were responsible for handling cloud storage operations and ensuring that the sensor data was uploaded to Google Drive correctly for remote access and storage. For more information about the google drive libraries, see here: [5]

- **google.auth.transport.requests**: Used for making authenticated requests to the Google API. This library handles the layer of authentication, ensuring that the requests to the Google Drive are secure.
- **google_auth_oauthlib.flow:** Manages OAuth2 authentication with Google APIs. It facilitates the authentication, enabling the Python script to securely access Google Drive and upload files.
- **googleapiclient.discovery:** This library enables interaction with the Google Drive API to upload and manage files in the cloud.
- **googleapiclient.http:** Handles HTTP requests for Google Drive file uploads and downloads. It was used to upload the CSV file to Google Drive, ensuring that the collected sensor data is stored remotely and can be accessed later.
- **Pickle**: Pickle was used to save and load authentication credentials and configuration data for Google Drive API access, allowing for the system to reuse saved session states.

2.3.3.3 Data Collection and Communication

This library was used for communicating with the Arduino and collecting data from the sensors.

- **Serial**: Enables serial communication between the Raspberry Pi and the Arduino to receive sensor data. This library allows the Raspberry Pi to read the data sent by the Arduino over the serial connection.

2.2.2.4 Data Processing and Analysis

This library was used for processing the collected sensor data and preparing it for visualization.

- **Pandas:** Pandas is a well known data manipulation and analysis library, and was used to read and process the CSV data, including handling missing values, formatting, and preparing the data for visualization.

2.2.2.5 Data Visualization (Dashboard)

These libraries were used to create the interactive dashboard that visualizes the sensor data.

- **dash**: dash is a framework for building interactive web applications, such as dashboards. Dash was used to build the UI for displaying the sensors data in real-time. It allows for interactive features, such as updating the graphs with new data every set amount of time.

- **plotly**: Plotly was used to generate graphs and plots that visually represent the collected sensor data on the website. These visualizations are embedded and hosted within the Dash dashboard itself.

2.3.4 Summary of Software Considerations

The integration of these selected libraries was critical in enabling seamless communication between the hardware components and the software layers of the system. The Arduino libraries such as DHT.h, Wire.h, and RTClib.h ensured accurate sensor readings, efficient I2C communication, and precise timekeeping. Meanwhile, the Python libraries facilitated the required functionalities, including serial communication, CSV management, Google Drive API operations, and data visualization. These software choices ensured that the system met its design specifications for real-time, autonomous environmental monitoring and data analysis.

3. Procedure

The development of the environmental monitoring system followed a structured process beginning with the design of an initial system architecture represented in the flowchart shown in Figure 1. This flowchart outlined the system's operations into five key stages: sensor data collection, data transmission, data logging, cloud storage, and visualization. By establishing a comprehensive framework, the team ensured a clear understanding of the systems dependencies and operational flow. Testing phases were also embedded within each of these five stages, enabling systematic validation before progressing onto the next stage. This approach allowed for early detection and resolution of potential issues, enhancing the overall efficiency and also optimizing the teams time management throughout the project.

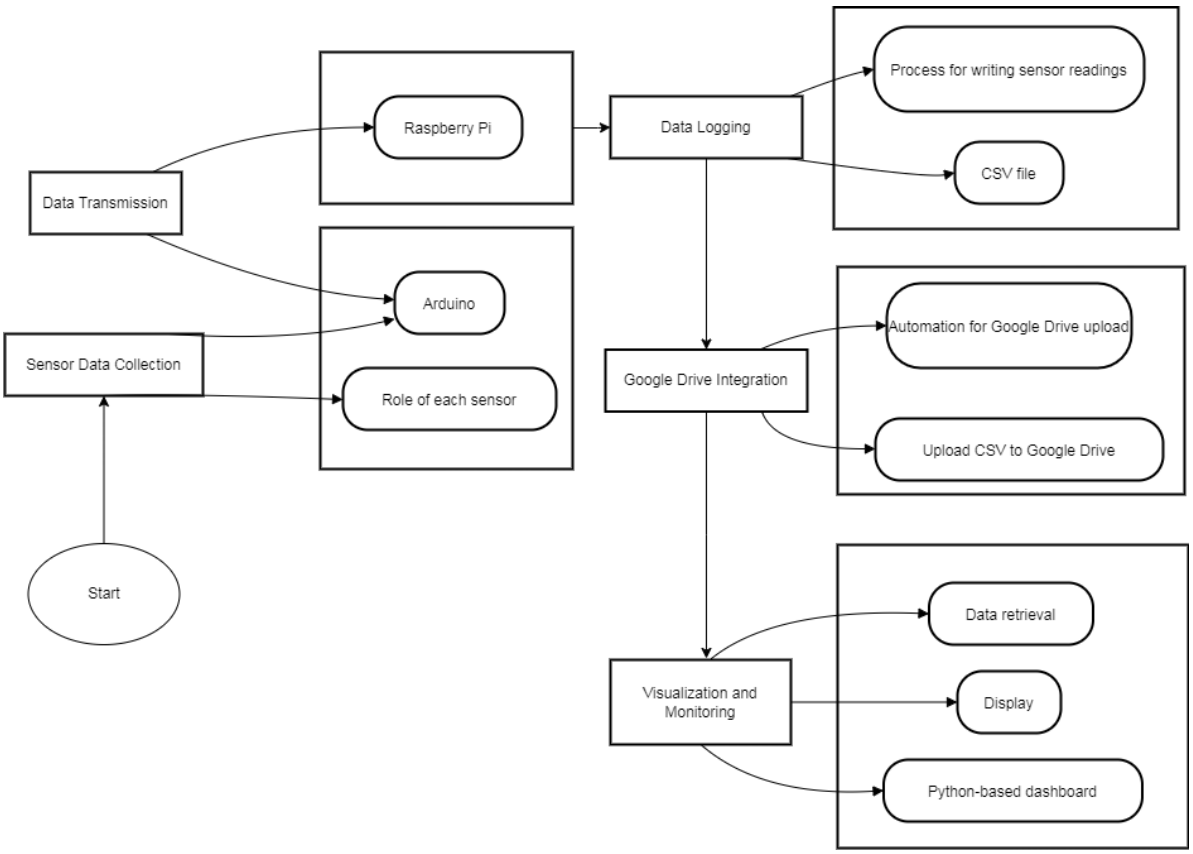

Figure 1: System Architecture

3.1 Hardware Implementation

The hardware schematic seen in Figure 2 and Figure 3, served as the blueprint for connecting the various components of the project. It provided a clear representation of the connections between the sensors, microcontroller, and power supplies, ensuring the system's stability and overall functionality. The schematic was designed to include all the critical hardware elements, including the DHT11, MQ135, LDR, DS1307 RTC, indicator LEDs and required resistors seen in Figure 3.

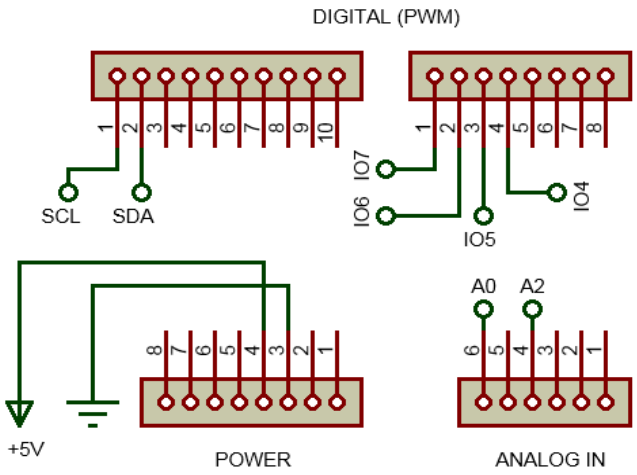

Figure 2: Simulated Arduino Schematic

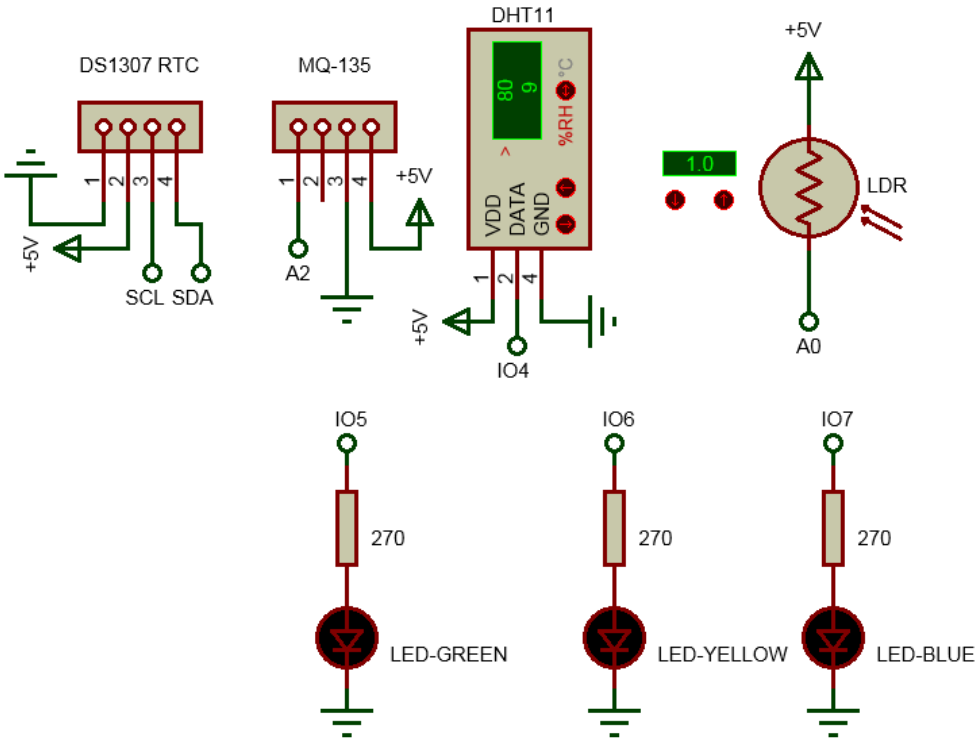

Figure 3: Simulated Breadboard Layout

During the implementation phase, the schematics allowed the team to connect components and verify their functionality independently prior to integrating them into the full system. For example, the DHT11 sensor was first tested using a simple program to validate temperature and humidity readings, ensuring its functionality before it was connected alongside the other components. Because of this the team was able to maintain a logical and systematic layout on the breadboard, reducing the risk of miswiring and potential shorts in the circuit. As a result, the schematic not only allowed for accurate breadboard implementations but also significantly enhanced the team's efficiency and confidence in the hardware setup before beginning the programming. Its important to note that the project incorporates three LEDs to meet the digital I/O requirements of providing visual feedback during system operation, however the system could easily be optimized by utilizing a single LED to perform the same task.

3.2 Software Implementation

With the hardware successfully set up and verified, the next phase involved developing and integrating the software required to control the system, process sensor data, and facilitate data storage and visualization. The software development was divided into two key parts: Arduino-based control/data acquisition, and the Python-based data processing, cloud integration and visualization. Below is an overall guide through the programming process.

3.2.1 Arduino-based control and Data Acquisition

The Arduino program begins by including the necessary libraries previously discussed, for interfacing with the sensors and the RTC module. The DHT.h library is used to communicate with the DHT11 temperature and humidity sensor, while the Wire.h library facilitates I2C communication, which is essential for connecting to the DS1307 real-time clock (RTC) module. The RTClib.h library is used to control the RTC and retrieve the current time.

The sensors are then initialized by defining their respective pins seen in Figure 3. The core of the program runs within the loop() function. Here, is where the sensor data is collected by reading the sensors values. The RTC is prompted for the current timestamp in the YYYY/MM/DD format as seen in Figure 4, and the system formats the data into a CSV-like structure and sends it over serial communication to the Raspberry Pi.

```
// Check if the DHT11 readings are valid
if (!isnan(humidity) && !isnan(temperature)) {
    // Send the timestamp from RTC along with the sensor data
    Serial.print(now.year(), DEC);
    Serial.print("/");
    Serial.print(now.month(), DEC);
    Serial.print("/");
    Serial.print(now.day(), DEC);
    Serial.print(" ");
    Serial.print(now.hour(), DEC);
    Serial.print(":");
    Serial.print(now.minute(), DEC);
    Serial.print(":");
    Serial.print(now.second(), DEC);
    Serial.print(",");
```

Figure 4: RTC Timestamp Format

Each reading includes the timestamp, light level, air quality value, humidity and temperature as seen below in Figure 5. If there is an error in reading the DHT11 sensor, an error message is printed instead, (this is important for manually checking the data in the serial monitor, however this will later be further addressed in the python script).

```
2024/11/24 16:20:49,9.97,675,68.00,19.70
```

Figure 5: CSV Formatted Data

Finally, a delay is implemented to ensure the system does not overload the sensors and communication channels with constant data transmission.

3.2.2 Data Logging To CSV Using Python

The Python script for data logging forms the backbone of the Raspberry Pi's functionality, ensuring the collection, validation, and storage of the sensor data sent by the Arduino. It is

designed for efficiency and reliability, focusing on structured data handling to allow for the future analysis and visualization.

3.2.2.1 Serial Communication with Arduino: The script begins by establishing a serial connection with the Arduino using the serial library as previously discussed. The connect_to_arduino function seen in Figure 6 handles this by specifying the serial port and baud rate. It then verifies the connection status and handles potential connection errors to ensure data reliability.

```
def connect_to_arduino():
    """
    Attempts to establish a serial connection to the Arduino board using the specified port and baud rate.
    SERIAL_PORT = '/dev/ttyACM0'
    BAUD_RATE = 9600
    If the connection is successful, it returns the serial object to interact with Arduino.
    If it fails, the error message is printed, and the program exits.

    Returns:
        arduino (Serial): A serial object for communication with the Arduino.

    Raises:
        SerialException: If unable to open the specified serial port.
    """
    try:
        arduino = serial.Serial(SERIAL_PORT, BAUD_RATE)         # Opens the serial connection to Arduino
        print("Connected to Arduino successfully!")
        return arduino
    except serial.SerialException as e:
        print(f"Error: Could not open port {SERIAL_PORT}. {e}") # Feedback for opening error
        raise
```

Figure 6: Arduino Connection Function

The arduino.readline() method seen in Figure 7 is used within the main loop to continuously fetch the data from the Arduino. Each data line is expected to be a comma-separated string containing the timestamp, light level, gas concentration, humidity, and temperature. This can be checked via the terminal as “Raw data”, with the “.decode('utf-8')” method converting the raw byte data into a readable string format, while “.strip()” removes any leading/trailing whitespace or newline characters for overall cleaner processing. The script also checks if the received data contains the word "Error". This was implemented as a safeguard to skip lines where sensor readings failed or are flagged as invalid by the Arduino code i.e. potentially missing sensor data.

```
# Loop for continuously collecting and logging data
while True:
    # Read a line of data from the Arduino and decode it
    data = arduino.readline().decode('utf-8').strip()
    print(f"Raw data received: '{data}'")

    # Skip bad data (mainly sensor errors)
    if "Error" in data:
        print("Sensor error, skipping entry.")
        continue
```

Figure 7: Method for Continuously Fetching Data

This filtering mechanism ensures that the CSV file only contains meaningful entries, intern preserving the accuracy and reliability of the stored dataset and the final dashboard.

3.2.2.2 CSV Initialization: The initialize_csv function seen in Figure 8 is responsible for preparing the CSV file used to log sensor data. This preparation process ensures that the file is correctly formatted and ready to receive structured data from the system. It also guarantees that the data will adhere to a standardized format, simplifying future analysis and visualization.

The function begins by checking whether the specified CSV file, identified by the constant CSV_FILENAME, already exists within the system. This check is performed using Python's os.path.exists() method. If the file is not present, the script recognizes that it needs to create a new file to store the data. Additionally, the function determines whether the file is empty by evaluating its size with os.path.getsize().

If the file does not exist or is empty, the function writes a header row seen in Figure 10 to the file using the csv_writer.writerow() method. This header row includes column names that correspond to the fields in the logged data: "Timestamp", "Light Level (%)", "Gas Concentration (MQ135)", "Humidity (%)", and "Temperature (°C)". These headers provide context for each column, improving the interpretability of the dataset and was designed in aiding future users or developers in understanding the file structure.

```
def initialize_csv():
    """
    Initializes the CSV file where sensor data will be stored. If the file doesn't exist or is empty,
    it writes the header row with the column names.

    Returns:
        csv_writer (csv.writer): A CSV writer object to write data to the file.
    """
    # Checks if the CSV file already exists
    file_exists = os.path.exists(CSV_FILENAME)

    # Opens the CSV file in append mode
    with open(CSV_FILENAME, mode='a', newline='') as csv_file:
        csv_writer = csv.writer(csv_file)

        # Writes the header if the file doesn't exist or is empty
        if not file_exists or os.path.getsize(CSV_FILENAME) == 0:
            csv_writer.writerow(["Timestamp", "Light Level (%)", "Gas Concentration (MQ135)", "Humidity (%)", "Temperature (C)"])

    return csv_writer
```

Figure 8: CSV Initialization

3.2.2.3 Processing and Logging Valid Data: Once the system filters out bad data, the next step involves parsing and logging valid sensor readings into a CSV file named sensor_data.csv. This section of the code handles the core functionality of storing the filtered data for later analysis.

```
# Split the received data into separate values
values = data.split(",")

# If the data contains the expected number of values (5), log it to the CSV
if len(values) == 5:
    with open(CSV_FILENAME, mode='a', newline='') as csv_file:
        csv_writer = csv.writer(csv_file)
        csv_writer.writerow(values)  # Write the data to the CSV file
        csv_file.flush()             # Ensure the data is written immediately
        print(f"Logged data: {values}")
```

Figure 9: Handling CSV Logging

The “data.split(",")” method segments the serial input string into discrete fields using the comma (,) delimiter. This approach was done to align with the CSV file structure and allows for consistent data formatting as seen below:

Input string: '2024/11/24 10:12:05,72.5,300,45,22.3'

Parsed array: ['2024/11/24 10:12:05', '72.5', '300', '45', '22.3']

The script checks whether the parsed array contains exactly five fields. This verification step is critical to maintaining data integrity, as it filters out incomplete or malformed entries that may result from sensor errors or communication issues. Any data that fails this check is discarded, ensuring that only valid and complete entries are stored. The validated data is then appended to a CSV file. By opening the file in append mode (mode='a'), the script ensures that each new entry is added without overwriting the existing data. The data is then written as a new row using the "csv_writer.writerow(values)" method, with each field occupying a distinct column. An example of the good data being stored successfully can be seen below in Figure 10.

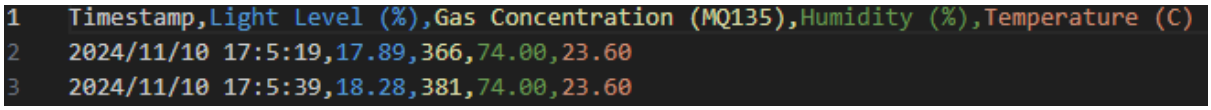

Figure 10: Valid Data Entry in CSV File

Throughout the process, the script provides real-time feedback by printing each logged entry to the terminal. This output not only confirms successful logging but also facilitated debugging and monitoring during the development. Any skipped entries due to errors or validation failures are similarly reported, which allowed the team to promptly identify and address any potential issues.

3.2.3 Automating the CSV Upload to Google Drive

The integration of Google Drive into the system requires a detailed configuration process to ensure secure and efficient file management. This section outlines the steps and technical considerations involved, from setting up the credentials on the Google Cloud Console to implementing the file upload functionality within the Python script.

3.2.3.1 Setting Up the Google Drive Console

The process starts by enabling the Google Drive API, which allows for interactions with Google Drive. To enable the API, access the Google Cloud Console [6], navigate to the "APIs & Services" section, and activate the Google Drive API for the project. This is a requirement for all applications intending to use the Google Drive services.

Once the API is enabled, OAuth 2.0 credentials must be created to authenticate the application. In the "Credentials" tab of the Cloud Console, a new set of credentials is generated by selecting the OAuth client ID option. For this project, the application type was specified as a Desktop App, since the system is going to work as a standalone application. After creating the credentials, the user downloads the JSON file (credentials.json) containing the client ID and client secret. This file is essential for establishing a secure connection with Google Drive and is placed in the project's working directory for the other scripts to access.

The Python script requires specific API scopes to define the permissions granted to the application. In this case, the "https://www.googleapis.com/auth/drive.file" scope is used, which limits access to files created or opened by the application. This will allow the program to access relevant files, while also enhancing the overall security of the project. [7] [8]

Below in figure 11 shows the function used for the authentication. The script uses the credentials.json file to initiate an OAuth 2.0 authentication flow using the google-auth and google-auth-oauthlib libraries. If valid credentials are available from a previously authenticated

session (stored in a token.pickle file), the script refreshes them as expected. Otherwise, it prompts the user to authenticate via a pop-up browser-based OAuth 2.0 flow, where they log in to their Google account and grant the requested permissions. Once authenticated, the script saves the credentials for future use, reducing the need for repeated user interactions.

```python
def authenticate_google_drive():
    """
    This handles the authentication process for the Google Drive API.
    It checks if valid credentials are stored in the 'token.pickle' file.
    If the credentials are expired or invalid, it attempts to refresh them.
    If refreshing fails, it prompts the user to authenticate via OAuth2 and saves the
    new credentials for future use.

    SCOPES = ['https://www.googleapis.com/auth/drive.file']
    CREDENTIALS_FILE = 'credentials.json'

    Returns:
        service (Resource): an authenticated Google Drive service object for file operations.
    """
    creds = None  # Stores the credentials

    # Checks if the token file exists and load the stored credentials
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)

    # This will initiate authentication if credentials are invalid or expired
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())   # Refresh expired credentials
        else:
            # Prompts user to authenticate via OAuth2 if no valid credentials exist
            flow = InstalledAppFlow.from_client_secrets_file(CREDENTIALS_FILE, scopes=SCOPES)
            creds = flow.run_local_server(port=0)  # Starts the local server for OAuth2 authentication

        # Saves the credentials for future use
        with open('token.pickle', 'wb') as token:
            pickle.dump(creds, token)

    # Return an authenticated Google Drive service
    return build('drive', 'v3', credentials=creds)
```

Figure 11: Google Drive Authentication

3.2.3.2 Google Drive Integration: File Management and Data Uploads

Once the Google Drive API credentials are established, the Python script uses the authenticated service object to interact with the Google Drive. This process mainly involves uploading the data file and ensuring any updates to the CSV are reflected in the Google Drive.

File Metadata and Upload Process: To upload the sensor data CSV file the script first prepares metadata for the file. Metadata being the file's name (e.g., sensor_data.csv) and other potential properties. The MediaFileUpload class from the googleapiclient.http module is then used to package the local file, specifying its MIMETYPE (text/csv).

The upload process uses the files().create() method of the Drive API to push the CSV file to the Google Drive. If the file doesn't already exist on Google Drive, it is created as a new entry, and the API returns the unique file ID for subsequent operations. The script captures and logs this file ID to facilitate additional updates.

File Updates and Overwriting Existing Files: For efficiency and accuracy, the script ensures that subsequent runs do not create duplicate files in the drive. Instead, it checks whether a file with the same name already exists in the drive. This is achieved by querying the Drive API using

the files().list() method with a filter condition based on the files name. If a match is found, the script retrieves the file ID of the existing file.

When the CSV file needs to be updated, the script uses the files().update() method, passing the file ID and new file content as arguments. This process replaces the old file with the updated version while retaining the same file ID, ensuring continuity for any external references to the file.

Data Consistency: The script ensures data consistency by verifying that file uploads or updates succeed before logging a success message via the terminal. If an upload or update fails, the script retries the operation or reports the failure via the terminal.

Below in Figure 12 shows how this function demonstrates metadata and media preparation, file existence checks, and API methods are utilized to interact with Google Drive.

```python
def upload_to_drive(service, csv_filename):
    """
    Uploads or updates the CSV file to Google Drive.
    It checks if the file already exists in the Google Drive. If it doesn't, it uploads the new file.
    If the file already exists, it updates the existing file with new data.

    Args:
        service (Resource): The authenticated Google Drive API service object.
        csv_filename (str): The name of the CSV file to upload or update.
    """
    # Metadata for the file being uploaded
    file_metadata = {'name': csv_filename}
    media = MediaFileUpload(csv_filename, mimetype='text/csv')  # Prepares the CSV file for upload

    # Checks if the file already exists on Google Drive
    query = f"name = '{csv_filename}'"
    results = service.files().list(q=query, fields="files(id, name)").execute()
    files = results.get('files', [])

    if not files:
        # Uploads file if it doesnt exist
        request = service.files().create(media_body=media, body=file_metadata, fields='id')
        file = request.execute()
        print(f"File uploaded successfully with ID: {file['id']}")
    else:
        # Updates the file if it exists
        file_id = files[0]['id']
        request = service.files().update(fileId=file_id, media_body=media).execute()
        print(f"File updated successfully with ID: {file_id}")
```

Figure 12: Upload to Google Drive

The Data Logging to CSV Using Python section highlighted the foundational step of collecting and organizing sensor data in a structured format. The integration with Google Drive extended the functionality by automating the upload of this CSV data, ensuring it remains accessible and secure in the cloud. At this stage in the report, the project has successfully implemented a fully operational system for acquiring, logging, and remotely storing the environmental data.

3.2.3 Dashboard Implementation

The dashboard component of the project was developed to provide real-time visualization of the data collected from the sensors. The goal was to create a user-friendly interface that allows users to monitor the environmental parameters. However firstly the data must be retrieved from the google drive to be processed.

3.2.3.1 Retrieving the Stored Data

The download_csv function below is responsible for downloading the CSV file from the Google Drive using its unique file ID and saving it locally. It starts by authenticating the user through the authenticate_google_drive() function as discussed before, which handles OAuth2 authentication and returns an authenticated Google Drive service object. The function then sends a request to Google Drive to retrieve the file in chunks, using an in-memory byte stream to temporarily store the data as it's being downloaded. Once the file is fully downloaded, the content is written to a local file, and the function returns the filename. This process ensures that the latest data is available for use in the dashboard. The script can be seen in Figure 13 below.

```
def download_csv(file_id, filename=CSV_FILENAME):
    """
    Downloads a CSV file from Google Drive using its file ID and saves it locally.
    Args:
    - file_id (str): The unique ID of the file on Google Drive.
    - filename (str): The name of the local file where the CSV will be saved (default is 'sensor_data.csv').

    Returns:
    - filename (str): The name of the saved file.
    """
    # Initialize Google Drive service to interact with the API
    drive_service = authenticate_google_drive()

    # Request the file from Google Drive
    request = drive_service.files().get_media(fileId=file_id)
    file = io.BytesIO()  # Using an in-memory byte stream to download the file
    downloader = MediaIoBaseDownload(file, request)

    # Download the file in chunks until fully downloaded
    done = False
    while not done:
        status, done = downloader.next_chunk()

    # Save the downloaded content into a local CSV file
    file.seek(0)    # Move back to the start of the byte stream
    with open(filename, 'wb') as f:
        f.write(file.read())

    return filename
```

Figure 13: Retrieving the Data from Google Drive

3.2.3.2 Processing the Retrieved Data

The load_data function is responsible for loading the downloaded CSV file into the Pandas DataFrame, which is a data structure that makes it easier to manipulate and analyse data. It utilizes the pd.read_csv() method from the pandas library to read the CSV file (which was previously downloaded using the download_csv function) and convert it into a Data Frame. Once the data is loaded, the function then returns the Data Frame, which can then be used for generating the visualizations or further analysis. This step is crucial for transforming the raw CSV data into a format suitable for real-time processing and graphing on the dashboard.

```
def load_data():
    """
    Loads the downloaded CSV file into a pandas DataFrame for further processing.

    Returns:
    - df (DataFrame): A pandas DataFrame containing the data from the CSV.
    """
    return pd.read_csv(CSV_FILENAME, encoding='utf-8')
```

Figure 14: Processing the Retrieved Data

3.2.3.3 Creating and Updating the Graphs

The create_figure function is responsible for generating visualizations using the Plotly library. It takes in a pandas Data Frame, and three parameters: the column names for the x-axis (x_col), y-axis (y_col), and a title for the plot. In this case, the x-axis represents the timestamps, while the y-axis can represent any of the 4 sensor readings, such as light level, temperature, humidity, or gas concentration. Using Plotly Express's px.line() function, a line plot is created based on the specified columns. This line plot represents the sensor data over time, with the respective title provided for context. The function returns the generated Plotly figure object, which can be rendered by the Dash framework for real-time updates in the dashboard. The layout of the dashboard is defined in the app.layout section which includes a title, multiple graphs, and an interval component. The graphs are used to display visualizations of the sensor data such as light level, temperature, humidity, and gas concentration over time. The dcc.Interval component triggers updates at regular intervals (every 60 seconds in this case) to ensure the displayed data is up to date.

```
def create_figure(df, x_col, y_col, title):
    """
    Creates a line plot figure using Plotly Express.
    Args:
    - df (DataFrame): The pandas DataFrame containing the sensor data.
    - x_col (str): The column name for the x-axis (typically 'Timestamp').
    - y_col (str): The column name for the y-axis (sensor readings like 'Light Level (%)').
    - title (str): The title of the plot.

    Returns:
    - fig (plotly.graph_objects.Figure): A Plotly figure object for displaying the chart.
    """
    return px.line(df, x=x_col, y=y_col, title=title)


# Initialize Dash
app = Dash(__name__)

# Define the layout of the app
app.layout = html.Div([
    html.H1("Sensor Data Dashboard"),  # Title of the dashboard
    # Interval component to refresh the data every 60 seconds
    dcc.Interval(id='interval-component', interval=60*1000, n_intervals=0),

    # Graphs
    dcc.Graph(id='light-level-over-time'),
    dcc.Graph(id='temperature-over-time'),
    dcc.Graph(id='humidity-over-time'),
    dcc.Graph(id='gas-concentration-over-time')
])


@app.callback(
    [Output('light-level-over-time', 'figure'),
     Output('temperature-over-time', 'figure'),
     Output('humidity-over-time', 'figure'),
     Output('gas-concentration-over-time', 'figure')],
    [Input('interval-component', 'n_intervals')]
)
```

Figure 15: Creating the Graphs

The update_graphs function, which is linked to the interval component, serves as a callback that is triggered by the interval timer. This function is responsible for downloading the latest CSV data from the Google Drive, loading it into the pandas Data Frame, and updating the figures in the dashboard. The function calls the download_csv function to fetch the latest file and then processes the data with the load_data function. It generates four different figures (one for each

sensor) by calling create_figure with the appropriate arguments. Finally, the function returns the updated figures, which are displayed in the respective graphs in the Dash dashboard. [9]

```python
def update_graphs(n):
    """
    Callback function to update the graphs with the latest data every time
    the Interval component triggers (every 60 seconds).
    This function downloads the latest CSV, loads it into a DataFrame,
    and updates the four graphs accordingly.
    """
    # Download the latest data file from Google Drive
    download_csv(FILE_ID)

    # Load the newly downloaded data into a pandas DataFrame
    df = load_data()

    # Create the four graphs based on the latest data
    light_fig = create_figure(df, 'Timestamp', 'Light Level (%)', "Light Level Over Time")
    temp_fig = create_figure(df, 'Timestamp', 'Temperature (C)', "Temperature Over Time")
    humidity_fig = create_figure(df, 'Timestamp', 'Humidity (%)', "Humidity Over Time")
    gas_fig = create_figure(df, 'Timestamp', 'Gas Concentration (MQ135)', "Gas Concentration Over Time")

    # Return the updated figures for each graph
    return light_fig, temp_fig, humidity_fig, gas_fig
```

Figure 16: Updating Graphs

3.2.3.4 Running the Dash Server

Finally, the Dash app is run using the app.run_server() function. This starts the web server locally, enabling users to access the dashboard via a web browser. The debug=True flag allows for live updates and error tracking during development, while host="127.0.0.1" binds the server to the local pi.

```python
# Run the Dash app server to launch the web dashboard
if __name__ == '__main__':
    app.run_server(debug=True, host="127.0.0.1")
```

Figure 17: Running the Dash Server

Below shows what one of the graphs look like on the website.

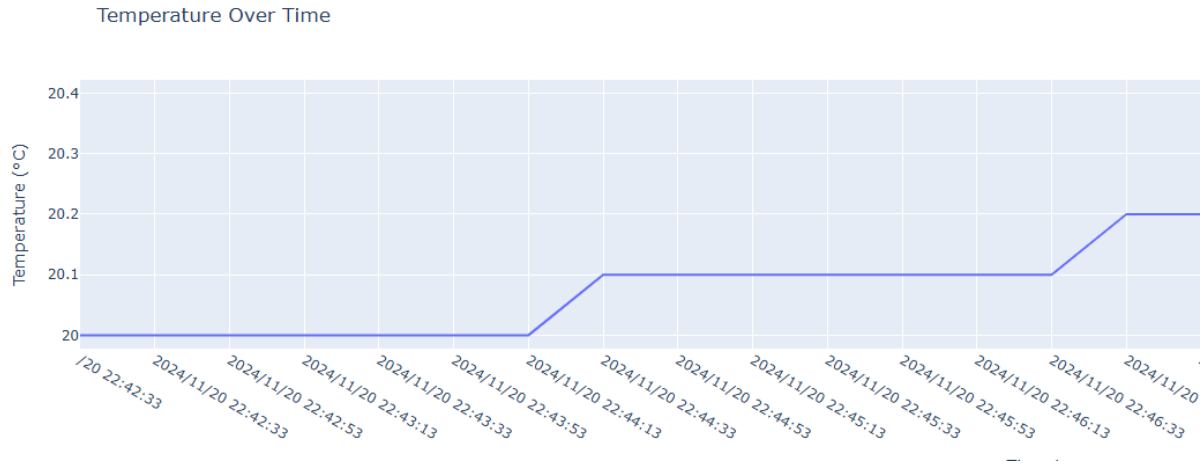

Figure 18: Graph on Dash Website

3.2.4 Setting Up Auto-Run on the Raspberry Pi

To ensure that both the data logging and dashboard scripts run automatically when the Raspberry Pi powers on, it is essential to set up an auto-run feature. This feature will make the system more reliable and allow for continuous operation without needing manual intervention after power cycles or reboots. The team found that the most effective way to achieve this on a Raspberry Pi for our project is by using systemd.

To set up an auto-run feature, the first step is to create a custom systemd service for both the data logging script and the dashboard script. The systemd service is a configuration file that will define how and when a script should be executed on boot. This service can be created in the /etc/systemd/system/ directory. For this project the team made a service file named sensor_dashboard.service, to manage the execution of the dashboard, and a file named dataToCSVToGD.service, to manage the data logging/uploading. Its important to note that cron could also be employed for scheduling tasks that need to be executed at specific intervals, however this project focuses on uploading the data as soon as it is received so system was chosen. Once the service file is created, it can be enabled so that it starts automatically when the Pi boots up using commands like “sudo systemctl enable sensor_dashboard.service”. [10]

By implementing the auto-run mechanisms, it can ensure that the data logging and dashboard scripts will function continuously without manual intervention, providing an automated and reliable system for long-term monitoring and data collection. This setup makes the project well-suited for unattended deployments where power reliability and operational continuity are key factors.

3.3 Problems Encountered During Development

Throughout the development of this project, multiple challenges arose that required troubleshooting and adaptation. Below are some of the problems encountered:

1. Google Drive Authentication Issues

Initially, there were challenges in authenticating with the Google Drive API. The OAuth2 flow required periodic token refreshing, and there were issues with the token file (token.pickle) being either expired or not stored correctly. Additionally, the file download functionality wasn’t working as expected due to incorrect configuration of Google API credentials or scope mismatches. To solve this the setup process was followed to ensure that the OAuth2 credentials file (credentials.json) and the Google Drive API scope were defined and pathed correctly especially on the raspberry pi. Once the configuration was correct, the file download process from Google Drive began working seamlessly.

2. LEDs Not Flashing During Data Collection

The LEDs connected to the sensors (photoresistor, MQ135, and DHT11) were not flashing as expected when new data was read. This was because the logic for controlling the LEDs wasn't integrated correctly into the data reading loop. The loop was modified so that the LEDs would be turned on briefly each time new data was read from the sensors. The logic was adjusted to ensure that once a sensor read was completed successfully, the corresponding LED would turn on for a short period before turning off again.

3. Dashboard Update Delay

Initially, the web dashboard was not updating in real-time with the new sensor data. This was primarily because the interval between data collection and updating the dashboard was too

long, causing noticeable delays when the data was displayed. The dcc.Interval component was used in Dash to set the data update frequency to every 60 seconds. This ensured that the dashboard was regularly updated with the latest data, even if the previous update had not fully completed leading to multiple data updates (varying with specific sensor reading delays). Additionally, the backend process was tuned to reduce the time it took for new data to be uploaded from the Raspberry Pi to Google Drive, and for the dashboard to load and render the new data.

In conclusion, careful planning played a crucial role in mitigating many of the common issues that could have derailed the project. By thoroughly researching the components and understanding their limitations beforehand, many challenges, for example sensor calibration errors were already anticipated. This proactive approach allowed for the implementation of solutions in advance, such as adding error-checking mechanisms for sensor data, or optimizing the Google Drive integration for seamless data uploads. Ultimately, the planning process streamlined the projects development, reducing the likelihood of encountering common problems and ensuring the project's overall success.

4. Analysis on the Projects Performance

The performance of this project was evaluated in terms of data collection speed, accuracy, and the systems responsiveness, which were essential to meet the project's objectives. Given the combination of sensor inputs and the integration with the cloud storage, it was crucial to ensure the system operated efficiently while maintaining its data integrity. This analysis will focus on how well the system performed in collecting and processing sensor data, the limitations faced, and the methods used to optimize the overall performance of the project.

4.1 Data Collection Speed and Accuracy

In the project, data collection intervals were tested at varying frequencies: 0.5-second, 1-second, 5-second, and 30-second intervals to observe how the system handled different data loads and performance factors. Initially, the sensors provided accurate data regardless of the interval chosen. However, performance aspects such as data transfer load and energy consumption became more apparent as the data collection interval decreased.

At faster data collection intervals, such as 0.5 seconds, the system had to handle a higher volume of data being transferred per second. This intern increased the data transfer load, as more data points were transmitted within the same period. For instance, at 0.5-second intervals, the system sent twice as much data per second compared to a 1-second interval. This additional load resulted in noticeable delays in data processing when manually observing the CSV file. However, this did not significantly impact the website's performance, as it only updates every 60 seconds.

On the other hand, longer intervals, such as 30 seconds, reduced the frequency of data transmission, lowering the data transfer load and allowing the system to process the data more efficiently. This made the system more resource-efficient and less prone to lag when observing the CSV file during the data processing.

While accuracy was maintained at all intervals, faster data collection rates, like 0.5-second intervals, placed a greater demand on system resources. More frequent data readings resulted in slower processing speeds and as expected, higher CPU temperatures. Conversely, slower

intervals, like 30 seconds, allowed continuous data collection and processing while maintaining a normal CPU temperature.

Given the nature of the project, a 30-second interval for data collection was deemed appropriate given the system's goal is to track environmental sensor data, faster intervals such as 0.5 seconds would provide excessive data points that wouldn't add significant value and would lead to unnecessary resource consumption. Slower intervals, such as 30 seconds, were sufficient to capture meaningful trends in environmental changes without overwhelming the system.

4.2 System Boot Time

The system's boot time was analysed to determine how quickly it becomes fully operational after power-up, focusing on when data logging begins and the web dashboard becomes accessible. On average, the system took about 30 to40 seconds to boot completely, with the operating system initializing at around 20 seconds, data logging scripts starting by 30 seconds, and the dashboard becoming accessible by approximately 35–40 seconds. This timeframe is suitable for the project, as environmental monitoring does not require immediate startup, and the system resumes data collection seamlessly after power cycles. Its worth noting that potential improvements to reduce the boot time could be implemented by using a lighter weight version of the pi operating system, disabling unused services on the pi or optimizing network configurations. While these optimizations are not strictly necessary for this project, they could enhance the overall responsiveness in scenarios requiring quicker recovery from power interruptions.

4.3 Long-Term Data Storage

Long-term data storage was assessed to ensure the system could reliably retain collected environmental data over extended periods without significant performance degradation. The system stores data locally in a CSV file, which is periodically uploaded to Google Drive for secure offsite storage and easy access. This approach provides redundancy, reducing the risk of data loss due to potential hardware failure. CSV format was chosen for its simplicity and compatibility with various data analysis tools, but over time, file size growth could lead to slower processing and increased storage requirements. To address this, the system could implement periodic archiving or splitting files by date ensuring manageable file sizes and faster query times. This strategy would balance the data retention needs with system performance, making it well-suited for long-term environmental monitoring applications.

At a data collection interval of 30 seconds, the system is estimated to store 2,880 entries daily. Although this suits for short testings to demonstrate the project, this delay could ultimately rise to approximately 5 minutes or more. With a 5 minute configuration, the stored data would be spread out enough to track trends and changes throughout the day, while avoiding excessive data generation that could potentially strain storage or processing capabilities.

4.4 Energy Efficiency

Energy efficiency is a critical consideration for the system as its designed to operate continuously, especially in long-term environmental monitoring applications. The project involves a Raspberry Pi running data collection scripts, processing sensor readings, and hosting a web dashboard, all of which contribute to the overall energy consumption. The energy efficiency of the system was evaluated based on its power usage during operation, including idle states, active data logging, and web server activity. [11]

The Raspberry Pi model used in the project typically consumes between 3W and 5W under normal operation. Measurements indicated that:

- **Idle State:** When the Raspberry Pi 3 B is powered on but not actively engaged in processing or data collection, it consumes approximately 260 mA (1.4 W).
- **Web Server Activity:** During web requests simulated with the Apache Bench test with 100 requests and a concurrency of 10, the power consumption rises to 480 mA (2.4 W) due to increased processing activity to handle incoming traffic.
- **Maximum Load:** Under a synthetic load of 400% CPU usage using the stress --cpu 4 utility, the power consumption peaks at 730 mA (3.7 W).

These measurements provide a range of power consumption scenarios, from idle to high computational demand, highlighting the energy profile of the system under typical and peak operating conditions.

Energy Cost Estimation

- **Average Operating Power:** The system operates between 1.4W and 3.7W, depending on the state. Assuming an average of 2.4W during active usage, the daily energy consumption is approximately 57.6Wh (2.4W x 24 hours).
- **Monthly Energy Use:** About 1.73kWh, costing approximately €0.26 per month at an electricity rate of €0.15 per kWh.
- **Annual Energy Use:** The yearly cost would be approximately €3.15, making it an affordable solution for continuous operation.

Optimizations for improving the system's energy efficiency may focus on reducing power consumption during idle states, optimizing task schedules, and exploring alternative hardware or power sources. Idle state optimization can be achieved by disabling unnecessary services and adjusting the CPU frequency, potentially lowering the base consumption from 1.4W. For future iterations, integrating a renewable energy source such as solar panels would allow for off-grid operation, and also enhance the sustainability of the project.

4.5 Meeting the Project Objectives

The project successfully met its primary objectives set out by the team of creating a real-time environmental monitoring system, collecting sensor data, and presenting it through a user-friendly web dashboard. The system was able to reliably collect and log data from the sensors, process it efficiently, and upload it for offsite storage, fulfilling the core set out objectives. Furthermore, the system performed well within the constraints of power consumption and boot time, making it more than suitable for long-term deployment in environmental monitoring scenarios.

One of the standout successes that was observed was the system's ability to maintain accurate data collection even under varying data collection intervals, ensuring that the performance did not degrade as the data load increased. This exceeded the teams expectations, as the system remained stable and responsive, even at the faster data collection rates.

While there were areas for potential improvement, such as further reducing boot time or further optimizing energy consumption during idle periods, the overall system demonstrated robust

performance in real-world testing scenarios. The project also met its sustainability goals by offering energy-efficient solutions and providing an adaptable framework for future enhancements, such as potentially implementing renewable energy sources.

In conclusion, the project was a success in terms of meeting the teams objectives, and with additional optimizations, it has the potential to evolve into an even more efficient and scalable solution for environmental monitoring.

5. Cost Effectiveness and Commercialization Plan

To assess the cost-effectiveness of this project as a finished product, it's important to consider both the hardware and development costs. The core components required for the system include a Raspberry Pi 3 B, sensors (DHT11, MQ135, photoresistor), a real-time clock (RTC) module, and other accessories like LEDs, power supply, and microSD card for storage. The Raspberry Pi 3 B costs around €35, while the sensors and other components total approximately €20-€30, depending on sourcing and quantity. The total hardware cost for a single unit would be around €60-€70. Additionally, if mass production were considered, the cost could decrease due to bulk purchasing. In terms of time, assuming a few weeks for development, coding, and testing, the development cost would depend on the hourly rate for the developer. If one estimates 30 hours of development at €30/hour, that would add another €900 to the project cost.

To turn the project into a commercially viable product however, the focus would need to shift to optimizing the manufacturing process, reducing hardware costs, and ensuring scalability. One option to reduce costs would be to potentially switch to a simpler microcontroller like the Raspberry Pi Zero, which would lower the power consumption and reduce the overall cost of the unit. Another potential area for saving is the integration of renewable energy sources as previously mentioned, like solar panels, which could eliminate electricity costs for off-grid applications. In terms of selling the product, a subscription model for cloud storage or additional data analysis services could also be implemented to help generate recurring revenue.

By focusing on minimizing manufacturing costs, optimizing hardware for energy efficiency, and exploring added-value services, the product could be made cost-effective for both production and sale.

6. Conclusion

In conclusion, this project successfully achieved its objectives of designing and implementing an environmental monitoring system that collects and logs sensor data, visualizes it through a web dashboard, and stores it securely for long-term use. Throughout the development process, careful planning and optimizations were applied to balance system performance, accuracy, and cost-effectiveness. The system's ability to handle different data collection intervals and manage energy consumption demonstrated its potential for real-time environmental monitoring in various settings.

The project also highlighted the importance of efficient hardware selection, project planning, data management strategies, and system responsiveness. While the Raspberry Pi 3 B provided sufficient processing power, there are opportunities to reduce costs and improve energy efficiency by exploring alternative microcontrollers and sensor components. Optimizing boot times and task scheduling further enhanced system performance, ensuring reliable operation without unnecessary delays.

From a commercial perspective, the project showed that with thoughtful optimizations and the use of cost-effective components, the system could be scaled and sold as a product at a competitive price, making it viable for various consumer and industrial applications. By focusing on key areas such as hardware simplification, energy efficiency, and cloud storage management, the project can be transformed into a marketable solution for real-time environmental data monitoring.

Overall, the project not only met but exceeded the teams expectations in terms of functionality, reliability, and potential for future improvements, positioning it as a valuable tool for both personal and professional environmental monitoring.

Appendices

Team Contributions

Member	Role	Description
Cathal	Arduino	Programming and configuring the Arduino to collect data from the sensors. This included writing code to read sensor values, controlling LEDs for feedback, and ensuring data integrity before transmission to the Raspberry Pi.
Alan	CSV and Setup	Setting up the Raspberry Pi environment and implementing the data logging system. Creating the functionality for saving sensor readings to a CSV file, managed the initial system configuration, and ensured smooth operation of the hardware and software components.
Jamie	Google drive and Dashboard	Handling the integration with the Google Drive API for data uploading. Designing the Plotly Dash dashboard for data visualization. Setting up secure authentication, managing cloud storage, and creating interactive graphs to display sensor readings in real-time.

Codebase

The code snippets included throughout the report effectively illustrate the overall structure and functionality of the program. Therefore, the team has opted not to duplicate the full code in the appendices. If access to the complete code is required, it can be made available upon request.

Hardware Components

Component	Specification	Purpose
Raspberry Pi 3 Model B	1 GB RAM, Quad-Core 1.2 GHz CPU	Main computing unit
DHT11 Sensor	Temperature and Humidity Sensor	Environmental data collection
MQ135 Sensor	Gas Sensor for Air Quality	Air quality monitoring
Photoresistor	Analog Light Sensor	Light intensity measurement
Micro SD Card	120 GB	Operating system and data storage

Bibliography

[1] "DHT11 Temperature and Humidity Sensor," Irish Electronics, [Online]. Available: https://www.irishelectronics.ie/product/DHT11-Temperature-and-Humidity-Sensor. [Accessed October 2024].

[2] "MQ-135 Gas Sensor Module," MakerShop, [Online]. Available: https://makershop.ie/MQ-135. [Accessed October 2024].

[3] "RTC DS1307 Real Time Clock Module," Irish Electronics, [Online]. Available: https://www.irishelectronics.ie/product/RTC-DS1307-Real-Time-Clock-Module. [Accessed October 2024].

[4] "Nova SDS011 High Precision Laser Dust Sensor," TinyTronics, [Online]. Available: https://www.tinytronics.nl/en/sensors/air/dust/nova-sds011-high-precision-laser-dust-sensor. [Accessed October 2024].

[5] Parthea, "Google API Client Library for Python Docs," Google, [Online]. Available: https://github.com/googleapis/google-api-python-client/blob/main/docs/README.md. [Accessed November 2024].

[6] Google, "Google Cloud Console," Google, [Online]. Available: console.cloud.google.com.

[7] A. Fadheli, "How to Use Google Drive API in Python," thepythoncode, [Online]. Available: https://thepythoncode.com/article/using-google-drive--api-in-python.

[8] Google, "google-api-python-client," Google, [Online]. Available: https://googleapis.github.io/google-api-python-client/docs/oauth.html.

[9] Dash, "Primer on Plotly Graphing Library," Dash, [Online]. Available: https://dash.plotly.com/dash-core-components/graph.

[10] MUD, "How to Use systemd to Launch Programs," MUD, [Online]. Available however the link from word may need to be copy pasted into a browser to work: https://www.makeuseof.com/what-is-systemd-launch-programs-raspberry-pi/#:~:text=To%20do%20this%2C%20type%20sudo,as%20per%20your%20provided%20instructions

[11] "Power Consumption Benchmarks," Raspberry Pi Dramble, [Online]. Available: https://www.pidramble.com/wiki/benchmarks/power-consumption.