HAW HAMBURG

BACHELORTHESIS
Igor Kojanin

Software Development for a Cloud-Connected IoT Air Purifier with Real-Time Air Quality Monitoring and Mobile Application Integration

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Igor Kojanin

Software Development for a Cloud-Connected IoT Air Purifier with Real-Time Air Quality Monitoring and Mobile Application Integration

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Paweł Buczek
Second examiner: Dr. H. Matthias Reiche

Day of delivery: 27. August 2024

Igor Kojanin

Title of Thesis

Software Development for a Cloud-Connected IoT Air Purifier with Real-Time Air Quality Monitoring and Mobile Application Integration

Keywords

Indoor Air Quality (IAQ), Air Purifiers, Environmental Sensors, Internet of Things (IoT), Embedded Systems, Cloud Infrastructure, Mobile Application Development

Abstract

This thesis develops an intelligent air purifier that improves indoor air quality (IAQ) through real-time monitoring and automation. Sensors detect pollutants, and a microcontroller adjusts the purifier's operation. Cloud infrastructure supports data storage and communication, with a mobile app enabling remote control.

Igor Kojanin

Thema der Arbeit

Softwareentwicklung für einen Cloud-verbundenen IoT-Luftreiniger mit Echtzeit-Luftqualitätsüberwachung und Integration mobiler Anwendungen

Stichworte

Innenraumluftqualität (IAQ), Luftreiniger, Umweltsensoren, Internet der Dinge (IoT), eingebettete Systeme, Cloud-Infrastruktur, Entwicklung mobiler Anwendungen

Kurzzusammenfassung

In dieser Arbeit wird ein intelligenter Luftreiniger entwickelt, der die Luftqualität in Innenräumen durch Echtzeitüberwachung und Automatisierung verbessert. Sensoren erkennen Schadstoffe, und ein Mikrocontroller regelt den Betrieb des Luftreinigers. Eine Cloud-Infrastruktur unterstützt die Datenspeicherung und -kommunikation, und eine mobile App ermöglicht die Fernsteuerung.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, indoor air quality has become an increasingly important issue as people spend the vast majority of their time indoors. In Central Europe, for instance, the average person now spends approximately 90 percent of their time inside [9]. This shift has raised concerns about the adverse effects of poor indoor air quality on human health. Air pollution within homes, offices, and other enclosed spaces has been linked to a variety of health problems, including respiratory issues, allergies, and chronic conditions [40]. According to the World Health Organization (WHO), air pollution is a major environmental threat, contributing to millions of premature deaths each year [52]. Ensuring clean air indoors is essential for maintaining health and well-being.

As awareness of indoor air pollution has grown, air purifiers have gained prominence as a practical solution to mitigate health risks. In fact, indoor air pollution can often be more severe than outdoor pollution, making the need for effective air purification systems even more critical [44]. This thesis presents the development of an intelligent air purifier system, specifically designed for indoor use, that integrates advanced sensor technology, cloud-based services, and a user-friendly mobile application.

The air purifier system is engineered to automatically monitor and improve indoor air quality in real-time. Equipped with multiple environmental sensors, it can detect pollutants, humidity, temperature, and other air quality metrics. The system dynamically adjusts its operation, regulating fan speed to optimize air purification based on the detected conditions. Additionally, it allows users to set schedules and customize the purifier's behavior to meet their individual preferences.

At the heart of this system is an embedded microcontroller that interfaces with the sensors, processes the collected data, and controls the purifier's components. The embedded system communicates with a cloud-based server, which handles data storage and remote device control. The server is connected to a mobile application, enabling users to receive real-time feedback on air quality and configure their devices with ease.

This thesis aims to provide a comprehensive overview of the development process, focusing on key components of the system: the embedded software, the cloud infrastructure, and the mobile application. By detailing each element, this work highlights the challenges encountered and the solutions implemented to create an efficient, reliable, and user-friendly air purifier system for indoor environments.

1.1 Objectives

The primary objective of this thesis is to develop the software for an intelligent indoor air purification system, with a focus on real-time air quality monitoring and control. This work specifically concentrates on the software side, leaving out the mechanical design, hardware, and printed circuit board (PCB) development. The main goals are as follows:

- **Air Quality Monitoring:** Develop software that manages sensors to detect indoor air pollutants such as Particulate Matter (PM2.5), Volatile Organic Compounds (VOCs), Carbon Dioxide (CO_2), Carbon Monoxide (CO), humidity, and temperature, continuously providing accurate real-time data to the user.
- **User Control and Scheduling:** Implement functionality that enables users to customize the air purifier's behavior, set operational schedules, and control the system remotely via a mobile application.
- **User-Friendly Mobile Application:** Design and develop a mobile application that provides an intuitive interface for monitoring air quality, configuring system settings, and receiving real-time updates.

These objectives aim to deliver a comprehensive and user-friendly software solution for improving indoor air quality through real-time monitoring and remote control.

1.2 Structure of the Thesis

This thesis is divided into the following core topics, each addressing a key component of the air purifier system:

- **Chapter 2: Requirements** defines the functional and non-functional requirements for the system, focusing on the embedded system, cloud infrastructure, and mobile application.
- **Chapter 3: Background** provides the necessary background information, explaining the key concepts, technologies, and context relevant to the development of the air purifier system.
- **Chapter 4: Embedded System Development** discusses the design and implementation of the embedded software responsible for sensor integration, data processing, and device control.
- **Chapter 5: Cloud Infrastructure Development** covers the backend infrastructure, including data storage, communication management, and the API that supports interaction between the mobile app and the embedded system.
- **Chapter 6: Mobile Application Development** covers the creation of the mobile application, focusing on real-time data display and device control.
- **Chapter 7: Testing and Validation** presents the methods used to test the system, ensuring that it meets the functional and non-functional requirements.
- **Chapter 8: Future Work** outlines potential improvements and extensions that could be made in future iterations of the system.
- **Chapter 9: Conclusion** summarizes the achievements of the project, reflecting on its impact and overall significance in the context of indoor air quality improvement.

This structure provides a logical flow from the technical development of the system to its real-world application and evaluation. Through this work, the thesis aims to demonstrate the feasibility and effectiveness of a smart air purifier system that enhances indoor air quality and provides a modern, user-friendly experience.

2 Requirements Analysis

This chapter outlines the functional and non-functional requirements for the air purifier system. The requirements are categorized into three major components: the embedded system, the cloud infrastructure, and the mobile application. Each component is essential for the overall functionality and performance of the air purifier system. The requirements ensure that the system operates efficiently, securely, and reliably while providing a seamless user experience.

2.1 Embedded System Requirements

Identifier	Description
REQ1.0	**Wi-Fi Connectivity:** The air purifier must connect to a Wi-Fi network to enable communication with remote platforms, such as a mobile application or cloud server, allowing users to remotely control and monitor the device.
REQ1.1	**Continuous Air Quality Monitoring:** The air purifier must continuously monitor indoor air quality using sensors that measure pollutants such as Volatile Organic Compounds (VOCs), carbon dioxide (CO2), carbon monoxide (CO), particulate matter (PM2.5), as well as environmental factors like temperature and humidity.
REQ1.2	**Real-time Data Transmission:** The air purifier must transmit sensor data, including air quality metrics and the calculated Indoor Air Quality Index (IAQI), to a mobile application in real-time, providing users with timely updates on their indoor environment.
REQ1.3	**Scheduled Operation:** The air purifier must operate based on predefined cleaning schedules, automatically adjusting fan speeds at specified intervals to align with user preferences.
REQ1.4	**Automated Fan Speed Adjustment:** The air purifier must automatically adjust fan speed according to real-time air quality measurements, ensuring efficient and responsive operation that maintains healthy indoor air conditions.

Table 2.1: Functional Requirements of the Air Purifier

Identifier	Description
REQ2.0	**Automatic Recovery:** The system must recover automatically from network disruptions, without requiring user intervention.
REQ2.1	**Scalability:** The embedded system should be designed to accommodate additional sensors or functionalities in future upgrades without requiring significant architectural changes.

Table 2.2: Non-Functional Requirements of the Air Purifier

2.2 Cloud System Requirements

Identifier	Description
REQ3.0	**Secure and Reliable Communication:** The cloud must facilitate secure and reliable communication between the air purifiers and the mobile application.
REQ3.1	**Real-time Communication:** The cloud must support real-time communication, allowing immediate updates and control commands to be sent and received without significant delays.
REQ3.2	**Data Storage:** The cloud must store data from the air purifiers, including device settings, and user information.
REQ3.3	**API for External Applications:** The cloud must provide an API that allows external applications, such as mobile apps, to interact with the system.

Table 2.3: Functional Requirements of the Cloud System

Identifier	Description
REQ4.0	**Scalability:** The server must be designed to scale as the number of devices and users increases, ensuring that performance remains consistent even under high demand.
REQ4.1	**Performance:** The server must process requests and deliver responses with minimal latency, ensuring that users experience real-time interactions with the system. It must handle large volumes of data efficiently, ensuring that retrieval and storage operations do not degrade performance as the system grows.

Table 2.4: Non-Functional Requirements of the Cloud System

2.3 Mobile Application Requirements

Identifier	Description
REQ5.0	**Wi-Fi Configuration:** The mobile application must allow users to configure Wi-Fi credentials for the air purifier device, enabling it to connect to a local network.
REQ5.1	**Real-time Data Display:** The application must retrieve and display real-time sensor data from the air purifier, allowing users to monitor indoor air quality.
REQ5.2	**Scheduling:** The application must allow users to create and manage schedules for the air purifier, specifying operational parameters such as start time, end time, and fan speed.

Table 2.5: Functional Requirements of the Mobile Application

Identifier	Description
REQ6.0	**Usability:** The app interface must be intuitive and responsive, making complex tasks such as Wi-Fi setup and scheduling simple for users to perform.
REQ6.1	**Scalability:** The application must be designed to accommodate additional features or integrations in future updates without requiring major architectural changes.
REQ6.2	**Security:** The application must ensure that sensitive data, such as Wi-Fi credentials and user information, is transmitted securely using encryption. The app must protect against unauthorized access, ensuring that only authenticated users can configure and control their devices.
REQ6.3	**Reliability:** The application must be reliable, ensuring consistent performance even in scenarios with intermittent network connectivity or temporary server unavailability.

Table 2.6: Non-Functional Requirements of the Mobile Application

2.4 Conclusion

The requirements outlined in this chapter provide a comprehensive framework for the development of the air purifier system. By defining both functional and non-functional requirements across the embedded system, cloud infrastructure, and mobile application, this chapter ensures that the system will operate effectively, securely, and reliably, while providing users with an intuitive and powerful tool to manage their indoor air quality. These requirements will serve as the foundation for the subsequent development and testing phases of the project.

3 Background

3.1 Air Purifiers and Their Role in Improving Indoor Air Quality

Air purifiers are essential devices designed to improve indoor air quality (IAQ) by filtering out pollutants, particulate matter, and harmful gases from the air. With growing awareness of indoor air pollution, which can often be more severe than outdoor pollution, air purifiers have gained significant attention as a means to mitigate health risks. The importance of air purifiers becomes apparent considering that most people spend approximately 90% of their time indoors. Indoor air pollutants originate from both outdoor sources and indoor activities such as cooking, cleaning, and the use of household products. Common contaminants include particulate matter (PM2.5 and PM10), volatile organic compounds (VOCs), mold, pollen, and various gases [46, 44].

Air purifiers employ several technologies to remove contaminants from the air. Mechanical filtration, such as High-Efficiency Particulate Air (HEPA) filters. Other technologies include ionization-based purifiers, which charge airborne particles, as well as electrostatic purifiers that use electrostatic forces to trap particles. Some advanced models also utilize activated carbon filters to remove gaseous pollutants or ultraviolet (UV) light to eliminate biological contaminants like bacteria and viruses [44].

3.2 Internet of Things (IoT)

The Internet of Things (IoT) represents a transformative technological shift, where everyday physical objects are embedded with sensors, actuators, and communication technologies that allow them to connect, exchange data, and interact over the internet. This paradigm extends connectivity beyond traditional devices like computers and smartphones to encompass a broad array of "things"—ranging from household appliances and

vehicles to industrial machinery—enabling remote monitoring and control. With rapid advancements in technology, IoT is creating vast opportunities for innovative applications that can significantly improve daily life. It has attracted considerable attention from researchers and practitioners worldwide for its potential to enhance connectivity and quality of life. The integration of ubiquitous devices into IoT systems is driving the development of cutting-edge applications in fields such as communication, security, and localization [14, 53].

3.3 Particle Photon

The Particle Photon is a powerful IoT development kit designed to build connected products. It integrates a Broadcom BCM43362 Wi-Fi chip and an STM32F205RGY6 ARM Cortex-M3 microcontroller, offering robust computational power and reliable wireless connectivity in a compact form factor. The Photon supports 802.11b/g/n Wi-Fi standards, enabling seamless cloud integration and interaction with various IoT platforms. With 1MB of flash memory and 128KB of RAM, it is well-suited for handling embedded applications. The device features a real-time operating system (FreeRTOS) and supports software enabled access point (SoftAP) setup, simplifying network configuration for users [31].

3.4 HDC1080 Temperature and Humidity Sensor

The HDC1080 is a low-power, high-accuracy digital humidity sensor with an integrated temperature sensor from Texas Instruments. It operates over a wide supply voltage range of 2.7V to 5.5V and is ideal for low-power applications, such as Heating, Ventilation, and Air Conditioning (HVAC) systems, smart thermostats, medical devices, and wireless sensors. The sensor offers a relative humidity accuracy of $\pm 2\%$ and a temperature accuracy of $\pm 0.2°C$, making it a reliable choice for precise environmental monitoring. The HDC1080 supports 14-bit measurement resolution and utilizes an I2C interface for communication [43].

3.5 SGP30 VOC and CO_2 Sensor

The SGP30, developed by Sensirion, is a digital multi-pixel gas sensor specifically designed for indoor air quality monitoring. This sensor provides outputs for Total Volatile Organic Compounds (TVOC) and CO2 equivalent (CO2eq) concentrations, making it an ideal choice for air purifiers, demand-controlled ventilation, and IoT applications. The SGP30 integrates a complete sensor system on a single chip, featuring a digital I2C interface and robust gas sensing capabilities.

One of the key advantages of the SGP30 is its long-term stability and resistance to contaminating gases, which ensures reliable performance in real-world applications. The sensor operates with low power consumption, drawing only 48 mA at 1.8V, making it suitable for energy-efficient applications. The compact 6-pin DFN package (2.45 x 2.45 x 0.9 mm) allows for easy integration into space-constrained devices. The SGP30's ability to accurately measure air quality over time makes it a vital component for enhancing indoor environmental conditions [41].

3.6 ME2-CO-14x50-C Carbon Monoxide Sensor

The ME2-CO-14x50-C is an electrochemical carbon monoxide (CO) sensor developed by Zhengzhou Winsen Electronics Technology Co., Ltd. This sensor operates on the principle of fuel cell technology, where carbon monoxide and oxygen undergo corresponding redox reactions on the working and counter electrodes, generating a current proportional to the CO concentration. The ME2-CO sensor boasts key features such as low power consumption, high precision, and high sensitivity, making it suitable for a wide range of applications, including home safety, underground garages, and generator monitoring. It offers a measurement range of 0-1000 ppm with a resolution of 1 ppm. Additionally, it is designed to operate within a temperature range of -20°C to 50°C and a humidity range of 15% to 90% RH, ensuring reliable performance in diverse environmental conditions [55].

3.7 ZPH02 Particulate Matter Sensor

The ZPH02 sensor, developed by Zhengzhou Winsen Electronics Technology Co., Ltd., integrates advanced PM2.5 detection technology. It is capable of detecting particulate matter with diameters greater than or equal to 1 μm, making it an ideal choice for applications such as air purifiers, HVAC systems, and air refreshers.

Key features of the ZPH02 include high sensitivity, good consistency, and long-term stability. It provides multiple interface outputs, including PWM and UART, which allows for flexible integration with different systems. The sensor is powered by a 5V DC supply and operates within a temperature range of 0°C to 50°C and humidity levels up to 90% RH. Before delivery, the ZPH02 undergoes calibration and aging to ensure high performance and accuracy [54].

3.8 Azure SQL Database

Azure SQL Database is a fully managed Platform as a Service (PaaS) that provides a highly reliable and scalable data storage solution, optimized for modern cloud applications. Running on the latest stable version of the Microsoft SQL Server engine, it ensures 99.99% availability and incorporates automatic updates, backups, and patching. With built-in advanced monitoring, intelligent performance tuning, and comprehensive security measures—including data encryption and threat detection—Azure SQL Database delivers high performance and security with minimal user intervention. These features make it a robust choice for cloud applications that require flexibility, efficiency, and scalability [22].

3.9 Azure App Service

Azure App Service is a fully managed platform as a service (PaaS) offering from Microsoft, designed to host web applications, REST APIs, and mobile back ends. It supports a wide range of programming languages and frameworks, including .NET, Java, Node.js, PHP, and Python, and it allows developers to build and deploy applications on both Windows

and Linux environments. App Service provides a scalable and secure platform with built-in features such as load balancing, autoscaling, and continuous integration. Additionally, it integrates seamlessly with development tools like Visual Studio and GitHub [18].

3.10 REST API

A REST API (Representational State Transfer Application Programming Interface) is a web API that adheres to the REST architectural style, which offers a lightweight and scalable way to connect applications and components in a microservices architecture. First defined by Dr. Roy Fielding in 2000, REST APIs operate using standard HTTP methods such as GET, POST, PUT, and DELETE to perform CRUD (Create, Read, Update, Delete) operations on resources. REST APIs are characterized by key principles, including a uniform interface, statelessness, client-server decoupling, cacheability, and support for layered system architectures. Each request contains all the necessary information for processing, and resources can be cached to improve performance [18].

3.11 MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight publish/subscribe messaging protocol designed for low-power devices and limited bandwidth scenarios. It is highly suitable for Internet of Things (IoT) applications, where resources are constrained and reliable communication is essential. MQTT operates on the principle of publishing messages to topics, and clients can subscribe to these topics to receive messages [23].

One of the key advantages of MQTT is its lightweight nature, which makes it well-suited for resource-constrained devices such as microcontrollers. The protocol's publish/subscribe model decouples data producers and consumers, enabling scalable and flexible communication. MQTT also provides reliable message delivery by supporting different levels of Quality of Service (QoS), ensuring that messages are delivered as required by the application. Security is also a priority with MQTT, as it facilitates the encryption of messages using TLS (Transport Layer Security) and allows for modern client authentication methods, such as OAuth, making it a secure choice for IoT environments [25].

3.12 HTTP

Hypertext Transfer Protocol (HTTP) is the core protocol that facilitates data exchange on the Web. It operates as a client-server protocol, where the client (usually a web browser) sends requests, and the server responds with the requested resources such as HTML documents, images, and videos. HTTP is a stateless protocol, meaning that each request is treated independently, although session management can be implemented through mechanisms like cookies. HTTP communication typically runs over TCP, ensuring reliable transmission of data between clients and servers.

Security has become a critical concern in HTTP-based systems, particularly with the widespread adoption of HTTPS, which encrypts data transmission using Transport Layer Security (TLS). Additionally, HTTP supports various authentication mechanisms, allowing servers to control access to resources. With its simplicity, flexibility, and adaptability, HTTP remains one of the most widely used protocols for web communication and continues to support a vast array of web applications [24].

3.13 SSL/TLS

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are cryptographic protocols that enable secure communication over the internet. They work by establishing an encrypted link between a client (such as a web browser) and a server, ensuring that sensitive data, such as login credentials or credit card details, can be transmitted securely. SSL/TLS uses certificates to authenticate the identity of the website and to encrypt data exchanged during the session. These certificates contain a public key, which allows the client to encrypt data sent to the server, and a private key, which the server uses to decrypt the received data [42].

4 Embedded Software Implementation

This chapter covers the embedded software that enables the air purifier's functionality. The embedded system is responsible for collecting real-time air quality data from various environmental sensors, managing Wi-Fi connectivity, and ensuring seamless communication with a mobile application through the MQTT protocol. Running on a microcontroller, the embedded system bridges the physical sensors with the digital world, enabling data acquisition and control of the air purifier's operations.

Developing this embedded system presents a range of challenges, particularly in ensuring reliability and robustness in real-world scenarios, where network interruptions, and changing environmental conditions can occur. The system must be designed to dynamically configure network settings, process sensor data, maintain stable communication channels, and control the air purifier's operation based on real-time inputs and scheduled tasks.

This chapter delves into design and implementation of the embedded system, covering the setup of network connectivity, sensor integration, secure data transmission, and automated control logic. It will cover the calculation of the Indoor Air Quality Index (IAQI), and the handling of cleaning schedules. By the end of this chapter, a comprehensive overview of the embedded system's structure, development challenges, and implemented solutions will be presented, demonstrating its pivotal role in ensuring the air purifier operates effectively and reliably.

To provide a better understanding of the overall system architecture, Figure 4.2 illustrates the different components of the air purifier and their interactions. The green-shaded area represents the embedded system, which is the primary focus of this chapter. It includes the Particle Photon microcontroller, the sensors (Temperature, Humidity, VOC ,CO2, CO and PM2.5), and the fan. These components communicate through various interfaces such as I^2C, UART, PWM, and Analog. The rest of the system, including the cloud services and mobile application, will be covered in Chapter 5 and Chapter 6.

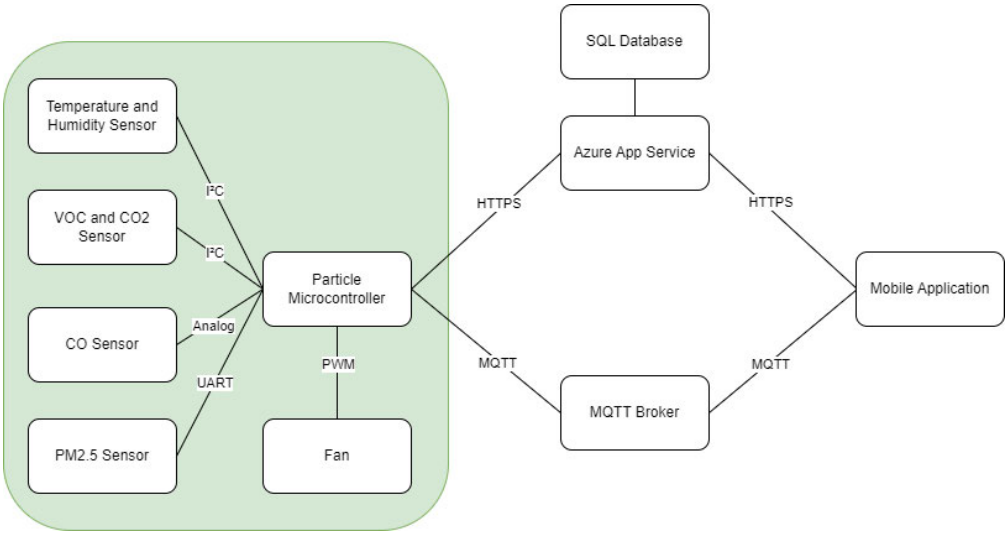

Figure 4.1: System Architecture of the Air Purifier. The green-shaded area highlights the embedded system, which is the focus of this chapter.

4.1 Embedded System Device Workflow

This section explains the general operation of the embedded system in the air purifier, detailing how it functions from Wi-Fi connectivity to sensor data processing, IAQI calculation, communication protocols, and scheduling operations. It outlines how the system handles data, communicates with the cloud and mobile application, and automatically adjusts the air purifier's functions to maintain optimal air quality.

The system begins by attempting to connect to a Wi-Fi network. If valid credentials are available, the microcontroller connects automatically. If not, the system enters "Listening Mode," enabling the mobile app to set Wi-Fi credentials via SoftAP. Once connected, the system continuously reads environmental sensor data, which is processed in real-time to adjust the air purifier's fan speed dynamically.

The system calculates the Indoor Air Quality Index (IAQI) from the sensor data, which is then communicated to the user through the mobile app. This IAQI value triggers automatic adjustments to the purifier's fan speed, helping to maintain optimal air quality.

Communication between the system, the cloud server, and the mobile application is handled through two primary protocols. MQTT is used for real-time transmission of sensor values to the mobile app, chosen for its efficiency and low latency. Additionally,

HTTP is used for periodic data transmission to the cloud server, where sensor data is stored, and schedules are retrieved.

Finally, the scheduling functionality allows users to configure the air purifier's operation through the mobile app. These schedules are stored in the cloud server and retrieved by the system during startup. Schedule updates made while the system is running are handled via MQTT, ensuring immediate adjustments to the air purifier's operation.

Figure 4.2 illustrates the system architecture of the air purifier, focusing on the device's workflow from initial setup through its operational state.

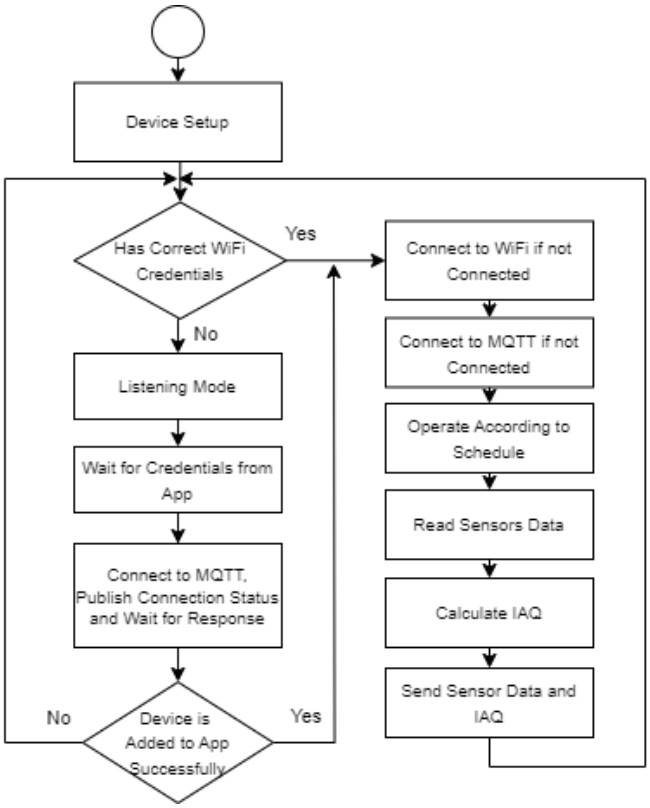

Figure 4.2: System Architecture of the Air Purifier. The diagram illustrates the workflow from device setup to normal operation, including Wi-Fi connection, MQTT communication, and sensor data processing.

4.1.1 Architectural Decisions: RTOS vs. Bare-Metal Programming

A crucial architectural decision during the development process was whether to use a Real-Time Operating System (RTOS) or bare-metal programming.

After carefully evaluating the system's requirements and constraints, bare-metal programming was selected for this implementation. This decision was informed by several key factors:

- **Task Complexity:** The tasks managed by the system are relatively straightforward and can be handled sequentially. Given the current complexity level, the overhead associated with an RTOS, which includes task scheduling and management, was considered unnecessary.
- **Resource Constraints:** The microcontroller powering the air purifier has limited memory and processing capabilities. By opting for bare-metal programming, the system could maximize resource efficiency and avoid the overhead introduced by an RTOS, thus better aligning with the microcontroller's resource limitations.
- **Scalability:** While bare-metal programming is suitable for the current implementation, the system architecture is designed to be flexible for future expansion. If the system's complexity increases or if more precise real-time control becomes necessary, migrating to an RTOS is a viable option that would facilitate more advanced task scheduling and management.

The choice of bare-metal programming enabled the development of a lightweight and efficient system that meets the air purifier's current performance requirements. Furthermore, the architecture remains adaptable, allowing for the integration of more sophisticated control mechanisms or an RTOS as the system evolves in the future.

4.2 Development Environment Setup

In order to program the underlying microcontroller , Particle Workbench is used within Microsoft Visual Studio Code (VSCode). Particle Workbench provides an integrated development environment (IDE) that supports cross-platform development on Windows, Linux, and macOS [32]. This section will guide you through setting up Particle Workbench in VSCode, assuming that VSCode is already installed.

4.2.1 Installing the Particle CLI and Logging In

Before setting up Particle Workbench, it is essential to install the Particle Command Line Interface (CLI). The CLI plays a critical role in interacting with Particle devices, including retrieving the device ID, which will be required in the next step.

Installing the Particle CLI: Download and install the Particle CLI using the Windows Installer provided in Particle's official documentation [29].

Creating and Logging In to Your Particle Account: If you do not have a Particle account, you can create one by visiting Particle's website [1]. Once your account is set up, log in via the Windows Command Prompt using the following command:

```
$ particle login
```

This will prompt you to enter the credentials associated with your Particle account.

4.2.2 Retrieving and Saving the Device ID

After logging in, you can retrieve your device ID by running the following command in the Command Prompt:

```
$ particle identify
```

Ensure that your Particle device is connected to your computer via USB. The device ID obtained from this command will be needed later when setting up your project in Visual Studio Code, so be sure to save it.

[1] https://www.particle.io/

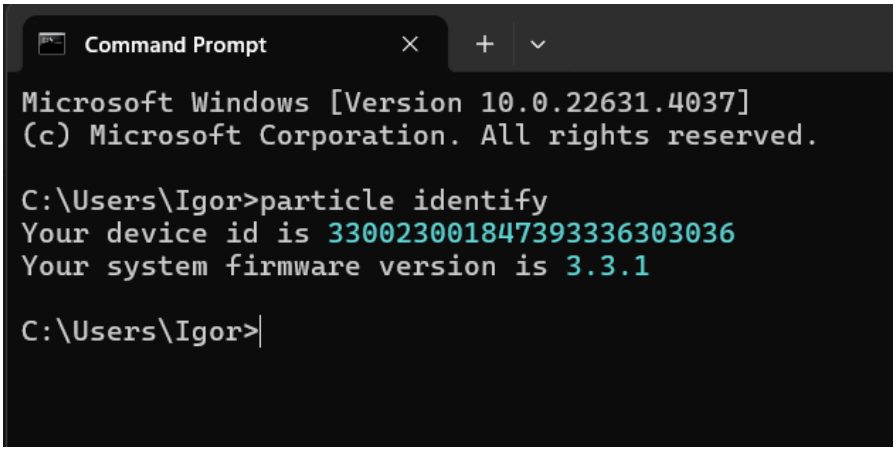

Figure 4.3: Running the ‘particle identify‘ command to retrieve and save the device ID.

The ‘particle identify‘ command will display your device ID along with the current system firmware version, as shown in Figure 4.3.

4.2.3 Installing the Particle Workbench Extension

Once you retrieved the device ID, the next step is to add the Particle Workbench extension, which includes all the necessary tools for developing and programming Particle devices.

1. **Open the VSCode Marketplace**: Inside Visual Studio Code, navigate to the Extensions view by clicking on the Extensions icon in the Activity Bar on the side of the window. Search for "Particle Workbench" in the Marketplace and click the "Install" button to add the extension, as shown in Figure 4.4.

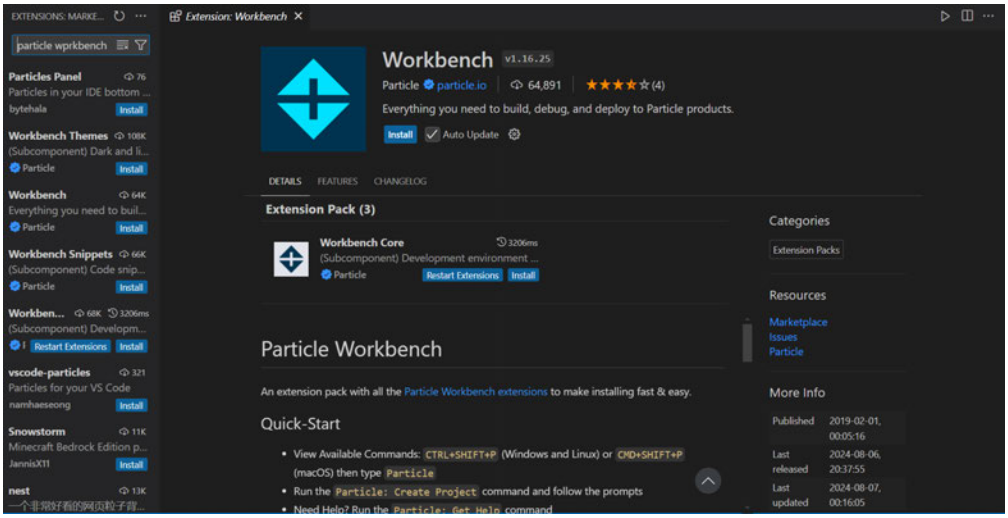

Figure 4.4: Install the Particle Workbench Extension in VSCode

2. **Confirm Installation**: After the installation completes, you may be prompted to reload VSCode. Click the "Reload Now" button to activate the extension. You will also be prompted to install additional components, such as the Particle Local Compiler, which is recommended for local development.

4.2.4 Creating a New Particle Project

1. Click on the Particle Workbench extension in VSCode, and choose the "CREATE A PROJECT" option as shown in Figure 4.5.

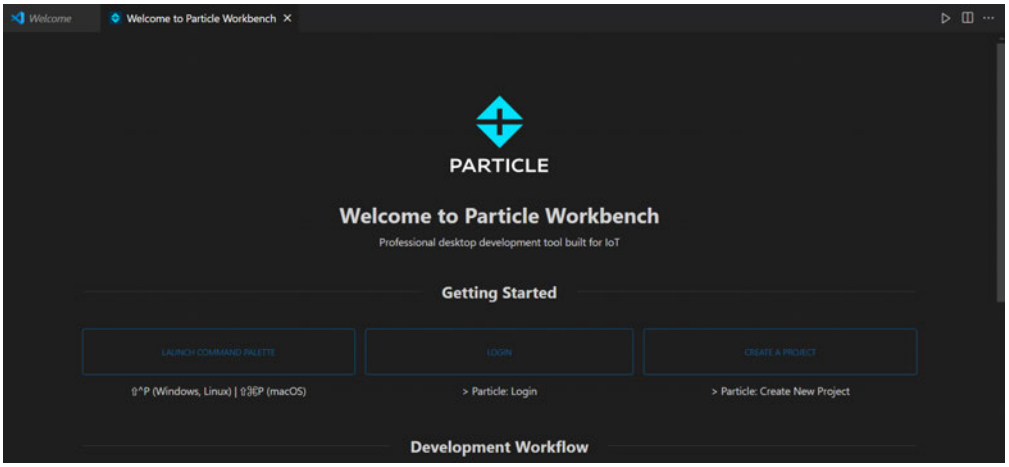

Figure 4.5: Create a New Project in Particle Workbench

2. Select a folder where you want to store the project and give the project a name, as shown in Figure 4.6.

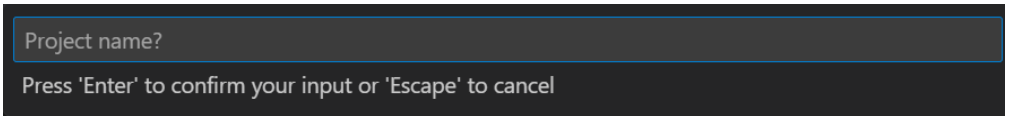

Figure 4.6: Enter Project Name and Select Folder

Project Workspace: Your first project workspace should look like the example in Figure 4.7, with a basic `setup()` and `loop()` function structure pre-configured.

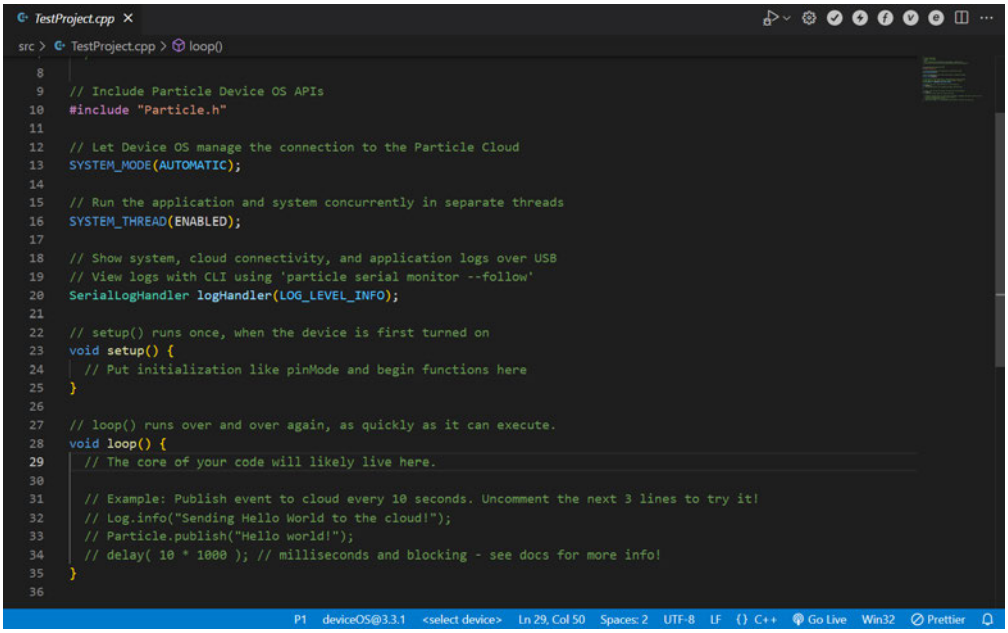

Figure 4.7: Basic Project Workspace with setup() and loop() Functions

Next, select the device OS version, the target platform, and the device ID from the menu at the bottom of the screen.

1. **Select Device OS**: Choose the appropriate device OS version, in this case, Device OS 3.3.1, as shown in Figure 4.8.

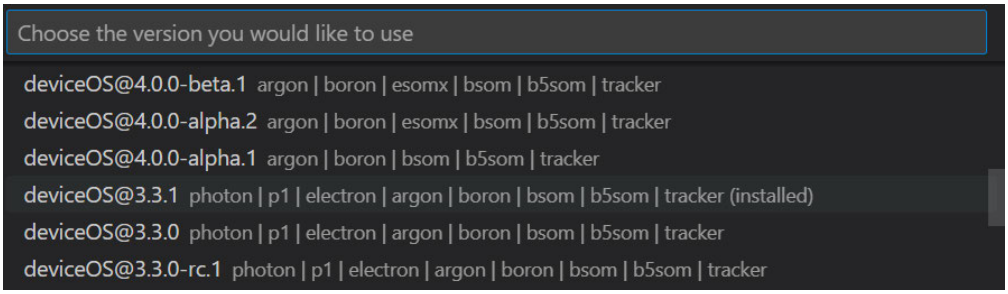

Figure 4.8: Select Device OS Version (3.3.1)

2. **Select Target Platform**: Choose the target platform for your project. For this tutorial, select "Photon" as shown in Figure 4.9.

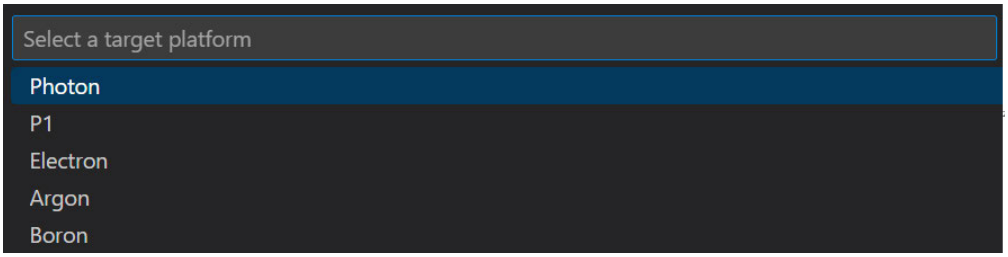

Figure 4.9: Select Target Platform (Photon)

3. **Add Device ID in VSCode**: Use the device ID that was retrieved in 4.2.2. Add the device ID in Visual Studio Code as shown in Figure 4.10.

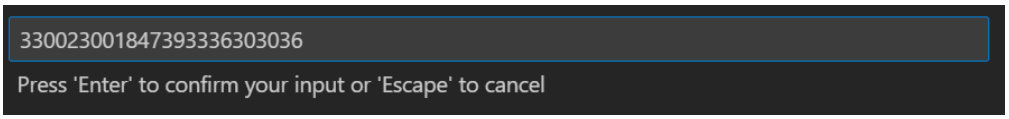

Figure 4.10: Adding the Device ID in Visual Studio Code

4.2.5 Flashing the Firmware

After all the configurations are done, you can now flash the firmware to your Particle device by clicking the "Flash Project" button at the top right menu, as shown in Figure 4.11.

Figure 4.11: Flash the Firmware to the Device

4.3 Wi-Fi Connectivity

The first step in ensuring device control and seamless data exchange is establishing a reliable Wi-Fi connection. For an air purifier to connect to Wi-Fi, it must first obtain the necessary credentials for the network. This section outlines the methods by which the Particle Photon manages Wi-Fi connectivity.

4.3.1 Wi-Fi Configuration Methods

The Particle Photon offers several methods for Wi-Fi configuration:

1. **USB Serial (CDC)**: The device can be connected to a computer via USB, and the Particle CLI (Command Line Interface) can be used with the command `particle serial wifi` to configure Wi-Fi credentials [36].
2. **SoftAP Setup**: This method involves connecting a mobile device to the Particle device's SoftAP (software enabled access point) Wi-Fi network to set up Wi-Fi credentials [36].
3. **Setup Web Application**: The Photon can be connected to the internet using the setup web application, which also involves a USB connection [35].

These methods provide different levels of convenience depending on the user scenario, with the SoftAP setup being the most intuitive for customer-facing products like the air purifier.

4.3.2 Wi-Fi Credentials Configuration

The software starts by checking for stored Wi-Fi credentials during the initialization phase. If the credentials are available, the device attempts to connect to the specified Wi-Fi network automatically.

4.3.3 Listening Mode

When the device lacks stored Wi-Fi credentials, it is programmed to automatically switch to a mode called Listening Mode [34]. While in Listening Mode, the device sets up a temporary access point (AP) and launches an HTTP server on port 80. This functionality allows the user to provide Wi-Fi credentials through the mobile application [33].

4.3.4 Wi-Fi Setup Confirmation Process

Ideally, the mobile application would seamlessly connect to the microcontroller's SoftAP, send the Wi-Fi credentials, and the microcontroller would then connect to the Wi-Fi network. However, as outlined in Section 6.3, configuring Wi-Fi credentials using the SoftAP method requires users to manually switch their mobile devices to the air purifier's SoftAP network. This process introduces the challenge of verifying whether the credentials have been correctly configured on the air purifier, as users must temporarily disconnect from their usual Wi-Fi network.

After the mobile application transmits the Wi-Fi credentials to the microcontroller, the device attempts to connect to the specified Wi-Fi network. At this point, the SoftAP is deactivated, preventing the application from directly verifying whether the connection was successful.

To overcome this limitation, the mobile application implements a confirmation mechanism that waits for up to 45 seconds for an MQTT message from the Particle Photon. This message serves as an indication that the Wi-Fi credentials are correct and that the air purifier has successfully connected to the network. Upon receiving this confirmation, the application sends an acknowledgment back to the Particle Photon via MQTT, signaling that the setup process has been successfully completed, allowing the device to proceed with normal operation.

The overall process involves the interaction between the air purifier, mobile application, and the MQTT broker, as depicted in Figure 4.13. This figure illustrates the message flow between the components during the Wi-Fi setup and connection confirmation process.

This approach ensures that the Wi-Fi setup process is reliable and robust, even in scenarios where issues arise on the application side. By implementing this feedback loop, the system can detect and recover from configuration failures, thereby improving the overall user experience and ensuring consistent connectivity.

4.3.5 Automatic Reconnection

Once the Wi-Fi credentials are successfully stored and the device is connected to the network, it continuously monitors the Wi-Fi connection status. If a disconnection occurs, the system will automatically attempt to reconnect to the previously configured network

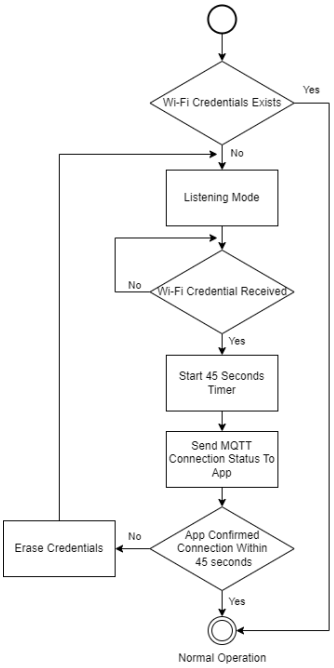

Figure 4.12: Flowchart depicting the Wi-Fi setup confirmation process for the Particle Photon. The flowchart shows the decision-making steps involved in verifying successful Wi-Fi configuration and handling potential failures.

as soon as it becomes available. This ensures minimal disruption in connectivity, allowing the device to function seamlessly even after temporary network interruptions.

4.3.6 Summary

The Wi-Fi connectivity management ensures that the air purifier remains connected to the internet by handling both the initial configuration of Wi-Fi credentials and any subsequent interruptions. The system is designed to be robust, automatically entering Listening Mode if no credentials are found, and reconnecting automatically if the connection is lost.

4.4 Indoor Air Quality

The core functionality of the air purifier system relies on reading sensor data to monitor indoor air quality. The embedded software interfaces with multiple environmental

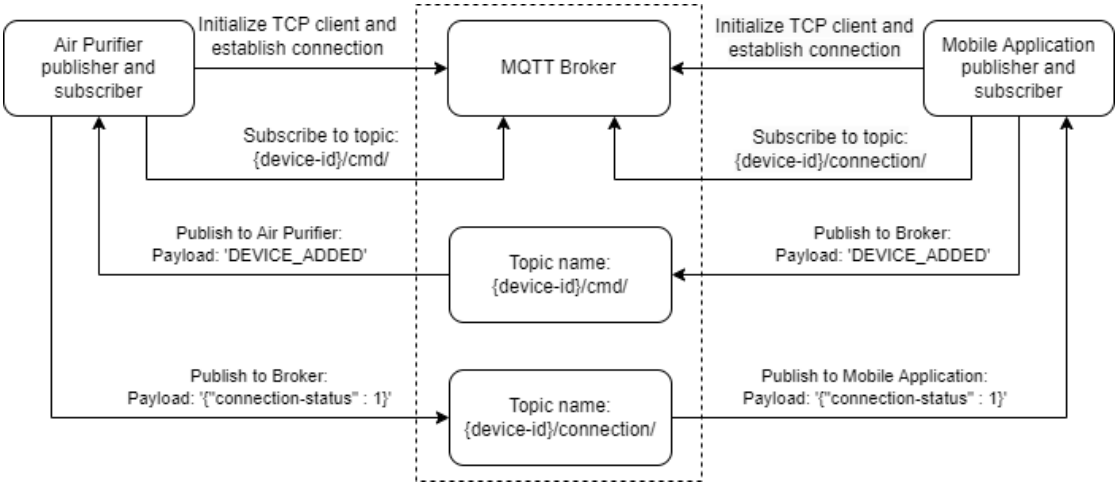

Figure 4.13: Publish-Subscribe model for handling connection status during the Wi-Fi setup process. The figure shows how the air purifier, mobile application, and MQTT broker communicate during the device setup process.

sensors that detect pollutants such as particulate matter (PM2.5), volatile organic compounds (VOCs), carbon dioxide (CO_2), and other relevant metrics like temperature and humidity. This data is continuously gathered, processed, and used to adjust the air purifier's operation in real-time, ensuring optimal air quality in the indoor environment. The following section details the methods used to collect, process, and display sensor readings within the embedded system and the mobile application.

4.4.1 Temperature and Humidity

Monitoring temperature and humidity is a critical aspect of indoor air quality (IAQ) management, particularly in enclosed environments like homes and offices. Humidity levels can significantly influence perceived air quality and have direct effects on health and comfort. For instance, low humidity levels can exacerbate sensory irritation in the eyes and upper airways, as well as contribute to complaints of dryness and discomfort [6]. High indoor humidity, on the other hand, can lead to the growth of mold and mildew, which are known to cause respiratory issues, including asthma and allergies [40]. Additionally, humidity plays a role in the transmission and survival of airborne viruses [13]. Therefore, regulating temperature and humidity in enclosed spaces can mitigate health risks associated with poor IAQ and enhance overall comfort.

Reading Temperature and Humidity with the HDC1080 Sensor

To monitor temperature and humidity effectively, the air purifier integrates the HDC1080 sensor [43]. The sensor is interfaced with the Particle microcontroller using the `Adafruit_HDC1000` library, which provides an easy-to-use API for accessing sensor data via the I^2C communication protocol [1]. Although the library is named after the HDC1000 sensor, it is fully compatible with the HDC1080 due to the similarity in their design and communication protocol, making it a suitable choice for this application.

The process of reading temperature and humidity data from the HDC1080 sensor consists of the following:

- **Create an HDC1000 Object**:

  ```
  Adafruit_HDC1000 hdc = Adafruit_HDC1000();
  ```

- **Initialize the Sensor in `setup()` and Wait for 15 Milliseconds Until the Sensor is Initialized**:

  ```
  hdc.begin();
  delay(15);
  ```

- **After Initialization, the Temperature and Humidity Data Can Be Read in the Main Loop**:

  ```
  float temperature = hdc.readTemperature();
  float humidity = hdc.readHumidity();
  ```

This process initializes the sensor during setup and ensures a 15-millisecond delay for proper startup. Once initialized, the sensor can be continuously read in the main loop to obtain temperature and humidity data.

4.4.2 VOC and CO_2 Measurement

Volatile Organic Compounds (VOCs) are gases emitted from various solids or liquids, such as paints, solvents, cleaning products, and building materials, which significantly impact indoor air quality. Indoors, VOC concentrations are often 2 to 10 times higher than outdoors, with certain activities like paint stripping leading to even greater levels. Health effects from VOC exposure can range from minor irritations, such as eye, nose, and throat discomfort, to severe outcomes, including liver and kidney damage, central nervous system effects, and potential cancer risks [49].

Carbon dioxide (CO_2) is a colorless, odorless gas that occurs naturally in the atmosphere. It is released into the atmosphere through both natural processes, such as volcanic eruptions, oceanic activity, animal and plant respiration, and decomposition, as well as through human activities like burning fossil fuels. Although CO_2 is not highly toxic, exposure to high concentrations can reduce the amount of oxygen available in the air, leading to various health effects. These effects can range from mild symptoms like headaches and drowsiness to more serious issues such as rapid breathing, confusion, and, in extreme cases, suffocation. The risks are particularly significant in enclosed or poorly ventilated areas, where CO_2 can accumulate to dangerous levels [8, 26].

To measure Volatile Organic Compounds (VOC) and Carbon Dioxide (CO_2) equivalent levels, the SGP30-2.5k sensor is used in conjunction with the Adafruit_SGP30 library. This library provides a straightforward interface for sensor initialization and data retrieval, enabling VOC and CO_2 readings with simple function calls. Additionally, the library supports humidity compensation, which can improve the accuracy of air quality measurements if temperature and humidity data are available [2].

Sensor Initialization

The SGP30 sensor communicates with the Particle Photon via the I^2C protocol [41]. The sensor is first instantiated and then initialized in the `setup()` function as follows:

Listing 4.1: Initialization of the SGP30 sensor for VOC and CO2 measurement

```
// Include the I2C library
#include <Wire.h>

// Create an SGP30 object
SensirionSGP30 sgp30 = SensirionSGP30();

// Initialize the sensor in setup()
sgp30.begin();
delay(15);
```

The object `sgp30` is created using the SensirionSGP30 library, and `sgp30.begin()` initializes the sensor. The delay allows the sensor to stabilize before readings are taken, ensuring accurate measurements.

Humidity Compensation and Reading VOC and CO_2 Data

Environmental factors such as humidity and temperature can significantly affect the sensor's ability to accurately detect VOCs and CO_2. To improve measurement accuracy, the SGP30 sensor incorporates on-chip humidity compensation. This compensation requires absolute humidity data from an external sensor, which is provided by the HDC1080 sensor mentioned in Section 4.4.1. By adjusting for the effects of humidity, the SGP30 can more accurately measure VOCs and CO_2 in varying environmental conditions [41].

The datasheet provides a formula for calculating absolute humidity $d_v(T, RH)$, which is crucial for the humidity compensation process. This formula, shown in Equation 4.1, takes into account the relative humidity (RH) and temperature (T) to calculate the absolute humidity. The equation is as follows:

$$d_v(T, RH) = 216.7 \cdot \left[\frac{\frac{RH}{100\%} \cdot 6.112 \cdot \exp\left(\frac{17.62 \cdot T}{243.12 + T}\right)}{273.15 + T} \right] \tag{4.1}$$

Here, d_v represents the absolute humidity in grams per cubic meter (g/m^3), RH is the relative humidity in percentage, and T is the temperature in degrees Celsius. By calculating the absolute humidity using this formula, the SGP30 can apply the necessary

compensation, resulting in more accurate VOC and CO_2 measurements across different environmental conditions.

Implementing Sensor Readings with Humidity Compensation

The code below illustrates how to read sensor data with humidity compensation, ensuring more accurate measurements under varying environmental conditions.

Listing 4.2: Humidity compensation and air quality measurement using the SGP30 sensor

```
// Set humidity compensation with HDC1080 sensor values
sgp30.setHumidity(getAbsoluteHumidity(temperature, humidity));

// Measure air quality
if (! sgp30.IAQmeasure()) {
    Serial.println("Measurement failed");
    return;
}

// Print the results
Serial.printlnf("TVOC %.2f ppb", sgp30.TVOC);
Serial.printlnf("eCO2 %.2f ppm", sgp30.eCO2);

delay(1000);
```

4.4.3 CO Measurement

Carbon monoxide (CO) is a colorless, odorless, and toxic gas that poses significant risks to indoor air quality, as it can be deadly without warning. Common sources of CO include unvented gas heaters, leaking chimneys, gas stoves, vehicle exhaust, and tobacco smoke. Health effects of CO exposure range from fatigue and chest pain at low concentrations to impaired vision, headaches, dizziness, confusion, and nausea at higher levels. Extremely high concentrations can be fatal. The acute effects are caused by carboxyhemoglobin formation in the blood, reducing oxygen intake [47].

CO Concentration Calculation

To measure carbon monoxide (CO), the ME2-CO-14x50-C sensor was utilized. This sensor generates a small current that is directly proportional to the concentration of CO in the air. According to the datasheet, the sensor's sensitivity ranges from 0.8 to 4 nA per ppm of CO [55].

In this implementation, a circuit utilizing an operational amplifier and a load resistor of 1 MΩ was chosen to convert the sensor's output current into a measurable voltage. The operational amplifier ensures that the output voltage remains within the 3.3V range, making it compatible with the Particle Photon's analog-to-digital converter (ADC) [30]. The relationship between the sensor's output current I_{sensor} and the resulting voltage V_{out} across the load resistor is defined by Ohm's law:

$$V_{\text{out}} = I_{\text{sensor}} \times R_{\text{load}}$$

With a 1 MΩ resistor, a current of $1\mu A$ generates a voltage of 1 V. This setup effectively converts the sensor's small current output into voltages that can be accurately read by the Particle Photon's analog input pins.

According to the sensor's datasheet (see Figure 4.14), for each volt measured at the output, the corresponding CO concentration is 750 ppm. Therefore, to calculate the CO concentration based on the analog reading, is as follows:

$$\text{CO Concentration (ppm)} = \left(\frac{\text{Analog Input} \times 3.3V}{2^{12}} \right) \times 750 \text{ ppm/V}$$

In this formula:

- **Analog Input** represents the digital value read by the Particle Photon's 12-bit ADC, which ranges from 0 to 4095 [30].
- The factor $3.3V$ accounts for the reference voltage of the Particle Photon.
- The division by 2^{12} converts the ADC reading to a voltage value.
- Finally, the result is multiplied by the sensor's sensitivity of 750 ppm per volt to calculate the CO concentration.

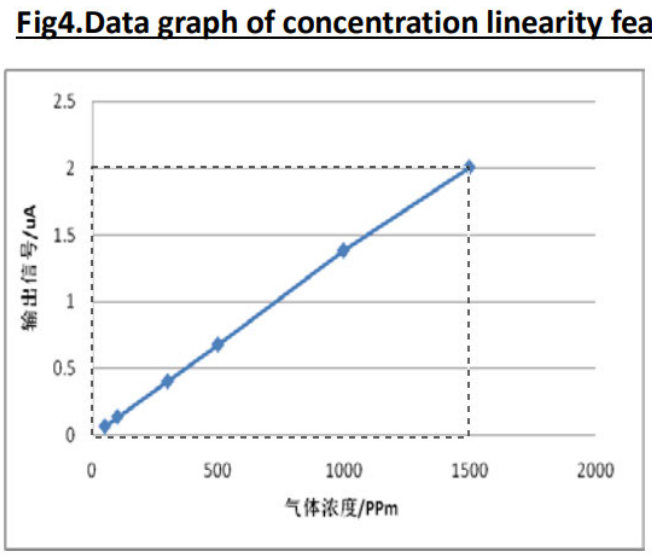

Figure 4.14: Data graph showing the concentration of CO in ppm as a function of the sensor output current in microamperes [55].

This calculation ensures that the raw analog data from the sensor is properly converted to a meaningful CO concentration in parts per million.

Temperature Dependency

The sensor's output varies with temperature, as illustrated in Figure 4.15. From the figure, it can be observed that the sensor returns accurate CO concentrations at 20°C. However, at temperatures above 20°C, the sensor begins to overestimate the CO concentration. Specifically, for every degree Celsius above 20°C, the sensor's reading increases by approximately 0.5% of the true concentration per degree Celsius.

This percentage increase can be quantified by calculating the slope m_1 of the linear increase in output:

$$m_1 = \frac{\Delta y}{\Delta x} = \frac{115\% - 90\%}{50°\text{C} - 0°\text{C}} = 0.5\frac{\%}{°\text{C}}$$

Conversely, at temperatures below 20°C, the sensor underestimates the CO concentration, with the reading decreasing by approximately 0.5% per degree Celsius.

It should be noted that the slope m_1 is only calculated for temperatures above 0°C, as the air purifier is designed to operate indoors, where room temperatures are not expected to drop below 0°C. This assumption simplifies the temperature compensation model to focus on typical indoor environments.

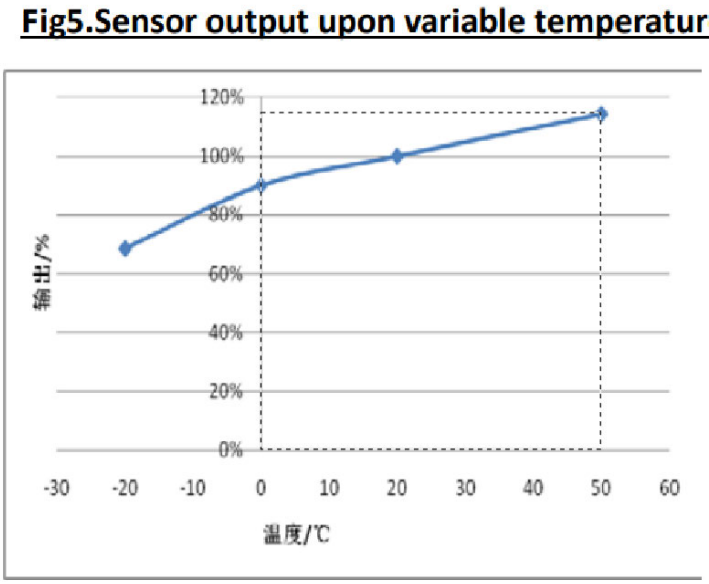

Figure 4.15: Sensor output as a function of temperature. The graph shows that the sensor's output is accurate at 20°C, but deviations occur at temperatures higher or lower than 20°C [55].

Additionally, the sensor's output experiences zero-point drift due to temperature variations, as shown in Figure 4.16. For temperatures above 20°C, the sensor's baseline output increases, meaning that the sensor overestimates the CO concentration even further. This temperature dependency can be quantified by calculating the slope m_1 of the linear increase in output:

$$m_2 = \frac{\Delta y}{\Delta x} = \frac{4 \text{ ppm} - 0 \text{ ppm}}{50°\text{C} - 20°\text{C}} = 0.1333 \frac{\text{ppm}}{°\text{C}}$$

Thus, for every degree Celsius above 20°C, 0.1333 ppm should be subtracted from the sensor reading to correct the CO concentration for zero-point drift. This correction, combined with the general temperature compensation of ±0.5% per degree Celsius, ensures

that the CO concentration values remain accurate across varying temperature conditions.

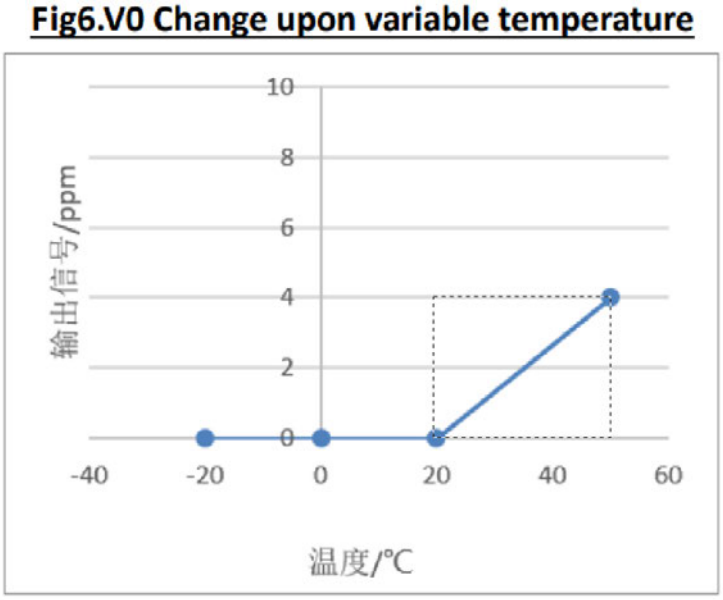

Figure 4.16: Graph showing the temperature dependency of the sensor's output (zero drift) for various temperatures [55].

This compensation approach ensures that the sensor readings remain reliable despite variations in temperature, which is critical for maintaining accurate air quality assessments.

4.4.4 PM2.5 Measurement

Particulate Matter (PM) is a complex mixture of tiny particles and liquid droplets suspended in the air, composed of various chemical compounds including inorganic ions, metals, elemental carbon, and organic materials. The size of these particles plays a critical role in their impact on health and the environment. Particulate Matter is categorized by its diameter, with PM10 referring to particles 10 microns or smaller, which are inhalable and can affect the lungs. PM2.5, a subset of PM10, consists of even finer particles with a diameter of 2.5 microns or smaller.

PM2.5 particles are especially concerning due to their ability to penetrate deep into the lungs and even enter the bloodstream. These particles primarily originate from

combustion processes, including vehicle emissions, industrial activities, and wildfires. Due to their small size, PM2.5 particles can bypass the body's natural defenses and deposit in the deepest parts of the lungs, causing inflammation and tissue damage.

Exposure to PM2.5 is linked to a variety of serious health issues. Short-term exposure can trigger respiratory problems, asthma attacks, and increased hospital admissions for cardiovascular and respiratory conditions. In vulnerable populations, such as children, older adults, and individuals with pre-existing health conditions, PM2.5 exposure can lead to premature mortality. Long-term exposure to PM2.5 has been associated with chronic respiratory diseases, reduced lung function, and an increased risk of heart disease. Studies have also linked PM2.5 exposure to lung cancer, highlighting its severe impact on public health [7].

Reading PM2.5 Data

The sensor ZPH02 communicates with the microcontroller using the Universal Asynchronous Receiver-Transmitter (UART) protocol. The sensor is configured to operate at a baud rate of 9600 bps, with 8 data bits, 1 stop bit and no parity bits. The microcontroller reads the data output from the sensor, which is sent automatically every second as a byte stream [54].

Once the sensor is initialized, the microcontroller begins reading PM2.5 data by receiving a data packet from the sensor. The PM2.5 concentration is determined by extracting two specific bytes from this packet: byte 3, which represents the integer part of the PM2.5 value, and byte 4, which represents the decimal part. These two bytes collectively encode the pulse rate corresponding to the PM2.5 concentration.

The combined value of these bytes is then multiplied by a factor of 9.091 to convert it into a concentration value in micrograms per cubic meter ($\mu g/m^3$). This conversion factor is based on the slope of the green line in Figure 4.17, which is derived from the sensor's average response behavior as represented by the black line.

Checksum Verification To ensure that the received data depicted in figure 4.18 has not been corrupted during transmission, the microcontroller calculates a checksum by subtracting bytes 1-7 from the start byte and adding +1. This calculated checksum is

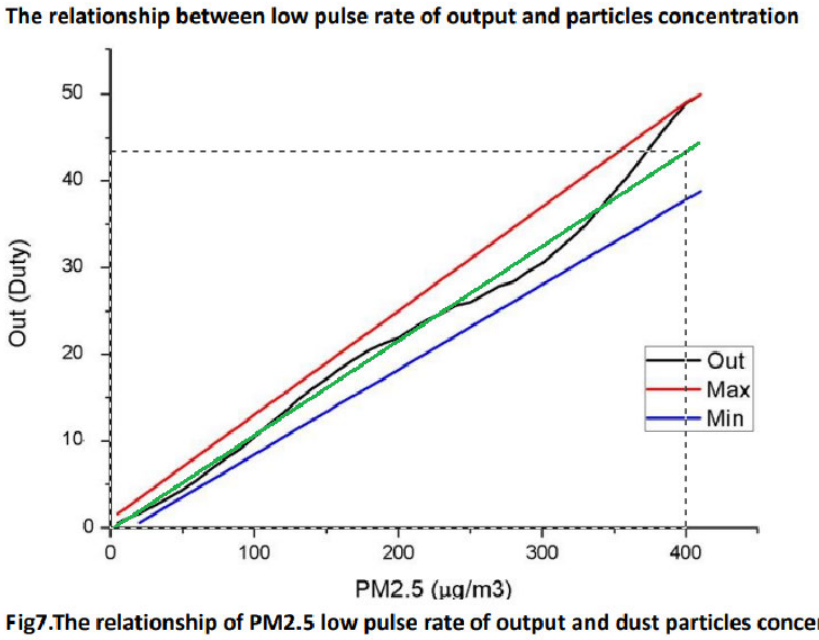

Figure 4.17: The relationship between the low pulse rate of output and particle concentration. The green line represents the average behavior of the sensor's output, leading to the conversion factor of 9.091 used in processing the sensor's PM2.5 data [54].

0	1	2	3	4	5	6	7	8
Start byte	Detection type name code	Unit (Low pulse rate)	Integer part of low pulse rate	Decimals part of low pulse rate	Reservation	Reservation	Reservation	Check value
0xFF	0x18	0x00	0x00-0x63	0x00-0x63	0x00	0x00	0x00	0x00-0xFF

Figure 4.18: Communication command structure sent by the sensor. The module sends the concentration value every other second. The data includes a start byte, detection type code, pulse rate, and a checksum for data integrity verification [54].

then compared with the checksum byte sent by the sensor, which is the last byte in the data packet [54].

If the calculated checksum matches the checksum byte received from the sensor, the data is considered valid, and the microcontroller proceeds to process it. If the checksums do not match, the data is discarded.

Example of Handling Data

The received bytes are processed to extract meaningful PM2.5 concentration data. For example, if the sensor transmits the following bytes:

```
0xFF 0x18 0x00 0x03 0x13 0x00 0x00 0x00 [Checksum]
```

These bytes can be interpreted as:

- **Start Byte:** 0xFF
- **Detection type name code:** 0x18 (PM2.5 Detection)
- **Pulse Rate:** 0x03 (Low Pulse Rate) and 0x13 (High Pulse Rate), combined as 3.19% duty cycle.
- **PM2.5 Concentration:** 3.19 * 9.091 = 29 µg/m^3
- **Checksum:** The last byte in the packet, used to verify the integrity of the data.

The structure of the data packet sent by the sensor is illustrated in Figure 4.18, which outlines the various components of the packet, including the start byte, detection type code, pulse rate, and checksum.

4.5 Indoor Air Quality Index Calculation

Interpreting raw data from air quality sensors in a meaningful way is crucial for assessing potential health risks associated with indoor air pollutants. However, it is often challenging for individuals to gauge the extent of air pollution or to comprehend the acceptable concentration limits for various pollutants. The calculation of an Indoor Air Quality Index (IAQI) serves as a critical tool for translating complex pollutant data into a format that is easy to interpret. This index helps to quantify the health risks posed by indoor air pollutants, providing clear and actionable information for informed decision-making across different levels of technical understanding.

Although various methodologies for calculating outdoor air quality indices, such as the U.S. Environmental Protection Agency's (EPA) Air Quality Index (AQI), are well documented, establishing a standardized method for indoor environments has proven to be

challenging, and currently, a binding IAQ document does not exist[48, 40, 39]. This complexity arises from the unique characteristics of indoor air environments and the varying health impacts of different pollutants[40].

To assess the health risks posed by a given pollutant, it is essential to understand how the human body responds to different concentrations in the air. Observations from individuals exposed to pollutants in occupational settings provide valuable data for determining the risks posed by specific pollutants. However, these findings may not always be directly applicable to the general public. Moreover, there is limited data on the combined effects of multiple pollutants on health. Currently, risk assessments are typically performed on individual chemicals, as there is a scarcity of relevant data and established methods to evaluate the health impacts of pollutant mixtures [40].

Furthermore, scientists continue to debate the precise effects of indoor air pollutants on health, and many global standards, including those from the World Health Organization (WHO), focus on exposure to pollutant concentrations over specific timeframes rather than on the development of comprehensive indoor air quality indices [40, 51]. This focus on time-based exposure highlights the challenges in creating a unified IAQI, as the health risks associated with pollutants can vary significantly depending on the duration and intensity of exposure.

4.5.1 Addressing the Gap in IAQI Standards

To address the gap in existing IAQI standards, this thesis proposes a unified IAQI framework that integrates sensor data, including measurements of carbon monoxide (CO), carbon dioxide (CO), volatile organic compounds (VOCs), and particulate matter (PM2.5). This integrated approach simplifies the complex data generated by multiple sensors into a single, interpretable index. The IAQI's primary goal is to provide actionable information about indoor air quality, allowing users to make informed decisions to protect their health.

Given the inconsistencies in current guidelines and the complexity of indoor air pollution, the IAQI framework developed here combines insights from various research studies and national guidelines to derive pollutant breakpoints [50, 48, 9, 15]. These breakpoints provide a flexible and practical approach to assessing indoor air quality. The IAQI is tailored to the specific pollutants measured by the air purifier and is designed to

trigger real-time alerts when air quality deteriorates, enabling immediate actions, such as increasing the purifier's fan speed, to mitigate potential health risks.

4.5.2 IAQI Framework

The proposed IAQI framework uses real-time data from sensors that measure concentrations of PM2.5, VOCs, CO, and CO_2. The index aggregates these pollutant measurements into a single value that reflects overall indoor air quality.

The IAQI is calculated using a method similar to the EPA's AQI formula[48]:

$$IAQI = \frac{I_{Hi} - I_{Lo}}{B_{Hi} - B_{Lo}} \times (C_p - T_{Lo}) + I_{Lo} \tag{4.2}$$

Where:

- I_{Hi} and I_{Lo} represent the upper and lower bounds of the IAQI range for a given pollutant.
- T_{Hi} and T_{Lo} represent the corresponding pollutant concentration breakpoint.
- C_p is the measured pollutant concentration.

This calculation allows the air purifier to provide real-time feedback by integrating pollutant data and generating a unified IAQI score. The system can then trigger alerts as soon as air quality deteriorates. Additionally, the system automatically adjusts the air purifier's fan speed to improve air quality, dynamically responding to changes in the environment.

4.5.3 IAQI Interpretation Table

The following table presents the interpretation of IAQI values. The IAQI is divided into four categories: Good, Moderate, Poor, and Critical, each associated with a corresponding color for easy visual interpretation.

Table 4.1: IAQI Value Interpretation

IAQI Value	Color	Interpretation
0-100	Blue	Good
101-150	Green	Moderate
151-250	Yellow	Poor
251-500	Red	Critical

4.5.4 Pollutant-Specific Breakpoints

The following table outlines the specific pollutant concentration ranges that correspond to each IAQI value, based on the synthesized guidelines from multiple sources. These thresholds ensure that users are alerted when pollutant levels reach a critical level, prompting immediate action.

Table 4.2: Pollutant-Specific Breakpoints for IAQI Categories

CO(ppm)	CO_2(ppm)	TVOC(ppb)	PM2.5(µg/m^3)	IAQI Value	Category
0-10	0-1000	0-100	0-35	0-100	Good
>11-14	1001-2000	101-300	36-55	101-150	Moderate
>15-30	2001-3000	301-1000	56-75	151-250	Poor
>30	>3001	>1001	>76	251-500	Critical

To summarize, while no universally accepted IAQI standard currently exists, the framework presented here offers a practical solution for real-time indoor air quality monitoring. By consolidating authoritative guidelines and focusing on real-time pollutant data, this IAQI framework enables immediate responses to deteriorating air quality, ensuring a healthier indoor environment.

4.6 Communication and Data Transmission

Once the system is connected to the Wi-Fi network, sensor data is acquired, and the Indoor Air Quality Index (IAQI) is calculated, the next step is to transmit this data for monitoring and control purposes. Communication in this system is achieved using two primary methods: MQTT for real-time data transmission and HTTP for scheduled communication with the server.

The Particle Photon microcontroller utilizes MQTT to transmit real-time sensor values to the mobile application, enabling continuous monitoring and instantaneous control. MQTT was chosen for real-time communication because performance testing shows that it can transmit data up to six times faster than HTTP under similar conditions [4]. This makes MQTT ideal for applications that require immediate data updates, such as monitoring indoor air quality in real time. Additionally, HTTP is employed for periodic communication with the backend server, including retrieving schedules on startup and sending averaged sensor data every 10 minutes. Both communication protocols are secured using appropriate encryption mechanisms to ensure data integrity and security.

4.6.1 MQTT for Real-Time Communication

MQTT is used for the real-time transmission of sensor values from the microcontroller to the mobile application. This ensures that the user can monitor indoor air quality and control the air purifier's operation in real-time. Given its efficiency in transmitting data with low latency and minimal overhead, MQTT is well-suited for this IoT system, where quick responsiveness is critical.

The MQTT broker, which facilitates the exchange of messages between the microcontroller and the mobile application, is deployed on a virtual machine in Microsoft Azure's cloud computing platform. The detailed setup and configuration of this MQTT broker are elaborated in Section 5.1.

For secure communication, the `MQTT-TLS` library is employed [10]. This library supports Transport Layer Security (TLS), ensuring that all data transmitted between the microcontroller and the broker is encrypted and secure. The MQTT broker operates on port 8884, which necessitates the use of TLS certificates, including a Certificate Authority (CA) certificate, a client certificate, and a client key.

Listing 4.3: MQTT client setup with TLS encryption and time synchronization

```
MQTT client("mqtt_broker_name.azure.com", 8884, callback);
#define ONE_DAY_MILLIS (24 * 60 * 60 * 1000)
unsigned long lastSync = millis();
void setup() {
    // Set Current Time
    if (millis() - lastSync > ONE_DAY_MILLIS) {
        Particle.syncTime();
        lastSync = millis();
    }

    client.enableTls(letencryptCaPem, sizeof(letencryptCaPem),
                     clientKeyCrtPem, sizeof(clientKeyCrtPem),
                     clientKeyPem, sizeof(clientKeyPem));
}
```

As shown in Listing 4.3, the ‘client.enableTls()‘ function is used to load the necessary TLS credentials, which are essential for authenticating the microcontroller with the broker and establishing a secure communication channel. The MQTT connection allows the system to transmit sensor data continuously, ensuring that the mobile application receives real-time updates.

Sending Sensor Values via MQTT

To ensure real-time monitoring of air quality, the system periodically sends sensor data to the cloud using the MQTT protocol. This allows the mobile application to receive updates on temperature, humidity, CO_2, VOC, CO, and PM2.5 levels in near real-time. MQTT was chosen for its efficiency and low latency, making it well-suited for transmitting sensor data in IoT systems.

The sensor values are formatted as a JSON string, which is then published to a specific MQTT topic corresponding to the device ID. The following code snippet demonstrates the process of sending sensor data via MQTT:

Listing 4.4: Sending Sensor Data via MQTT

```
payload = "{"
payload += "\"temperature\": " + temperature + ","
payload += "\"humidity\": " + humidity + ","
payload += "\"co2\": " + co2 + ","
payload += "\"voc\": " + voc + ","
payload += "\"co\": " + co + ","
payload += "\"pm25\": " + pm25 + ","
payload += "\"iaqi\": " + iaqi
payload += "}"

topic = device_id + "/sensors/"

client.publish(topic, payload)
```

By publishing the sensor data to the ‘device_id/sensors/‘ topic, the system ensures that the cloud infrastructure and mobile application receive timely updates, enabling efficient monitoring and control of the air purifier.

4.6.2 HTTP for Periodic Communication

HTTP is employed for scheduled communication between the Particle Photon microcontroller and the backend server. This communication occurs at specific intervals and includes tasks such as retrieving schedules on startup and sending averaged sensor data every 10 minutes. The `TlsTcpClient` library is used to establish secure HTTP connections over TLS, ensuring that all data exchanged with the server remains encrypted and protected [11].

An example of sending sensor values using this library is provided in the appendix (see Appendix B.1 for the full implementation). This implementation allows the microcontroller to securely transmit data to the server, ensuring that the system operates in a secure and efficient manner.

4.7 Scheduled and Automated Fan Control Logic

In this section, the implementation of the scheduling functionality for the air purifier system is discussed. The schedule is crucial for the correct operation of the air purifier, enabling it to operate at specific times and at different fan speeds as defined by the user. The schedule is received by the microcontroller either through an HTTPS request from the server during system startup or directly from the mobile application via MQTT if a new schedule is set while the system is running. Upon receiving the schedule, the microcontroller must immediately update its internal schedule structure to reflect the new instructions.

4.7.1 Schedule Data Structure

The schedule received by the microcontroller is encoded in a specific format. Each schedule entry is represented by a string of hexadecimal values that encode the time, duration, and fan speed settings. The structure of the schedule string has the following format and can be seen in Figure 4.19:

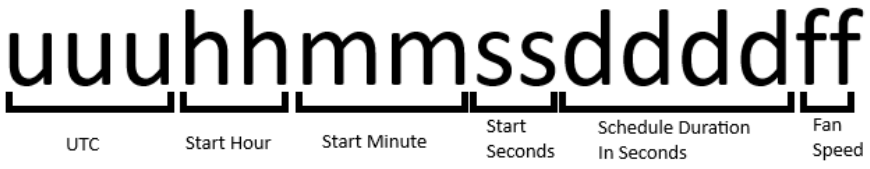

Figure 4.19: Schedule String Structure

- The first three bytes represent the UTC offset. The UTC offset is encoded to accommodate both positive and negative values and to allow for minute-level precision:
  - The first byte (`u`) indicates the sign of the UTC offset, where 0 represents a positive offset and 1 represents a negative offset.
  - The second byte (`u`) represents the integer part of the UTC offset, ranging from 0 to 14.

  - The third byte (`u`) represents the decimal part of the UTC offset, with values from 0 to 3, where 0 corresponds to 0 minutes, 1 corresponds to 15 minutes, 2 corresponds to 30 minutes, and 3 corresponds to 45 minutes.

- The next two bytes (`hh`) represent the hour at which the schedule should start.
- The following two bytes (`mm`) represent the start minute.
- The subsequent two bytes (`ss`) represent the start second.
- The next four bytes (`dddd`) represent the duration of the operation in seconds.
- The last two bytes (`ff`) represent the fan speed (1-7).

This structure allows the microcontroller to easily parse the schedule string and extract the necessary information to control the air purifier's operation.

4.7.2 Fan Speed Control via PWM

The fan speed is controlled using a simple Pulse Width Modulation (PWM) technique. The duty cycle of the PWM signal corresponds to the desired fan speed, allowing the microcontroller to adjust the fan's power level according to the schedule. The fan speed values, represented by the last two bytes of the schedule string, are mapped to specific PWM duty cycles that regulate the fan's operation from low speed to maximum speed.

4.7.3 Schedule Parsing and Storage

Users can configure multiple schedule entries, specifying different days, times, and fan speeds. The complete schedule received from the mobile application is represented as a single string, with individual schedule entries separated by commas.

To efficiently manage multiple schedule entries, the system dynamically allocates an array of schedule structures. Each schedule entry is extracted from the received string and parsed into this array. The parsing process consists of the following steps:

- First, the system determines the number of schedule entries in the received string by counting the commas separating each entry.

- The UTC offset is extracted and converted from its hexadecimal representation. The system handles both the sign and the fractional part of the offset.
- Each schedule entry is then parsed by extracting the hour, minute, second, day of the week, duration, and fan speed from their respective positions in the string.
- These values are stored in a struct that holds the schedule information, and the struct is placed in the schedule array. This process ensures that schedules are saved in memory, allowing the air purifier to continue operating according to the stored schedules, even if the Wi-Fi connection is lost.

4.7.4 Schedule Implementation in the Main Loop

The operation of the schedule is carried out in the main loop of the microcontroller's firmware. The system checks the current time against the stored schedules every second and activates the air purifier accordingly.

The process is as follows:

- Every second, the system checks if the current time matches any of the stored schedules for that day.
- If a matching schedule is found, the system determines the fan speed based on the schedule. If no match is found, the fan speed is set to 0 (off).
- The system sets the fan speed according to the schedule by adjusting the PWM duty cycle.

The following pseudo code summarizes this process:

Listing 4.5: Pseudocode for fan speed control based on schedules

```
if secondPassed:
    currentTime = getCurrentTime()
    fanSpeed = checkSchedules(currentTime)
    setFanSpeed(fanSpeed)
```

This approach ensures that the air purifier operates based on the schedule while allowing for dynamic changes in fan speed as needed.

4.7.5 Fan Speed Control Based on IAQI

In addition to following user-defined schedules, the air purifier's fan speed is dynamically adjusted according to the calculated Indoor Air Quality Index (IAQI), as calculated in Section 4.5.2. This functionality ensures that the air purifier responds in real-time to changing air quality conditions, automatically adjusting its operation to maintain optimal indoor air quality.

The IAQI is categorized into four levels: Good, Moderate, Poor, and Critical, as outlined in Table 4.1. Based on the current IAQI level, the fan speed is adjusted as follows:

- **Good (IAQI Level: 0-100)**: The fan speed remains at the user-defined level, as the air quality is satisfactory.
- **Moderate (IAQI Level: 101-150)**: The fan speed increases by one level to improve air circulation and maintain healthy indoor air quality.
- **Poor (IAQI Level: 151-250)**: The fan speed increases by three levels to more aggressively address the deteriorating air quality.
- **Critical (IAQI Level: 251+)**: The fan operates at full power (fan speed 7) to rapidly purify the air and reduce harmful pollutants.

By dynamically adjusting the fan speed based on IAQI, the air purifier enhances its ability to maintain a safe and comfortable environment. This automatic adjustment occurs alongside the scheduled fan speed changes, ensuring that the air purifier responds effectively to both user preferences and real-time air quality conditions.

4.8 Conclusion

The development and implementation of the embedded software for the air purifier system underscore the importance of creating robust, efficient, and scalable software solutions in IoT-based applications. Through the integration of various sensors, handling real-time data, ensuring secure communication via MQTT with TLS, and enabling automated control based on dynamic air quality data, this system effectively demonstrates the impact of embedded software on enhancing indoor environments. The decision to utilize bare-metal programming ensures optimal resource efficiency, meeting the system's performance requirements.

5 Cloud Infrastructure Development

This chapter outlines the development of the server infrastructure that facilitates communication between the embedded software and the mobile application for the air purifier system. The goal is to implement a backend that enables the transfer of data, such as sensor readings and device schedules, between the embedded system and the mobile application. As shown in Figure 5.1, the core cloud components include an Azure App Service, SQL Database, and MQTT Broker, which collectively enable data communication, storage, and real-time messaging.

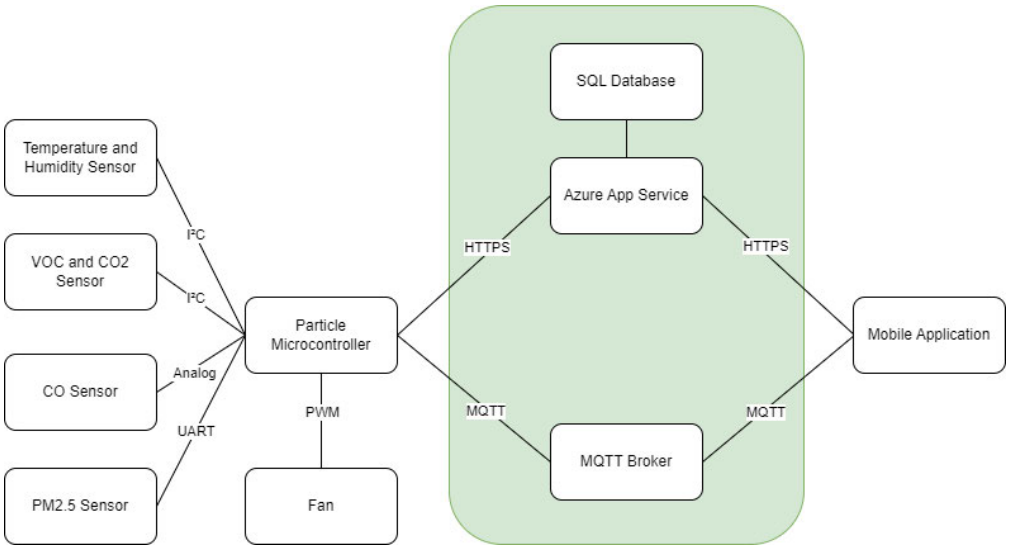

Figure 5.1: System Architecture Highlighting Cloud-Based Components in Green

The cloud infrastructure comprises an MQTT broker for real-time communication between the Particle microcontroller and the cloud, an SQL database for storing data such as sensor readings and schedules, and an Azure App Service that acts as a bridge between the database, the air purifier and the mobile application via HTTPS. This architecture

allows the mobile application to retrieve data and send commands, facilitating the full functionality of the air purifier system.

This chapter will focus on building this foundational cloud infrastructure, leaving advanced topics such as authentication, security, performance optimization, and load balancing for future development. The implementation serves as a base that can be enhanced with more complex features as needed.

By the end of this chapter, readers will understand how the cloud infrastructure supports the basic operations of the air purifier system, ensuring seamless communication between the embedded system and the mobile application.

5.1 Creating an MQTT Broker

This section details the process of setting up a private MQTT broker on an Azure virtual machine running Ubuntu's 20.04 LTS. The MQTT broker serves as the central hub for managing the secure and reliable transmission of data between IoT devices and a mobile application. To achieve this, a private broker is deployed on a cloud-based virtual machine, offering enhanced control and security over the data flow.

A step-by-step tutorial provided by Microsoft in their learning module, *Deploy a private MQTT broker* supported the MQTT broker deployment[19].

5.1.1 Preparing the Azure Virtual Machine

The first step in creating an MQTT broker involves setting up a virtual machine (VM) on Azure. This VM serves as the host for the MQTT broker. The following steps outline the creation and configuration of the VM:

1. **Create an Azure Virtual Machine:** The process begins by navigating to the Azure portal and creating a resource by selecting *Virtual Machine*. The image *Ubuntu Server 20.04 LTS Gen 1* is chosen for its compatibility and long-term support. The virtual machine is configured with a Standard_B1ls size (1 vCPU, 0.5 GiB memory) to balance performance and cost. An SSH public key is generated to ensure secure access to the VM.

2. **Configure Network Ports:** Once the VM is created, it is necessary to configure the network security group (NSG) to allow traffic on specific ports that are essential for the broker's operation. The following ports are configured:

   - **TCP 80:** This port is used for Let's Encrypt certificate renewal, ensuring that the broker's communication remains secure.
   - **TCP 8884:** This port is dedicated to encrypted MQTT communication, which requires a client certificate.
   - **TCP 8091:** This port handles MQTT over WebSockets, which is encrypted and authenticated.

3. **Enable Just-in-Time Access:** Just-in-Time (JIT) access is a security feature provided by Microsoft Defender for Cloud that helps protect Azure virtual machines (VMs) from unauthorized network access. JIT access reduces the attack surface by ensuring that access to VMs is only allowed when necessary and for a specified period of time. This access is configured in the Azure portal under the VM's *Configuration* pane [20].

4. **Connect to the Virtual Machine:** After the VM has been configured and JIT access is enabled, the next step is to connect to the VM. This is done via SSH using the private key that was generated during the VM creation process. The following command is used to establish the SSH connection:

   ```
   ssh -i <private_key_path> <username>@<host_dns_name>
   ```

   It may take a few moments to connect, especially if it's the first time accessing the VM.

5.1.2 Installing and Configuring the MQTT Broker

With the VM set up and secured, the next phase involves installing and configuring the Mosquitto MQTT broker. The Mosquitto broker is chosen for its reliability and widespread support in the IoT community.

1. **Install Required Packages:** Begin by updating the system packages to ensure that all dependencies are up-to-date. Then, install the Mosquitto broker along with its client tools using the following commands:

```
sudo apt update && sudo apt -y upgrade
sudo apt install -y mosquitto mosquitto-clients
```

2. **Generate and Install Certificates:** To secure communication between the MQTT broker and clients, a self-signed certificate is generated using OpenSSL. This process involves creating a certificate authority (CA), generating a server certificate, and issuing client certificates. The commands used are as follows:

```
openssl req -new -x509 -days 730 -nodes -extensions v3_ca \
-keyout ca.key -out ca.crt

openssl genrsa -out server.key 2048
openssl req -new -out server.csr -key server.key \
-subj "/CN=<Your_DNS_Name>"
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 730
```

   These certificates are crucial for ensuring encrypted communication between the broker and connected devices.

3. **Configure Mosquitto:** After generating the necessary certificates, the Mosquitto configuration file is edited to specify the ports on which the broker will listen, and the locations of the certificate files. The configuration file is modified using the following command:

```
sudo nano /etc/mosquitto/conf.d/default.conf
```

   The configuration should include the following entries:

```
listener 8884
cafile /etc/mosquitto/ca_certificates/ca.crt
certfile /etc/mosquitto/ca_certificates/server.crt
keyfile /etc/mosquitto/ca_certificates/server.key
```

```
require_certificate true

listener 8091
protocol websockets
cafile /etc/mosquitto/ca_certificates/ca.crt
certfile /etc/mosquitto/ca_certificates/server.crt
keyfile /etc/mosquitto/ca_certificates/server.key
```

This configuration ensures that the broker operates securely and that all MQTT messages are encrypted.

5.1.3 Starting the MQTT Broker

Once the configuration is complete, the Mosquitto broker is started as a daemon service to run in the background. This ensures that the broker is always available to handle incoming MQTT messages:

```
sudo systemctl enable mosquitto
sudo systemctl start mosquitto
```

To verify that the broker is functioning correctly, it is advisable to test the broker using Mosquitto client tools from both a remote machine and the Particle Photon device. These tests help confirm that the broker is correctly configured and can securely handle MQTT communications.

5.1.4 Conclusion

Setting up a private MQTT broker on an Azure VM significantly enhances the security and reliability of IoT systems, such as the air purifier system discussed in this thesis. By controlling the entire communication chain, sensitive data is protected, and the system's performance is predictable under varying network conditions.

5.2 Azure SQL Database

This section delves into the design and structure of a Structured Query Language (SQL) database that forms the backbone of the server-side data storage. For this project, an Azure SQL Database has been selected to handle the storage needs. Azure SQL Database is a fully managed relational database service provided by Microsoft, designed for high availability, scalability, and security.

The SQL database is responsible for storing various types of data, including user information, device details, sensor data, cleaning schedules, and other relevant information. This database plays a critical role in ensuring that data is organized, easily accessible, and efficiently managed. By leveraging the capabilities of Azure SQL Database, the system benefits from automated backups, built-in security features, and seamless scalability, making it well-suited for handling the dynamic data generated by the air purifier system [22].

5.2.1 Database Schema Design

The design of the SQL database schema is a critical step in ensuring that the data generated by the air purifier and the mobile application is stored efficiently and can be accessed and managed in a structured manner.

The following tables represent the core components of the database schema:

- **Sensor Data Table:** This table stores the sensor data generated by the devices.
  - **Columns:**
    - `device_id` (Foreign Key) - Links to the device that generated the data.
    - `temperature` - The recorded temperature.
    - `humidity` - The recorded humidity level.
    - `voc` - The recorded Volatile Organic Compounds (VOC) level.
    - `pm2_5` - The recorded PM2.5 concentration.
    - `co2` - The recorded CO2 concentration.

    * `co` - The recorded Carbon Monoxide (CO) concentration.

- **Device Properties Table:** This table stores information about the devices, including device details and user-defined schedules.
  - **Columns:**
    * `device_id` (Primary Key) - Unique identifier for each device.
    * `device_name` - User-defined name for the device.
    * `device_schedule` - A text field that stores the device's schedule in a structured format (e.g., JSON or serialized data).
- **Users Table:** This table stores user information and links users to their devices.
  - **Columns:**
    * `user_id` (Primary Key) - Unique identifier for each user.
    * `email` - The user's email address.
    * `password_hash` - Hashed password for authentication.
    * `api_key` - API key associated with the user for authentication.
- **User Devices Table:** This junction table handles the many-to-many relationship between users and devices, as a user can own multiple devices, and a device can be associated with multiple users (e.g., in a household).
  - **Columns:**
    * `user_id` (Foreign Key) - Links to the Users table.
    * `device_id` (Foreign Key) - Links to the Device Properties table.

The schema is designed to support scalability and future expansion, allowing for the addition of new sensors, devices, and features as the system evolves.

Figure 5.2 illustrates the database schema, highlighting the relationships between the tables. The design ensures that each device is linked to its corresponding data and users, enabling efficient data management and access. The structure supports future scalability, allowing the addition of new devices, sensors, and features as the system evolves.

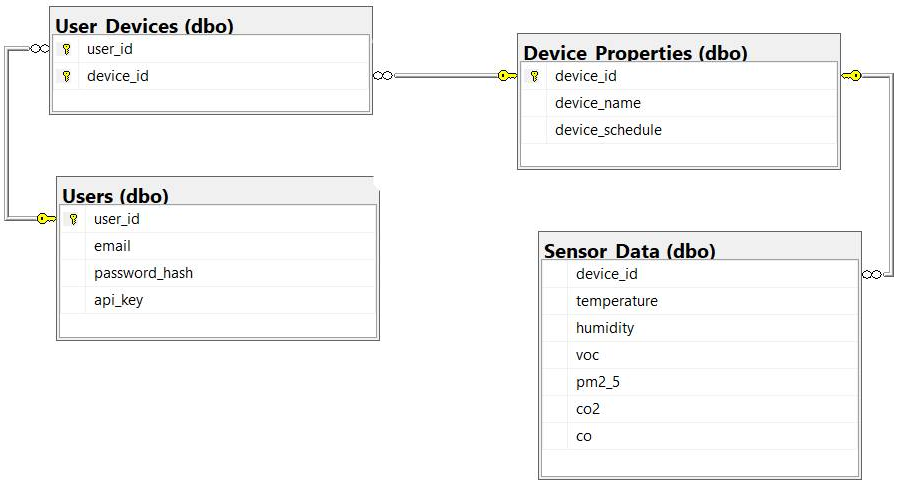

Figure 5.2: Azure SQL Database Schema Design

5.3 REST API Development with Azure App Service

The Representational State Transfer (REST) API plays a vital role in the server-side architecture of the air purifier system, acting as the interface between the mobile application and the SQL database. The API is responsible for fetching data from the database, such as sensor readings and device schedules, as well as adding new data, such as user-generated schedules or device configurations. This API is deployed on Azure App Service and built using Node.js, providing a scalable and reliable backend for the air purifier system.

5.3.1 Rationale for Azure App Service and Node.js

The decision to use Azure App Service and Node.js for developing and deploying the REST API was driven by several key factors that align with the requirements of the air purifier system:

- **Scalability and Flexibility:** Azure App Service provides a highly scalable environment, which is essential for handling varying loads as more devices are added to the system. The automatic scaling features of Azure App Service allow the backend

to handle increased traffic seamlessly, ensuring reliable performance as the system grows [21].

- **Integration with Azure Services:** Azure App Service integrates seamlessly with other Azure services, such as Azure SQL Database, enabling streamlined access to storage, databases, and other cloud functionalities [16].
- **Security Features:** Azure App Service automatically manages HTTPS for the application, ensuring that all data transmitted between the client and the server is securely encrypted. Additionally, it provides options for enforcing modern secure protocols like TLS 1.2, while disabling outdated protocols, to protect against vulnerabilities and ensure secure communication [17].
- **Node.js Efficiency:** Node.js's non-blocking, event-driven architecture allows it to handle thousands of concurrent connections efficiently without introducing the complexity of managing thread concurrency, which can be a significant source of bugs [27].

Given these advantages, Azure App Service combined with Node.js was chosen as the optimal platform for developing and deploying the REST API. This choice provides the necessary scalability, performance, and ease of integration needed to support the real-time demands of the air purifier system, while also ensuring that the backend remains secure and maintainable.

5.3.2 REST API Architecture

The REST API architecture is designed to facilitate communication between mobile application SQL database. This API acts as an intermediary, allowing the mobile application to retrieve sensor data, submit new schedules, and manage device configurations. The architecture follows REST principles, using standard HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE` to perform CRUD (Create, Read, Update, Delete) operations on the system's resources.

Endpoints and Resources

The REST API is structured around key resources, each of which represents a core entity in the system, such as devices, schedules, and sensor data. All endpoints are versioned,

starting with `api/v1/`, to ensure future-proofing and easy version control. The following are the primary endpoints:

- **GET /api/v1/devices/{device_id}:** Retrieves detailed information about a specific device, including its current configuration and schedule.
- **POST /api/v1/devices:** Adds a new device to the system, linking it to the authenticated user.
- **GET /api/v1/sensors/{device_id}:** Retrieves the sensor data for a specific device.
- **POST /api/v1/sensors:** Submits new sensor data to the system.
- **GET /api/v1/schedules/{device_id}:** Retrieves the current schedule for a specific device.
- **PUT /api/v1/schedules/{device_id}:** Updates the schedule of a specific device.

Each endpoint interacts with the SQL database, performing operations such as fetching sensor data, updating schedules, or retrieving device configurations. The data is transferred between the client (mobile application) and the server in JSON format, ensuring compatibility and ease of use.

6 Mobile Application Development

The mobile application is the primary interface for users to monitor and control the air purifier. Built using React Native and focusing on Android, the app enables users to configure Wi-Fi credentials, access real-time sensor data, set operational schedules, and manage the air purifier remotely. These functionalities rely on secure communication with the embedded system and cloud infrastructure to deliver an efficient user experience.

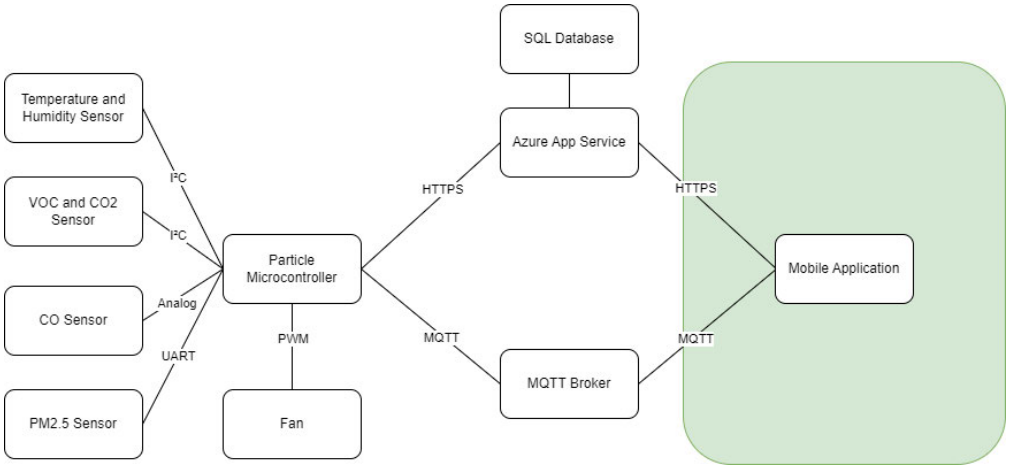

Figure 6.1: System Architecture Highlighting The Mobile Application in Green

This chapter details the technical implementation of the app's core features, including communication protocols, sensor data handling, and user interface design, and how these components integrate with the overall system.

6.1 Mobile Application Workflow

The mobile application provides a user-friendly interface for managing the air purifier, focusing on three primary functions: setting Wi-Fi credentials, configuring schedules, and

displaying real-time sensor data. Each of these functions plays a critical role in ensuring the smooth operation of the air purifier system.

The Wi-Fi credentials setup allows users to connect the air purifier to their local network. The app communicates with the air purifier via MQTT to confirm that the Wi-Fi credentials are correctly set. Once connected, the app registers the device in the central database via HTTP, enabling remote management through the cloud infrastructure.

For monitoring indoor air quality, the app displays real-time sensor data. Initially, it fetches the latest sensor readings from the cloud database using an HTTP request, ensuring that users can view up-to-date information immediately. Once the app establishes an MQTT connection, it begins receiving real-time sensor updates from the air purifier, providing continuous monitoring of indoor air quality.

The scheduling feature allows users to configure the air purifier's operation. When a user sets a new schedule, the app stores it in the cloud database via HTTP, ensuring the schedule is available for the air purifier during its next startup. If the air purifier is online, the app also sends the schedule via MQTT for immediate application, allowing real-time adjustments to the device's operation.

By leveraging both HTTP for structured data management and MQTT for real-time communication, the mobile application ensures a seamless and responsive user experience.

6.2 Permissions and Initial Setup

Before implementing the core functionality of the mobile application, it is essential to configure the necessary permissions and install the required libraries. These configurations will enable communication between the application and the air purifier, ensure secure network connections, and allow for seamless user interactions. This section outlines the steps required to configure these permissions and initialize the necessary libraries, providing a solid foundation for the subsequent implementation.

6.2.1 Permissions Configuration

To enable the mobile application to interact with Wi-Fi networks and communicate with the Particle Photon's SoftAP, specific permissions must be configured. These permissions are crucial for allowing the application to scan for available Wi-Fi networks, connect to the Particle Photon's SoftAP, and transmit HTTP requests over the network.

Android Permissions

On Android devices, the `ACCESS_FINE_LOCATION` permission must be added to the `AndroidManifest.xml` file to enable Wi-Fi network scanning, especially for devices running Android 6.0 or later [12]:

```
<uses-permission android:name=
    "android.permission.ACCESS_FINE_LOCATION" />
```

Network Security Configuration

In addition to the permissions outlined above, it is necessary to allow cleartext HTTP (unencrypted) communication with the Particle Photon's SoftAP, which operates on a local IP address `192.168.0.1` [33]. This is achieved by configuring a `network_security_config.xml` [3].

The `network_security_config.xml` file should be placed in the `android/app/src/main/res` directory, with the following configuration:

Listing 6.1: Network Security Configuration XML Example

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config cleartextTrafficPermitted="true">
        <domain includeSubdomains="true">192.168.0.1</domain>
    </domain-config>
</network-security-config>
```

This configuration permits cleartext traffic to the specified domain, ensuring that the application can communicate with the Particle Photon's SoftAP for setting Wi-Fi credentials.

Runtime Permission Requests

In addition to declaring permissions, it is necessary to request them at runtime, especially on Android devices running version 6.0 or later. The `react-native-permissions` library can be utilized to manage these runtime permission requests [12].

The library can be installed using the following command:

```
npm install react-native-permissions
```

An example of requesting permissions within the application is provided in the appendix (see Appendix B.2 for the full implementation).

6.2.2 Libraries Setup

To successfully interact with the Particle Photon's SoftAP and manage the Wi-Fi credential setup process, several libraries are required. The following section details the libraries necessary for this implementation and provides instructions on their installation.

React Native Wi-Fi Manager

The `react-native-wifi-reborn` library is employed to manage Wi-Fi connections, including scanning for available networks and connecting to the Particle Photon's SoftAP [12].

The library can be installed using the command:

```
npm install react-native-wifi-reborn
```

HTTP Requests (Axios)

To facilitate HTTP communication with the Particle Photon's SoftAP and with the cloud, the `axios` library is utilized. This library simplifies the handling of HTTP requests within the application [5].

The library can be installed using the following command:

```
npm install axios
```

MQTT Client

For communication between the mobile application and the air purifier via MQTT, the `react-native-mqtt` library is employed. This library enables the application to publish and subscribe to MQTT messages [45].

The library can be installed using the command:

```
npm install react-native-mqtt
```

Configuring Navigation

The mobile application includes multiple screens for managing various features, such as setting Wi-Fi credentials, monitoring sensor data, and configuring schedules. To navigate between these screens seamlessly, React Navigation is used [38].

React Navigation provides a flexible and robust solution for handling screen transitions and managing navigation stacks in React Native applications. To install React Navigation and the necessary dependencies, the following commands can be executed:

```
npm install @react-navigation/native
npm install @react-navigation/stack
```

The complete navigation setup for a React Native application, including the configuration of screens in the `App.js` file, is provided in the appendix (see Appendix B.3 for the full implementation).

6.3 Configuring Wi-Fi Credentials

In modern IoT applications, enabling users to intuitively and swiftly configure and manage devices is vital. This section covers the process of setting up Wi-Fi credentials for the Particle Photon microcontroller that powers the air purifier. The Particle Photon, acting as the core unit for capturing sensor data and controlling the air purifier, requires a connection to a local Wi-Fi network to communicate with a cloud-based MQTT broker and the mobile application.

To facilitate this, the user must securely set their own Wi-Fi credentials via the mobile application rather than relying on hardcoded credentials, which would lack flexibility and pose security risks. The process involves connecting to the Particle Photon's SoftAP (Software Enabled Access Point), retrieving available networks, securely transmitting credentials, and verifying a successful connection. Additionally, this setup process includes registering the device in the database to ensure the air purifier's functions, such as data transmission and remote control, are fully accessible.

The sequence diagram in Figure 6.2 outlines this process. In this section, each step shown in the diagram will be thoroughly explained, covering necessary interactions between the mobile application, the air purifier, and the backend server.

6.3.1 ParticleDeviceSetup Class

The `ParticleDeviceSetup` class is an adaptation of a code example provided in the Particle documentation, which demonstrates how to interact with the Particle Photon microcontroller through its SoftAP (Soft Access Point) HTTP server. The code is used to set Wi-Fi credentials for the device by creating a temporary access point and an HTTP server on port 80. The server is responsible for configuring the Wi-Fi access points that the Particle Photon attempts to connect to. The HTTP URLs provided by the system allow interaction with the Particle Photon's API, enabling the setup of Wi-Fi credentials [33].

The code example from the Particle documentation details various endpoints necessary for setting Wi-Fi credentials, such as fetching the device ID, obtaining the public key, retrieving the network list, sending Wi-Fi credentials, and connecting to the network. These elements are crucial for ensuring the device can securely connect to the desired Wi-Fi network (see Table 6.1 for a summary of these methods).

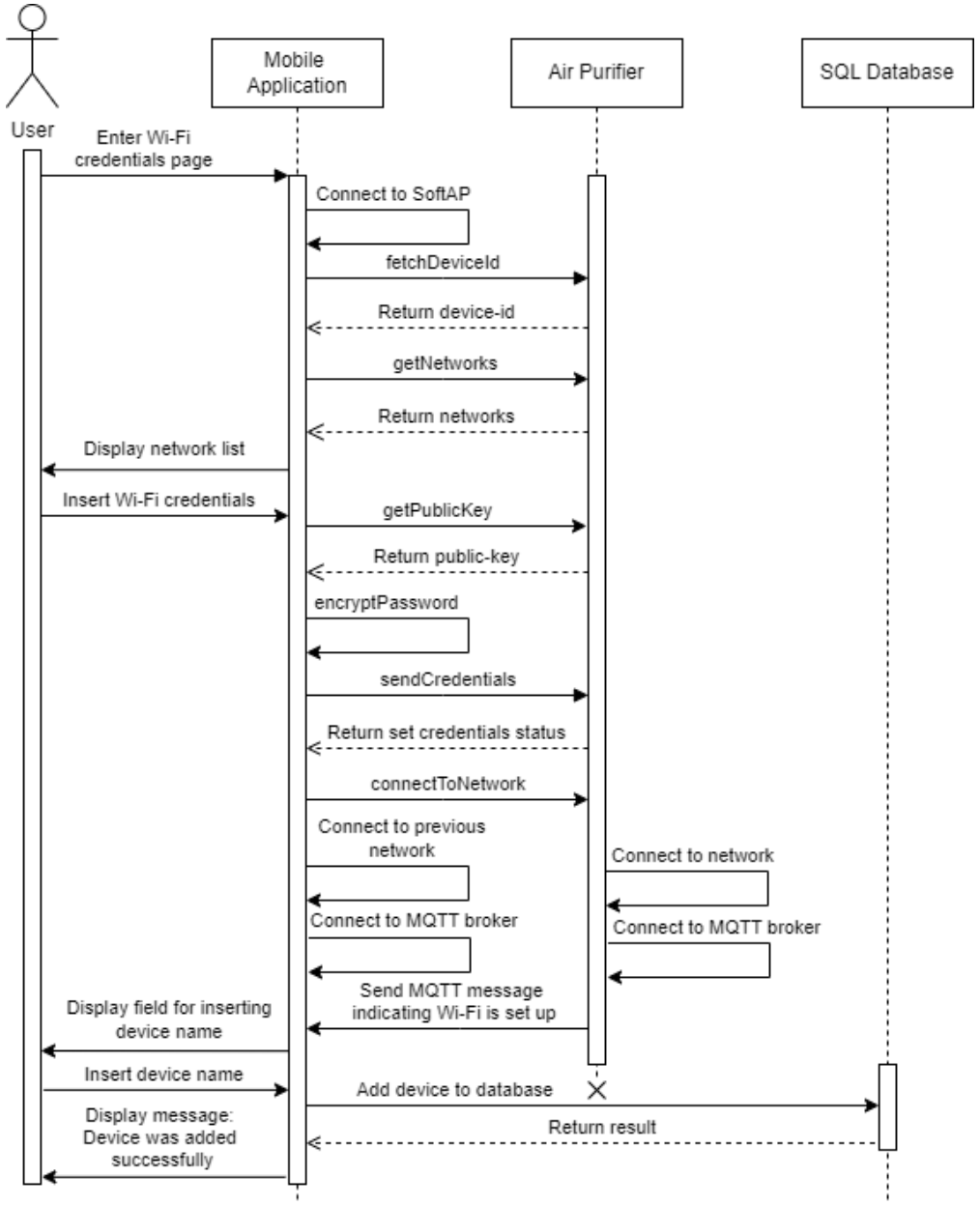

Figure 6.2: Sequence diagram of the Wi-Fi setup and device registration process.

For a detailed explanation of the methods implemented in the `ParticleDeviceSetup` class, including the pseudocode for each method, please refer to Appendix B.4.

Method	Description	Endpoint and HTTP Method
`fetchDeviceId`	Retrieves the unique device ID of the Particle Photon, which is essential for device identification during setup.	`/device-id` (HTTP GET)
`connectToNetwork`	Initiates the connection between the Particle Photon and the selected Wi-Fi network.	`/connect-ap` (HTTP POST)
`getNetworks`	Retrieves a list of available Wi-Fi networks that the Particle Photon can detect, returning details such as SSID, RSSI, security, and channel.	`/scan-ap` (HTTP GET)
`getPublicKey`	Retrieves the public RSA key from the Particle Photon, which is necessary for encrypting the Wi-Fi password.	`/public-key` (HTTP GET)
`encryptPassword`	Encrypts the Wi-Fi password using the RSA public key obtained from the Particle Photon, ensuring secure transmission.	N/A (Local Method)
`sendCredentials`	Configures the Particle Photon by sending the selected Wi-Fi network details including the encrypted password.	`/configure-ap` (HTTP POST)

Table 6.1: Summary of Particle Photon API Methods

6.3.2 Wi-Fi Credential Setup Process

The process of setting up Wi-Fi credentials on the Particle Photon through a React Native mobile application involves several key steps, each of which is essential to ensuring a secure and reliable connection to the desired Wi-Fi network. Additionally, this process includes adding the device to the database, allowing the mobile application to manage and display the device's status and data. This subsection outlines the step-by-step workflow of the Wi-Fi credential setup process, from connecting to the Particle Photon's SoftAP to verifying the connection and registering the device within the mobile application and the database.

Step 1: Connecting to the Particle Photon's SoftAP

The first step in the Wi-Fi credential setup process involves connecting the mobile application to the Particle Photon's SoftAP. The Particle Photon, when in SoftAP mode, creates a temporary Wi-Fi network with an SSID typically prefixed with "Photon-". The mobile application scans for available Wi-Fi networks, identifies the Particle Photon's SoftAP, and connects to it.

1. Request necessary permissions to access and modify Wi-Fi settings.
2. Scan for available Wi-Fi networks.
3. Identify the Particle Photon's SoftAP by its SSID (e.g., `Photon-XXXX`).
4. Connect to the identified SoftAP.

Step 2: Fetching Device ID and Network List

Once connected to the Particle Photon's SoftAP, the mobile application needs to retrieve essential information from the device, including its unique device ID and the list of available Wi-Fi networks that the Particle Photon can detect. This information is crucial for the subsequent steps in the setup process.

1. Use the `fetchDeviceId` method from the `ParticleDeviceSetup` class to fetch the device ID.
2. Use the `getNetworks` method from the `ParticleDeviceSetup` class described in 6.3.1 to retrieve the list of available Wi-Fi networks.
3. Display the list of detected Wi-Fi networks to the user within the mobile application.

Step 3: User Selection and Credential Input

In this step, the user selects the desired Wi-Fi network from the list provided by the mobile application and enters the corresponding password. This information will be securely transmitted to the Particle Photon to configure the network connection.

1. Allow the user to select a Wi-Fi network from the list displayed in the application.

2. Prompt the user to enter the Wi-Fi password for the selected network.
3. Optionally provide the user with the ability to toggle the visibility of the password field.

Step 4: Sending Wi-Fi Credentials and Connecting to the Network

Once the user has entered the Wi-Fi credentials, the mobile application uses the `ParticleDeviceSetup` class to encrypt the password and send the credentials to the air purifier. The `sendCredentials` method is used to transmit the network details and encrypted password to the device.

1. Use the `getPublicKey` method from the `ParticleDeviceSetup` class to retrieve the RSA public key.
2. Encrypt the Wi-Fi password using the `encryptPassword` method from the `ParticleDeviceSetup` class.
3. Use the `sendCredentials` method from the `ParticleDeviceSetup` class to send the encrypted credentials and network details to the Particle Photon.
4. Use the `connectToNetwork` method from the `ParticleDeviceSetup` class to initiate the connection to the Wi-Fi network.

Step 5: Verifying Connection and Registering the Device

As mentioned in the Section 4.3.4, the mobile application cannot directly determine whether the Wi-Fi credentials are correct and whether the Particle Photon successfully connected to the network. To overcome this limitation, MQTT is used to verify the connection status.

1. Reconnect the mobile device to its original Wi-Fi network.
2. Subscribe to the MQTT topic `{device-id}/connection-status/` to monitor the connection status of the Particle Photon.
3. Wait for an MQTT message indicating that the Particle Photon has successfully connected to the Wi-Fi network, confirming that the credentials were correct.

4. Upon receiving the connection confirmation, send a confirmation message back to the Particle Photon via the MQTT topic `{device-id}/cmd/` to indicate that the setup was successful.

5. If the connection is successful:

   a) Register the device in the mobile application by sending a request to the backend server.

   b) Add the device to the user's list of registered devices.

   c) Display the device and its status in the application.

6.4 Error Handling and Edge Cases

In the Wi-Fi credential setup process for the Particle Photon, various errors and edge cases may arise that need to be addressed to ensure a smooth and reliable user experience. This section discusses potential issues that could occur during the setup process and outlines strategies for handling these scenarios effectively.

6.4.1 Handling Wi-Fi Connection Issues

There are scenarios where the Particle Photon either fails to connect to the specified Wi-Fi network due to incorrect credentials or successfully connects but fails to communicate the connection status back to the mobile application. When the Wi-Fi setup is unsuccessful or the connection status is not communicated, the mobile application does not receive the expected MQTT message indicating that the Wi-Fi setup was correct. As a result, the app is unable to determine whether the air purifier has successfully established a connection to the Wi-Fi network, leaving the user uncertain about the device's status. In both cases, the mobile application responds in a similar manner:

1. **Timeout Mechanism:** A timeout mechanism was implemented in the mobile application to wait for an MQTT message from the Particle Photon confirming a successful connection. The timeout period was set to 45 seconds, which is sufficient for the mobile application to reconnect to the home Wi-Fi network, subscribe to the MQTT broker, and receive a confirmation message. This duration ensures that the process is not overly long, preventing unnecessary waiting time for the user. If no

message is received within this period, the application assumes that the connection attempt has failed and prompts the user to retry.

2. **Error Notification:** If the connection fails, either due to incorrect Wi-Fi credentials or unsuccessful feedback from the device, an error message is displayed to the user. The application then prompts the user to re-enter the Wi-Fi credentials and retry the setup process.

6.4.2 Network Disconnection During Setup

Another potential issue is the disconnection of the mobile device from the Particle Photon's SoftAP during the setup process. Without a stable connection to the SoftAP, the mobile application cannot fetch the device ID, retrieve the list of available networks, or, most importantly, send Wi-Fi credentials to the air purifier. This disconnection can disrupt the setup flow and prevent the successful configuration of the Wi-Fi credentials.

1. **Connection Monitoring:** Connection monitoring was implemented in the mobile application to detect if the device lost connection to the Particle Photon's SoftAP.

2. **Reconnection Attempt:** If a disconnection was detected, the mobile application automatically attempts to reconnect to the SoftAP network.

3. **User Notification:** If automatic reconnection failed, the user is notified of the disconnection, and the application automatically moves to the page that scans for available Wi-Fi networks.

6.4.3 Failure to Register the Device in the Application

Even if the Particle Photon successfully connects to the Wi-Fi network, there may be cases where the mobile application fails to register the device in the backend database. This issue can prevent the device from being visible or accessible in the application.

1. **Validation of Device Registration:** After receiving the MQTT message confirming a successful connection, the mobile application should send a request to the backend server to register the device. The response from the server indicates whether the device registration is completed successfully.

2. **User Notification:** If the registration continues to fail, the application notify the user of the issue and moves to the page that scans for available Wi-Fi networks.

Summary

The Wi-Fi credential setup process is a critical component of ensuring that the air purifier can reliably connect to a Wi-Fi network and communicate with the mobile application. By following the steps outlined above, the mobile application can guide the user through the setup process, handle potential errors, and ensure that the device is successfully registered and ready for use. This process involves interacting with the `ParticleDeviceSetup` class for configuration, securing the transmission of sensitive data, and verifying the connection through MQTT, ultimately resulting in a seamless user experience. Additionally, the process includes adding the device to the database ensuring it is fully integrated into the user's environment.

6.5 Displaying Sensor Values in the Mobile Application

After successfully setting the Wi-Fi credentials in the air purifier and registering the device in the database, users can monitor the sensor values transmitted by the air purifier. This section focuses on the process of retrieving and displaying these sensor values within the mobile application. In addition to the sensor data, the application also displays the Indoor Air Quality Index (IAQI), which is calculated directly on the microcontroller. The IAQI is sent to the mobile application via MQTT alongside the real-time sensor readings. This comprehensive data gives users a clear view of their indoor air quality, including temperature, humidity, VOC levels, CO2 levels, particulate matter concentrations, and CO levels. The data transmission mechanism relies on the MQTT protocol, as detailed in Section 4.6.1.

6.5.1 Initial Data Retrieval and Display

Upon entering the sensor data display page in the mobile application, an initial step is performed to fetch the sensor values from the server via an HTTPS request to the endpoint `/api/v1/sensors/{device_id}`. This pre-fetching is necessary due to the inherent delay in establishing an MQTT connection, subscribing to the appropriate

topic, and beginning to receive real-time sensor data. By retrieving the most recent sensor data from the database, the application ensures that users are immediately presented with relevant information upon navigating to the page, thereby enhancing the user experience.

6.5.2 Real-Time Data Update via MQTT

Once the MQTT connection is successfully established, the sensor values and the calculated IAQI are updated in real-time. The MQTT protocol, which operates with a publish-subscribe model, delivers sensor data, including the IAQI, to the mobile application every second. As new data is received, the application dynamically updates the displayed values to reflect the current state of the indoor air quality.

6.5.3 Visualization of Sensor Values

The sensor values and IAQI are presented to the user through a visually intuitive interface. The values are displayed using a bar that fills according to the magnitude of the sensor readings and the calculated IAQI. Additionally, the color of the bar is dynamically adjusted based on the air quality levels, as defined in Table 4.1.

This color-coded representation provides users with an immediate understanding of the air quality in their environment, allowing them to take necessary actions if the air quality deteriorates.

6.6 Setting a Schedule in the Air Purifier

In the mobile application, users can create and manage schedules for their air purifier, customizing its operation based on their preferences. The scheduling feature allows users to select specific days of operation, define start and end times, and set the desired fan speed. Once the user inputs these details, the application processes the input and converts it into a specific string format that the air purifier's microcontroller can parse and execute. This string format is based on a predefined structure that encodes all the necessary scheduling details, as described in detail in Section 4.7.1.

6.6.1 User Input for Scheduling

The scheduling feature offers several configurable parameters, including:

- **Days of Operation**: Users can choose specific days during which the air purifier should operate according to the schedule.
- **Start Time**: Users specify the exact hour and minute when the air purifier should begin operation.
- **End Time**: The user also sets the time at which the air purifier should stop operating.
- **Fan Speed**: The user selects a fan speed between 1 and 7, with 1 being the lowest speed and 7 being the highest.

After confirming their choices, the application converts the user's input into a string format that the air purifier can interpret, ensuring that the device operates according to the defined schedule.

6.6.2 Sending the Schedule to the Air Purifier

Once the schedule is formatted, the application sends it to the backend server using a PUT request to the following endpoint:

```
/api/v1/schedules/{device_id}
```

This process stores the schedule in the database, preserving it even if the air purifier is temporarily offline or disconnected from the network. When the air purifier reconnects, it retrieves the stored schedule from the server and adjusts its operation accordingly.

Simultaneously, the schedule is also transmitted directly to the air purifier via MQTT. The message sent to the MQTT topic ‘device-id/cmd/‘ includes the keyword "SCHEDULE" followed by the encoded schedule string. The air purifier then parses this message and adjusts its operation based on the received instructions.

If the air purifier is online when the schedule is created, it will immediately receive the schedule via MQTT and adjust its operation in real time. This dual mechanism

ensures that the device always stays synchronized with the user's preferences, regardless of network connectivity.

7 Testing and Validation

Testing is a critical phase in the development of the intelligent air purifier system, ensuring that all components—embedded software, cloud infrastructure, and the mobile application—function correctly and cohesively. The goal of the testing phase is to validate the system's performance, reliability, and robustness in real-world scenarios. This section outlines the comprehensive testing process undertaken to evaluate the system's behavior across various conditions, including Wi-Fi connectivity, sensor data acquisition, data transmission, and user interaction via the mobile application.

This section details the methodologies, tools, and results of the testing procedures, highlighting any challenges encountered and how they were addressed. The outcomes of the testing phase provide insights into the system's reliability and readiness for deployment.

7.1 Testing Sensors and IAQ

Due to the absence of proper calibration tools, measuring the accuracy of the sensors and performing their calibration is not currently feasible. This calibration process will be addressed in future work. However, during testing, sensor values were observed in real-time to observe changes in response to varying environmental conditions.

For instance, when blowing on the sensors, noticeable changes were observed in the readings for CO_2 and VOC concentrations, indicating that the sensors are responsive to environmental changes. This provides a basic validation that the sensors are functioning as expected, even though precise accuracy measurements were not performed.

The next step in testing was to evaluate the Indoor Air Quality (IAQ) by calculating the Indoor Air Quality Index (IAQI). Figure 7.1 shows the sensor readings at a specific point in time during testing. Using the IAQI calculation formula from Equation 4.2, and the pollutant-specific breakpoints from Tables 4.1 and 4.2, the following IAQI values were computed for each pollutant:

- **CO**: Based on a concentration of 0.40 ppm, the calculated IAQI is **9**.

$$\text{IAQI} = \frac{100 - 0}{4 - 0} \times (0.40 - 0) + 0 = 10$$

- **CO_2**: Based on a concentration of 448 ppm, the calculated IAQI is **44**.

$$\text{IAQI} = \frac{100 - 0}{1000 - 0} \times (448 - 0) + 0 \approx 44$$

- **TVOC**: Based on a concentration of 5 ppb, the calculated IAQI is **5**.

$$\text{IAQI} = \frac{100 - 0}{100 - 0} \times (5 - 0) + 0 = 5$$

- **PM2.5**: Based on a concentration of 18.18 µg/m^3, the calculated IAQI is **51**.

$$\text{IAQI} = \frac{100 - 0}{35 - 0} \times (18.18 - 0) + 0 \approx 51$$

It is important to note that the highest IAQI value among all the calculated values is used to determine the overall air quality. In this case, the highest IAQI value is **51**, which corresponds to the PM2.5 concentration. Indeed, as shown in Figure 7.1, the air purifier also calculated an IAQI value of **51**, confirming the system's accurate response to the sensor data.

During testing, it was also observed that as the IAQ deteriorated and the IAQI increased, the air purifier automatically adjusted its fan speed to a higher setting. Conversely, when the IAQ improved and the IAQI decreased, the fan speed was automatically reduced.

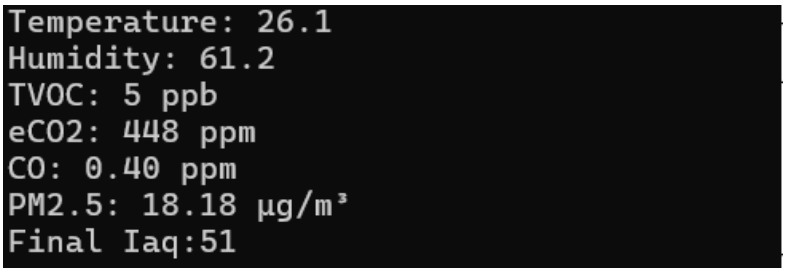

Figure 7.1: Sensor readings at a specific point in time during testing as generated by the microcontroller.

These tests confirm that the system successfully meets both **REQ1.1** and **REQ1.4** in Table 2.1. REQ1.1 requires continuous air quality monitoring using various sensors, while REQ1.4 specifies automated fan speed adjustment based on real-time air quality measurements. Although this testing approach is limited in scope, the results provide an initial validation of the embedded system's ability to read sensor values, calculate the IAQI, and adjust the air purifier's operation accordingly. Future work will focus on calibrating the sensors and verifying their accuracy under controlled conditions.

7.2 Wi-Fi Connectivity Testing

Wi-Fi connectivity is a crucial aspect of the air purifier's operation, as it enables communication with the cloud and the mobile application. To test the system's Wi-Fi functionality, a hotspot was created using a laptop, allowing for better control over network conditions. Three different test cases were performed to evaluate how the air purifier handles various Wi-Fi connectivity scenarios:

- **Case 1: Starting the Air Purifier with Wi-Fi Available**

  In this test, the air purifier was turned on while the Wi-Fi hotspot was already active. The device successfully connected to the network and maintained a stable connection throughout the test, continuing its operation as expected.

- **Case 2: Starting the Air Purifier without Wi-Fi**

  In this scenario, the air purifier was powered on while the Wi-Fi hotspot was turned off, simulating a situation where no network is available at startup. As expected, the device was unable to connect initially. However, once the Wi-Fi hotspot was activated, the air purifier automatically connected to the available network and resumed normal operation. This test demonstrated the air purifier's ability to recover from a network outage during startup.

- **Case 3: Network Interruption During Operation**

  For this test, the air purifier was turned on with the Wi-Fi hotspot active. After connecting to the network and operating normally, the hotspot was turned off to simulate a network interruption. The air purifier continued its operation, though it was unable to send sensor data to the mobile application. Once the hotspot was

reactivated, the air purifier automatically reconnected to the network and resumed its full functionality.

The results of these tests indicate that the air purifier's embedded system effectively handles different network conditions. Whether starting with an active network, recovering from a network outage, or reconnecting after a network interruption, the air purifier demonstrated reliable Wi-Fi connectivity and resilience to connectivity changes. Therefore, the requirement **REQ1.0** outlined in Table 2.1 has been successfully met.

7.3 MQTT Connectivity Testing

Ensuring reliable MQTT connectivity is essential for the air purifier to transmit sensor data to the cloud and mobile application. Three test cases were conducted to evaluate how the MQTT client behaves under various network conditions:

- **Case 1: Starting the Air Purifier with Wi-Fi Available**

  The air purifier was started with an active Wi-Fi hotspot, and it successfully established an MQTT connection. Sensor data was transmitted consistently, confirming that the system operates as expected when the network is available from startup.

- **Case 2: Starting the Air Purifier without Wi-Fi**

  The air purifier was started without an active Wi-Fi network, resulting in a failed initial MQTT connection. After turning on the Wi-Fi hotspot, the air purifier automatically connected to the network and re-established the MQTT connection, resuming sensor data transmission. This demonstrated the system's ability to recover from network outages.

- **Case 3: Network Interruption During Operation**

  After starting the air purifier with Wi-Fi, an MQTT connection was successfully established, and sensor data was transmitted. However, when the Wi-Fi was turned off and back on, the air purifier reconnected to the network but failed to re-establish the MQTT connection. Consequently, sensor data transmission did not resume, indicating a need for improved reconnection logic within the MQTT client.

The tests revealed that while the system successfully handles normal startup conditions (Case 1) and recovers from network outages during startup (Case 2), it struggles to reconnect to the MQTT broker following a network interruption during operation (Case 3). This indicates a need for enhanced error handling and reconnection mechanisms in the MQTT client to ensure consistent data transmission in fluctuating network conditions.

As a result, requirement **REQ2.0** from Table 2.2, which specifies automatic recovery from disruptions, is not fully met. Further development is needed to address this issue and improve the system's resilience to network interruptions.

7.4 Testing Schedule Parsing

In testing the schedule parsing functionality, a string consisting of multiple schedule entries was created. Each schedule entry was separated by a comma, following the format outlined in Section 4.7. An example string used for testing was:

```
0200c000002465001,0200c000004465002,020071e00039ab003
```

The parsed results for these schedules followed the expected format, confirming that the schedule parsing function accurately extracted the necessary information from the schedule strings. This allowed the system to execute the corresponding operations at the correct times with the designated fan speeds.

Figure 7.2 illustrates the parsed schedule entries as displayed by the microcontroller. The figure shows each schedule, including the corresponding UTC offset, start and end times, day of the week, and fan speed settings. The detailed breakdown of each parsed schedule entry can be found in Appendix A, where the corresponding schedule components are outlined.

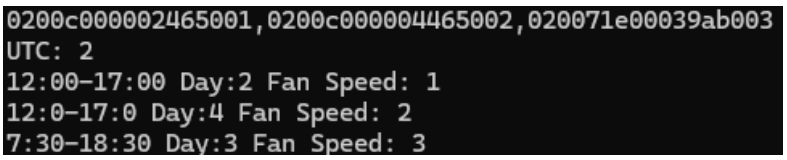

Figure 7.2: Schedule parsing output as generated by the microcontroller.

7.5 API Testing

To test the API endpoints, manual testing was performed using Postman, a popular API testing tool. Postman allows developers to easily interact with APIs by crafting HTTP requests and inspecting their responses in a user-friendly interface. It supports various HTTP methods, such as 'GET', 'POST', 'PUT', and 'DELETE', making it an ideal tool for testing REST APIs [37].

The testing process in Postman was as follows:

- **Device Registration:** A 'POST' request was sent to the '/api/v1/devices' endpoint with a payload containing 'device_id', 'device_name', and 'device_schedule', which successfully added a new device to the Device Properties Table. The response confirmed that the device had been registered with the correct properties.
- **Device Data Retrieval:** The newly added device was retrieved using a 'GET' request to the '/api/v1/devices/{device_id}' endpoint. The response returned the correct device data, confirming that the 'POST' operation had been successful.
- **Schedule Update:** The scheduling functionality was tested by sending a 'PUT' request to '/api/v1/schedules/{device_id}', which updated the 'device_schedule' field in the Device Properties Table. This test ensured that the schedule was properly stored in the database.
- **Schedule Retrieval:** Finally, a 'GET' request was sent to '/api/v1/schedules/{device_id}', which returned the updated schedule.

Additional tests simulated various edge cases, such as invalid input, missing data, and network failures. These scenarios helped ensure that the server responded with appropriate error messages and maintained stability under adverse conditions.

These tests confirm that requirements **REQ3.2** and **REQ3.3** from Table 2.3 have been successfully met. **REQ3.2** ensures that the cloud system is capable of storing data, including device settings and user information, while **REQ3.3** mandates an API for external applications to interact with the system.

7.6 Testing MQTT Communication

Testing the MQTT communication between the air purifier and the MQTT broker was performed using MQTT Explorer, a popular tool for monitoring and interacting with MQTT topics and messages. MQTT Explorer provides a graphical interface that allows users to subscribe to topics, publish messages, and visualize data flowing through the broker in real-time [28]. This made it an ideal tool for testing the communication between the air purifier's microcontroller and the MQTT broker.

To perform the test, both MQTT Explorer and the air purifier's microcontroller were connected to the same MQTT broker. A schedule string was published to the topic 'device-id/messages/cmd/', which was received and processed by the microcontroller. The schedule string used for this test was the same one tested in Section 7.4:

Figure 7.3: Publishing a schedule string via MQTT Explorer.

```
SCHEDULE:0200c000002465001,0200c000004465002,020071e00039ab003
```

The expected outcome was that the microcontroller would receive the MQTT message, parse it, and produce the same output as in Section 7.4. The microcontroller successfully parsed the schedule string, and the output matched the previously validated schedules, confirming that the MQTT communication was functioning correctly.

This process is illustrated in Figures 7.3 and 7.4. Figure 7.3 shows the MQTT Explorer interface where the schedule string was published to the appropriate topic. Figure 7.4 displays the microcontroller's output after receiving and parsing the MQTT message, which is consistent with the expected results.

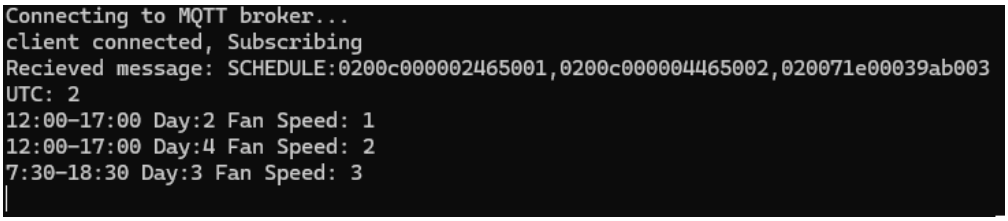

Figure 7.4: Microcontroller's output after receiving and parsing the MQTT message.

7.7 Testing Device Onboarding and Wi-Fi Setup

This section outlines the testing process for adding a new device to the air purifier system and configuring Wi-Fi credentials through the mobile application. The device onboarding process is a critical part of the user experience, as it ensures that the air purifier connects to the home network and becomes available for remote control and monitoring via the mobile application. Different scenarios were tested to validate the system's behavior under various conditions, from successful setups to potential failures such as incorrect credentials or network disconnections.

7.7.1 Successful Wi-Fi Credential Setup

In the ideal scenario, the process of setting up Wi-Fi credentials in the air purifier is completed without any issues. The user inputs the correct Wi-Fi credentials into the mobile application, which then sends these credentials to the air purifier. The air purifier successfully connects to the specified Wi-Fi network, and the connection is confirmed by an MQTT message sent from the air purifier to the mobile application.

Upon receiving the confirmation, the mobile application proceeds to register the device in the backend database. The registration process involves adding the device to the *Device Properties Table*, ensuring that the air purifier is now recognized within the system. Once this registration is complete, the device becomes available within the mobile application,

allowing the user to view real-time sensor values, adjust settings, and configure schedules for the air purifier.

This seamless integration demonstrates the core functionality of the system, where the air purifier, mobile application, and backend server work together to provide a smooth user experience.

7.7.2 Handling Wi-Fi Connection Issues

To test the handling of Wi-Fi connection issues, two different scenarios were simulated. In the first scenario, incorrect Wi-Fi credentials were entered into the mobile application, resulting in a failed connection attempt. The application responded by displaying an error message, prompting the user to re-enter the correct credentials. In the second scenario, the correct Wi-Fi password was entered, and the connection process began. However, before the mobile application could receive the confirmation message from the air purifier via MQTT, the phone's Wi-Fi was turned off. This interruption prevented the mobile application from completing the setup process. In both cases, the mobile application employed a 45-second timeout mechanism. If no confirmation message was received within this period, the application assumed that the connection attempt had failed and prompted the user to retry.

7.7.3 Network Disconnection During Setup

The network disconnection scenario was tested by simulating a sudden disconnection of the air purifier during the setup process. This was done by unplugging the air purifier, which caused the SoftAP network to shut down and subsequently disconnected the mobile device from the air purifier. During this test, the mobile application continuously monitored the connection status. Upon detecting the disconnection, the application attempted to automatically reconnect to the SoftAP network. If this reconnection attempt failed, the user was notified, and the application redirected them to the Wi-Fi scanning page, allowing them to restart the setup process.

7.7.4 Failure to Register the Device

Testing the failure to register the device involved simulating scenarios where, even after the Particle Photon successfully connected to the Wi-Fi network, the mobile application failed to register the device in the backend database. To test this, the API was temporarily modified to return a response code other than the "OK status code" 200, even when the registration process was technically correct. This allowed for testing the application's handling of registration failures. When the registration failed, the mobile application displayed an error message notifying the user of the issue. The air purifier automatically entered listening mode, generating a SoftAP network so that the user could reconnect the mobile application and restart the setup process.

Through these tests, it was ensured that the mobile application could effectively handle connection issues, disconnections, and registration failures, providing a reliable and user-friendly experience during the device setup process.

The following requirements were met during the testing of the device onboarding and Wi-Fi setup:

- **REQ5.0** from Table 2.5: The mobile application allows users to configure Wi-Fi credentials for the air purifier device, enabling it to connect to a local network.
- **REQ6.0** from Table 2.6: The app interface is intuitive and responsive, making complex tasks such as Wi-Fi setup simple for users to perform.
- **REQ6.3** from Table 2.6: The application is reliable, ensuring consistent performance even in scenarios with intermittent network connectivity or temporary server unavailability.

7.7.5 Testing Scheduling Functionality

The scheduling functionality of the air purifier was tested by creating several schedules with short intervals between them to observe how the system handled consecutive operations. Each schedule was set to run for a few minutes, with a minute or two difference between the schedules. Additionally, the schedules specified different fan speeds for each period.

During testing, it was observed that the air purifier followed the schedules as expected. The device started and stopped according to the specified times, and the fan speed adjusted automatically based on the schedule parameters. This confirmed that the scheduling feature was functioning correctly and that the air purifier could handle multiple schedules without issue.

This method of testing ensured that the air purifier could maintain optimal indoor air quality by executing user-defined schedules reliably, even when multiple schedules were set close to each other in time. As a result, REQ5.2 from Table 2.5 is met, confirming that the mobile application allows users to create and manage schedules for the air purifier, specifying operational parameters such as start time, end time, and fan speed.

Furthermore, this testing verifies that REQ1.3 from Table 2.1 is also met, as the air purifier successfully operates based on predefined cleaning schedules, automatically adjusting fan speeds at specified intervals to align with user preferences.

7.8 Testing IAQ Monitoring

To validate the sensor data display in the mobile application, a simple yet effective test was performed. Air was manually blown onto the air purifier's sensors to simulate a rapid change in air quality. The expected outcome was a real-time update of sensor values in the mobile application, reflecting the immediate change in the air environment.

Upon performing this test, it was observed that the sensor values in the mobile application updated immediately, confirming that the sensor data was being accurately transmitted from the air purifier to the app via MQTT. This real-time feedback demonstrated that the system effectively captured and displayed changes in air quality, ensuring that users can reliably monitor indoor air conditions through the mobile application.

This test confirmed the functionality of several key requirements across different components of the system. Specifically, it demonstrated that REQ5.2 from Table 2.5, which pertains to the mobile application's ability to manage and display real-time sensor data, was successfully met. Furthermore, the test validated that REQ3.1 from Table 2.3, related to real-time communication between the cloud and the mobile application, was also fulfilled, as immediate updates were received and displayed. Additionally, the air purifier met REQ1.2 from Table 2.1, by successfully transmitting real-time sensor data,

including the calculated Indoor Air Quality Index (IAQI), to the mobile application, thus providing users with timely updates on indoor air quality conditions.

7.9 Conclusion

This chapter provided a detailed overview of the testing and validation process for the intelligent air purifier system. The tests were designed to evaluate key functional and non-functional requirements developed during this thesis, covering areas such as Wi-Fi connectivity, sensor data processing, real-time communication, and user interaction. A complete summary of the requirements that were met during development and testing provided in Appendix C.

8 Future Work

While this thesis has successfully developed and tested an intelligent air purifier system, there are several areas for potential enhancement and expansion. This chapter outlines possible future work, focusing on improving system performance, scalability, user experience, and expanding functionality.

8.1 Sensor Calibration and Accuracy Testing

Future work should focus on properly calibrating the sensors to ensure that they provide accurate and reliable data. This may involve using professional calibration tools and testing the sensors under controlled environmental conditions to validate their accuracy and responsiveness. Accurate sensor data is essential for making informed decisions about air quality and adjusting the air purifier's operation accordingly.

8.2 Enhanced Sensor Integration

Future improvements could focus on integrating additional sensors to offer a more detailed and accurate assessment of indoor air quality. Adding sensors for detecting harmful gases like nitrogen dioxide (NO_2) and other nitrogen oxides (NOx), as well as Radon and PM1.0, would significantly enhance the system's ability to monitor indoor pollutants. With these additions, the system could generate a more comprehensive Indoor Air Quality Index (IAQI), providing users with deeper insights into their indoor environment and empowering them to take more informed actions to maintain a healthier living space.

8.3 Optimizing MQTT Communication

The current implementation of MQTT communication faced challenges, particularly in re-establishing connections after Wi-Fi disruptions. Future work could focus on optimizing MQTT settings, exploring alternative communication protocols (e.g., CoAP, AMQP, or WebSocket), or implementing more robust reconnection strategies to ensure uninterrupted data transmission between the embedded system and the cloud infrastructure.

8.4 Certificate Management

One crucial aspect of maintaining a secure IoT system is the management of digital certificates used for communication between the embedded devices, the server, and the MQTT broker. These certificates play a vital role in ensuring secure and encrypted data transmission, but they come with an expiry date. Once expired, communication between the devices and the cloud infrastructure can be disrupted, potentially leading to a loss of functionality and data security risks.

Future work should focus on implementing a robust certificate management system that can automatically detect and handle certificate expirations. This could involve setting up automated renewal processes for certificates, as well as integrating notification mechanisms to alert administrators when certificates are approaching their expiry dates. Additionally, designing fallback strategies, such as automatic switching to backup certificates, could ensure uninterrupted communication in case of unexpected certificate failures. Proper certificate lifecycle management is essential to maintaining a secure and resilient system in the long term.

8.5 Improving Scalability and Performance

As outlined in REQ4.0 and REQ4.1, future work will need to focus on optimizing the system for scalability and performance. This will require additional testing and evaluation to determine the current system's ability to handle increasing loads of devices, users, and data.

Performance and scalability testing will involve simulating a high number of concurrent users and devices to assess how the system performs under heavy loads. By doing so,

potential bottlenecks in the cloud infrastructure, such as database performance or server-side processing limitations, can be identified.

Upgrading the cloud infrastructure, including improving database query efficiency, could help reduce latency. Additionally, implementing advanced mechanisms like load balancing and auto-scaling would allow the system to dynamically adjust to higher demand without degrading performance. Since the current system has not undergone thorough testing for scalability and performance, future work must include this critical step to ensure the system is robust enough to handle real-world usage at scale.

8.6 User Interface Enhancements

The mobile application's user interface could be enhanced to provide more detailed insights and visualizations of air quality data. For example, adding graphs to display historical trends and integrating notifications for air quality alerts would make the application more informative and user-friendly.

8.7 Security Enhancements

Security remains a critical aspect for any IoT system, particularly when handling sensitive user data and controlling devices remotely. One area for future work is enhancing the system's authentication mechanisms to ensure that only authorized users can access and control the air purifier. Implementing multi-factor authentication (MFA) for the mobile application could significantly improve security by requiring users to verify their identity through multiple channels. Additionally, exploring token-based authentication for API access and device management would help prevent unauthorized actions and ensure that all interactions with the system are properly authenticated.

9 Conclusion

9.1 Summary of Achievements

This thesis has successfully designed, developed, and implemented an intelligent air purifier system that integrates advanced sensor technology, embedded software, cloud infrastructure, and a mobile application to enhance indoor air quality. The system is capable of real-time monitoring of multiple environmental parameters, dynamic control of air purification based on data inputs, and remote user interaction through a user-friendly mobile interface.

The research addressed key technical challenges in areas such as embedded system design, IoT communication, cloud integration, and user interface development. Through rigorous testing and validation, as summarized in the Appendix C, the system was shown to meet most of the functional and non-functional requirements laid out in the early stages of the project, providing a comprehensive solution for improving indoor air quality.

9.2 Key Contributions

The primary contributions of this work include:

- The development of a robust embedded system capable of continuous air quality monitoring and automated fan speed adjustment based on sensor data.
- The creation of a scalable cloud infrastructure that facilitates real-time data transmission, remote control, and secure data storage.
- The implementation of a mobile application that offers users real-time insights into their indoor environment and the ability to control their air purifiers from anywhere.

- A comprehensive testing and validation framework that ensured the system operates reliably under various conditions and scenarios.

These contributions advance the field of smart home and IoT technologies by delivering a practical and user-centered approach to indoor air quality management.

9.3 Challenges and Limitations

While the project achieved its primary goals, several challenges were encountered throughout the development process. One of the key challenges was maintaining reliable MQTT communication under certain network conditions. Further optimizations are necessary to improve the system's ability to handle network interruptions and ensure consistent data transmission.

Another significant limitation of the project was the lack of sensor accuracy testing and calibration. Although the sensors integrated into the system were functional and responsive to environmental changes, no calibration was performed to validate the accuracy of the sensor data. This limitation means that the current system cannot guarantee precise air quality measurements, particularly in extreme conditions or edge cases where sensor accuracy is critical.

Authentication with the server also presented challenges. The current implementation lacks robust authentication mechanisms to securely manage communication between the air purifier, the cloud, and the mobile application. While basic communication was established, further work is required to implement secure authentication protocols, such as OAuth or token-based authentication, to ensure that only authorized devices and users can access the system. This is especially important as the system scales and handles more sensitive data.

Additionally, the current system is optimized for a single-device scenario, and scaling the solution to manage multiple devices across different environments presents further challenges. This includes considerations related to data management, cloud performance, and user interface complexity, all of which will require further investigation to ensure the system can scale effectively while maintaining performance and usability.

9.4 Impact and Significance

This intelligent air purifier system demonstrates the potential of IoT technology in addressing critical health and environmental concerns related to indoor air quality. By providing real-time monitoring, automated control, and remote access, the system offers a significant improvement over traditional air purifiers, which typically lack these intelligent features. The integration of cloud services and mobile applications allows users to have greater control and insights into their environment, enhancing their ability to maintain healthier indoor spaces.

9.5 Final Thoughts

The development of this intelligent air purifier system represents a step forward in the evolution of smart home devices, showcasing the power of IoT and real-time data integration to improve quality of life. While there are still areas for improvement, this project lays the groundwork for future innovations in the field of environmental monitoring and smart appliances. The work presented in this thesis highlights the importance of continued research and development in the pursuit of smarter, healthier living environments.

Bibliography

[1] Adafruit: *Adafruit_HDC1000 (community library)*. 2024. – URL https://docs.particle.io/reference/device-os/libraries/a/Adafruit_HDC1000/. – Zugriffsdatum: 20.08.2024

[2] Adafruit: *Adafruit_SGP30 (community library)*. 2024. – URL https://docs.particle.io/reference/device-os/libraries/s/SensirionSGP30/. – Zugriffsdatum: 20.08.2024

[3] Android: *Cleartext / Plaintext HTTP*. 2024. – URL https://developer.android.com/privacy-and-security/risks/cleartext. – Zugriffsdatum: 20.08.2024

[4] Atmoko, R A. ; Riantini, R ; Hasin, M K.: IoT real time data acquisition using MQTT protocol. In: *Journal of Physics: Conference Series* 853 (2017), may, Nr. 1, S. 012003. – URL https://dx.doi.org/10.1088/1742-6596/853/1/012003

[5] Axios Developers: *Axios*. 2021. – URL https://www.npmjs.com/package/axios. – Zugriffsdatum: 20.08.2024

[6] Brightman, H. S. ; Milton, D. K. ; Wypij, D. ; Burge, H. A. ; Spengler, J. D.: Evaluating building-related symptoms using the US EPA BASE study results. In: *Indoor Air* 18 (2008). – URL https://asquifyde.es/uploads/documentos/Evaluating-building-related-symptoms-using-the-US-EPA-BASE-study-results-2008%20(ingl%C3%A9s).pdf

[7] California Air Resources Board: *Inhalable Particulate Matter and Health (PM2.5 and PM10)*. 2024. – URL https://ww2.arb.ca.gov/resources/inhalable-particulate-matter-and-health. – Zugriffsdatum: 20.08.2024

[8] FSIS Environmental, Safety, and Health Group: *Carbon Dioxide Health Hazard Information Sheet*. 2024. – URL https://www.fsis.usda.gov/sites/default/files/media_file/2020-08/Carbon-Dioxide.pdf

[9] German Environment Agency: *German Committee on Indoor Air Guide Values*. 2023. – URL https://www.umweltbundesamt.de/en/topics/health/commissions-working-groups/german-committee-on-indoor-air-guide-values. – Zugriffsdatum: 20.08.2024

[10] hiroktakster: *MQTT-TLS (Community Library)*. 2024. – URL https://docs.particle.io/reference/device-os/libraries/m/MQTT-TLS/. – Zugriffsdatum: 20.08.2024

[11] hirotakaster: *TlsTcpClient (community library)*. 2024. – URL https://docs.particle.io/reference/device-os/libraries/t/TlsTcpClient/. – Zugriffsdatum: 20.08.2024

[12] JuanSeBestia: *react-native-wifi-reborn*. 2024. – URL https://www.npmjs.com/package/react-native-wifi-reborn. – Zugriffsdatum: 20.08.2024

[13] Lowen, Anice C. ; Mubareka, Samira ; Steel, John ; Palese, Peter: Influenza Virus Transmission Is Dependent on Relative Humidity and Temperature. In: *PLOS Pathogens* 3 (2007). – URL https://journals.plos.org/plospathogens/article?id=10.1371/journal.ppat.0030151

[14] Madakam, Somayya ; Ramaswamy, R. ; Tripathi, Siddharth: Internet of Things (IoT): A Literature Review. In: *Journal of Computer and Communications* 3 (2015), S. 164–173. – URL https://www.scirp.org/pdf/JCC_2015052516013923.pdf

[15] Meng, Xiaoliang ; Wang, Feng ; Xie, Yichun ; Song, Guoqiang ; Ma, Shifa ; Hu, Shiyuan ; Bai, Junming ; Yang, Yiming: An Ontology-Driven Approach for Integrating Intelligence to Manage Human and Ecological Health Risks in the Geospatial Sensor Web. In: *Sensors* 18 (2018), Nr. 11. – URL https://www.mdpi.com/1424-8220/18/11/3619. – ISSN 1424-8220

[16] Microsoft: *Quickstart: Use Node.js to query a database in Azure SQL Database or Azure SQL Managed Instance*. 2023. – URL https://learn.microsoft.com/en-us/azure/azure-sql/database/connect-query-nodejs?view=azuresql&tabs=windows. – Zugriffsdatum: 20.08.2024

[17] Microsoft: *Security in Azure App Service*. 2023. – URL https://learn.microsoft.com/en-us/azure/app-service/overview-security. – Zugriffsdatum: 20.08.2024

[18] Microsoft: *App Service overview*. 2024. – URL https://learn.microsoft.com/en-us/azure/app-service/overview. – Zugriffsdatum: 20.08.2024

[19] Microsoft: *Deploy a private MQTT broker on Azure Sphere*. 2024. – URL https://learn.microsoft.com/en-us/training/modules/altair-azure-sphere-deploy-mqtt-broker. – Zugriffsdatum: 20.08.2024

[20] Microsoft: *Enable just-in-time access on VMs*. 2024. – URL https://learn.microsoft.com/en-us/azure/defender-for-cloud/just-in-time-access-usage. – Zugriffsdatum: 20.08.2024

[21] Microsoft: *Scale up an app in Azure App Service*. 2024. – URL https://learn.microsoft.com/en-us/azure/app-service/manage-scale-up. – Zugriffsdatum: 20.08.2024

[22] Microsoft: *What is Azure SQL Database?* 2024. – URL https://learn.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview?view=azuresql. – Zugriffsdatum: 20.08.2024

[23] Mosquitto: *MQTT man page*. 2024. – URL https://mosquitto.org/man/mqtt-7.html. – Zugriffsdatum: 20.08.2024

[24] Mozilla Developer Network: *An Overview of HTTP*. 2024. – URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. – Zugriffsdatum: 20.08.2024

[25] MQTT: *MQTT: The Standard for IoT Messaging*. 2024. – URL https://mqtt.org. – Zugriffsdatum: 20.08.2024

[26] National Energy Technology Laboratory (NETL): *Carbon Dioxide 101*. 2024. – URL https://netl.doe.gov/carbon-management/carbon-storage/faqs/carbon-dioxide-101. – Zugriffsdatum: 20.08.2024

[27] Node.js: *Introduction to Node.js*. 2024. – URL https://nodejs.org/en/docs/guides/getting-started-guide/. – Zugriffsdatum: 20.08.2024

[28] Nordquist, Thomas: *MQTT Explorer - An all-round MQTT client*. 2024. – URL https://mqtt-explorer.com/. – Zugriffsdatum: 20.08.2024

[29] Particle: *CLI Command Reference*. 2024. – URL https://docs.particle.io/reference/developer-tools/cli/. – Zugriffsdatum: 20.08.2024

[30] Particle: *Input/Output*, 2024. – URL https://docs.particle.io/reference/device-os/api/input-output/analogread-adc/. – Zugriffsdatum: 20.08.2024

[31] Particle: *Photon Datasheet*, 2024. – URL https://docs.particle.io/assets/pdfs/datasheets/photon-datasheet.pdf. – Zugriffsdatum: 20.08.2024

[32] Particle: *Quick Start: Workbench*. 2024. – URL https://docs.particle.io/quickstart/workbench/. – Zugriffsdatum: 20.08.2024

[33] Particle: *SoftAP HTTP pages*. 2024. – URL https://docs.particle.io/reference/device-os/api/softap-http-pages/softap-http-pages/. – Zugriffsdatum: 20.08.2024

[34] Particle: *Status LED - Photon*. 2024. – URL https://docs.particle.io/troubleshooting/led/photon/. – Zugriffsdatum: 20.08.2024

[35] Particle: *What's in the Box*. 2024. – URL https://docs.particle.io/quickstart/photon/. – Zugriffsdatum: 20.08.2024

[36] Particle: *Wi-Fi Setup Options*. 2024. – URL https://docs.particle.io/reference/device-os/wifi-setup-options/. – Zugriffsdatum: 20.08.2024

[37] Postman: *What is Postman?* 2024. – URL https://www.postman.com/product/what-is-postman/. – Zugriffsdatum: 20.08.2024

[38] React Navigation Contributors: *@react-navigation/native*. 2024. – URL https://www.npmjs.com/package/@react-navigation/native. – Zugriffsdatum: 20.08.2024

[39] Saffell, John ; Nehr, Sascha: Improving Indoor Air Quality through Standardization. In: *Standards* 3 (2023), Nr. 3, S. 240–267. – URL https://www.mdpi.com/2305-6703/3/3/19. – ISSN 2305-6703

[40] Scientific Committee on Health and Environmental Risks (SCHER): *Opinion on Risk Assessment on Indoor Air Quality*. 2007. – URL https://ec.europa.eu/health/ph_risk/committees/04_scher/docs/scher_o_055.pdf

[41] Sensirion: *Datasheet SGP30 Indoor Air Quality Sensor for TVOC and CO2eq Measurements*. 2020. – URL https://www.mouser.de/datasheet/2/682/Sensirion_Gas_Sensors_Datasheet_SGP30-2320451.pdf

[42] SSL Support Team: *What is SSL/TLS: An In-Depth Guide*. 2023. – URL https://www.ssl.com/article/what-is-ssl-tls-an-in-depth-guide/. – Zugriffsdatum: 20.08.2024

[43] Texas Instruments: *HDC1080 Low Power, High Accuracy Digital Humidity Sensor with Temperature Sensor Datasheet*, 2016. – URL https://www.ti.com/lit/ds/symlink/hdc1080.pdf?ts=1724102017669&ref_url=https%253A%252F%252Fwww.mouser.com%252F

[44] Thakran, Praveen ; Narsimha, Malla Sainadh R. ; Charaya, Nisha ; Kaur, Karamjit: Air Quality Monitoring and Purification Devices: A Review. In: *International Journal of Innovative Research in Computer Science & Technology (IJIRCST)* 8 (2020), Nr. 4, S. 285–289. – URL https://ssrn.com/abstract=3673119

[45] tuanpm: *react-native-mqtt*. 2016. – URL https://www.npmjs.com/package/react-native-mqtt. – Zugriffsdatum: 20.08.2024

[46] United States Environmental Protection Agency: *Guide to Air Cleaners in the Home: 2nd Edition*, 2018. – URL https://www.epa.gov/sites/default/files/2018-07/documents/guide_to_air_cleaners_in_the_home_2nd_edition.pdf. – Zugriffsdatum: 20.08.2024

[47] United States Environmental Protection Agency: *Carbon Monoxide's Impact on Indoor Air Quality*. 2024. – URL https://www.epa.gov/indoor-air-quality-iaq/carbon-monoxides-impact-indoor-air-quality

[48] United States Environmental Protection Agency: *Technical Assistance Document for the Reporting of Daily Air Quality - the Air Quality Index (AQI)*. 2024. – URL https://document.airnow.gov/technical-assistance-document-for-the-reporting-of-daily-air-quailty.pdf

[49] United States Environmental Protection Agency: *Volatile Organic Compounds' Impact on Indoor Air Quality*. 2024. – URL https://www.epa.gov/indoor-air-quality-iaq/volatile-organic-compounds-impact-indoor-air-quality. – Zugriffsdatum: 20.08.2024

[50] Wisconsin Department of Health Services: *Carbon Dioxide*. 2023. – URL https://www.dhs.wisconsin.gov/chemical/carbondioxide.htm. – Zugriffsdatum: 20.08.2024

[51] World Health Organization: *Air Quality Guidelines: Global Update 2005. Particulate Matter, Ozone, Nitrogen Dioxide, and Sulfur Dioxide*. World Health Organization, 2006. – URL https://iris.who.int/bitstream/handle/10665/107823/9789289021920-eng.pdf?sequence=1. – ISBN 92 890 2192 6

[52] World Health Organization: *New WHO Global Air Quality Guidelines aim to save millions of lives from air pollution*. 2021. – URL https://www.who.int/news/item/22-09-2021-new-who-global-air-quality-guidelines-aim-to-save-millions-of-lives-from-air-pollution. – Zugriffsdatum: 20.08.2024

[53] Xia, Feng ; Yang, Laurence T. ; Wang, Lizhe ; Vinel, Alexey: Internet of Things. In: *Int. J. Commun. Syst.* 25 (2012), sep, Nr. 9, S. 1101–1102. – URL https://doi.org/10.1002/dac.2417. – ISSN 1074-5351

[54] Zhengzhou Winsen Electronics Technology Co., Ltd: *Qirquality and Particles Sensor (Model: ZPH02) Manual*. 2016. – URL https://www.winsen-sensor.com/d/files/zph02-particles-and-voc-module-manual-v1_0.pdf

[55] Zhengzhou Winsen Electronics Technology Co., Ltd: *Electrochemical Carbon Monoxide Sensor (Model: ME2-CO-14x50-C)*, 2023. – URL https://www.winsen-sensor.com/d/files/manual/me2-co-%D1%8414x50-c.pdf

A Detailed Schedule Parsing Results

The detailed breakdown of the parsed schedule entries is provided below. Each schedule was successfully parsed and resulted in the following expected outcomes:

- **First Schedule (0200c000002465001):**
  - **UTC Offset:** 2
  - **Start Time:** 12:00
  - **Duration:** 0x4650 seconds, which equals 17:00 as the end time
  - **Day:** 2
  - **Fan Speed:** 1
- **Second Schedule (0200c000004465002):**
  - **UTC Offset:** 2
  - **Start Time:** 12:00
  - **Duration:** 0x4650 seconds, translating to 17:00 as the end time
  - **Day:** 4
  - **Fan Speed:** 2
- **Third Schedule (020071e00039ab003):**
  - **UTC Offset:** 2
  - **Start Time:** 07:30
  - **Duration:** 0x39ab seconds, translating to 18:30 as the end time
  - **Day:** 3

- **Fan Speed:** 3

B Selected Code Implementation

B.1 Code Implementation for Sending Sensor Data Using TlsTcpClient

The full implementation of the function to send sensor data to a server using the `TlsTcpClient` library in a C++ application is provided below in pseudocode format. This function establishes a secure HTTPS connection and transmits the sensor values (humidity, temperature, CO2, VOC, CO, and PM2.5) in JSON format.

Listing B.1: Function to Send Sensor Data Using TlsTcpClient (Pseudocode)

```
#include "TlsTcpClient/TlsTcpClient.h"

function sendSensorData(device_id, humidity, temp,
                        co2, voc, co, pm25):
    buffer = createBuffer(512)  // Allocate buffer for data
    client = TlsTcpClient()     // Initialize TLS client
    response = 0                // Response code variable

    // Setup Root CA pem
    client.init(letencryptCaPem, sizeof(letencryptCaPem))

    // Connect to HTTPS server
    if client.connect("{server address}", 443):

        // Verify server certificate
        if not client.verify():
            print("Server certificates are invalid")
            return 0
```

```
        // Prepare the payload in JSON format
        payload = "{"
        payload += "\"device_id\": " + device_id + ","
        payload += "\"humidity\": " + toString(humidity) + ","
        payload += "\"temperature\": " + toString(temp) + ","
        payload += "\"co2\": " + toString(co2) + ","
        payload += "\"voc\": " + toString(voc) + ","
        payload += "\"co\": " + toString(co) + ","
        payload += "\"pm25\": " + toString(pm25)
        payload += "}"

        // Prepare HTTP POST request
        requestLength = sprintf(buffer,
            "POST /api/v1/sensor_data HTTP/1.1\r\n" +
            "Host: {server address}\r\n" +
            "Content-Type: application/json\r\n" +
            "Content-Length: " + toString(payload.length()) +
            "\r\n\r\n" + payload)

        // Send HTTP request
        client.write(buffer, requestLength)

        // Get Response Code
        header = readHttpHeader(client)
        content = strstr(header, " ")
        if content != NULL:
            content += 1
            response = toInt(content)
            print("Response Code: " + toString(response))

        // Close the connection
        client.stop()
    else:
        print("Connection failed")

    return response  // Return the response code
```

B.2 Code Implementation for Runtime Permission Requests

The full implementation of the runtime permission request functionality in a React Native application is provided below:

Listing B.2: Full Code for Runtime Permission Requests in React Native

```
import { PermissionsAndroid } from 'react-native';

const granted = await PermissionsAndroid.request(
      PermissionsAndroid.PERMISSIONS.ACCESS_FINE_LOCATION,
      {
        title: 'Location permission is required for WiFi' +
               'connections',
        message:
          'This app needs location permission as this is' +
          'required to scan for wifi networks.',
        buttonNegative: 'DENY',
        buttonPositive: 'ALLOW',
      },
);
if (granted === PermissionsAndroid.RESULTS.GRANTED) {
    // You can now use react-native-wifi-reborn
} else {
    // Permission denied
}
```

B.3 Code Implementation for Navigation Setup

The full implementation of the navigation setup in a React Native application is provided below:

Listing B.3: Full Code for Navigation Setup in React Native

```
import * as React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import HomeScreen from './screens/HomeScreen';
import SetupScreen from './screens/SetupScreen';
import SensorsScreen from './screens/SensorsScreen';
import ScheduleScreen from './screens/ScheduleScreen';

const Stack = createStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Setup" component={SetupScreen} />
        <Stack.Screen name="Sensors" component={SensorsScreen}
        />
        <Stack.Screen name="Schedule" component={ScheduleScreen}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

B.4 ParticleDeviceSetup Class Implementation

The following section provides detailed explanations and pseudocode for the key methods implemented in the `ParticleDeviceSetup` class. These methods allow interaction with the Particle Photon microcontroller to set up WiFi credentials through its SoftAP HTTP server.

B.4.1 fetchDeviceId

The `fetchDeviceId` method is responsible for retrieving the unique device ID of the Particle Photon. This ID is essential for identifying the specific device during the setup process. The method sends an HTTP `GET` request to the `/device-id` endpoint and returns the device ID.

Listing B.4: Pseudocode for fetchDeviceId method

```
function fetchDeviceId:
    Send HTTP GET request to /device-id
    Receive response and parse JSON
    Extract 'id' from the response
    Return device ID
```

B.4.2 connectToNetwork

The `connectToNetwork` method initiates the connection between the Particle Photon and the selected WiFi network. This method sends an HTTP `POST` request to the `/connect-ap` endpoint with the network index. A successful connection is indicated by a specific response value.

Listing B.5: Pseudocode for connectToNetwork method

```
function connectToNetwork:
    Send HTTP POST request to /connect-ap with network index
    Receive response and parse JSON
    if response contains 'r' == 0:
        Return True (connection successful)
    else:
        Return False (connection failed)
```

B.4.3 getNetworks

The `getNetworks` method retrieves a list of available WiFi networks that the Particle Photon can detect. This method sends an HTTP `GET` request to the `/scan-ap` endpoint and returns a list of networks with their details.

Listing B.6: Pseudocode for getNetworks method

```
function getNetworks:
    Send HTTP GET request to /scan-ap
    Receive response and parse JSON
    Extract 'scans' array from the response
    Return list of networks (SSID, RSSI, security, channel)
```

B.4.4 getPublicKey

The `getPublicKey` method is used to retrieve the public RSA key from the Particle Photon, which is necessary for encrypting the WiFi password before transmission. The method sends an HTTP `GET` request to the `/public-key` endpoint and returns the public key.

Listing B.7: Pseudocode for getPublicKey method

```
function getPublicKey:
    Send HTTP GET request to /public-key
    Receive response and parse JSON
    Extract public key from the response
    Return public key
```

B.4.5 encryptPassword

The `encryptPassword` method is responsible for encrypting the WiFi password using the RSA public key obtained from the Particle Photon. This method processes the public key and the password, then returns the encrypted password, ensuring secure transmission over HTTP.

Listing B.8: Pseudocode for encryptPassword method

```
function encryptPassword(publicKey, password):
    Extract modulus and exponent from publicKey
    Initialize RSA encryption object with modulus and exponent
    Encrypt password using RSA object
    Return encrypted password
```

B.4.6 sendCredentials

The `sendCredentials` method configures the Particle Photon to connect to the selected WiFi network. It first retrieves the public RSA key and encrypts the password using the `encryptPassword` method. Then, it sends the network details and encrypted password to the `/configure-ap` endpoint via an HTTP `POST` request. A successful configuration is indicated by a specific response value.

Listing B.9: Pseudocode for sendCredentials method

```
function sendCredentials(network, password):
    publicKey = getPublicKey()
    encryptedPassword = encryptPassword(publicKey, password)
    Prepare network configuration payload (SSID, security,
        channel, encryptedPassword)
    Send HTTP POST to /configure-ap with network configuration
    Receive response and parse JSON
    if response contains 'r' == 0:
        Return True (configuration command received)
    else:
        Return False (configuration command not received)
```

C System Requirements and Verification

This appendix provides a summary of the functional and non-functional requirements for the air purifier system, along with an indication of whether each requirement has been met based on the testing results.

C.1 Functional Requirements

C.1.1 Air Purifier Functional Requirements

Identifier	Description	Status
REQ1.0	**Wi-Fi Connectivity:** The air purifier must connect to a Wi-Fi network to enable communication with remote platforms, such as a mobile application or cloud server, allowing users to remotely control and monitor the device.	Met
REQ1.1	**Continuous Air Quality Monitoring:** The air purifier must continuously monitor indoor air quality using sensors that measure pollutants such as VOCs, CO2, CO, PM2.5, as well as temperature and humidity.	Met
REQ1.2	**Real-time Data Transmission:** The air purifier must transmit sensor data, including air quality metrics and the calculated IAQI, to a mobile application in real-time.	Met
REQ1.3	**Scheduled Operation:** The air purifier must operate based on predefined cleaning schedules, automatically adjusting fan speeds at specified intervals to align with user preferences.	Met
REQ1.4	**Automated Fan Speed Adjustment:** The air purifier must automatically adjust fan speed according to real-time air quality measurements.	Met

Table C.1: Functional Requirements of the Air Purifier

C.1.2 Air Purifier Non-Functional Requirements

Identifier	Description	Status
REQ2.0	**Automatic Recovery:** The system must recover automatically from network disruptions, without requiring user intervention.	Not Met
REQ2.1	**Scalability:** The embedded system should be designed to accommodate additional sensors or functionalities in future upgrades.	Met

Table C.2: Non-Functional Requirements of the Air Purifier

C.1.3 Cloud System Functional Requirements

Identifier	Description	Status
REQ3.0	**Secure and Reliable Communication:** The cloud must facilitate secure and reliable communication between the air purifiers and the mobile application.	Met
REQ3.1	**Real-time Communication:** The cloud must support real-time communication, allowing immediate updates and control commands.	Met
REQ3.2	**Data Storage:** The cloud must store data from the air purifiers, including device settings, and user information.	Met
REQ3.3	**API for External Applications:** The cloud must provide an API that allows external applications to interact with the system.	Met

Table C.3: Functional Requirements of the Cloud System

C.1.4 Cloud System Non-Functional Requirements

Identifier	Description	Status
REQ4.0	**Scalability:** The server must scale with the number of devices and users, ensuring performance remains consistent.	Not Tested
REQ4.1	**Performance:** The server must process requests with minimal latency, handling large volumes of data efficiently.	Not Tested

Table C.4: Non-Functional Requirements of the Cloud System

C.1.5 Mobile Application Functional Requirements

Identifier	Description	Status
REQ5.0	**Wi-Fi Configuration:** The mobile app must allow users to configure Wi-Fi credentials for the air purifier.	Met
REQ5.1	**Real-time Data Display:** The app must retrieve and display real-time sensor data from the air purifier.	Met
REQ5.2	**Scheduling:** The app must allow users to create and manage schedules for the air purifier.	Met

Table C.5: Functional Requirements of the Mobile Application

C.1.6 Mobile Application Non-Functional Requirements

Identifier	Description	Status
REQ6.0	**Usability:** The app interface must be intuitive and responsive.	Met
REQ6.1	**Scalability:** The app must accommodate additional features or integrations in future updates.	Met
REQ6.2	**Security:** The app must ensure secure transmission of sensitive data using encryption.	Met
REQ6.3	**Reliability:** The app must perform consistently even with intermittent network connectivity.	Met

Table C.6: Non-Functional Requirements of the Mobile Application

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

City Date Signature