ESP32-Based Environmental Monitoring and Cloud Data Logging Using Azure IoT Central and Google Sheets

Ian Wallace

Bethune-Cookman University

Senior Design II and CIS 498A

Term Project

11/28/2025

Abstract

This is a low-cost, cloud-connected environmental plant monitoring system using an ESP32 microcontroller, a DHT11 temperature/humidity sensor, and an analog soil-moisture sensor. It reads local environmental conditions and sends the data to two different cloud targets: first, Microsoft Azure IoT Central using MQTT over TLS with Device Provisioning Service (DPS) and Shared Access Signature (SAS) tokens; second, Google Sheets via an HTTP JSON API backed by Google Apps Script. Together, these two paths demonstrate both an enterprise-style IoT architecture and a budget-friendly logging option that still provides real-time data capture.

What began as a simple experiment in LED blinking on the ESP32 evolved into a complete end-to-end pipeline, coalescing hardware, secure networking, cloud services, and data visualization. Data from IoT Central was exported as CSV using Azure Data Explorer and turned into structured charts and dashboards using Power BI. At the same time, the Google Sheets implementation showed how the same sensors could log directly into a spreadsheet for quick analysis. This report follows the department's technical report template, documents the design decisions, explains key parts of the code, and reflects on the learning experience from spring to fall.

1. Introduction

This report presents a senior-level CIS project that connects hands-on embedded hardware with modern cloud services. The system is built around the ESP32 microcontroller, which reads temperature, humidity, and soil-moisture data and sends it to two different cloud targets, Azure IoT Central and Google Sheets. From there, the data can be visualized and analyzed using tools like Power BI or directly inside a spreadsheet.

As a Computer Information Systems student, I wanted a project that did more than just blink LEDs. I wanted something that tied together cloud computing and data analysis, the same areas I hope to work in after graduation. At the same time, I had to design something realistic in terms of budget, lab time, and my responsibilities outside of the classroom.

1.1. Project Context and Motivation

The idea for this project grew from several concepts discussed in Senior Design I. My professor recommended that I start by working with an ESP32 microcontroller and a DHT11 sensor. First, I made the onboard LED turn on and off on the ESP32 to get comfortable with the development environment and the basic upload cycle. After that, I attached the DHT11 so the board could read temperature and humidity and print the values to the serial monitor.

Over the summer, I completed an internship at an insurance company and was exposed to the data analyst side of the business. That experience made me more interested in data and reporting, and I wanted a project that could move real sensor data “off the board” into a cloud platform. Azure IoT Central became the next step, where I learned how to connect the ESP32 to Azure over MQTT, configure a device template, and stream telemetry in real time. Later, I used Azure Data Explorer to export the telemetry as a CSV file and brought that file into Power BI to create clean, organized reports and dashboards.

Finally, I extended the system by adding a soil-moisture sensor. This required solving a physical constraint: both the DHT11 and the soil probe needed access to the 3.3 V rail, and the original breadboard only had enough space for one sensor. The solution I came up with was to merge two breadboards and place the ESP32 across the middle so the 3.3 V and GND buses could fan out on both sides. This small hardware trick allowed me to power everything cleanly and complete the three-sensor design.

1.2. Problem Statement

Conventional environmental monitoring often relies on manual readings or stand-alone devices that do not store readings over time. In many small settings, such as a dorm room, small garden, or indoor plants, people may want to keep track of temperature and soil moisture but lack an easy, low-cost way to log and visualize that data.

The problem this project addresses is how to design an affordable, flexible system that continuously measures environmental conditions with an ESP32, securely sends that data to the cloud, and stores the readings in formats that support graphs, dashboards, and further analysis.

1.3. Scope

The scope of this project covers the end-to-end design of an ESP32-based environmental monitoring system that measures temperature, humidity, and soil moisture using a DHT11 sensor and an analog soil-moisture sensor. On the cloud side, the project implements a secure Azure IoT Central solution that uses DPS, MQTT over TLS, and SAS tokens for device authentication and telemetry.

In parallel, it includes another data path that sends the same sensor readings to Google Sheets via an HTTP JSON API and Google Apps Script. The work also extends into exporting telemetry from Azure into Power BI via CSV for organized reporting and visualization, and documenting the entire process in a professional technical report that still reflects a college student's perspective and learning journey.

2. Background

The Internet of Things (IoT) describes physical devices such as sensors, microcontrollers, and actuators that are connected to the internet and can exchange data with cloud services or other devices. In this context, the ESP32 is an "edge device" that sits in the physical environment, collects measurements, and sends them upstream to cloud backends.

2.1. Environmental Monitoring Use Cases

Common environmental monitoring applications deploy IoT devices in farms, greenhouses, or buildings to measure temperature, humidity, soil moisture, and other factors. Traditional monitoring usually relies on manual measurement, stand-alone thermometers, or simple devices that do not store data over time.

In contrast, an IoT-based system can continuously collect data at regular intervals, timestamp each reading, send data to the cloud in real time, and visualize trends over hours, days, or weeks. This supports better decisions, such as deciding when to water plants based on soil-moisture trends or understanding how temperature changes throughout the day in a specific room.

Soil moisture is a direct indicator of whether the plant's roots are getting enough water, but temperature and humidity also strongly affect plant health. Temperatures that are too high or too low can slow or stop growth, while very low humidity can dry out leaves and very high humidity

can encourage fungal diseases. Because of this, logging all three factors—soil moisture, temperature, and humidity—provides a more complete picture of plant stress than watching soil moisture alone.

2.2. Azure IoT Central, DPS, MQTT/TLS, and SAS Tokens

Azure IoT Central is a managed IoT application platform. It simplifies device enrollment, telemetry visualization, and integration with other Azure services. For production-style deployments, Azure supports the Device Provisioning Service (DPS), which acts as a secure front door for devices. DPS checks each device's ID and key and then assigns it to an appropriate IoT Hub.

MQTT (Message Queuing Telemetry Transport) is a lightweight publish-subscribe protocol that fits well in IoT applications. It is efficient for low-power devices and is a good match for resource-constrained microcontrollers such as the ESP32. In this project, MQTT runs over TLS (Transport Layer Security), which means communication between the ESP32 and Azure is encrypted.

Authentication uses Shared Access Signature (SAS) tokens. Each SAS token is a signed string that proves the device that holds a valid secret key is authorized to connect. The token includes an expiration time, which improves security compared to a static password that never changes. These tokens are generated on the ESP32 using the mbedTLS cryptography library.

2.3. Google Sheets and Apps Script as an Alternative Backend

While Azure IoT Central provides a powerful, scalable solution, it may be more complex and feature-rich than some small projects need. For that reason, I also built a second path where the ESP32 sends the same sensor data to a Google Sheets document.

In this design, a short Google Apps Script exposes a Web App URL. The ESP32 sends an HTTP POST request to that URL containing a JSON document with fields for Temperature, Humidity, and Soil. The script parses the JSON and appends a new row in the spreadsheet with the current timestamp and the sensor values.

This alternative path is more limited in terms of security and scalability compared to Azure, but it is easy to understand, inexpensive, and still powerful for quick dashboards and data sharing.

3. System Design Overview

3.1. Functional Requirements

The main functional requirements are to reliably collect and transmit environmental data using low-cost hardware and modern cloud services. Specifically, the ESP32 should:

The system must read temperature and humidity from a DHT11 sensor and soil-moisture levels from an analog soil sensor, then connect to the campus Wi-Fi network (BCUStudents) to send that data outward. It also needs to synchronize its internal clock with NTP servers so that SAS token expiration times are valid, use Azure DPS to register the device and discover the assigned IoT Hub and device ID, and generate SAS tokens on the ESP32 using HMAC-SHA256 with mbedTLS.

Once provisioned, the device must send telemetry as JSON over MQTT/TLS to both Azure IoT Hub and Azure IoT Central, and support exporting that data into Power BI from Azure for visualization. In addition to the Azure path, the system must also support an alternate data path where the same telemetry is sent to Google Sheets using HTTP POST and JSON, providing a simpler, spreadsheet-based logging option.

3.2. Non-Functional Requirements

The non-functional requirements describe how the system should behave, not just what it should do. From a security point of view, the design must use TLS and SAS tokens and avoid sending any credentials in clear text. The system should be reliable by automatically reconnecting Wi-Fi and MQTT if connections drop and by handling bad or missing sensor readings gracefully without crashing. It should be scalable because, through Azure DPS, the same approach can support not just one prototype device but many devices later on.

Maintainability is also important, meaning that the code should be divided into clear, logical functions with comments that another student or future developer can understand. Finally, the

overall cost must remain reasonable by using hardware and cloud platforms that make sense for a college student budget.

3.3. Overall Architecture

The architecture for this project can be described in three main layers. The device layer consists of the ESP32 microcontroller, which collects environmental data from a DHT11 sensor connected on GPIO 5 and a soil-moisture sensor connected on GPIO 34 (ADC1).

Above that, the first cloud layer (Path 1) handles the enterprise-style integration. The ESP32 connects to global.azure-devices-provisioning.net using MQTT over TLS. Azure DPS validates the SAS token and returns the assigned IoT Hub and device ID, and then the ESP32 connects to that IoT Hub via MQTT/TLS and publishes telemetry to a device-specific topic. Azure IoT Central consumes this data, provides dashboards, and supports data export for further analysis.

In parallel, a second cloud layer (Path 2) offers a simpler path where the ESP32 sends an HTTP POST request with a JSON payload to a Google Apps Script Web App URL. The script parses the JSON and appends each new reading as a row in a Google Sheet that acts like a lightweight database for charts and basic analysis.

3.4. Development Process and Iterations

LED Blinking. I started with the classic “blink” example on the ESP32. This phase focused on becoming comfortable with the board, installing and configuring the Arduino IDE, and verifying that sketches uploaded and ran correctly. It gave me a low-stress way to confirm that the hardware and development environment were working before adding any sensors or cloud features.

DHT11 + Serial Monitor. Next, I connected the DHT11 sensor to GPIO 5 and, using the DHT library, read temperature and humidity. The ESP32 printed these values to the serial monitor every few seconds, which confirmed that it could communicate correctly with a digital sensor and that the readings were reasonable. This provided a solid baseline for moving into cloud connectivity.

ESP32 + DHT11 + Azure IoT Central. Once local readings were stable, I added Wi-Fi and MQTT code and connected the board to Azure IoT Central. This required creating a device template, working with SAS tokens for authentication, and troubleshooting some issues with the board connecting to the campus Wi-Fi. By the end of this phase, the ESP32 was streaming live telemetry to Azure.

Export to Power BI. Once data was consistently flowing into Azure, I used Azure Data Explorer to export the telemetry as a CSV file. That CSV was imported into Power BI, where I created basic charts showing how temperature, humidity, and later soil moisture changed over time. This step connected the IoT work to data visualization and reporting, which is important in a CIS context.

Adding the Soil-Moisture Sensor. Adding the soil-moisture sensor turned out to be more complicated than just connecting one additional wire. Both sensors required 3.3 V and ground, and the original breadboard layout did not have enough room on the rails. The solution was to combine two breadboards and place the ESP32 so that it spanned the middle and could power both sides. This minor hardware issue highlighted that system design also includes physical layout and power distribution, not just code.

Google Sheets Path. Once the Azure path was stable, I created a parallel sketch that sent the same sensor data to Google Sheets. The ESP32 used HTTP POST with JSON to call a Google Apps Script Web App, which logged each reading into a spreadsheet. This path provided both a backup logging option and a hands-on way to practice building simple REST-style APIs with Apps Script.

This step-by-step process allowed me to debug each layer in isolation, which was important given my academic workload and responsibilities outside the class.

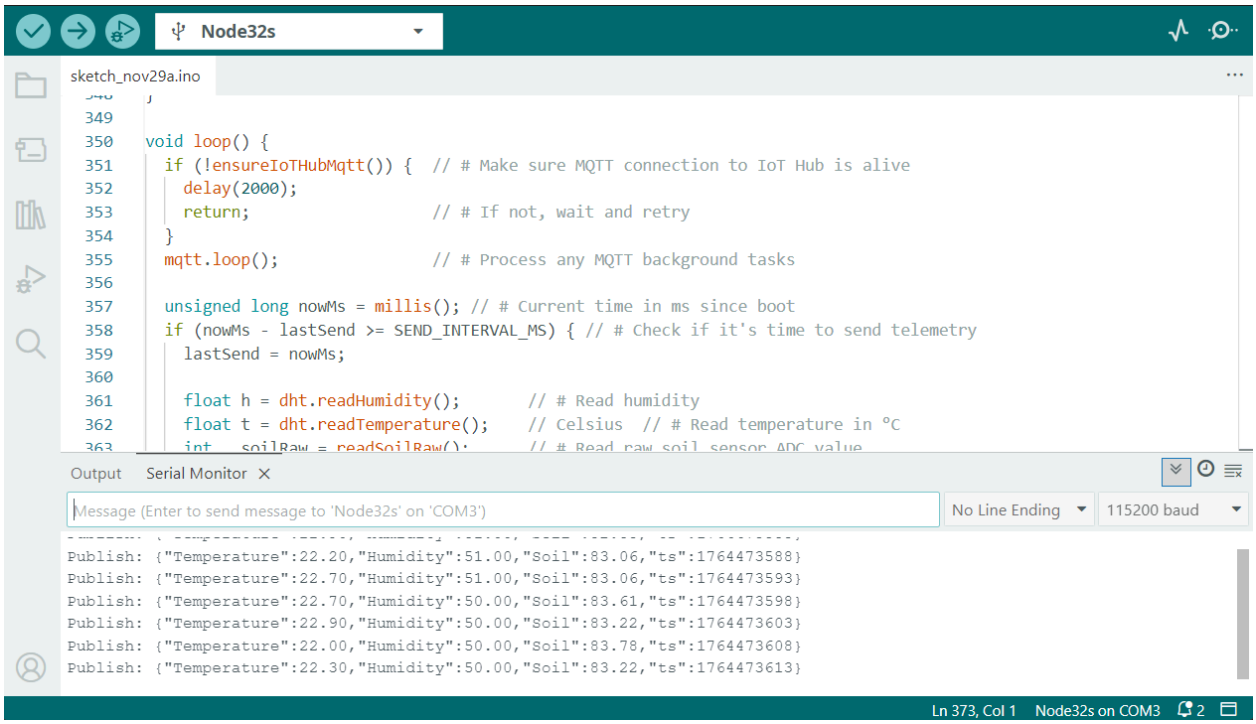

Figure 1. Arduino IDE with ESP32 environmental monitoring sketch and serial output. The serial monitor shows live temperature, humidity, and soil-moisture readings along with connection status messages used for debugging.

4. Hardware Design

The ESP32 microcontroller was chosen because it has built-in Wi-Fi with support for secure sockets, multiple GPIO pins and analog inputs for connecting sensors, and enough processing power and memory to handle TLS encryption, MQTT messaging, and continuous sensor sampling at the same time. Throughout development, the board was powered over USB from a laptop, which also provided a serial connection for debugging and monitoring sensor readings and connection status in real time.

The DHT11 temperature and humidity sensor has three key connections: VCC, GND, and a data pin. In this project, VCC is connected to the ESP32's 3.3 V output, GND is wired to one of the ESP32's ground pins, and the data pin is connected to GPIO 5. The Arduino DHT library handles the strict timing requirements of the sensor and provides simple functions like readTemperature() and readHumidity() so the code can easily access readings. While not the most precise sensor available, the DHT11 is accurate enough for this prototype and works well for demonstrating core IoT concepts in a teaching and learning context.

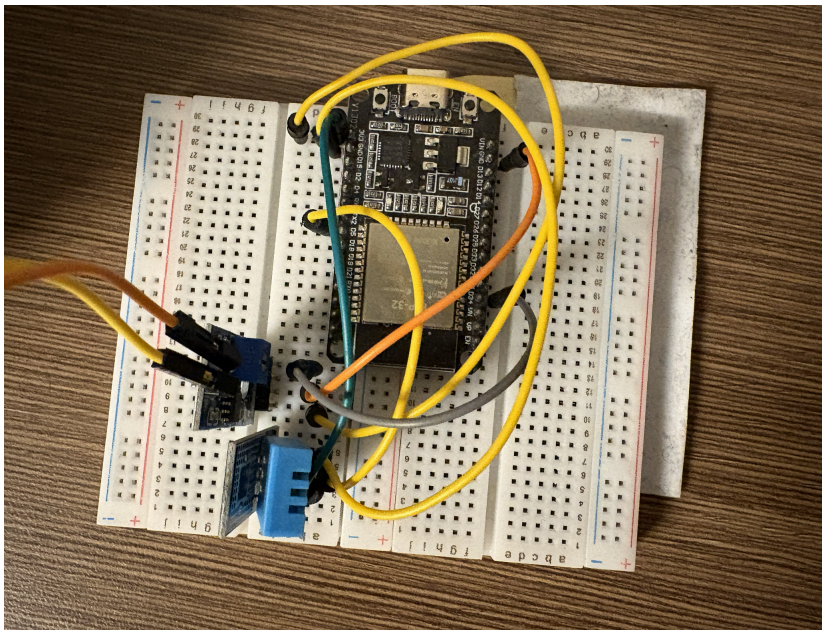

Figure 2. ESP32 prototype mounted across two breadboards with DHT11 and soil-moisture sensor. The layout shows how the 3.3 V and GND rails are shared to power both sensors while keeping the wiring organized for testing.

4.3. Soil-Moisture Sensor and Calibration

The soil-moisture sensor outputs an analog voltage that changes depending on how wet the soil is. In this project, its VCC pin is connected to the ESP32's 3.3 V output, GND is tied to one of the ESP32 ground pins, and the analog output (AO) is connected to GPIO 34, which is an ADC1 channel. The raw analogRead(SOIL_PIN) values vary from 0 to 4095.

To make these raw readings easier to understand, two calibration constants are defined in the code:

```
int SOIL_RAW_DRY = 3000;  // approximate value in dry soil or air

int SOIL_RAW_WET = 1200;  // approximate value in very wet soil or water
```

These values were obtained experimentally by placing the sensor in dry soil and then in water and recording the readings. The function soilRawToPercent() then maps any raw value between these limits onto a 0–100% scale, giving a more intuitive soil-moisture percentage.

Soil moisture is critical because it regulates how much water the roots can take up, but it is only one part of the plant's environment. Temperature affects growth and metabolism, where very high or very low temperatures can slow or stop growth. Humidity affects how quickly a plant loses water through its leaves: low humidity can pull moisture out of the plant and cause wilting or leaf scorch, while high humidity can promote fungal diseases. By monitoring soil moisture together with temperature and humidity, the system can highlight situations where the soil looks fine, but air conditions are stressing the plant.

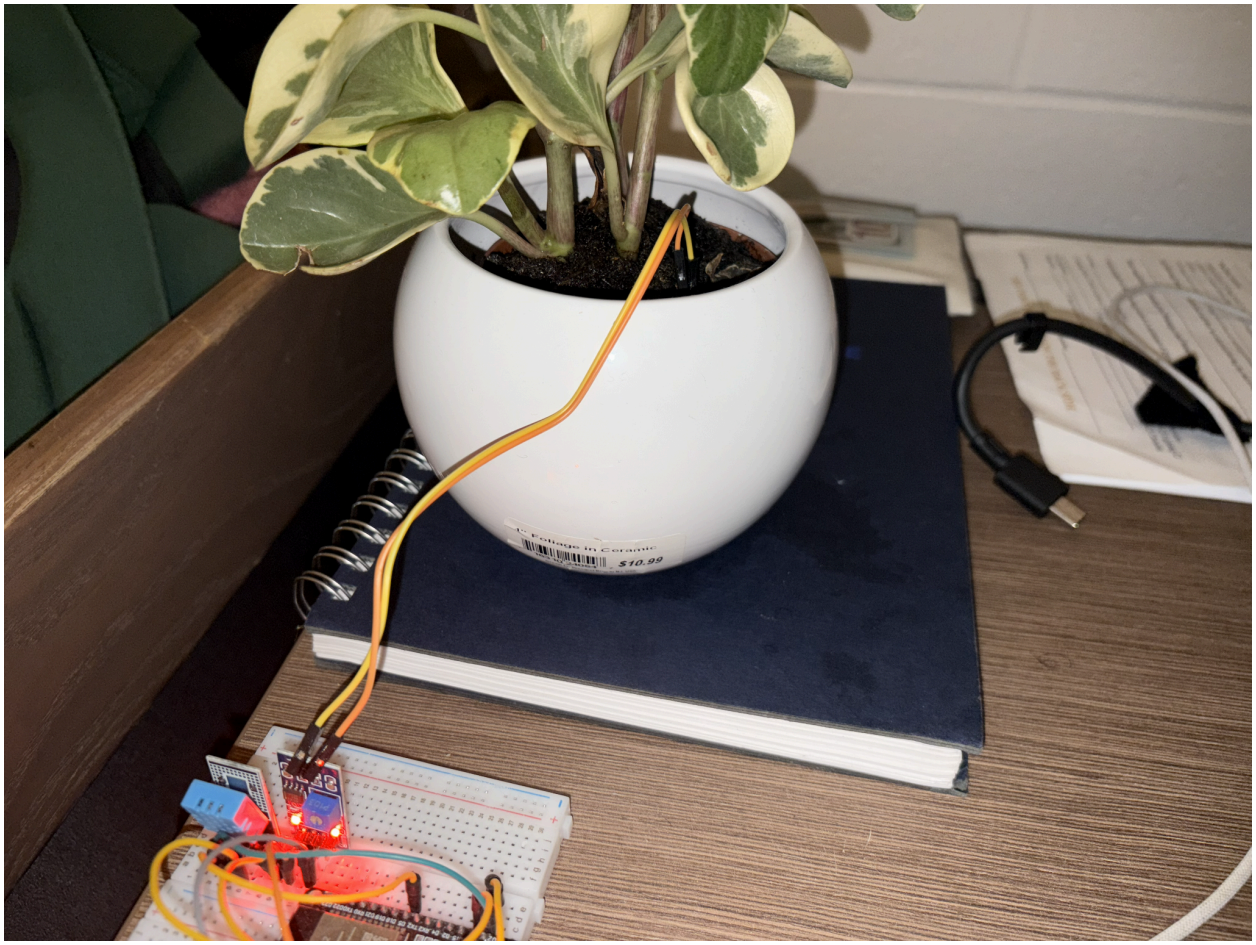

Figure 3. Soil-moisture sensor placed in a plant pot for calibration and testing. The probe is inserted into real soil so the raw ADC readings can be compared between dry and wet conditions and converted to a percentage.

4.4. Breadboard Layout and Power Considerations

Initially, the entire circuit was built on a single breadboard. However, because of the physical size of the ESP32, the number of required connections, and the shared 3.3 V rail, it was difficult to route all the wires cleanly. I solved that by using two breadboards placed side by side and placing the ESP32 across the center gap so that different sets of pins could feed each board. The 3.3 V rails were tied together, as were the GND rails, so power was shared across both boards.

This layout effectively doubled the available space and allowed the DHT11 and the soil-moisture sensor to each have their own organized sections. It also helped reduce noise on the analog line by keeping the soil sensor wiring farther away from other high-frequency signal lines where possible.

5. Azure IoT Implementation

5.1. Network and Security Design

On the network side, the ESP32 uses several key libraries to handle connectivity and security. WiFi.h is used to connect in station mode to the BCUStudents network, giving the board internet access. On top of that, WiFiClientSecure creates a TLS-encrypted TCP socket so that data sent to the cloud is protected in transit. The PubSubClient library then runs the MQTT protocol over this secure socket, allowing the ESP32 to publish telemetry messages to Azure.

In the Arduino code, the client is configured with net.setInsecure() for development purposes, which means it does not validate the server's certificate. In a real production system, the root CA certificate should be installed and validated. However, since this is running on a campus network as a student project, this trade-off reduced complexity while still ensuring that the communication is encrypted on the wire.

5.2. SAS Token Generation with mbedTLS

Azure uses Shared Access Signature (SAS) tokens to authenticate devices. In the code, SAS tokens are built by the function: String buildSasToken(const String &resourceUri, const String &keyBase64, time_t expiry) This function works in five main steps:

Decode the device key.

The device key in DPS_DEVICE_KEY_BASE64 is Base64-decoded into raw bytes using mbedtls_base64_decode.

Build the string to sign.

The code lowercases the resource URI, URL-encodes it, and concatenates it with the expiry time separated by a newline character, forming "sr\nse".

Compute the HMAC-SHA256 hash.

Using the mbedTLS HMAC-SHA256 API (mbedtls_md_* functions), the ESP32 computes a hash over the string to sign, using the decoded device key as the secret key.

Encode the signature.

The HMAC result is Base64-encoded and then URL-encoded so that it is safe to place inside a URL-style token.

Construct the final SAS token.

The function returns a string in the format:
SharedAccessSignature sr=<resource>&sig=<signature>&se=<expiry>.

The same pattern is used for both Azure DPS (with a DPS-specific resource URI) and Azure IoT Hub (with a {hub}/devices/{deviceId} resource URI), so the device can authenticate securely in both the provisioning phase and the normal telemetry phase.

5.3. Device Provisioning Service (DPS) Registration Flow

The function dpsRegisterAndGetHub() implements the complete DPS flow for this project. First, it configures MQTT to connect to global.azure-devices-provisioning.net on port 8883, then builds a DPS SAS token and an MQTT username that include the ID scope, registration ID, and API version. With these values, it connects the ESP32 to DPS using the registration ID as the MQTT client ID and the SAS token as the password, subscribes to the $dps/registrations/res/# topic to receive responses, and publishes a registration request that includes the registration ID in the JSON payload.

DPS responds with either a 202 status code and an operationId, which indicates the device is in the middle of being assigned, or a 200 status code with a registrationState that already contains the assigned hub and device ID. When a 202 is returned, the code saves the operationId and periodically publishes GET requests to the DPS operation-status topic until the status becomes "assigned". At that point, the ESP32 uses jsonGetNested() to extract the assignedHub and

deviceId values from the response payload and then disconnects from DPS so it can connect directly to the assigned IoT Hub.

Figure 4. Azure portal view of the ESP32 device registration through DPS and IoT Hub. The registered device entry verifies that DPS successfully assigned the ESP32 to the correct IoT Hub and created a cloud identity for telemetry.

5.4. IoT Hub MQTT Telemetry

Once DPS returns the hub and device ID, the ESP32 constructs a new SAS token for the IoT Hub and reconnects using MQTT over TLS. The function ensureIoTHubMqtt() checks whether the connection is active and reconnects if necessary. Telemetry is published to a topic of the form:

devices/{deviceId}/messages/events/

The payload is a JSON string constructed in the publishTelemetry() function, which packages the current temperature, humidity, soil-moisture percentage, and timestamp into a single message:

```
snprintf(payload, sizeof(payload),
"{\"Temperature\":%.2f,\"Humidity\":%.2f,"
"\"Soil\":%.2f,\"ts\":%lu}",
tC, h, soilPct, ts);
```

This format matches the device template in IoT Central so that the fields can be graphed and analyzed.

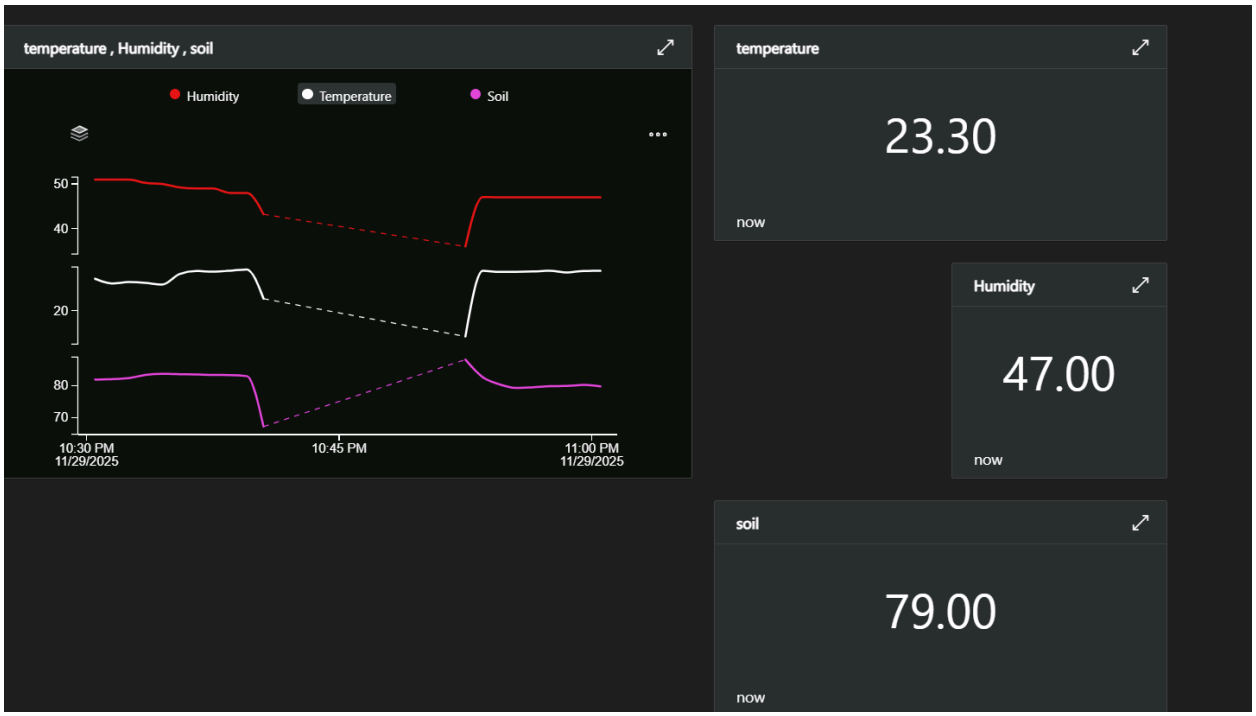

Figure 5. Azure IoT Central device dashboard for the ESP32 environmental monitor. The dashboard confirms that the device is connected and shows live telemetry values for temperature, humidity, and soil moisture.

5.5. Exporting Data to Power BI

Once telemetry reaches Azure IoT Central, it can be viewed in real time on the device dashboard, queried using Azure Data Explorer, and exported as a CSV file. For this project, I downloaded CSV data from Azure Data Explorer and imported it into Power BI, where the fields Temperature, Humidity, Soil, and ts became columns in a dataset. From there, I built line charts to show how each reading changed over time and used filters and slicers to focus on specific days, test periods, or sensor conditions, turning raw telemetry into organized, easy-to-read visual reports.

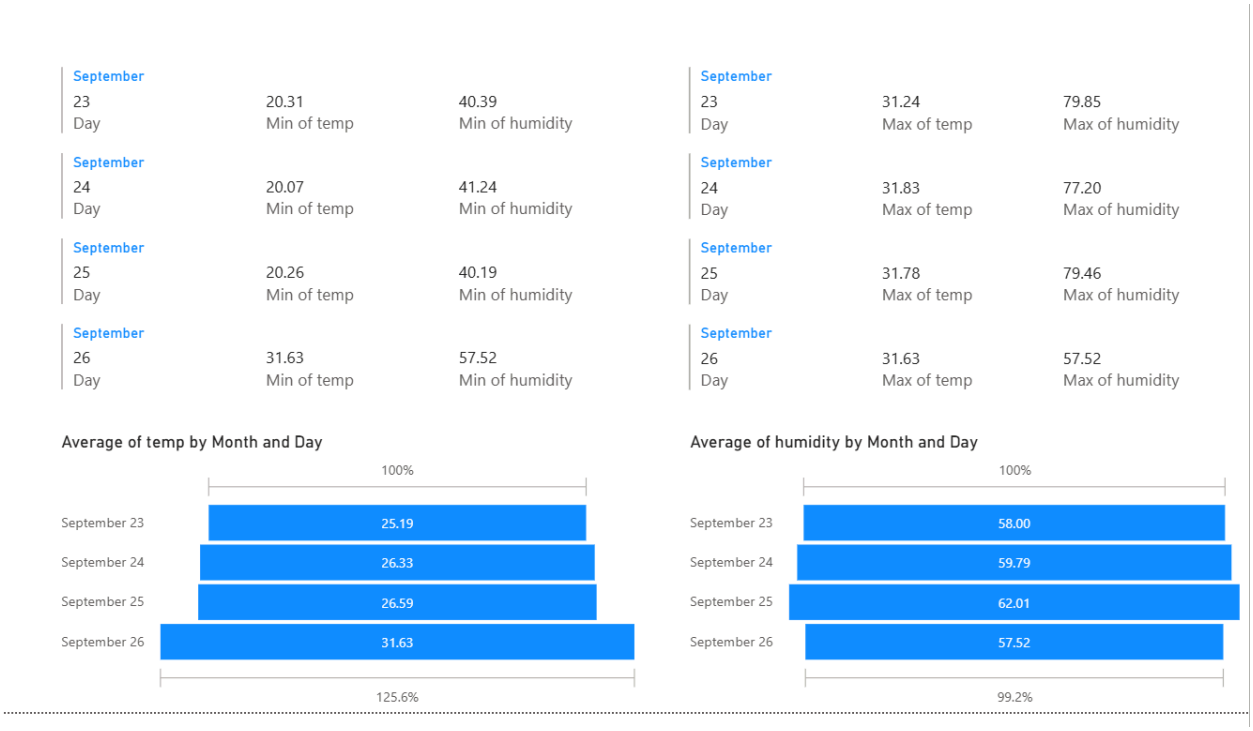

Figure 6. Power BI dashboard created from Azure IoT Central telemetry. The line charts track temperature, humidity, and soil moisture over time, turning raw ESP32 readings into clean, readable visual reports.

6. Implementing the Google Sheets Path

6.1. Motivation for a Second Data Path

The Azure path is powerful, but its setup can introduce extra complexity and possible cost, especially when connecting to additional services such as Power BI. That level of overhead is not always needed for smaller projects or early learners. In contrast, the Google Sheets approach is essentially free and much easier to access, which makes it a practical option not only for college students but also for younger or less experienced tech learners.

By sending data directly from the ESP32 to Google Sheets via HTTP and Apps Script, the project demonstrates an alternative that keeps the core IoT concepts, sensing, transmitting, and visualizing data,while using simple, familiar, and widely available tools.

6.2. Google Apps Script Web Service (doPost)

On the Google side, a short Apps Script defines a doPost(e) function and is published as a Web App. The script first checks whether the incoming request contains POST data, then parses the JSON payload into a JavaScript object so that it can access the individual fields. It appends a new row to the active sheet, inserting the current timestamp along with the sensor values from the ESP32, and returns a simple text response that the ESP32 can read and print to the serial monitor, confirming whether the data was logged successfully.

	A	B	C	D
7313	11/17/2025 10:21:28	22.5	49	58
7314	11/17/2025 10:21:32	22.3	49	58
7315	11/17/2025 10:21:37	22.4	49	58
7316	11/17/2025 10:21:48	22.4	49	58
7317	11/17/2025 10:21:53	22.6	49	58
7318	11/17/2025 10:21:57	22	49	58
7319	11/17/2025 10:22:02	22.5	49	58
7320	11/17/2025 10:22:07	22.6	49	58
7321	11/17/2025 10:22:12	22.4	49	58
7322	11/17/2025 10:22:18	22.5	49	58
7323	11/17/2025 10:22:22	22.7	49	58
7324	11/17/2025 10:22:28	22.2	49	58
7325	11/17/2025 10:22:33	22.9	49	58
7326	11/17/2025 10:22:38	22.6	49	58
7327	11/17/2025 10:22:42	22.4	49	58
7328	11/17/2025 10:22:48	22	49	58
7329	11/17/2025 10:22:53	22.9	49	58
7330	11/17/2025 10:22:57	22.6	49	58

Figure 7. Google Sheet populated by the ESP32 using the Apps Script Web App. Each row records the timestamp, temperature, humidity, and soil-moisture values, providing a simple, low-cost alternative to the Azure path.

6.3. ESP32 HTTP Client Code for Sheets

At a high level, the ESP32 Google Sheets sketch uses the WiFi.h, HTTPClient.h, and DHT.h libraries to handle networking and sensor input. The code connects to the BCUStudents Wi-Fi network, reads the current temperature, humidity, and soil-moisture percentage, and then constructs a JSON string such as:

```
String json = "{\"Temperature\":" + String(t) +

",\"Humidity\":" + String(h) +

",\"Soil\":" + String(soil) + "}";
```

This JSON payload is sent in an HTTP POST request to the Apps Script Web App URL, and the sketch logs the HTTP status code and any errors to the serial monitor for debugging. This path does not use TLS or SAS tokens in the same way Azure does, but because Google's Web App URL is served over HTTPS, the data is still encrypted in transit. For a classroom prototype, this is an acceptable level of security, although it should be hardened for any sensitive or production deployment.

6.4. Comparison: Azure IoT Central vs. Google Sheets

The Azure IoT Central path offers several advantages for this project. It provides an enterprise-style architecture built around DPS and IoT Hub, similar to what real organizations use in production. It has a strong security model based on SAS tokens and supports rich, built-in dashboards for monitoring device telemetry. Azure IoT Central also integrates well with advanced tools like Power BI and other Azure services, making it easier to extend the solution into more complex analytics and enterprise workflows.

However, this approach requires more configuration steps. Setting up DPS, IoT Hub, device templates, and permissions takes time, and the overall learning curve is steeper than simpler options, especially for students or beginners who are just starting to work with cloud platforms and IoT.

The Google Sheets path has its own strengths and weaknesses. On the positive side, it is very easy to learn and use, making it great for quick experiments or simple charts with minimal setup. Because the data is stored in a familiar spreadsheet, it is easy to share with classmates or instructors and to collaborate on basic analyses.

On the downside, this approach provides less control over security and device identity than a full Azure solution, and it is not aimed at large-scale deployments with many devices. From a CIS perspective, implementing both paths provided a balanced learning experience: Azure taught me about enterprise-level cloud patterns and security, while Google Sheets showed how to build lightweight APIs and practical data flows that are still useful in real projects.

7. Software Architecture and Code Walkthrough

7.1. Configuration Constants and Globals

Both the Azure and Google Sheets-based sketches start with configuration sections that define the core settings for the system. These include the Wi-Fi SSID and password, the cloud endpoints (such as the DPS ID scope and device key for Azure, or the Web App URL for Google Sheets), the sensor pins and types (DHT11 on GPIO 5 and soil sensor on GPIO 34), timing settings like SEND_INTERVAL_MS, and the calibration constants SOIL_RAW_DRY and SOIL_RAW_WET. In the Azure sketch, several global objects are also defined, such as:

```
WiFiClientSecure net;

PubSubClient mqtt(net);

DHT dht(DHTPIN, DHTTYPE);
```

In addition, strings like assignedHub and deviceId are declared at global scope because they must be accessible across the DPS registration, IoT Hub connection, and telemetry publishing parts of the code.

7.2. Wi-Fi Connection and Time Synchronization

The connectWiFi() function in the Azure sketch and the Wi-Fi setup in the Google Sheets sketch follow a similar pattern. They call:

```
WiFi.begin(WIFI_SSID, WIFI_PASS);
```

to start connecting to the configured Wi-Fi network, then loop until WiFi.status() == WL_CONNECTED, printing dots to the serial monitor to show connection progress, and finally log the assigned IP address once the connection is successful.

In the Azure path, the syncTime() function then uses configTime() with public NTP servers and a loop that checks time(nullptr) until a valid epoch time is returned. This accurate time is critical for SAS token expiry, because without a correct clock Azure would treat tokens as already expired or not yet valid and reject the device's connection.

7.3. Sensor Reading and Data Conversion

In both sketches, the main loop reads the sensors roughly every five seconds, based on SEND_INTERVAL_MS = 5000. Each cycle calls dht.readTemperature() and dht.readHumidity() to get the current temperature and humidity, and uses analogRead(SOIL_PIN) to read the soil-moisture sensor.

In the Azure sketch, the soil percentage is calculated using:

float soilPct = soilRawToPercent(soilRaw);

which maps the raw ADC value into a 0–100% range using the calibration constants SOIL_RAW_DRY and SOIL_RAW_WET. In the Google Sheets sketch, the soil percentage is computed using map() and then constrained between 0 and 100. If either the temperature or humidity reading returns NaN, the code logs an error message and skips sending telemetry for that cycle, which helps avoid publishing bad data and makes debugging easier.

7.4. Telemetry Payload Design

The Azure payload is a compact JSON string with four fields: Temperature, which stores the temperature in degrees Celsius; Humidity, which stores the relative humidity in percent; Soil, which represents the soil-moisture percentage from 0–100; and ts, which records the epoch timestamp in seconds.

The Google Sheets payload is almost the same but omits the timestamp, because the Apps Script uses new Date() on the server side to record the time of each entry.

This keeps the ESP32 payload simple and lets the spreadsheet control how timestamps are formatted. From a design perspective, using simple, flat JSON objects makes it easy for both Azure and Apps Script to parse the data, and for other tools or services to consume the readings later.

7.5. Error Handling and Logging

Because this project was built and tested by a single student, good error handling and logging were important. The sketches log key events such as Wi-Fi connection progress and the assigned IP address, any errors reported by the DHT sensor, DPS status codes and payloads in the Azure

sketch, MQTT connection successes or failures, and HTTP response codes from the Google Sheets Web App.

Most debugging was done using the Arduino Serial Monitor. Real-time logs of SAS token generation, DPS responses, and telemetry payloads made it much easier to see exactly where a problem occurred—whether it was a Wi-Fi issue, a DPS configuration error, or a bug in the code.

8. Testing and Evaluation

8.1. Test Plan

The test plan was divided into five phases that followed the development order of the project and ensured that each part of the system was working before moving on to the next:

Basic Connectivity – Check that the ESP32 can connect to BCUStudents and maintain a stable connection.

Sensor Validation – Verify that the DHT11 and soil sensors return reasonable values under different conditions (for example, hot/cold or dry/wet).

Azure Path Testing – Validate that the device can register via DPS, connect to IoT Hub, and send telemetry.

Data Export Testing – Verify that data from IoT Central can be exported via Data Explorer and imported into Power BI.

Google Sheets Path Testing – Confirm that HTTP POST requests reach the Apps Script and that rows are correctly appended to the spreadsheet.

Together, these stages provided a structured way to test connectivity, sensor accuracy, cloud integration, data export, and the alternative Google Sheets logging path.

8.2. Experimental Results

Several observations from testing showed that the system was behaving realistically. Indoor temperature readings generally ranged between 23–27 °C, while relative humidity readings hovered around 50–65%. Soil-moisture percentage started near 90–100% immediately after watering and dropped gradually over time as the soil dried out. The Azure dashboards and Google Sheets graphs both showed these expected curves over time, confirming that the system was capturing real environmental changes rather than random noise.

In a plant context, these trends matter because sustained high temperature with low humidity can increase water loss through the leaves, while high humidity at warm temperatures can support disease even if the soil moisture looks normal. Seeing all three readings together made it easier to interpret what conditions a plant would actually experience.

8.3. Reliability and Performance

With stable network conditions, the ESP32 handled both the Azure and Google Sheets paths reliably. The 5-second interval was conservative and allowed enough time for the cryptographic and networking operations to complete.

Occasional Wi-Fi drops or DPS timeouts occurred, especially on the campus network, but the sketches were often able to recover by reconnecting. During development, manual resets were used when necessary.

8.4. Security Review and Limitations

From a security perspective, this project has several strengths and some clear limitations. On the strong side, the Azure path uses TLS for MQTT connections, so telemetry is encrypted in transit, and it relies on SAS tokens with limited lifetimes instead of long-term static passwords. The ESP32 also uses NTP to keep accurate time, which is required for proper SAS token expiry and prevents tokens from being accepted outside their allowed window.

However, there are also limitations. In the Azure sketch, the use of net.setInsecure() disables certificate validation. Wi-Fi and DPS keys are stored directly in the code, and the Google Sheets path depends on a single Web App URL without strong device authentication. These are acceptable trade-offs for a senior CIS project focused on learning cloud concepts and staying

within a student budget, but they would need to be tightened and redesigned for any real-world or production deployment.

9. Project Management and Learning Outcomes

9.1. Balancing RA Duties and Senior Design

A unique part of this project was learning something completely new. IoT, Azure, and Google Sheets integrations, while balancing the load of being a CIS student. During this semester, I worked off campus in Orlando at Universal, held an on-campus job, and still tried to keep up with classes and this senior project. I also do not come from a deep coding or research background, so much of the material, especially around MQTT, TLS, SAS tokens, and cloud services, was new to me.

Because of that, I had to be very deliberate about how I managed my time and energy. I split the project into small, focused tasks that I could work on in short blocks, such as “get the DHT11 reading correctly,” “test soil sensor calibration,” or “fix one Azure connection error.” I kept notes on what I tried, what worked, and what broke so that when I came back from a long shift or a busy day on campus, I could pick up where I left off without starting over.

Although this project pushed me outside my comfort zone, the final result reflects not just technical progress but also the real-world challenge of balancing a job, school, and a complex CIS project at the same time.

9.2. Technical Growth as a CIS Student

From a learning perspective, this project touched almost every component of the CIS curriculum, even though I relied on several external resources for the Arduino C++ code. I did not just download one library, “fill in the blanks,” and upload it. I started with very little C++ background, so a major part of the work was carefully studying existing sketches, doing deep research, and going through a lot of trial and error to get everything working on the ESP32.

I pulled in libraries and example code from different people who had used an ESP32, DHT11 sensors, soil-moisture sensors, Azure IoT Central, Power BI, and Google Sheets before, then

mixed pieces of that code over time. Each change required testing, breaking things, reading error messages, and fixing issues until the final system became stable.

Programming. Instead of writing C++ from scratch, I focused on reading, understanding, and debugging existing ESP32 sketches. I combined libraries, tried different versions, and learned how small changes in includes, function calls, or configuration could completely break or fix the project. This forced me to understand what the code was doing, not just copy and paste it.

Networking. I gained hands-on experience with Wi-Fi, TCP, TLS, MQTT, HTTP, and NTP by seeing how each part of the stack was used in the code and then verifying that the device actually connected, authenticated, and sent data the way it was supposed to. Many problems only showed up when the whole system was running, so I had to debug across both code and network settings.

Cloud Platforms. I set up Azure IoT Central, DPS, and IoT Hub and learned how device identities, SAS tokens, and MQTT topics fit together in the bigger picture. On the Google side, I created the Apps Script Web App and linked it to Google Sheets as a simple cloud backend. Getting both paths working meant understanding enough about the cloud services to know whether an error was caused by my code, my keys, or my configuration.

Data. I practiced exporting CSV data from Azure, importing it into Power BI, and comparing those results with the data stored in Google Sheets. This helped me see the full path from raw sensor readings on the ESP32 to visual reports and charts that someone else could actually use.

Systems Thinking. Most importantly, I learned to view the project as a system made of interacting components, hardware, firmware, Wi-Fi, Azure, Google Sheets, and Power BI, rather than just a single piece of code. Each component had dependencies and its own type of failure, so getting everything to work together was a systems problem, not just a programming problem.

Even though I am not a C++ expert and did not write a completely original sketch from a blank file, I put in the work to understand how the code operates, to choose and combine compatible

libraries, and to test the system thoroughly until all the pieces worked together. That process connects directly back to CIS coursework such as systems analysis and design, where we focus on requirements, architecture, integration, and the practical value of cloud-based environments.

9.3. Skills Gained for Future Roles

Cloud Engineering / DevOps.

This project gave me hands-on practice connecting a physical device to cloud services instead of just reading about it. By working through the ESP32 code and mixing different libraries, I learned how the board authenticates with Azure IoT Central using SAS tokens and how DPS works as a provisioning layer that assigns devices to an IoT Hub. I also had to troubleshoot real connection problems, figuring out whether an issue was Wi-Fi, MQTT settings, bad keys, or time sync. This is the same type of problem-solving that cloud engineers and DevOps professionals use when they wire up services, manage secrets, and keep systems talking to each other securely.

Data Engineering / Analytics.

On the data side, I took raw sensor readings and converted them into something that could actually be analyzed. I exported telemetry from Azure as CSV, imported it into Power BI, and compared it with data logged in Google Sheets through Apps Script. I built simple charts that showed how temperature, humidity, and soil moisture changed over time and checked that the values stayed consistent across tools. Moving data between systems, checking its quality, and making it easy to understand is very similar to what data engineers and analysts do in real projects.

IoT and Embedded Systems Integration.

For IoT and embedded work, I had to understand how the ESP32, the DHT11, and the soil-moisture sensor fit together with Wi-Fi, Azure, and Google Sheets as one system. I did not just plug in a single library and hit upload; I pulled examples from different sources, mixed code from multiple sketches, and debugged the errors that came from combining them. It took a lot of testing to get the sensors, networking, and both cloud paths (Azure and Google Sheets) stable at the same time. That experience is at the heart of IoT integration—getting hardware, firmware, and cloud platforms to work together in a reliable, end-to-end solution that goes beyond a simple “hello world” demo.

10. Future Work

10.1. Hardware Extensions

Potential hardware enhancements for this project include replacing the DHT11 with a more accurate sensor such as the DHT22 or BME280, which would improve the quality of the temperature and humidity readings. A light sensor could be added to correlate soil moisture with sunlight exposure and better understand how lighting conditions affect plant health.

The system could also be upgraded by integrating a relay or MOSFET to drive a small water pump, turning it into an automatic watering solution instead of just a monitoring tool. Finally, deploying multiple ESP32 nodes in different locations and aggregating their data in the same Azure IoT Central application would make the setup feel more like a real-world IoT network rather than a single-device prototype.

10.2. Software and Security Improvements

Software and security improvements for this project could focus on several key areas. First, the Azure path could be strengthened by enabling full certificate validation instead of using setInsecure(), so the ESP32 properly verifies the server's identity. The sketch itself could be refactored into separate modules or classes for sensors, networking, and cloud logic, making the code easier to read, maintain, and extend.

Over-the-air (OTA) updates could be added so the firmware can be updated without physically reconnecting the board, which is important if the system were ever deployed in harder-to-reach locations. Finally, the device could support configurable thresholds and simple rules directly on the ESP32, allowing it to make basic local decisions, such as flagging dry soil or high temperature, even if the cloud connection is temporarily unavailable.

10.3. Integration with Ontologies and Data Analytics

The data from this project can also support more advanced academic work beyond basic charts and dashboards. In a database or web application course, the sensors, locations, and measurements could be modeled as an ontology in Protégé, with each reading represented as an

instance of a measurement class. The same data could be imported into tools like Weka to explore simple clustering or basic predictive models, such as estimating when soil moisture will drop below a certain threshold.

In addition, the overall system could be used as a case study in enterprise architecture discussions, comparing the Azure and Google Sheets paths in terms of scalability, cost, security, and long-term maintenance, and showing how different cloud patterns fit different project goals and budgets.

11. Conclusions

This senior project presents a complete, working IoT pipeline that connects physical sensors to two different cloud platforms. Starting from simple LED blinking and DHT11 tests, the design evolved into a system that supports secure Azure IoT Central integration using DPS and SAS tokens, exports data to Power BI, and also logs readings to Google Sheets through a lightweight HTTP API.

The solution emphasizes not only technical correctness but also clear documentation and maintainability. It reflects a realistic experience balancing coursework, out-of-class duties, and personal growth, while still delivering a quality report. The project provides a solid foundation for future work in cloud computing, data analytics, and IoT, and it shows how a relatively small set of hardware components, combined with the right cloud tools, can produce a powerful learning platform.

12. References

BI Tricks. (n.d.). How to create a KPI visual in Power BI | Power BI KPI Card Tutorial [Video]. YouTube. https://youtu.be/AnrGyoYsTmI

DIY TechRush. (n.d.). Introduction to ESP32 Board - Getting Started (Step by Step) [Video]. YouTube. https://youtu.be/aLEKiGNfHZw

DIY TechRush. (n.d.). Connect ESP32 to WiFi - ESP32 Beginner's Guide [Video]. YouTube. https://youtu.be/aH3sLEQI4_w

DIY TechRush. (n.d.). ESP32 Web Server - ESP32 Beginner's Guide [Video]. YouTube. https://youtu.be/z-I-r3PX2lU

DIY TechRush. (n.d.). ESP32 Tutorial - How to use Serial Monitor (NEW Arduino IDE) [Video]. YouTube. https://youtu.be/wia2sUjNpwI

DIY TechRush. (n.d.). ESP32 Tutorial - DHT11/DHT22 (Temperature and Humidity Sensor) [Video]. YouTube. https://youtu.be/K98h51XuqBE

DIY TechRush. (n.d.). ESP32 NTP Server - Real Time Clock (NO RTC Module) [Video]. YouTube. https://youtu.be/YBOks3mM4Ng

How To Electronics. (n.d.). Getting Started with Microsoft Azure IoT Central using NodeMCU ESP8266 [Video]. YouTube. https://youtu.be/jmMWel9Y3Qg

How To Electronics. (n.d.). Getting started with Azure IoT Central using ESP8266. *How2Electronics*.https://how2electronics.com/getting-started-with-azure-iot-central-using-esp8266/

Teach Me Something. (n.d.). Send Data to Google Spread Sheet using ESP32: IoT projects, IoT training, DHT11, Google Sheet [Video]. YouTube. https://youtu.be/5k2ztjHOEi8