# 1 Cuprins

# 2 Summary

## 2.1 State of the Art

Modern IoT systems that integrate environmental sensors and drones are widely used in agriculture, air quality monitoring, natural disaster prevention, and more. They are based on a modular architecture, where data is collected by sensors and transmitted via MQTT to a cloud platform for real-time analysis [1].

In agriculture, IoT systems based on ESP32 and Raspberry Pi help optimize irrigation by measuring temperature, air and soil humidity, and light levels [1][2]. The security of these systems is managed through encryption using symmetric algorithms, such as "Expeditious Cipher" [2].

In urban environments, a pollution monitoring system was proposed in [3] that uses MQ-2 and DHT22 sensors connected to Arduino and sends alerts via the Blynk application. The data is analyzed using machine learning algorithms to support decisions such as traffic routing.

The forest fire detection system in [4] uses smoke and flame sensors connected to a Raspberry Pi and is visualized in real time using MATLAB. The architecture is scalable and allows for the creation of automated alerts.

Due to their price-performance ratio, Raspberry Pi and ESP32 are recommended hardware platforms. In [5], the system uses DHT11 and MQ-135 sensors and LM393 comparators to automatically control fans based on weather conditions, demonstrating the interactive capabilities of the Internet of Things.

SEMAR is an open source platform that uses ESP32 and MQTT to monitor environmental parameters in real time. The architecture and components are fully described in [6] and are easy to replicate and extend.

Academic solutions such as [3] and [5] show that functional IoT systems can be built using affordable components and open source software. It can be extended for use with drones and mobile applications.

MQTT has the advantage of low resource consumption and is the most widely used protocol in IoT systems. According to [7], adding TLS for security requires a trade-off between security and autonomy, resulting in an increase in energy consumption from 18.76 J to 73.25 J.

Current limitations are:

- The drone has a short flight time (less than 30 minutes) [8]
- The hardware costs and complexity of specialized sensors are high [9]
- Wireless communication problems affected by interference and congestion can be solved by quality of service improvement algorithms [10].

In my current project, using ESP32 with MQTT and TLS requires a balance between security and energy efficiency.

## 2.2 Theoretical Fundamentals

IoT systems consist of internet-connected devices that transmit and process data independently. The classic architecture consists of three layers: perception (sensors), network (data transmission), and application (presentation/interaction) [11][12]. Enhanced applications will use communication technologies such as LoRa, Sigfox, and 5G [12][13].

Edge computing emerged as a solution to the limitations imposed by the cloud model. It allows local processing, reduces latency, and improves network traffic. The disadvantage is that local computing power is limited. Hybrid architectures (edge + cloud) offer a balance between speed and scalability [14][15][16][17].

The ESP32-S2 used in this project is a high-performance Wi-Fi microcontroller with an Xtensa LX7 processor (240 MHz), 320 KB of SRAM, and 43 GPIO pins. It integrates several peripherals (ADC, DAC, $I^2C$, SPI, UART, USB OTG) and supports AES, RSA, SHA encryption, and secure boot [18].

Compared to other IoT boards (Arduino Uno R3, Raspberry Pi Pico W, ESP8266, STM32), the ESP32-S2 offers superior performance, optimal power consumption, and wireless connectivity at a lower cost [18].

Sensors used:

- DHT22: Temperature and humidity accuracy is ±0.5 °C, relative humidity is 2-5 %. Range: from -40 to +80 °C, relative humidity 0 to 100% [23].
- SGP40: MOx sensor for volatile organic compounds (VOCs), output as "VOC indicator", low power consumption. Communicates via $I^2C$ and has automatic temperature and humidity compensation [24][25].

IoT systems need to be scalable and modular. Adding sensors via standard data buses ($I^2C$, SPI) allows for expansion without refactoring. In [27], an ESP32 system with humidity, pressure, light and VOC sensors was presented, all connected via $I^2C$ and communicating with a central server via Wi-Fi.

The MQTT protocol is ideal for the Internet of Things. It is energy efficient, works well in unstable networks, uses a publish/subscribe architecture, and is compatible with microcontrollers [29][30].

- Quality of Service 0: Delivery is not guaranteed
- Quality of Service 1: At least once (repeatable)
- Quality of Service 2: Exactly once (most secure, but slowest)

Messages are small, and the broker handles asynchronous communications. Additional encryption via TLS is recommended for protection [31].

TLS, the standard for securing communications, provides confidentiality, integrity, and authentication. This includes the handshake, encrypted channel, and cipher suite (ECDHE, AES, RSA, SHA256, etc.) [35][36][37].

TLS in IoT can impose significant energy costs. For microcontrollers, improved variants (e.g., ECC) are recommended [38][39].

Django is a back-end framework used for managing aggregated data. It is based on the Model-View-Template (MVT) architecture, which provides a clear separation between logic, data, and interfaces [40][41][42].

- Model: Database structure and rules
- View: Application logic
- Template: Data view

The REST API developed in Django is used for the system interface that communicates via HTTP requests and returns JSON data. It is compatible with modern user interfaces (React, Vue) and allows dynamic rendering [45][46].

The logical structure of an IoT system includes:

1. Sensors (DHT22, SGP40) connected to ESP32-S2
2. Wi-Fi communication via MQTT
3. Python scripts as a bridge to Django
4. Database + web interface for visualization

The logical architecture of the proposed IoT system is shown in Figure 1. It is simple, reproducible, and does not require expensive technology [47][48].

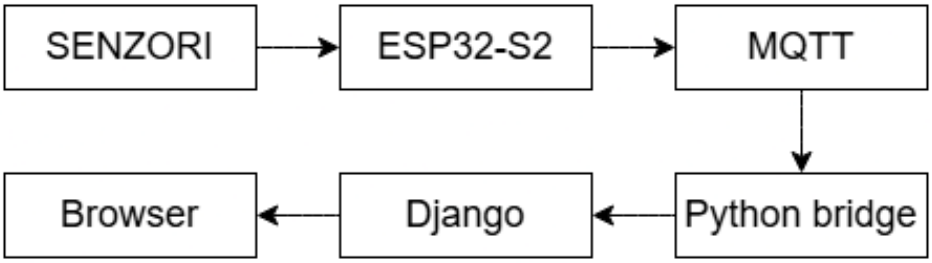



Figure 1. Logical architecture of the proposed IoT system

## 2.3 Implementation

The overall architecture of the proposed system consists of a drone equipped with an ESP32-S2 module that collects data from DHT22 (temperature and humidity) and SGP40 (VOC indicator) sensors. The data is securely transmitted via MQTT + TLS to a local server, processed by Python scripts, and displayed in a web interface developed in Django. The complete data flow is illustrated in Figure 2.

Hardware components:

- ESP32-S2 Thing Plus
- DHT22
- SGP40
- Breadboard + jumper wires
- 3.7 V Li-Po battery

Software components:

- Arduino firmware (main.ino)
- Mosquitto MQTT broker
- Python bridge script (mqtt_to_django.py)
- Django + SQLite backend
- Web interface (sensor_dashboard.html)

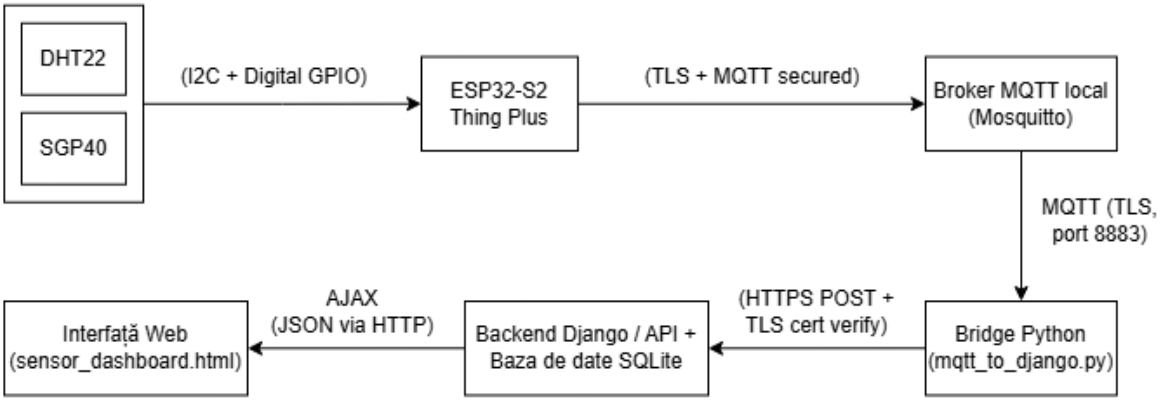



Figure 2. General architecture of the proposed IoT monitoring system

The system is based on a modular architecture, where the ESP32-S2 board periodically collects peripheral data and publishes it asynchronously using the MQTT protocol. Data packets in JSON formats are sent securely to the local Mosquitto MQTT broker using TLS encryption. A custom Python program running on the server subscribes to these MQTT topics and forwards the received data to a Django REST API endpoint for storage in an SQLite database. The data is then displayed in a web interface that is updated in real time using AJAX requests.

Sensor values are read using the standard Arduino library. The DHT22 communicates via digital GPIO pins, while the SGP40 uses the $I^2C$ protocol. Firmware developed in the Arduino IDE handles Wi-Fi connectivity, TLS negotiation, MQTT communications, and data collection from the sensors. The connection is configured using PEM certificates and mutual TLS authentication. An automatic reconnect logic is implemented to ensure stable operation even in the event of temporary connection problems.

A Python bridge script called "mqtt_to_django.py" listens to three MQTT topics: temperature, humidity, and air quality, and sends the values to the Django API via POST requests. The backend is built using the Django REST framework, and the data is validated through a serializer and stored in a simple form, with fields for timestamp, temperature, humidity, and VOC. The API is secure and only supports authenticated TLS connections.

On the front-end, data is automatically updated and displayed in a simple HTML page that does not require a page reload. Every 5 seconds, an AJAX poll retrieves the latest records via a custom JavaScript function. The CSV export function allows you to download the final record for offline analysis. To ensure system resilience, error handling and reconnection mechanisms are implemented at each communication level.

Regarding the physical assembly, the sensor is placed on a breadboard with appropriate connecting cables. A 10 kΩ pull-up resistor is added to stabilize the data signal from the DHT22 sensor. The 3.7 V Li-Po battery provides sufficient autonomy for portable operation. The ESP32-S2 did not require any additional voltage regulation, but its proximity to the DHT22 sensor caused slight thermal interference, which was later observed during experimental validation.

Configuring TLS was one of the main challenges of the implementation. The process of creating the client's certificate authority and validation required some modifications, especially regarding the "subjectAltName" field. To solve this issue, the OpenSSL configuration file was modified to include the necessary extensions to make it compatible with the ESP32-S2 certificate resolver.

Throughout the implementation process, the focus was on data integrity, secure transmission, and real-time availability. Each subsystem was tested individually before the full system integration. The final system demonstrated reliable functionality, from taking measurements to securely displaying them in a web interface, with the potential to expand by adding new types of sensors and connecting to cloud platforms.

## 2.4 Experimental Results

The testing phase aims to ensure the complete operation of the system in real scenarios. Each of the individual components was analyzed and integrated, from obtaining environmental values to displaying them in the web interface. The assembly consisting of the ESP32-S2 board and two sensors was placed in an environment with fixed parameters, and variations were introduced by making changes such as turning on the air conditioner or moving a marker closer to the SGP40 sensor.

To verify the functionality of the software, system components such as the local server, the Mosquitto MQTT broker, the Python bridge and the Django application were launched. The web interface can be accessed locally over the network, allowing real-time data monitoring. Exporting values in CSV format was also successfully tested in Microsoft Excel. As important aspects for assessing scalability, the frequency of interface updates, connection stability and the application behavior during outages were monitored.

In normal operation, the system operates for approximately 20 minutes, taking values every 5 seconds. The DHT22 and SGP40 sensors operated continuously, and the data was transmitted and displayed without interruption. The values were stable and fluctuated around a reference average. This is illustrated in Figure 3.

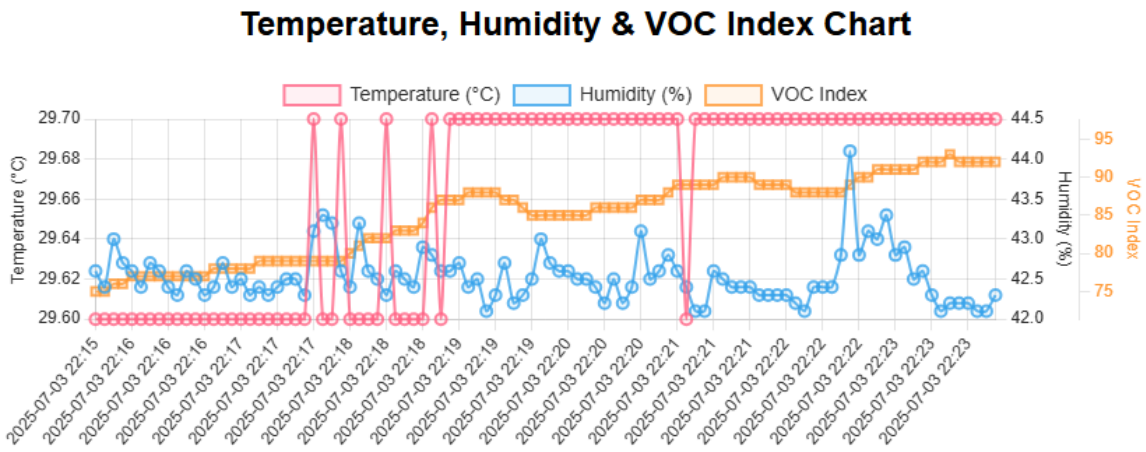



Figure 3. Graph of sensor values over approximately 8 minutes

To evaluate the accuracy of the DHT22 sensor, we compared it with a reference device, an indoor digital thermometer and hygrometer, placed under the same conditions. After installation, the DHT22 indicated a stable temperature of 31.4 °C, while the reference value indicated 28.8 °C, resulting in a difference of 2.6 °C. Regarding humidity, the difference was about 1.8%. These deviations are justified by local thermal effects caused by their proximity to the ESP32-S2 board, unlike the reference sensor, which is protected inside a sealed housing. These differences are acceptable for system purposes, but for professional applications a more sensitive sensor is recommended.

The sensor's response to direct stimulation was tested under controlled conditions. For the DHT22 sensor, the temperature was rapidly increased from 31.7 °C to 38 °C using a hair dryer, and the humidity decreased from 35% to 26%. The values returned to normal within 5-6 minutes. In another test, exposure to cold air from a fan lowered the temperature by 3.5 °C and increased the humidity to over 42%. This is shown in Figure 4.

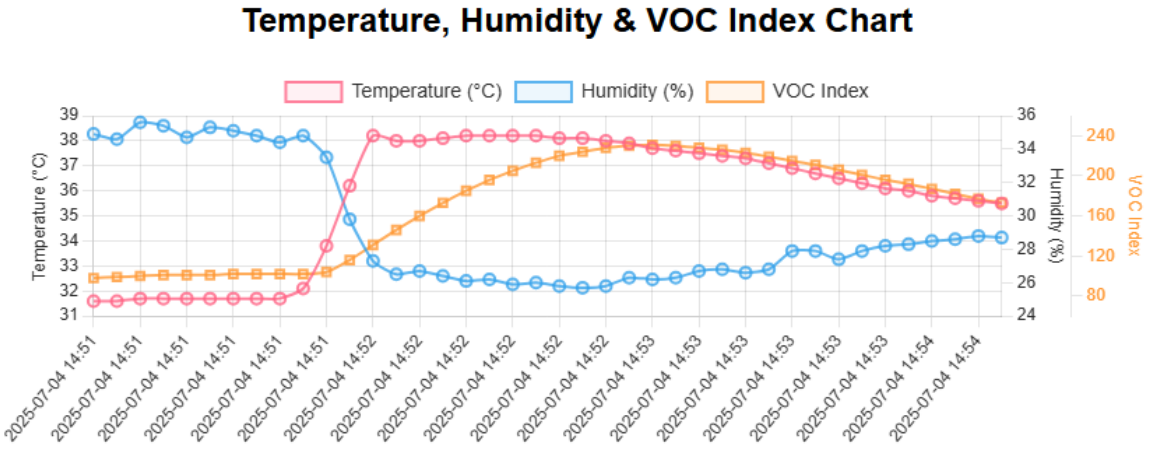



Figure 4. DHT22 sensor response to hot air exposure

The SGP40 sensor responded immediately to the proximity of the marker, with a sudden increase in the COV value from 100 to 245. This behavior can be observed in the web interface graph shown in Figure 5. The values stabilized approximately three minutes after the source was removed.

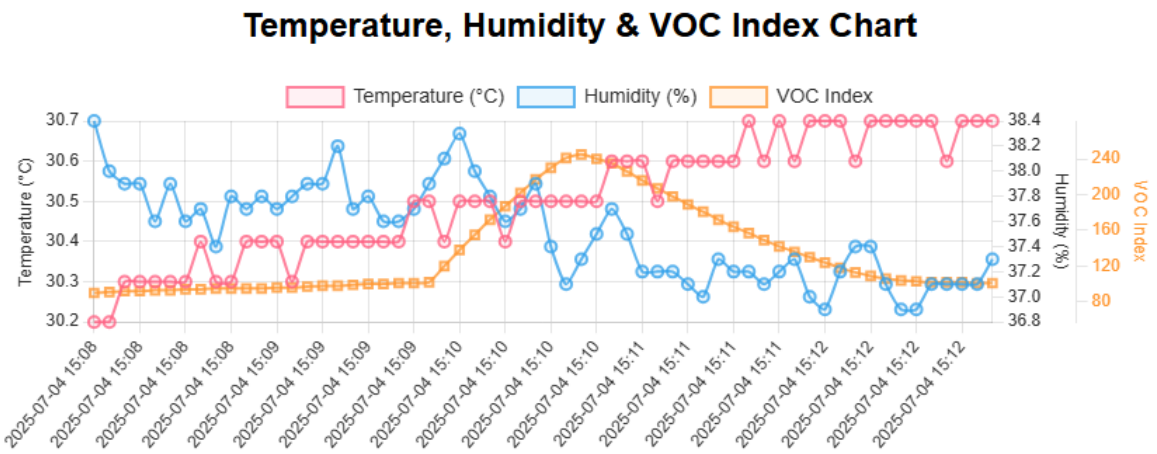



Figure 5. Sudden increase in VOC value when approaching a marker

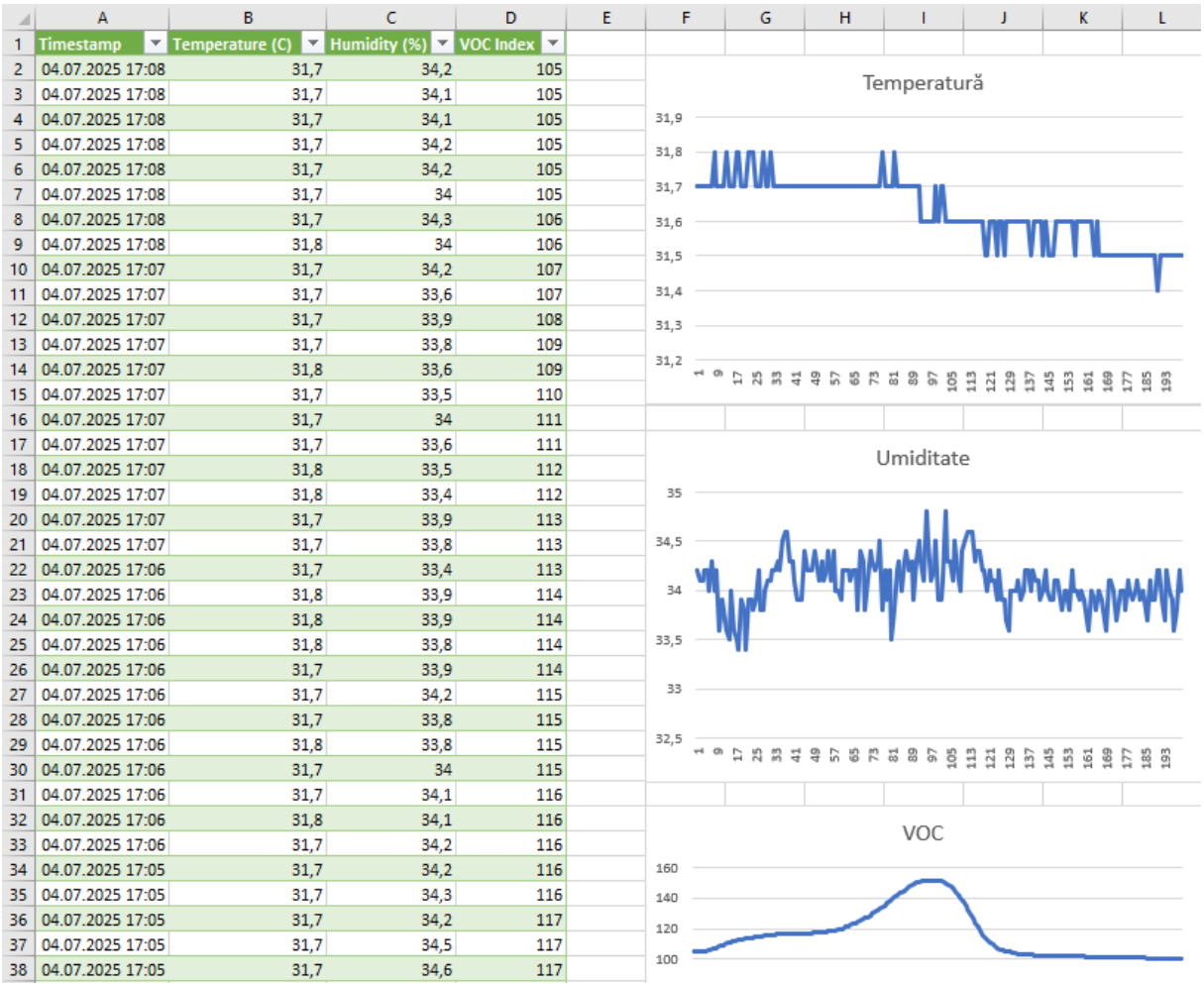
Boundary conditions were also simulated to verify the robustness of the system. If the Wi-Fi connection is lost, the ESP32-S2 board automatically tries to reconnect and continues to send values once the connection is re-established. As shown in Figure 6. In another scenario, the MQTT broker was intentionally stopped, generating an error message in the serial monitor, but the ESP32 module periodically tried to reconnect.

```
---------------------------------
DHT22 - Temp: 31.60 °C, Hum: 34.30 %
SGP40 - VOC Index: 112
All sensor data published.
---------------------------------
Connecting to WiFi..E (6785008) wifi:sta is connecting, return error
.............. Connected.
ESP32-S2 IP address: 192.168.[illegible]
Connecting to MQTT broker with TLS... Connected.
DHT22 - Temp: 31.60 °C, Hum: 34.50 %
SGP40 - VOC Index: 112
All sensor data published.
---------------------------------
DHT22 - Temp: 31.60 °C, Hum: 34.40 %
SGP40 - VOC Index: 112
All sensor data published.
---------------------------------
```

Figure 6. ESP32 board automatically reconnects to Wi-Fi network after signal loss

The behavior of the system when the Django application is stopped was tested by stopping the server while the Python bridge was receiving data. The script displays the error that occurred because the HTTPS connection could not be established and continues to run while waiting for the server to continue. After restarting the backend, the data flow was fully restored without any manual intervention.

The system also includes the ability to export data in CSV format, which can be accessed via a dedicated button in the web interface. The data uses ";" as a separator and can be analyzed in external applications such as Microsoft Excel. Accordingly, three graphs were created to evaluate the changes in temperature, humidity and VOC index, shown in Figure 7.




| | Timestamp | Temperature (C) | Humidity (%) | VOC Index |
|---|---|---|---|---|
| 2 | 04.07.2025 17:08 | 31,7 | 34,2 | 105 |
| 3 | 04.07.2025 17:08 | 31,7 | 34,1 | 105 |
| 4 | 04.07.2025 17:08 | 31,7 | 34,1 | 105 |
| 5 | 04.07.2025 17:08 | 31,7 | 34,2 | 105 |
| 6 | 04.07.2025 17:08 | 31,7 | 34,2 | 105 |
| 7 | 04.07.2025 17:08 | 31,7 | 34 | 105 |
| 8 | 04.07.2025 17:08 | 31,7 | 34,3 | 106 |
| 9 | 04.07.2025 17:08 | 31,8 | 34 | 106 |
| 10 | 04.07.2025 17:07 | 31,7 | 34,2 | 107 |
| 11 | 04.07.2025 17:07 | 31,7 | 33,6 | 107 |
| 12 | 04.07.2025 17:07 | 31,7 | 33,9 | 108 |
| 13 | 04.07.2025 17:07 | 31,7 | 33,8 | 109 |
| 14 | 04.07.2025 17:07 | 31,8 | 33,6 | 109 |
| 15 | 04.07.2025 17:07 | 31,7 | 33,5 | 110 |
| 16 | 04.07.2025 17:07 | 31,7 | 34 | 111 |
| 17 | 04.07.2025 17:07 | 31,7 | 33,6 | 111 |
| 18 | 04.07.2025 17:07 | 31,8 | 33,5 | 112 |
| 19 | 04.07.2025 17:07 | 31,8 | 33,4 | 112 |
| 20 | 04.07.2025 17:07 | 31,7 | 33,9 | 113 |
| 21 | 04.07.2025 17:07 | 31,7 | 33,8 | 113 |
| 22 | 04.07.2025 17:06 | 31,7 | 33,4 | 113 |
| 23 | 04.07.2025 17:06 | 31,8 | 33,9 | 114 |
| 24 | 04.07.2025 17:06 | 31,8 | 33,9 | 114 |
| 25 | 04.07.2025 17:06 | 31,8 | 33,8 | 114 |
| 26 | 04.07.2025 17:06 | 31,7 | 33,9 | 114 |
| 27 | 04.07.2025 17:06 | 31,7 | 34,2 | 115 |
| 28 | 04.07.2025 17:06 | 31,7 | 33,8 | 115 |
| 29 | 04.07.2025 17:06 | 31,8 | 33,8 | 115 |
| 30 | 04.07.2025 17:06 | 31,7 | 34 | 115 |
| 31 | 04.07.2025 17:06 | 31,7 | 34,1 | 116 |
| 32 | 04.07.2025 17:06 | 31,8 | 34,1 | 116 |
| 33 | 04.07.2025 17:06 | 31,7 | 34,2 | 116 |
| 34 | 04.07.2025 17:05 | 31,7 | 34,2 | 116 |
| 35 | 04.07.2025 17:05 | 31,7 | 34,3 | 116 |
| 36 | 04.07.2025 17:05 | 31,7 | 34,2 | 117 |
| 37 | 04.07.2025 17:05 | 31,7 | 34,5 | 117 |
| 38 | 04.07.2025 17:05 | 31,7 | 34,6 | 117 |

Figure 7. Graph generated in Excel based on exported data

The pilot test verifies that the system operates correctly, is stable in normal mode, and is able to respond correctly in fault scenarios. The integration of hardware and software components has proven reliable, and the modular structure of the system allows it to adapt to a variety of local monitoring applications.

## 2.5 Conclusions

This study aimed to develop an IoT system capable of monitoring three environmental parameters in real time: temperature, humidity and VOC index. The system is designed to operate independently, transfer data securely and display it via a locally accessible web interface.

The application is based on the ESP32-S2 module, which was chosen for its built-in support for Wi-Fi connectivity and TLS encryption. It is connected to two sensors: DHT22 (for temperature and humidity) and SGP40 (for air quality). The data was sent to a locally installed Mosquitto broker via MQTT using a TLS encrypted connection. This data is received by a Python bridge script, transmitted to a Django backend and stored in a SQLite database.

The architecture is designed to be modular and scalable, so that each component (acquisition, transport, processing, storage, display) can be extended or replaced without major changes. The web interface, built using HTML and JavaScript, allowed for asynchronous data updates via AJAX without page reloads and included CSV export functionality for further analysis in applications such as Excel, Python, and MATLAB.

The main objective of this study is to secure communication. We configured one-way TLS authentication between the ESP32-S2 and the broker and between the Python bridge and the

Django server. The use of self-signed certificates and server validation prevents “man-in-the-middle” attacks and data transmission corruption.

The tests examined not only the functionality of the system in normal mode, but also its reaction to unforeseen situations. The sensor clearly responded to artificial stimuli, such as hot air, cold air, and volatile compounds. The differences from the reference device were explained by incremental effects (e.g., local heating caused by the ESP32-S2). We also tested the behavior in case of Wi-Fi network disconnection, MQTT broker shutdown, and Django server outage. In both cases, the system was able to recover independently, without any external intervention.

This work gave us the opportunity to directly apply our theoretical knowledge in a real-world context, including development boards, sensors, communication protocols, and interaction with modern backend technologies. Problems that arose during development (e.g., missing “subjectAltName” field in certificates, synchronization of MQTT messages) were solved in concrete ways, and the lessons learned are important for future IoT projects.

The result is a stable, functional and scalable system. It can be integrated as an accessory for commercial drones or as a fixed node in an IoT sensor network. The proposed architecture and technical solution can be easily adapted to more advanced requirements, such as the use of more precise sensors, more complex databases and the implementation of advanced data analysis capabilities.

# 3 Planificarea activității

| Plan de lucru pentru elaborarea tezei de licență (19 săptămâni - *27.02.2025-09.07.2025* ) | | |
|---|---|---|
| **Obiective** | **Titlul pachetului de lucru/Intervalul de implementare** | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19<br>2025 |
| **Ob. 1** | **Managementul tezei de diplomă** | |
| | Livrabil: Documentare inițială, alegerea senzorilor și platformei ESP32 | |
| **Ob. 2** | **Stadiul actual** | M1 |
| | Livrabil: Material pentru cap. Stadiul actual | |
| **Ob. 3** | **Alegerea și dezvoltarea soluției tehnice** | M2 |
| **Ob. 3.1** | **Fundamentare teoretică a soluției** | |
| | Livrabil: IoT, MQTT, ESP32, securitate TLS, Django | |
| **Ob. 3.2** | **Implementarea propriu-zisă a soluției** | |
| | Livrabil: Hardware, software, firmware, backend, web | |
| **Ob. 3.3** | **Migrarea la arhitectura securizată** | |
| | Livrabil: MQTT ---> TLS + backend Django | |
| **Ob. 4** | **Testare și validarea soluției** | |
| | Livrabil: Rezultate experimentale, grafice, testare cu stimuli, erori | |
| **Ob. 5** | **Redactarea și finalizarea lucrării** | M3 |
| **Ob. 5.1** | **Redactarea capitolelor** | |
| | Livrabil: cap. Fundamentare teoretică, cap. Implementarea soluției adoptate, cap. Rezultate experimentale, cap. Concluzii | |
| **Ob. 5.2** | **Revizuire finală, anexe și traduceri** | |
| | Livrabil: cap. Sumarry, cap. Anexe, corecturi finale | |
| | M1 - Miletsone 1, M2 - Milestone 2, M3 - Milestone 3 | |

# 4 Stadiul actual

## 4.1 Sisteme existente cu drone și senzori de mediu

Monitorizarea mediului folosind sisteme IoT (Internet of Things) bazate pe senzori integrați și pe drone cu senzori a avut parte de o dezvoltare alertă în diverse domenii precum agricultura, monitorizarea calității aerului și prevenirea dezastrelor naturale. Aceste soluții ies în evidență prin colectarea de date ale mediului înconjurător în timp real. Ele oferă o perspectivă mai profundă asupra condițiilor atmosferice sau a parametrilor distincți pentru aplicațiile industriale.

### 4.1.1 Aplicații IoT în agricultură

Agricultura modernă utilizează tot mai mult sisteme IoT bazate pe drone sau pe microcontrolere precum ESP32 și Raspberry Pi. Prin intermediul acestor sisteme se pot măsura și monitoriza anumiți parametri cum ar fi temperatura, umiditatea aerului, umiditatea solului și nivelul de lumină. În [1] se prezintă o arhitectură modulară în trei straturi: dispozitiv, cloud și aplicație. Senzorii de mediu trimit date către o platformă cloud prin intermediul unor protocoale de comunicație. Datele sunt analizate, procesate și pot genera acțiuni automate. Această abordare permite agricultorilor să ia decizii bazate pe date concrete și actualizate în timp real. Rezultatul este un randament îmbunătățit al culturilor și reducerea consumului de resurse naturale.

Un alt exemplu relevant este prezentat în [2], unde este dezvoltat un sistem de irigare complet automat. Sistemul integrează senzori de sol, de temperatură și de lumină, conectați la un nod ESP32. Comunicarea dintre senzori și ESP32 este realizată în mod criptat prin MQTT (Message Queuing Telemetry Transport). Criptarea se face printr-un algoritm simetric numit „Expeditious Cipher”, adaptat pentru microcontrolere. În această lucrare se evidențiază provocările securității rețelelor IoT și oferă soluții concrete pentru a proteja integritatea datelor în scenarii reale.

### 4.1.2 Monitorizarea poluării în orașele mari

Urbanizarea accelerată a dus la o necesitate de monitorizare a calității aerului tot mai mare în orașele aglomerate. Articolul [3] propune o soluție acestei probleme prin monitorizarea poluanților atmosferici, folosind senzori de tip MQ-2 pentru detecția de gaze precum butanul și dioxidul de carbon și senzori de tip DHT22 pentru temperatura și umiditatea aerului. Acești senzori sunt conectați la un nod Arduino care transmite datele prin protocolul MQTT către o infrastructură de tip cloud. Utilizatorii primesc alerte în timp real, prin intermediul unei aplicații mobile numite Blynk. Datele sunt afișate și analizate prin intermediul platformei ThingSpeak. Pentru interpretarea lor sunt folosiți algoritmi de învățare automată. Prin integrarea metodelor de analiză predictivă, sistemul permite estimarea nivelurilor viitoare de poluare și sprijină luarea deciziilor automate, cum ar fi redirecționarea traficului în zonele cu risc crescut sau chiar închiderea unor astfel de zone.

### 4.1.3 Gestionarea dezastrelor naturale

Un alt domeniu important în care sunt implicate sistemele de monitorizare cu senzori este detectarea timpurie a incendiilor în agricultură sau păduri. Articolul [4] are în vedere utilizarea unui sistem de supraveghere bazat pe senzori de fum și de flăcări, conectați la un Raspberry Pi. Datele citite sunt transmise către platforma ThingSpeak. Procesarea și afișarea în timp real a datelor se face cu ajutorul software-ului MATLAB. Sistemul este structurat pe trei niveluri: dispozitiv, cloud și aplicație. Acesta permite generarea alertelor automate la depășirea unor

praguri stabilite în prealabil. Arhitectura prezentată este scalabilă și se potrivește în contextele în care este nevoie de detecție rapidă și de intervenție eficientă.

### 4.1.4 Platforme hardware utilizate

Din punct de vedere hardware, platformele cele mai des utilizate în proiectele studiate sunt Raspberry Pi și ESP32. Acestea sunt, de regulă, preferate datorită echilibrului între performanță, flexibilitate și preț. În articolul [5] este descris un sistem de monitorizare a calității aerului care folosește senzori de temperatură (DHT11), de gaze (MQ-135) și un comparator (LM393) pentru detectarea ploii. Toți senzorii sunt conectați la un Raspberry Pi care transmite datele către platforma ThingSpeak pentru vizualizare și control. În plus, este inclus un mecanism de control automat, fiind vorba despre un ventilator activat printr-un releu. Acest lucru ilustrează capacitatea sistemelor IoT de a reacționa în mod automat la schimbarea condițiilor de mediu.

### 4.1.5 Soluții open-source și academice

Apariția multiplelor soluții open-source și academice bazate pe drone și sisteme de monitorizare a mediului, spre deosebire de dezvoltarea sistemelor comerciale sau industriale, demonstrează fezabilitatea abordărilor tehnice accesibile și care pot fi replicate. Aceste proiecte reprezintă o sursă de inspirație valoroasă pentru dezvoltatori, cercetători și studenți prin punerea la dispoziție a codului sursă, schemelor electrice, infrastructurilor și metodologiilor de integrare, care pot fi ulterior adaptate și îmbunătățite.

### 4.1.6 Platforma SEMAR

Platforma SEMAR (Smart Environmental Monitoring and Analytical in Real-Time) este un exemplu open-source care integrează microcontrolerul ESP32 și protocolul MQTT. În [6], autorii propun această platformă care preia date precum temperatura, umiditatea și compușii volatili de la senzori, după care le transmite prin protocolul MQTT către un server central. Acest server rulează local și este bazat pe software open-source. Oferă o interfață dedicată atât vizualizării datelor în timp real, cât și administrării. Platforma se poate extinde prin integrarea mai multor noduri de senzori și poate fi adaptată pentru alte aplicații, cum ar fi cele din domeniul agriculturii, ale orașelor mari sau pentru monitorizare industrială.

Atât arhitectura software, cât și componentele hardware utilizate sunt descrise în detaliu și facilitează replicarea sistemului menționat. Platforma SEMAR poate fi o platformă didactică ideală pentru proiectele academice și este o alternativă viabilă la soluțiile comerciale mai costisitoare.

### 4.1.7 Soluții academice

În literatura de specialitate apar mai multe exemple de prototipuri academice care utilizează componente accesibile și abordări open-source. Unul dintre acestea ar fi sistemul de monitorizare a calității mediului descris în [5]. Acest sistem format din placa de dezvoltare Raspberry Pi și senzori standard precum DHT11 și MQ-135, cu integrare în platforma ThingSpeak, este un sistem care poate fi extins cu ușurință. Codul este scris în Python, iar autorii evidențiază aplicabilitatea acestuia în domenii precum controlul industrial și metode de agricultură inteligente.

De asemenea, sistemul de monitorizare a poluării aerului din [3] se concentrează pe aplicații urbane precum măsurarea în timp real a temperaturii, umidității, dioxidului de carbon și a gazului butan. Arhitectura formată din senzori Arduino, împreună cu tehnologiile IoT și transmiterea datelor prin protocolul MQTT sunt toate tehnologii open-source. Acestea pot fi adaptate și scalate

la alte sisteme inteligente, cum ar fi utilizarea pe drone, platforme mobile sau în sisteme de trafic automatizat.

### 4.1.8 Relevanță pentru proiectul propus

Soluțiile open-source disponibile demonstrează că este fezabilă construirea sistemelor de monitorizare a mediului. Acestea nu trebuie să implice investiții semnificative, precum cele industriale, fiind totodată funcționale, scalabile și sigure. Utilizarea de microcontrolere, precum ESP32, care oferă conectivitate Wi-Fi, reprezintă o soluție practică pentru aplicații IoT. Aceasta, împreună cu folosirea protocoalelor de comunicație precum MQTT și a interfețelor web personalizabile construite cu framework-uri moderne pentru prezentarea datelor, reprezintă o abordare relevantă în mod direct pentru proiectul de față. Avantajele majore ale acestor sisteme sunt adaptabilitatea și ușurința de a le extinde prin adăugarea funcționalităților suplimentare.

## 4.2 Tehnologii și protocoale utilizate în practică

Cel mai frecvent protocol de comunicare utilizat în sistemele IoT este MQTT. Acest protocol funcționează pe principiul publish-subscribe. Datorită consumului redus de resurse, a eficienței în benzile limitate și a compatibilității cu microcontrolere precum ESP32, acesta este ideal pentru transmiterea datelor colectate de la senzori către un server sau un broker.

MQTT nu oferă criptare nativă a datelor, dar se pot folosi protocoale precum TLS (Transport Layer Security) pentru a asigura o comunicație protejată. Conform articolului [7], autorii au implementat și testat un sistem format dintr-un ESP32 care transmite date către un broker MQTT instalat pe un Raspberry Pi. Au fost comparate consumul de energie, latența și complexitatea generală pentru două cazuri de funcționare a sistemului. În primul caz s-a testat fără conexiune TLS, iar în al doilea caz, criptarea a fost activată.

Conform rezultatelor obținute, consumul de energie a crescut de aproximativ patru ori în contextul folosirii TLS, la nivelul QoS 2 (Quality of Service), de la 18,76 J fără TLS la 73,25 J. Cu toate că securitatea datelor are un rol important, autorii concluzionează că este necesar un compromis între securitate și autonomie, mai ales în cazul aplicațiilor mobile sau alimentate de o baterie [7].

## 4.3 Limitări în sistemele existente

Chiar dacă sistemele IoT bazate pe microcontrolere și senzori au evoluat semnificativ în perioada recentă, există totuși o serie de limitări, atât tehnice, cât și practice. Acestea influențează performanța și aplicabilitatea lor în scenariile reale. Principalele limitări sunt cele legate de autonomie, consumul energetic, costuri, stabilirea comunicațiilor wireless fără întreruperi și complexitatea implementării securității.

### 4.3.1 Autonomia sistemelor mobile

Una dintre principalele limitări în utilizarea dronelor este autonomia redusă a bateriei. În articolul [8], autorii folosesc dronele în scopul cercetărilor ecologice. Deși cercetarea acestora este explicită pe un grup specific de viețuitoare marine, concluziile despre tehnologia utilizată sunt general valabile.

Dronele comerciale au în mod obișnuit o autonomie sub 30 de minute. În momentul în care este necesară montarea senzorilor, a camerelor video sau a altor echipamente, consumul energetic crește considerabil și limitează durata unei sesiuni de observație aeriană.

Autorii subliniază necesitatea planificării fiecărui zbor, luând în considerare toți factorii care ar putea afecta durata de zbor a dronei. În plus, întreruperile necesare schimbării bateriei pot compromite calitatea datelor în cazurile în care este esențială continuitatea observației.

### 4.3.2 Costurile și complexitatea hardware

Un alt aspect important este cel legat de costul total al sistemelor cu senzori montați pe drone. În [9] se discută despre starea actuală a senzorilor utilizați pe drone. Deși senzorii existenți pot oferi performanțe bune, trebuie să se țină cont de mai mulți factori înainte de alegerea acestora. Prețul poate deveni unul costisitor dacă este nevoie de calibrare precisă sau transmisie securizată. Soluțiile comerciale sunt eficiente, dar greu de adaptat. Soluțiile open-source oferă mai multă flexibilitate, dar necesită și cunoștințe tehnice mai avansate.

### 4.3.3 Provocări în comunicația wireless

Un alt set de probleme apare în rețelele wireless, mai ales atunci când calitatea semnalului este slabă sau există mai multe dispozitive conectate pe aceeași rețea. În [10], autorii discută despre tehnicile de optimizare a calității serviciului (QoS) în rețelele IoT. Funcționarea normală a aplicațiilor poate să fie afectată de interferențe, pierderi de pachete, congestie și variații de latență.

Autorii recomandă utilizarea unor tehnici de optimizare bazate pe algoritmi euristici, stocastici și memetici. Prin ajustarea inteligentă a resurselor și a fluxului de date, aceste soluții contribuie la menținerea unei calități acceptabile în rețelele wireless.

### 4.3.4 Observații generale

Toate aceste limitări sunt importante și trebuie luate în calcul încă din faza de proiectare. În proiectul propus, utilizarea protocolului MQTT cu TLS aduce un plus de securitate, dar și un cost energetic mai mare microcontrolerului ESP32. Este necesară justificarea fiecărei alegeri în funcție de context și de nevoile aplicației.

*Instrumentele de inteligență artificială au fost utilizate doar pentru corectarea gramaticală și stilizarea formală a textului. Conținutul lucrării nu a fost redactat cu ajutorul acestora. Intervențiile au vizat claritatea, lizibilitatea și coerența exprimării, fără a modifica fondul ideilor.*

# 5 Fundamentare teoretică

## 5.1 Internet of Things (IoT)

### 5.1.1 IoT și aplicații în telecomunicații

Prin conceptul de IoT înțelegem o rețea de dispozitive fizice („lucruri") dotate cu senzori și procesoare, conectate la internet, care transmit și prelucrează date, fără a fi nevoie de intervenție umană. Aceste dispozitive sunt identificabile, capabile să comunice între ele și pot funcționa autonom în rețelele distribuite. Sunt sisteme care interacționează cu mediul înconjurător și cu utilizatorii, contribuind la atingerea obiectivelor comune [11].

Din punct de vedere istoric, ideea de IoT nu este chiar nouă. Acest termen a fost folosit pentru prima dată în 1999, când Kevin Ashton a introdus sintagma „Internet of Things" în contextul utilizării tehnologiei RFID (Radio-Frequency Identification), introducând totodată ideea de obiecte capabile să comunice date despre ele însele. De fapt, primele aplicații reale asemănătoare cu cele IoT au fost sistemele M2M (machine-to-machine), în care mașinile își transmiteau singure starea către un centru de control. Fiind vorba despre un sistem închis care monitoriza și gestiona datele, lipsea componenta de rețea globală specifică domeniului IoT de astăzi [11].

Sistemele IoT nu au o arhitectură standard, dar în general sunt distribuite pe mai multe straturi. Cea mai frecvent întâlnită arhitectură este cea pe trei niveluri:

1. Primul este stratul de percepție, reprezentând nivelul fizic în care are loc interacțiunea cu mediul înconjurător. Din acest strat fac parte senzorii și dispozitivele care au rolul de a măsura temperatura, de a detecta mișcările sau nivelul de luminozitate.
2. Al doilea este stratul de rețea, responsabil de transmiterea datelor. Informațiile primite de la senzori sunt transmise către sistemele care procesează datele prin tehnologii precum Wi-Fi, rețele celulare sau protocoale specializate.
3. Al treilea este stratul de aplicație, care procesează și afișează informațiile relevante pentru utilizator. Acesta poate interacționa cu sistemul prin intermediul aplicațiilor de tip smart home, al sistemelor de control industrial sau al platformelor de monitorizare [11][12].

În literatură sunt prezentate și alte tipuri de arhitecturi, precum cele pe cinci niveluri care includ un strat middleware sau un strat business, sau cele orientate spre servicii SoA (service-oriented architecture). Cu toate acestea, funcționalitatea de bază rămâne aceeași în ceea ce privește logica de colectare, transmitere și prelucrare a datelor. Rolul middleware-ului este de a permite integrarea mai ușoară a dispozitivelor diferite, fără ca dezvoltatorul să cunoască toate detaliile tehnice [11].

În ceea ce privește legătura cu domeniul telecomunicațiilor, conectarea dispozitivelor IoT depinde mult de distanța care trebuie acoperită și de nevoile aplicației. Pentru acoperirea unor zone mici, cum ar fi o casă sau un birou, se folosesc tehnologii precum Bluetooth, ZigBee sau Wi-Fi, reprezentând soluții simple și cu consum redus. În schimb, pentru aplicații care trebuie să acopere suprafețe mari, cum ar fi monitorizarea mediului sau dronele, este nevoie de tehnologii cu rază lungă de acțiune, precum LoRa sau Sigfox, care oferă o autonomie mare și consum redus de resurse. Pentru aplicațiile complexe, care necesită viteze mari și latență mică, tehnologia 5G devine soluția potrivită. Fiecare tehnologie are avantajele și limitările sale, prin urmare, alegerea depinde de context [12][13].

În aplicațiile IoT destinate agriculturii inteligente, monitorizării orașelor sau mediului înconjurător, majoritatea sistemelor trimit datele colectate de la senzori, prin rețele wireless, către un server local. În aceste cazuri, se folosesc protocoale precum MQTT, care permit transmiterea eficientă a datelor cu un consum de energie redus, inclusiv în condițiile unei rețele instabile.

### 5.1.2 Arhitecturi IoT – edge, cloud și hibride

Un factor important în cadrul sistemelor IoT este cel referitor la procesarea datelor. Sistemele IoT pot genera volume mari de date acumulate de la senzori, iar trimiterea integrală a acestora direct către serverele centrale din cloud poate să producă întârzieri care influențează negativ performanța aplicațiilor. Aplicațiile care se bazează pe un răspuns instant, ca în cazul sistemelor de transport autonom, a rețelelor electrice sau a celor de supraveghere video, nu pot opera eficient dacă răspunsul nu vine imediat după producerea unui eveniment declanșator [14].

Pentru o perioadă îndelungată, modelul cloud computing a reprezentat soluția predominantă pentru procesarea datelor IoT. Avantajul major era datorat puterii de calcul ridicate și capacității de stocare virtuală și a resurselor practic nelimitate. Datele erau trimise, cu scopul de a fi analizate, corelate și arhivate, către puncte numite centre de date. Această metodă rămâne în continuare eficientă în cazul analizei complexe, agregărilor de date la scară largă sau în cazul aplicațiilor în care latența nu este un factor critic [15].

Totuși, în timp, au apărut o serie de limitări. Pe de o parte, trimiterea constantă a datelor către cloud crește semnificativ traficul de rețea și consumul de bandă. Pe de altă parte, funcționalitatea anumitor aplicații, care depind de o conexiune constantă la internet, poate fi compromisă în situațiile de instabilitate sau întrerupere a conectivității. În plus, centralizarea integrală a procesării datelor în cloud poate ridica probleme privind confidențialitatea și securitatea, mai ales în cazul datelor sensibile [16][17].

Ca răspuns pentru aceste limitări a apărut paradigma edge computing. Aceasta implică mutarea unei părți de procesare a datelor mai aproape de sursele generatoare, adică în apropierea senzorilor și dispozitivelor IoT. Elementele intermediare, precum gateway-uri, ruterele inteligente sau terminalele finale, pot efectua procesări locale fără să depindă integral de infrastructura cloud [14]. Avantajul major oferit de edge computing este reducerea latenței. Datele nu trebuie să mai parcurgă distanțe lungi prin rețea, ci sunt interpretate aproape instant. Acest aspect permite aplicațiilor care necesită un răspuns rapid să funcționeze fiabil, chiar și în condiții de conectivitate limitată. Un alt beneficiu semnificativ este optimizarea traficului prin rețea. Dacă filtrarea și procesarea se fac local, doar informațiile relevante sunt transmise mai departe, optimizând astfel consumul de resurse [14][16].

Dezavantajul pentru edge computing este faptul că dispozitivele locale nu dispun de aceeași capacitate de calcul precum cea a unui centru de date. Rularea locală a algoritmilor avansați de învățare automată sau a analizei de date masive este limitată de resursele hardware disponibile. Administrarea unei rețele distribuite, cu mai multe noduri de procesare, poate deveni mai costisitoare și complexă [15][16].

Arhitecturile hibride sunt cele care îmbină edge computing cu cloud. Acestea oferă o sinergie între viteza de reacție oferită de edge computing și puterea de analiză și scalabilitate oferite de infrastructura cloud. Această abordare asigură un grad mai mare de reziliență a sistemului. Dacă cloud-ul devine temporar inaccesibil, sistemul continuă să funcționeze parțial pe baza procesării locale. Astfel, pentru sistemele în care este nevoie de o reacție rapidă sau în care se poate face o filtrare inițială, se trimit doar informațiile utile către cloud. Ulterior se pot realiza operații suplimentare de analiză, corelare sau stocare pe termen lung [14][16].

Prin urmare, alegerea tipului de arhitectură dintre edge, cloud sau hibrid poate fi stabilită în funcție de particularitățile fiecărei aplicații IoT. Decizia este influențată de mai mulți factori, incluzând importanța latenței și a disponibilității constante, fezabilitatea procesării locale și nivelul de securitate necesar [16][17].

## 5.2 Microcontrolere utilizate în IoT

### 5.2.1 Microcontrolerul ESP32-S2 – caracteristici tehnice

Modulul ESP32-S2 este un sistem integrat pe cip (System-on-Chip – SoC) destinat aplicațiilor IoT care necesită conectivitate la Wi-Fi și consum redus de energie. Pentru proiectul realizat, a fost utilizată o placă de dezvoltare SparkFun Thing Plus ESP32-S2, care are la bază modulul ESP32-S2 WROOM.

Sistemul funcționează la o frecvență de până la 240 MHz. Procesorul este un Xtensa LX7 pe 32 de biți. Memoria de bază este formată din 128 KB ROM (Read-Only Memory) și 320 KB SRAM (Static Random-Access Memory). Pentru aplicațiile care necesită mai multă capacitate de stocare, se pot adăuga memorii externe prin interfețele SPI (Serial Peripheral Interface)/QSPI (Quad SPI)/OSPI (Octal SPI) pentru stocare flash sau RAM (Random-Access Memory) [18].

Din punct de vedere al conectivității, ESP32-S2 este compatibil cu standardul IEEE 802.11 b/g/n și poate atinge viteze de transmisie de până la 150 Mbps în modul Wi-Fi. Acesta permite funcționarea atât în modul Station, cât și în modul SoftAP (Software-enabled Access Point) [18].

Un avantaj major al acestei platforme este gama extinsă de periferice hardware integrate. Dispune de 43 de GPIO-uri (General Purpose Input/Output), două convertoare DAC (Digital-to-Analog Converter) pe 8 biți, două convertoare ADC (Analog-to-Digital Converter) pe 12 biți și suport pentru comunicație USB OTG (On-The-Go). În plus, are multiple interfețe precum $I^2C$ (Inter-Integrated Circuit), SPI, $I^2S$ (Integrated Inter-IC Sound), UART (Universal Asynchronous Receiver/Transmitter), LED_PWM (Pulse Width Modulation) și DVP (Digital Video Port) [18].

Din perspectiva securității, ESP32-S2 include suport hardware pentru criptare AES (Advanced Encryption Standard), SHA (Secure Hash Algorithm) și RSA (Rivest-Shamir-Adleman), precum și suport pentru semnături digitale, criptare flash și boot securizat. Aceste caracteristici asigură protecția datelor și o securitate ridicată a comunicațiilor în arhitecturile IoT [18].

### 5.2.2 Compararea ESP32-S2 cu alte platforme IoT

Alegerea unui microcontroler potrivit depinde foarte mult de contextul aplicației IoT. Deși nu există o alegere ideală, unele variante au ajuns să fie considerate standard în cazul proiectelor realizate pentru uz personal sau semiprofesionale. Criteriile vizate sunt cele de performanță, consum, compatibilitate extinsă, conectivitate și preț.

Cele mai răspândite plăci de dezvoltare pentru proiectele de IoT sunt Arduino Uno R3, Raspberry Pi Pico W, ESP8266 NodeMCU v3 și STM32F103. Acestea acoperă majoritatea nevoilor utilizatorilor la un preț accesibil. Comparația între plăcile menționate și placa ESP32-S2 este prezentată în Tabelul 1.

Tabelul 1. Comparație între microcontrolere frecvent utilizate în IoT

| Caracteristică | ESP32-S2 [18] | ESP8266 NodeMCU v3 [19] | Arduino Uno R3 [20] | STM32F103C8 [21] | Raspberry Pi Pico W [22] |
|---|---|---|---|---|---|
| Frecvență CPU | Xtensa LX7 @ 240 MHz | Tensilica L106 @ 80/160 MHz | ATmega328P @ 16 MHz | ARM Cortex-M3 @ 72 MHz | RP2040 dual-core @ 133 MHz |
| RAM intern | 320 KB SRAM | 80 KB SRAM | 2 KB SRAM | 20 KB SRAM | 264 KB SRAM |
| Flash intern | 4MB | 4MB | 32 KB | 64 KB Flash | 2 MB |
| Wi-Fi | 802.11 b/g/n | 802.11 b/g/n | Nu | Nu | 802.11n |

| Bluetooth | Nu | Nu | Nu | Nu | Nu |
|---|---|---|---|---|---|
| Interfețe | SPI, I²C, UART, ADC, DAC, USB | SPI, I²C, UART, ADC | SPI, I²C, UART, ADC, PWM | SPI, I²C, UART, ADC, PWM, CAN | SPI, I²C, UART, ADC, PWM |
| Număr GPIO | 43 | 17 | 20 (14D + 6A) | 37 | 26 |
| Sleep modes | Da | Limitat | Nu | Da | Da |
| Securitate hardware | AES, SHA, RSA, HMAC, Secure | WEP/WPA | Nu | Parțial (watchdog, brown-out) | TLS prin software |

ESP32-S2 se remarcă prin procesorul mult mai puternic decât celelalte variante, cu o frecvență de până la 240 MHz și o arhitectură Xtensa LX7. De asemenea, capacitatea memoriei RAM, conectivitatea Wi-Fi și suportul pentru USB OTG fac din acesta o alegere solidă pentru o varietate de aplicații practice în domeniul IoT.

ESP8266 rămâne o alegere bună pentru proiectele cu un buget mic, dar este limitat din punct de vedere al memoriei. Oferă un mod de repaus simplu, dar mai puțin configurabil. De asemenea, nu dispune nici de suport pentru criptare hardware avansată.

STM32 dispune de un procesor performant, de tip ARM, dar nu are conectivitate wireless. Utilizarea lui este mai potrivită pentru aplicații industriale sau care cer control fin al perifericelor. În plus, permite doar moduri standard de consum redus, în funcție de implementare.

Raspberry Pi Pico W ar fi o alternativă solidă pentru ESP32-S2, având un procesor dual-core la 133 MHz și suport Wi-Fi. De asemenea, oferă suport pentru moduri de repaus, utile în aplicații cu resurse de energie limitate, asemănătoare cu cele ale ESP32-S2.

Arduino Uno R3 nu oferă conectivitate wireless, iar specificațiile tehnice sunt modeste. Rămâne totuși una dintre cele mai populare platforme datorită simplității, prețului accesibil și suportului larg din partea comunității. Sunt disponibile foarte multe proiecte open-source, ceea ce o face o alegere bună pentru aplicații simple.

Concluzionând, ESP32-S2 oferă un echilibru foarte bun între performanță, conectivitate, securitate și consum, la un preț rezonabil. Compatibilitatea cu Arduino IDE și suportul pentru librării de MQTT și TLS o face alegerea potrivită pentru proiecte IoT moderne, inclusiv pentru aplicații mobile precum dronele.

## 5.3 Senzori utilizați în IoT

### 5.3.1 Senzorii DHT22 și SGP40

Senzorii relevanți documentației sunt DHT22, utilizat pentru măsurarea temperaturii și a umidității, și SGP40, folosit pentru măsurarea calității aerului.

Senzorul DHT22, cunoscut și sub denumirea de AM2302, este o soluție digitală accesibilă, dar suficient de precisă pentru prototipuri și proiecte. Acesta integrează un senzor capacitiv pentru umiditate și un termistor pentru temperatură. Conversia analog-digitală a datelor este realizată de un cip intern, transmițând apoi datele printr-un semnal digital ușor de interpretat de un microcontroler. Precizia acestui senzor pentru temperatură este de ±0.5°C, iar pentru umiditate între 2% și 5%. Intervalul măsurabil pentru temperatură este de la -40 până la +80°C și pentru umiditate, între 0-100% RH. Un aspect important al acestui senzor este că, pentru a obține citiri corecte, senzorul nu trebuie interogat mai des de o dată la două secunde. De asemenea, varianta modulului cu 3 pini (VCC, DAT și GND) necesită o rezistență de pull-up de aproximativ 10kΩ

pe pinul de date. În cazul modulului cu 4 pini, logica este aceeași, iar pinul 3 rămâne neconectat [23].

Senzorul SGP40 de la Sensirion este un senzor de tip MOx (metal-oxid) specializat în detectarea compușilor organici volatili (VOC – Volatile Organic Compounds) din aerul ambiental. Spre deosebire de senzorii care oferă valori exacte în ppm (părți per milion), SGP40 generează un scor relativ numit „VOC Index", cuprins între 0 și 500. Acest indice reflectă intensitatea poluanților prin comparație cu o referință stabilită pe baza ultimelor 24 de ore. Practic, senzorul „învață" profilul tipic al aerului dintr-o încăpere și semnalează evenimente care diferă semnificativ față de acea referință. Comunicarea cu senzorul SGP40 se realizează prin interfața $I^2C$. Consumul acestui senzor este destul de redus, de doar 2,6 mA la 3,3 V. Dispune, de asemenea, de un algoritm intern de compensare pentru umiditate și temperatură, ceea ce îl face mai fiabil în condiții de mediu diverse. Ceea ce îl recomandă pentru sistemele montate în exterior sau pe drone, unde înlocuirea senzorului nu este facilă, este rezistența la contaminanți și durata de viață extinsă. Spre deosebire de senzorii tradiționali care măsoară doar concentrații brute de gaze, designul senzorului SGP40 este gândit să funcționeze împreună cu un algoritm software, pus la dispoziție de Sensirion, care analizează nu doar prezența, ci și intensitatea, durata și frecvența evenimentelor VOC [24][25].

### 5.3.2 Scalabilitatea în utilizarea senzorilor IoT

Unul dintre atributele principale ale unui sistem IoT este capacitatea de a fi extins. Acest lucru se poate realiza fie prin creșterea complexității funcționale a dispozitivelor deja existente, fie prin adăugarea de noduri sau senzori în rețea. În contextul senzorilor, acest lucru se numește modularitate. Modularitatea reprezintă capacitatea unui sistem de a permite adăugarea, înlocuirea sau eliminarea unor componente hardware, fără să necesite o reconstrucție integrală a arhitecturii. Un sistem care nu este modular este un sistem rigid și dificil de adaptat noilor cerințe. Prin urmare, poate îngreuna evoluția aplicației IoT [26].

Conform literaturii de specialitate, scalabilitatea este recunoscută ca un criteriu esențial de proiectare în rețelele IoT. Aceasta implică flexibilitatea arhitecturii fizice, nu doar a capacității serverului sau a lățimii de bandă. Un sistem scalabil este un sistem capabil să integreze un număr crescut de senzori, fără să necesite modificări majore ale codului sau ale structurii hardware. Interfețe precum $I^2C$ sau SPI permit conectarea mai multor dispozitive pe aceeași magistrală, în timp ce bibliotecile software permit gestionarea paralelă a modulelor cu funcționalități diverse. Astfel, rolul acestor interfețe este fundamental în scalabilitatea sistemelor [27].

Un exemplu concret îl reprezintă sistemele formate din mai multe dispozitive ESP32. Fiecare modul are senzori proprii conectați și transmite datele către o platformă centralizată sau către un dispozitiv master, cum ar fi un Raspberry Pi. Într-un astfel de caz, fiecare nod din rețea funcționează independent, dar contribuie la un sistem de monitorizare interconectat. Un avantaj suplimentar al acestei configurații este că, în cazul în care un nod se defectează, restul rețelei nu este compromisă. Rețeaua este în continuare operațională, ilustrând astfel un nivel înalt de modularitate și capacitate de extindere fără a perturba funcționalitatea celorlalte componente. În [27] se prezintă un proiect conceput pentru monitorizarea calității aerului și a parametrilor meteorologici, bazat pe platforma ESP32. Pentru acest proiect au fost utilizați diverși senzori care măsoară umiditatea, temperatura, presiunea atmosferică, intensitatea luminii și concentrația compușilor gazoși. Toți senzorii au fost conectați prin magistrala $I^2C$. Au funcționat concomitent, transmițând date prin Wi-Fi către un server cloud, fără să genereze interferențe sau conflicte de date. Biblioteca asociată fiecărui senzor asigură comunicarea standardizată cu microcontrolerul și are rol în izolarea implementării [27].

O altă necesitate legată de scalabilitatea utilizării senzorilor este extinderea domeniului de măsură. Aceasta devine esențială pentru aplicațiile în care condițiile de mediu fluctuează în mod frecvent. Un sistem construit inițial doar cu un senzor de temperatură, conceput pentru monitorizare ambientală, poate să fie ulterior extins cu un senzor de calitate a aerului, un senzor

de presiune sau chiar un modul GPS, în funcție de cerințele aplicației. În cazul sistemului modular, integrarea unui senzor nou nu înseamnă reconstruirea întregului ansamblu [26].

Câțiva senzori adesea întâlniți în proiectele IoT sunt: BMP280, utilizat pentru măsurarea presiunii și a altitudinii; MQ135, folosit pentru monitorizarea calității aerului. Pentru determinarea concentrației de dioxid de carbon se utilizează frecvent module precum MH-Z19. TGS2600 sau SGP30 sunt opțiuni viabile pentru monitorizarea generală a compușilor organici volatili (VOC). Toți acești senzori sunt compatibili cu plăci de dezvoltare precum ESP32 și sunt ușor de integrat datorită existenței bibliotecilor bine documentate [27][28].

Ca notă finală, scalabilitatea în contextul senzorilor IoT nu este doar o opțiune tehnică, ci o condiție esențială pentru funcționarea sistemelor pe termen lung. Modularitatea și utilizarea interfețelor de comunicare standardizate transformă un sistem simplu într-o platformă robustă și adaptabilă la extinderi ulterioare.

## 5.4 Protocolul MQTT

### 5.4.1 Introducere și principii – MQTT

MQTT (Message Queuing Telemetry Transport) este un protocol de mesagerie conceput pentru a asigura o comunicare eficientă între dispozitive cu resurse limitate și rețele instabile sau cu lățime de bandă redusă. A fost dezvoltat în 1999 de Arlon Nipper și Andy Stanford-Clark pentru a gestiona transferul de date din sistemele industriale folosind conexiuni prin satelit [29][30].

Acest protocol funcționează pe o arhitectură simplă, după modelul publish-subscribe, în care există două entități: clienții, care pot fi publisheri sau subscriberi, și brokerul, care acționează ca un server. Brokerul primește, filtrează și distribuie mesajele pe anumite topicuri [29].

MQTT este un protocol de nivel aplicație care se bazează în mod obișnuit pe TCP (Transmission Control Protocol)/IP (Internet Protocol). Cu toate că inițial nu a fost un protocol gratuit, acest lucru s-a schimbat din 2010, iar în 2014 a fost declarat drept protocol standard de către OASIS (Organization for the Advancement of Structured Information Standards). Implementarea ușoară, consumul redus de energie și capabilitatea de a trimite rapid mesaje scurte în format optimizat fac ca acest protocol să fie o alegere frecventă în aplicațiile IoT [29][30].

Un alt motiv pentru care MQTT a devenit o soluție viabilă pe scară largă este arhitectura sa foarte eficientă. Folosește mesaje de câțiva octeți, nu necesită neapărat conexiuni persistente și oferă opțiuni pentru controlul calității livrării mesajelor în funcție de nivelul de fiabilitate dorit. În plus, MQTT este o alegere potrivită pentru microcontrolere și module wireless, deoarece nu necesită multă memorie sau putere de procesare [29][30].

### 5.4.2 Cum funcționează protocolul MQTT

Modelul publish/subscribe pe care se bazează protocolul MQTT decuplează complet comunicația dintre emițător și receptor. Nu există trimitere directă de la un nod la altul, ci totul trece printr-un intermediar numit broker. Un client poate să publice date pe un anumit topic, iar alt client, dacă este abonat la acel topic, va primi datele, fără să știe cine l-a trimis. Totul trece prin broker, care face legătura între toți clienții, iar aceștia nu mai sunt obligați să rețină informații suplimentare unii despre alții [30].

Brokerul este elementul central al acestui sistem. Primește mesaje de la clienți, le filtrează, dacă este nevoie, le stochează temporar și le trimite exclusiv celor care sunt abonați la topicul respectiv. Acesta poate gestiona simultan mii de sesiuni și conexiuni, inclusiv autentificări și reconectări automate. Mai mulți brokeri precum Mosquitto sau HiveMQ oferă implementări gratuite sau comerciale [29].

Topicurile sunt asemănătoare unor etichete care se scriu în format text și definesc despre ce e mesajul. Cu cât topicul este mai specific, cu atât este mai particularizat, iar controlul este mai fin. De asemenea, există și posibilitatea de abonare la mai multe subtopicuri o dată, prin folosirea caracterelor „wildcard" cum ar fi „+" sau „#" [29].

Quality of Service (QoS) stabilește cât de sigur trebuie să fie livrat un mesaj. Există trei niveluri pentru QoS:

1. QoS 0 – mesajul este transmis o singură dată, fără confirmare. Este cel mai rapid, dar nu oferă garanții despre livrarea corectă a mesajului.
2. QoS 1 – mesajul este livrat cel puțin o dată. Există confirmare de primire, dar pot să apară duplicate ale mesajului.
3. QoS 2 – mesajul este livrat exact o dată. Se folosește un mecanism de tip handshake în patru pași. Este cel mai sigur, dar și cel mai lent [29][30].

Datorită faptului că publicatorii și abonații nu trebuie să fie activi în același timp, întrucât dispozitivele nu sunt mereu online și nu pot avea sesiuni deschise permanent, iar brokerul ține evidența mesajelor și le livrează când este necesar, tot sistemul devine mai robust și mai scalabil [30].

### 5.4.3 Avantajele protocolului MQTT în IoT

În contextul IoT, protocolul MQTT răspunde problemei sistemelor cu resurse limitate, rezolvând probleme precum lățimea de bandă redusă, conexiunile instabile și nevoia de putere de procesare mare.

Un avantaj imediat este dimensiunea redusă a mesajelor. Studiile experimentale subliniază că limitarea mesajelor la 5 kilobyți contribuie semnificativ la eficiența sistemului, atât în viteza de transmisie, cât și în privința consumului de resurse. Chiar dacă MQTT nu are mecanisme proprii de criptare, poate fi adaptat cu ușurință pe partea de securitate, prin integrarea cu TLS și chiar prin folosirea unor suite criptografice precum AES-GCM (Advanced Encryption Standard – Galois/Counter Mode) sau CHACHA20-POLY1305. Pe baza testelor din sursa [31], deși aceste măsuri introduc o ușoară penalizare de performanță, ele pot totuși să asigure un echilibru între securitate și o latență acceptabilă, inclusiv în scenarii dificile cum ar fi transmisia prin rețele TOR (The Onion Router) [31].

Din punct de vedere al infrastructurii, un broker MQTT are capabilitatea de a gestiona simultan mii de conexiuni, fără probleme majore. Clienții se pot conecta și deconecta în mod dinamic, iar acest comportament este o necesitate în aplicații la scară largă [30].

Implementarea infrastructurilor expuse care folosesc MQTT trebuie să includă filtrare, limitări ale ratelor și autentificare, deoarece există riscul ridicat de atacuri. Protocolul MQTT este vulnerabil în fața formelor diverse de suprasolicitări precum flooding, brute-force sau exploatări ale mecanismelor de stabilire a conexiunii. Cel mai comun este atacul DDoS (Distributed Denial of Service), care se concentrează exclusiv pe epuizarea resurselor brokerului prin trimiterea excesivă a mesajelor sau inițierea simultană a unui număr mare de conexiuni false [30].

### 5.4.4 Protocoale alternative la MQTT și comparații

Cu toate că MQTT este un protocol des utilizat în aplicațiile IoT, există și alte variante care pot fi luate în considerare. Se pot folosi, după caz, și protocoale precum HTTP (HyperText Transfer Protocol), CoAP (Constrained Application Protocol), AMQP (Advanced Message Queuing Protocol) sau protocoale hibride. Fiecare dintre acestea are avantajele și dezavantajele sale. În funcție de aplicație, infrastructură sau constrângeri tehnice, nu există o soluție general valabilă, ci mai degrabă o decizie de compromis [32].

Protocolul HTTP rămâne unul dintre cele mai cunoscute, în special în contextul arhitecturilor REST (Representational State Transfer). Acesta este folosit pe scară largă în aplicațiile web

datorită compatibilității extinse cu browserele și serverele existente. Cu toate acestea, în aplicațiile IoT, protocolul HTTP poate deveni o alegere nepotrivită. Acest lucru se datorează modelului său clasic de tip cerere-răspuns, care necesită existența unor conexiuni stabile și persistente. HTTP demonstrează o eficiență redusă în cazul transmisiilor frecvente de date scurte. În plus, overhead-ul este mai mare comparativ cu alte protocoale mai optimizate [33].

În schimb, CoAP a fost conceput special pentru a opera în contextul rețelelor cu resurse limitate. Acesta se bazează pe protocolul UDP (User Datagram Protocol) și adoptă o arhitectură asemănătoare cu REST (Representational State Transfer), dar mai simplificat. Protocolul CoAP are avantaje semnificative în ceea ce privește viteza în rețea datorită utilizării UDP, dar dezavantajul este lipsa unui mecanism robust de control al erorilor sau de retransmisie. Prin urmare, CoAP este considerat nesigur fără pachete ACK (Acknowledgement) suplimentare. Rămâne totuși o opțiune viabilă pentru dispozitivele cu resurse limitate, dar nu este cea mai potrivită alegere în cazul unei conexiuni instabile. Este folosit cu succes în aplicații M2M sau acolo unde se dorește o latență extrem de redusă [33].

Protocolul AMQP permite comunicarea de tip publish-subscribe și client-server. Integrează mecanisme de calitate a serviciului (QoS) similare cu cele ale protocolului MQTT. AMQP este un protocol deschis, de nivel aplicație și rulează peste TCP. În ceea ce privește performanța protocolului AMQP, acesta generează mesaje de dimensiuni mai mari. În unele cazuri, acest lucru poate influența cantitatea de date transmise în rețea. Cu toate că încearcă să gestioneze eficient transmisia datelor în prezența erorilor, poate fi mai puțin adecvat pentru aplicațiile în care rețeaua este instabilă, înregistrând rate semnificative de pierdere a pachetelor [34].

Caracteristicile principale precum modelul de comunicare, protocolul de transport, dimensiunea pachetelor, suportul pentru QoS, securitatea și scalabilitatea au fost comparate în Tabelul 2.

Tabelul 2. Tabel comparativ între protocoalele IoT (date sintetizate după [32][33][34])

| Protocol | Model de comunicare | Protocol de transport | Overhead | QoS | Securitate | Scalabilitate | Complexitatea implementării |
|---|---|---|---|---|---|---|---|
| MQTT | Publish/Subscribe | TCP | Mic | Da (0, 1, 2) | TLS, autentificare posibilă | Bună | Redusă |
| HTTP | Request/Response | TCP | Mare | Nu | TLS prin HTTPS (nu nativ) | Limitată, conexiuni costisitoare | Scăzută |
| CoAP | Request/Response (poate fi asincron) | UDP | Foarte mic | Limitat | DTLS | Bună pentru rețele mici | Redusă |
| AMQP | Message-oriented (cu broker) | TCP | Mediu-Mare | Confirmări, livrare fiabilă | - | Bună, dar cu cost mare | - |

Conform Tabelului 2, din analiza comparativă a protocoalelor reiese faptul că MQTT oferă un echilibru bun între simplitate, eficiență și scalabilitate. CoAP este un protocol rapid, dar prezintă anumite limitări în ceea ce privește fiabilitatea transmisiilor. HTTP este robust, dar mai puțin eficient pentru mediile cu lățime de bandă limitată. Pe de altă parte, AMQP se distinge prin funcții avansate, dar nu este adecvat pentru implementarea în aplicații încorporate.

Prin urmare, alegerea protocolului potrivit depinde de cerințele specifice ale aplicației și de constrângerile de resurse disponibile. În contextul proiectului de față, protocolul MQTT este cea mai justificată opțiune.

## 5.5 Securitatea comunicațiilor TLS/SSL

TLS (Transport Layer Security) este un protocol de securitate care are rolul de a proteja comunicațiile dintre două sisteme interconectate în rețea. Scopul principal al acestui protocol este de a asigura confidențialitatea și integritatea datelor, precum și autentificarea. Prin termenul de „confidențialitate" se înțelege faptul că datele nu pot fi citite de către alt cineva în afară de destinatar. Prin „integritate" se presupune că datele nu pot fi modificate pe traseu fără ca acest lucru să fie observat, iar prin „autentificare" fiecare parte trebuie să știe cu cine comunică. TLS a apărut ca succesor al protocolului SSL (Secure Sockets Layer). În prezent este standardul despre care se vorbește în contextul securizării comunicațiilor pe internet, fiind prezent de la paginile HTTPS (HyperText Transfer Protocol Secure), la aplicații de e-mail și mesagerie, VPN-uri (Virtual Private Network) și protocoale de comunicații în domeniul IoT [35][36].

Protocolul TLS are la bază două componente principale: handshake-ul și canalul de date. Handshake-ul este o legătură inițială între client și server. În această etapă, cele două părți stabilesc modul în care vor comunica mai departe. Se stabilește ce algoritmi de criptare vor fi folosiți, se verifică identitățile părților și se generează cheile secrete în baza cărora acestea două vor comunica protejat. Ulterior acestui pas va începe transmiterea propriu-zisă a datelor, în care informațiile sunt criptate și semnate [36][37].

Toate aceste decizii de securitate cu privire la ce metode se folosesc pentru schimbul de chei, cum se realizează criptarea și cum se verifică integritatea sunt decise prin cipher suite. Cipher suite reprezintă o combinație de algoritmi care se ocupă de toate componentele importante din TLS cum ar fi algoritmul de schimb de chei (ECDHE, RSA), algoritmul de criptare (AES, CHACHA20), semnătura digitală (RSA, ECDSA) și funcția hash folosită pentru a face verificările (SHA256, SHA384) [37]. Numele complet al unui cipher suite conține informații exacte despre operațiile care vor fi realizate.

O altă variantă existentă este TLS cu PSK (Pre-Shared Key), care este de obicei folosită în cazurile în care nu sunt suportate certificatele digitale sau pentru aplicații restrânse. În acest caz, cheia dintre client și server este deja cunoscută. Este o metodă mai eficientă din punct de vedere al resurselor, dar și riscurile sunt mai mari, întrucât cheia trebuie să rămână permanent protejată și să nu fie expusă sub nicio formă [36][38].

Folosirea certificatelor digitale și a unei infrastructuri de încredere, cunoscută sub numele de PKI (Public Key Infrastructure) este, de asemenea, o parte esențială a protocolului TLS. Certificatele sunt emise de autorități de certificare (CA), care au rolul de a verifica identitatea emițătorului, atribuindu-le o anumită cheie publică. Aceste certificate sunt folosite ulterior în handshake pentru a confirma clientului că acesta comunică cu serverul legitim. Această infrastructură funcționează bine la scară mare, dar are și vulnerabilități. În cazul în care o autoritate de certificare este compromisă, atacurile pot fi realizate cu succes, chiar dacă TLS este implementat corect [35].

Un alt avantaj al TLS este gama mare de opțiuni de care dispune, în funcție de nevoile aplicației sau de capabilitatea elementelor hardware disponibile. Există variante de criptare mai complexe, care sunt foarte sigure, și variante mai simple, destinate dispozitivelor cu resurse limitate. Spre exemplu, algoritmii care folosesc criptografie pe curbe eliptice (ECC) oferă un nivel bun de securitate, cu calcule mai rapide și chei mai scurte, fiind o alegere bună în cazul dispozitivelor IoT [37].

În contextul protocoalelor de comunicație precum MQTT, TLS este folosit ca un strat suplimentar de protecție pentru a securiza toate datele care circulă între client și broker sau între broker și subscriberi. Deși este o soluție solidă pentru protejarea comunicațiilor pe sisteme cu

putere mare de procesare, precum PC-uri, servere sau gateway-uri, această soluție poate deveni o problemă pe microcontrolere sau senzori, din cauza consumului de memorie și procesare. În unele cazuri, implementarea TLS trebuie să fie optimizată prin alegerea unui cipher suite mai simplu sau prin criptarea datelor înainte de transmitere [38][39].

## 5.6 Django

### 5.6.1 Arhitectura aplicațiilor web cu Django

Django este un framework web scris în Python, folosit des în proiectele în care sunt necesare baze de date, interfețe dinamice și logici de backend mai complexe. Acest framework este cunoscut pentru gama mare de funcționalități pe care le oferă și care sunt deja pregătite sau parțial implementate. Dezvoltatorul nu trebuie să construiască totul de la zero. Django înglobează un sistem de autentificare, gestionare a sesiunilor, conectare la bazele de date, un dashboard pentru administrare și multe alte facilități [40][41][42].

Una dintre ideile promovate de Django este „Don't Repeat Yourself (DRY)" [40]. Practic încearcă să reducă cât mai mult scrierea a aceluiași cod în locuri diferite. O regulă sau o structură de date care a fost deja definită o dată, poate fi folosită în mai multe părți, fără a mai fi necesară rescrierea ei. Acest principiu „DRY" este util pe măsură ce aplicația devine tot mai complexă. Ajută la menținerea unui cod mai organizat și mai ușor de înțeles de către alți colaboratori care vor lucra ulterior în proiect [40][41].

Django este folosit des în aplicații web, dar mai ales în cadrul proiectelor în care este nevoie de interacțiuni constante cu o bază de date sau în care datele sunt gestionate dinamic, precum platformele web, interfețele administrative, dashboard-uri pentru IoT sau alte sisteme în care conținutul se schimbă frecvent [40][42].

### 5.6.2 Arhitectura Model-View-Template (MVT)

Modelul de organizare a codului pe care îl urmează Django se numește Model-View-Template (MVT). Este varianta adaptată a arhitecturii MVC (Model-View-Controller). Structura acestor arhitecturi este foarte asemănătoare. Cea specifică pentru Django este organizată astfel:

1. Model: gestionează partea de date. Pe acest strat se definesc structurile care corespund tabelelor din baza de date, regulile de validare și relațiile dintre entități.
2. View: este stratul în care se scrie logica aplicației. Primește cereri de la utilizator, cum ar fi accesarea unei pagini web sau trimiterea unui formular, le procesează și apoi decide ce răspuns va trimite. View este echivalentul lui Controller din arhitectura MVC.
3. Template: este partea care se ocupă de prezentarea datelor. Template-ul este un fișier HTML care primește date din View și le afișează într-un format vizibil pentru utilizator. Este echivalentul lui View din arhitectura MVC clasică [41][42][43].

Această separare pe straturi, ilustrată în Figura 1, ajută ca aplicațiile Django să fie mai ușor de întreținut, de testat și de extins. Dacă se dorește modificarea interfeței unei aplicații, doar template-ul trebuie să fie editat. Logica celorlalte straturi rămâne neschimbată.

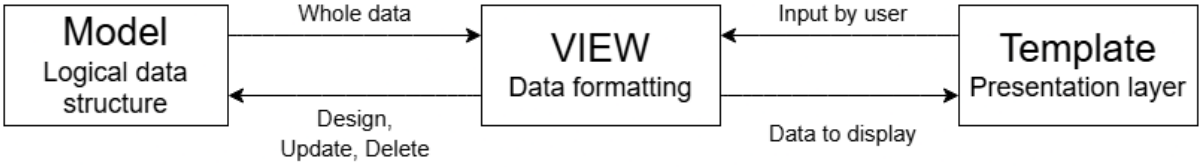



Figura 1. Arhitectura MVT în Django (adaptare după [44])

### 5.6.3 Crearea unui REST API în Django

REST API (Application Programming Interface) este un mecanism prin care clientul trimite cereri către server și primește înapoi date. Datele sunt de regulă în format JSON (JavaScript Object Notation). Pentru Django, se definesc view-uri care răspund cererilor HTTP de tip GET, POST, PUT, DELETE și returnează răspunsuri de tip JSON, folosind clase precum „JsonResponse". View-urile pot verifica tipul cererii, iar în cazul în care cererea este validă, le pot serializa și trimite clientului sub forma unei liste de dicționare [45].

Frontend-ul interacționează cu astfel de endpoint-uri folosind funcția nativă JavaScript numită „fetch()" sau metode AJAX (Asynchronous JavaScript and XML) din jQuery. Fluxul de cereri este reprezentat în Figura 2. Django poate să răspundă acestor cereri, fără să încarce toată pagina, fiind o funcționalitate esențială pentru aplicațiile care consumă date dinamic, inclusiv aplicații bazate pe frameworkuri JavaScript precum React sau Vue [40][45].


Figura 2. Fluxul cererii între client, AJAX și Django view (adaptare după [46])

### 5.6.4 Utilitate Django

Frameworkul Django este potrivit pentru aplicații web care necesită o structură solidă și logică, separată de interfață, și o modalitate simplă de interacțiune cu datele. Django este utilizat în diverse domenii, inclusiv în domeniul IoT, unde este necesară stocarea în baza de date a datelor de la senzori, actualizarea rapidă a conținutului, iar interfețele trebuie să fie ușor de folosit. Marele avantaj al acestui framework este că oferă toate aceste lucruri din start, fără să fie necesară instalarea multiplelor pachete externe sau repetarea codului în mai multe părți ale proiectului [40][42][45].

## 5.7 Arhitectura logică a unui sistem IoT – model general

Un sistem IoT nu presupune doar alegerea componentelor potrivite, ci și o înțelegere a interconectării acestora. Spre deosebire de conectarea fizică, arhitectura logică se axează pe schimburile de informație dintre componente și pe rolul funcțional al fiecăreia. Este o reprezentare conceptuală, care facilitează perspectiva de ansamblu înainte de a trece la implementarea tehnică. Unii autori definesc această arhitectură drept un model funcțional care integrează senzori, protocoale de comunicație, mecanisme de procesare și interfețe finale. Toate acestea sunt organizate pe straturi sau module [47][48].

Tipurile de arhitecturi IoT variază de la modele complexe la abordări mai abstracte, cu mai multe straturi și funcții auxiliare. Cu toate acestea, majoritatea arhitecturilor includ o sursă de date (senzori), un nod care citește datele (microcontroler), o rețea pentru transportul informației, o componentă de procesare (server, cloud sau edge computing) și o aplicație dedicată afișării sau utilizării datelor [47].

Modelul propus în cadrul acestei lucrări abordează o structură similară celor descrise anterior și este ilustrat în Figura 3. Modulul ESP32-S2 preia datele de la senzori, iar apoi le trimite prin protocolul MQTT, utilizând o conexiune Wi-Fi, către un broker. Ulterior, mesajele sunt trimise mai departe către un script Python, care acționează ca o punte de legătură. Scriptul stochează informațiile într-o bază de date gestionată de un backend Django. Vizualizarea datelor se

realizează prin intermediul unei interfețe web, accesibilă într-un browser. Această arhitectură simplificată facilitează testarea și replicarea sistemului în diverse scenarii. Sistemul nu folosește tehnologii costisitoare sau dependențe externe.

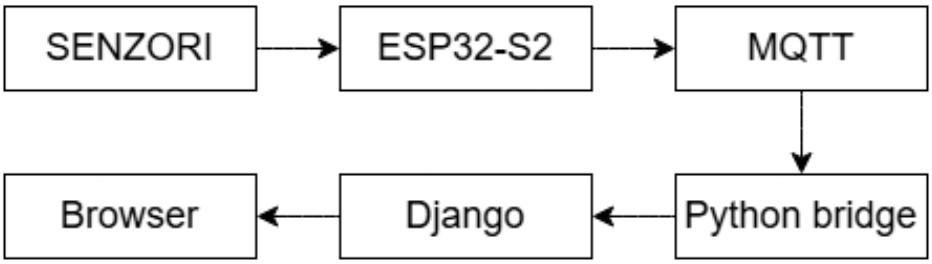



Figura 3. Arhitectura logică a sistemului IoT propus

Modelul prezentat are o separare clară între etapele de achiziție a datelor, comunicare, procesare și interfață. Aceste elemente contribuie la robustețea și claritatea sistemică. Folosirea unui protocol asincron precum MQTT, alături de o arhitectură de tip REST bazată pe Django, permite extinderea sistemului fără să afecteze componentele deja funcționale.

*Instrumentele de inteligență artificială au fost utilizate doar pentru corectarea gramaticală și stilizarea formală a textului. Conținutul lucrării nu a fost redactat cu ajutorul acestora. Intervențiile au vizat claritatea, lizibilitatea și coerența exprimării, fără a modifica fondul ideilor.*

# 6 Implementarea soluției adoptate

## 6.1 Arhitectura generală a sistemului implementat

Arhitectura propusă în cadrul acestei lucrări de licență facilitează un sistem robust de monitorizare a parametrilor de mediu. Sistemul folosește o dronă echipată cu un modul ESP32-S2 care este responsabil de colectarea datelor de la doi senzori. Pentru măsurarea temperaturii și a umidității a fost integrat senzorul DHT22, iar pentru măsurarea concentrației compușilor organici volatili din aer (indicele VOC) este utilizat senzorul SGP-40. Transmisia datelor se realizează securizat, prin intermediul protocolului MQTT, către un server local. Ulterior, datele sunt procesate și vizualizate în timp real printr-o interfață web dedicată. Arhitectura este una modulară, scalabilă și orientată spre securitatea conexiunilor.

Din punct de vedere hardware, sistemul este alcătuit din următoarele componente:

- ESP32-S2 Thing Plus (SparkFun);
- DHT22;
- SGP40;
- Breadboard și cabluri de conexiune;
- Acumulator Li-Polymer 3.7V cu conector JST.

Din perspectiva părții software, sistemul cuprinde:

- Codul firmware (main.ino);
- Broker MQTT (Mosquitto);
- Script Python bridge (mqtt_to_django.py);
- Backend Django;
- Interfață web (sensor_dashboard.html).

Fluxul de date între componente este evidențiat prin diagrama bloc a arhitecturii logice a sistemului ilustrată în Figura 4.

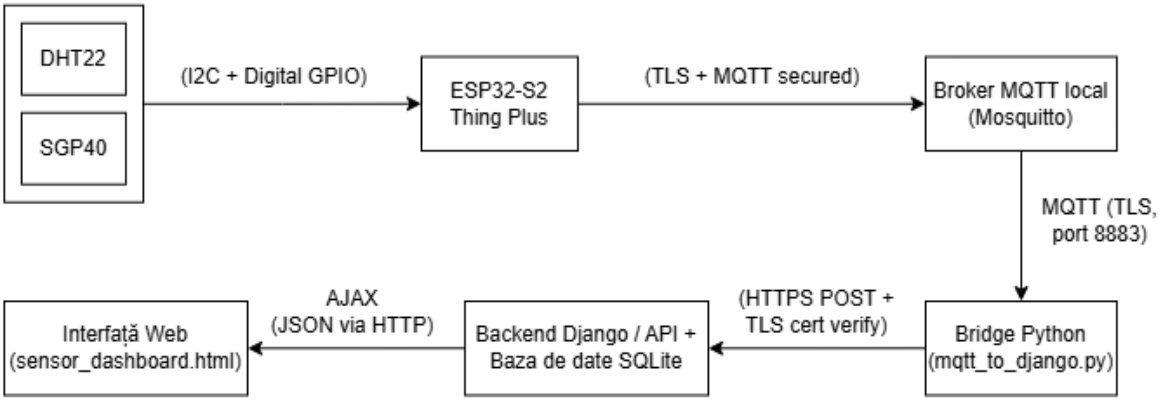



Figura 4. Arhitectura generală a sistemului de monitorizare IoT propus

Prin intermediul acestei diagrame se reflectă în mod clar separarea între straturile de achiziție, comunicație, procesare și prezentare.

Achiziția datelor se realizează prin intermediul senzorilor DHT22 și SGP40, care sunt conectați direct la placa ESP32-S2. Microcontrolerul interoghează periodic senzorii și construiește un mesaj de tip JSON pe baza valorilor obținute. Acest mesaj este ulterior transmis prin protocolul MQTT, folosind o conexiune TLS securizată, către un broker Mosquitto MQTT implementat local.

La nivelul serverului, un script Python se abonează la topicurile MQTT respective și operează în mod continuu. Rolul său este de a prelua mesajele primite și de a le transforma în obiecte compatibile cu structura bazei de date Django. Ulterior, datele procesate sunt stocate într-

o bază de date SQLite (Structured Query Language Lite). Acestea sunt preluate la fiecare cinci secunde prin solicitări AJAX ale interfeței web pentru a le afișa utilizatorului. Fluxul de date este asincron și permite monitorizarea în timp real, fără a necesita reîncărcarea paginii web sau intervenții manuale.

Această arhitectură oferă multiple avantaje. Scalabilitatea sistemului permite integrarea facilă de senzori sau noduri suplimentare fără modificări majore. Securitatea datelor este asigurată prin integrarea protocolului TLS în comunicația MQTT. De asemenea, costurile sunt reduse datorită utilizării de componente open-source și hardware accesibil. În plus, sistemul este portabil și ușor de replicat.

## 6.2 Montajul fizic și conexiunile componentelor

Montajul fizic al sistemului a fost realizat cu ajutorul unui breadboard, a plăcii de dezvoltare ESP32-S2 Thing Plus și a celor doi senzori utilizați pentru achiziția parametrilor de mediu: DHT22 și SGP40.

Toate componentele au fost montate pe breadboard cu ajutorul firelor jumper pentru a facilita testarea și modificările rapide, evitând astfel necesitatea lipiturilor.

Senzorul de temperatură și umiditate DHT22 necesită o linie de date digitală, alimentare și masă. Pinii au fost conectați astfel:

- VCC – pinul 3V3 al ESP32-S2
- DAT – pinul GPIO13
- GND – GND

Semnalul DAT al senzorului este de tip open-drain, prin urmare senzorul nu poate aplica singur un nivel logic „1” pe linia de date. Acest tip de ieșire limitează transmiterea la tragerea liniei la masă (0 V) pentru a reprezenta un bit logic „0”, menținând o stare de impedanță ridicată în celelalte situații. Pentru ca linia să revină în mod natural la nivelul logic „1”, este necesară o rezistență de tip pull-up conectată între pinii DAT și VCC.

În montaj a fost utilizată o rezistență de 10 kΩ, fapt ilustrat în Figura 5. Această rezistență este una standard în aplicațiile DHT22 și stabilizează semnalul de ieșire, prevenind citirile incorecte. Ea asigură faptul că, în absența unei comenzi de la senzor, tensiunea de pe linia de date se stabilizează la 3.3 V. În lipsa acestei rezistențe, linia DAT ar rămâne flotantă, ceea ce ar conduce la semnale instabile și erori de comunicare.

În schimb, senzorul de compuși organici volatili SGP40 comunică cu ESP32-S2 prin intermediul magistralei $I^2C$. Pinii au fost conectați astfel:

- 3V3 – pinul 3V3 al ESP32-S2
- GND – GND
- SDA – SDA
- SCL – SCL

În Figura 6 este reprezentat ansamblul complet al sistemului fizic, cu toate componentele montate pe breadboard. Acest montaj a fost folosit pentru proiectarea, dezvoltarea și testarea aplicației de monitorizare a temperaturii, umidității și concentrației compușilor volatili din mediul ambiant.

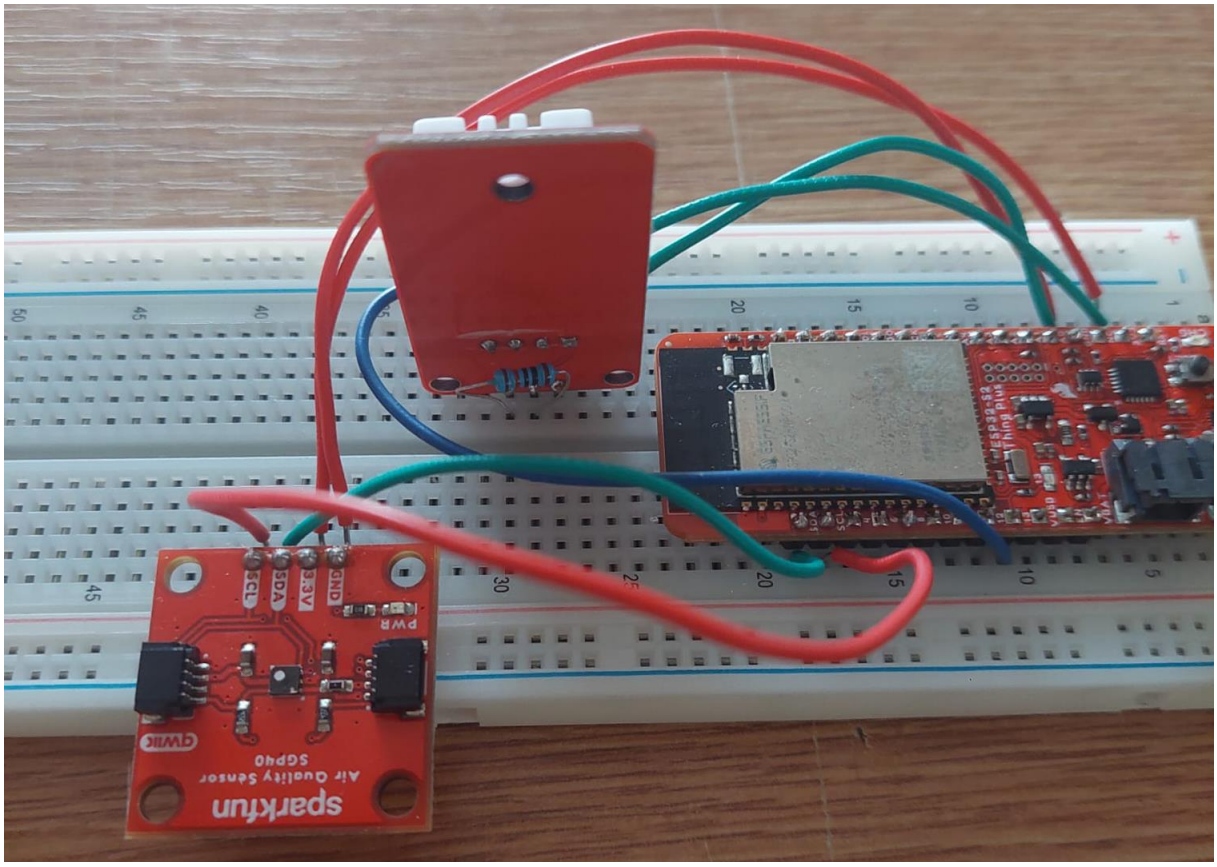



Figura 5. Conexiunea rezistenței de 10 kΩ

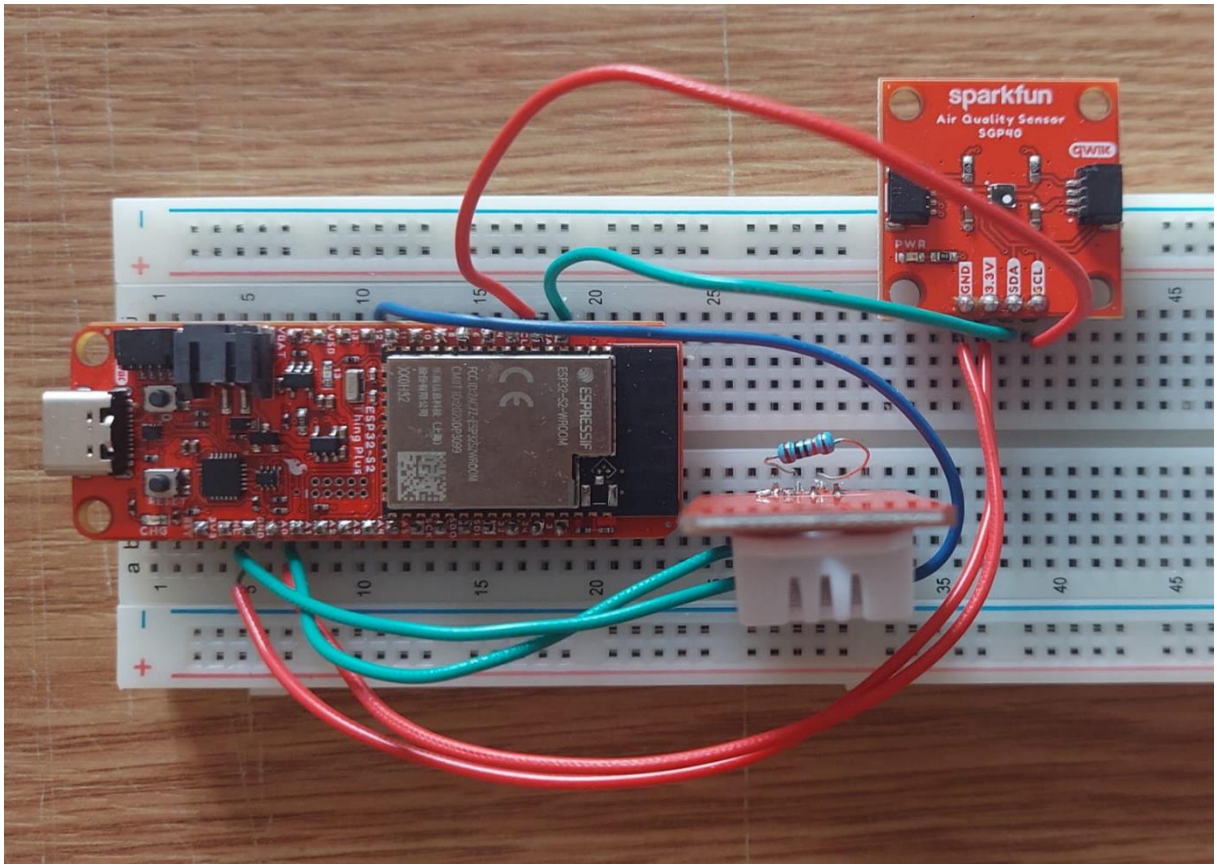



Figura 6. Montajul pe breadboard al sistemului

## 6.3 Citirea și prelucrarea datelor cu ESP32-S2

Placa de dezvoltare ESP32-S2 are rolul central în achiziția, prelucrarea și transmiterea securizată a datelor către brokerul MQTT Mosquitto. Firmware-ul a fost dezvoltat în Arduino IDE. Acesta cuprinde inițializarea conexiunii Wi-Fi cu autentificare TLS. Senzorii DHT22 și SGP40 sunt interogați periodic. Valorile obținute sunt publicate pe trei topicuri MQTT distincte.

Conexiunea la rețeaua Wi-Fi este inițializată odată cu alimentarea modulului ESP32-S2. Ulterior, clientul „WiFiClientSecure" este configurat pentru a utiliza un certificat TLS stocat local în fișierul „secrets.h". Codul de inițializare, prezentat în Figura 7, tratează, de asemenea, potențialele erori de conectare care pot apărea în intervalul de timp alocat.

```
// --- Initializare si gestionare conexiune WiFi ---
void connectWiFi() {
  Serial.print("Connecting to WiFi..");
  WiFi.begin(ssid, password);
  unsigned long startAttemptTime = millis();
  // limitare timp de conectare WiFi la 20 sec
  while (WiFi.status() != WL_CONNECTED && millis() - startAttemptTime < 20000) {
    Serial.print(".");
    delay(500);
  }
  if (WiFi.status() == WL_CONNECTED) {
    Serial.println(" Connected.");
    Serial.print("ESP32-S2 IP address: "); // adresa IP a ESP32
   Serial.println(WiFi.localIP());
  } else {
    Serial.println(" Connection failed!");
  }
}
```

Figura 7. Funcție pentru inițializarea conexiunii Wi-Fi

După stabilirea conexiunii, este setat certificatul brokerului pentru autentificarea acesteia în cadrul conexiunii MQTT. Acest proces este ilustrat în Figura 8.

```
  // certificat conexiune TSL
  wifiClientSecure.setCACert(certificate);
  // adresa si portul serverului MQTT
  mqttClient.setServer(mqtt_server, mqtt_port);
```

Figura 8. Configurarea conexiunii TLS și a brokerului MQTT

Pentru a comunica cu brokerul Mosquitto, ESP32-S2 utilizează un client MQTT cu suport TLS. Conform codului din Figura 9, conexiunea este inițiată utilizând un identificator unic, generat pe baza adresei MAC a dispozitivului. Astfel se asigură unicitatea sesiunii. Implementarea software monitorizează starea conexiunii și gestionează automat reconectarea în cazul unei întreruperi.

```
// --- Conectare ESP32 la brokerul MQTT ---
bool connectMQTT() {
  if (mqttClient.connected()) return true;
  Serial.print("Connecting to MQTT broker with TLS... ");
  // creare clientId unic folosind adresa MAC a ESP32
  String clientId = "ESP32Client-" + WiFi.macAddress();
  // se incearca conectarea la broker folosind clientId
  if (mqttClient.connect(clientId.c_str())) {
    Serial.println("Connected.");
    return true;
  } else {
    Serial.print("Connection failed!, rc=");
    Serial.println(mqttClient.state());
    return false;
  }
}
```

Figura 9. Funcție pentru conectarea la brokerul MQTT folosind TLS

```
// --- Citire date de la senzori ---
void readSensors() {
  float t = dht.readTemperature();
  float h = dht.readHumidity();
  if (!isnan(t) && !isnan(h)) {
    temperature = t;
    humidity = h;
    Serial.printf("DHT22 - Temp: %.2f °C, Hum: %.2f %%\n", temperature,
humidity);
  } else {
    Serial.println("Failed to read DHT22.");
    temperature = NAN;
    humidity = NAN;
  }

  if (sgp40Initialized) {
    int voc = sgp40.getVOCindex();
    if (voc != -100) {
      vocIndex = voc;
      Serial.printf("SGP40 - VOC Index: %d\n", vocIndex);
    } else {
      Serial.println("Failed to read SGP40 VOC.");
      sgp40Initialized = false; // se incearca reinitializarea
      vocIndex = -100;
    }
  }
}
```

Figura 10. Funcție pentru citirea datelor de la senzori

Funcția „readSensors()”, ilustrată în Figura 10, este responsabilă de achiziția valorilor de temperatură și umiditate de la senzorul DHT22, precum și a valorii indicelui VOC de la senzorul SGP40. În cadrul acestei funcții se efectuează verificări pentru a asigura validitatea datelor citite, iar în cazul unei erori, valorile sunt marcate ca invalide, prevenind astfel trimiterea de informații eronate.

Această funcție este apelată la fiecare cinci secunde din bucla principală pentru a respecta frecvența recomandată de citire a senzorilor.

După validarea datelor, acestea sunt publicate individual pe trei topicuri MQTT distincte: „cosmin/temperature”, „cosmin/humidity” și „cosmin/air_quality”. Înainte de a fi transmise, valorile numerice sunt convertite în șiruri de caractere și expediate utilizând funcția „publishSensorData()”. Acest proces este ilustrat în Figura 11.

```
// --- Publicare date senzori pe MQTT ---
void publishSensorData() {
  char buf[16]; // 15 caractere + terminatorul \0
  bool success = true;

  if (!isnan(temperature)) {
    snprintf(buf, sizeof(buf), "%.2f", temperature);
    if (!mqttClient.publish(temperature_topic, buf)) {
      Serial.println("Failed to publish temperature");
      success = false;
    }
  }

  if (!isnan(humidity)) {
    snprintf(buf, sizeof(buf), "%.2f", humidity);
    if (!mqttClient.publish(humidity_topic, buf)) {
      Serial.println("Failed to publish humidity");
      success = false;
    }
  }

  if (vocIndex != -100) {
    snprintf(buf, sizeof(buf), "%d", vocIndex);
    if (!mqttClient.publish(air_quality_topic, buf)) {
      Serial.println("Failed to publish air quality");
      success = false;
    }
  }
}
```

Figura 11. Funcție pentru publicarea datelor pe topicurile MQTT

Pentru a spori stabilitatea operațională a sistemului, a fost implementat un watchdog software. Acesta monitorizează continuu execuția aplicației și inițiază o resetare a sistemului în cazul unei blocări. De asemenea, funcția „loop()” include o secvență de reconectare automată pentru Wi-Fi și MQTT, precum și o rutină de reinițializare a senzorului SGP40 în situațiile de pierdere a comunicației.

Această abordare permite modulului ESP32-S2 să opereze în mod autonom și sigur, chiar și în condiții de rețea instabilă. Datele de mediu continuă să fie transmise în mod constant către sistemul central.

## 6.4 Configurare broker MQTT (Mosquitto) cu TLS

Securitatea comunicației dintre microcontrolerul ESP32-S2 și serverul local este asigurată prin intermediul unei conexiuni criptate folosind protocolul TLS, în combinație cu MQTT. Această măsură este esențială în arhitecturile IoT pentru a proteja datele transmise de la

dispozitivele de tip edge către infrastructura centrală, împotriva tentativelor de interceptare sau modificare neautorizată, precum un atac „man-in-the-middle”.

Brokerul MQTT utilizat în cadrul acestei implementări este Eclipse Mosquitto, instalat pe un sistem local Windows. Configurarea acestuia permite operarea pe două porturi distincte: portul 1883 pentru conexiuni nesecurizate, utilizat doar pentru depanare și testare, și portul 8883 pentru conexiuni criptate prin TLS. Configurația a fost realizată în fișierul „mosquitto.conf”, iar codul este ilustrat în Figura 12.

```
listener 8883
certfile C:/mqtt/certs/mosquitto.crt
keyfile C:/mqtt/certs/mosquitto.key
cafile C:/mqtt/certs/mosquitto.crt
require_certificate false
allow_anonymous true
```

Figura 12. Activarea conexiunii TLS pe portul 8883

Configurația indică activarea autentificării TLS prin utilizarea unui certificat auto-semnat (mosquitto.crt) și a unei chei private (mosquitto.key). Setarea parametrului „require_certificate false” permite clientului ESP32-S2 să se conecteze fără a prezenta un certificat propriu, bazându-se exclusiv pe verificarea identității serverului.

Certificatul auto-semnat a fost generat cu ajutorul utilitarului OpenSSL, în baza unui fișier de configurare personalizat denumit „mosquitto_openssl.cnf”. Acest fișier include atât identitatea serverului, cât și extensia „subjectAltName” necesară pentru validarea adresei IP. Comanda specifică de generare a certificatului a fost cea prezentată în Figura 13. Certificatul a fost utilizat atât pentru autentificarea serverului (certfile), cât și în calitate de autoritate de certificare (cafile).

```
openssl req -x509 -nodes -days 730 -newkey rsa:2048 -keyout mosquitto.key -out
mosquitto.crt -config mosquitto_openssl.cnf
```

Figura 13. Comandă OpenSSL pentru generarea certificatului și a cheii private pentru brokerul MQTT

Pentru a permite modulului ESP32-S2 să valideze conexiunea TLS, certificatul „mosquitto.crt” a fost convertit în format PEM (Privacy-Enhanced Mail) și integrat într-un fișier de configurare separat, numit „secrets.h”. Acest fișier este inclus în proiectul Arduino și este folosit pentru inițializarea clientului Wi-Fi securizat, menționat în Figura 14. Certificatul este definit în cod ca un șir de caractere multi-linie și este ilustrat în Figura 15. Această abordare permite verificarea autenticității serverului MQTT la stabilirea conexiunii, fără a necesita un certificat client.

```
wifiClientSecure.setCACert(certificate);
```

Figura 14. Asocierea certificatului CA pentru conexiunea TLS pe ESP32

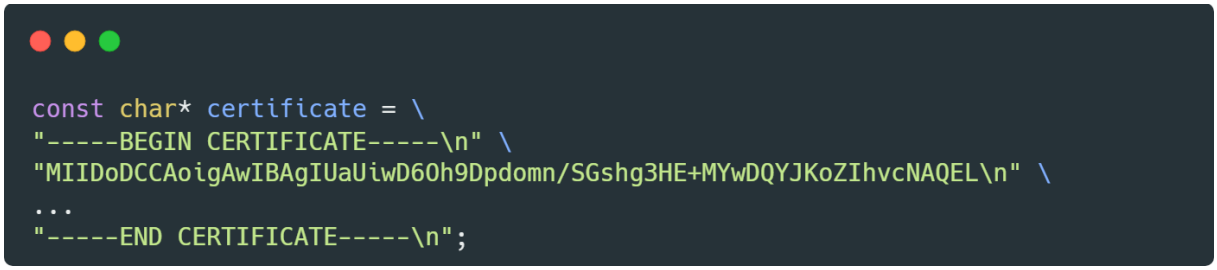



Figura 15. Integrarea certificatului TLS în ESP32-S2 pentru validarea serverului Mosquitto

Pentru a verifica funcționarea corectă a conexiunii TLS, a fost utilizat clientul MQTT de test „mosquitto_sub". Acesta a fost configurat folosind parametrii „--cafile", „--port" și „-t" pentru a se abona, prin conexiune securizată, la toate topicurile disponibile. În plus, modulul ESP32-S2 a fost testat în scenarii cu certificat invalid sau lipsă. În aceste cazuri, conexiunea a fost refuzată, confirmând astfel funcționalitatea mecanismelor de validare implementate.

Pe parcursul dezvoltării, au fost identificate și soluționate diverse probleme. În unele cazuri, ESP32 refuza conexiunea din cauza absenței câmpului „subjectAltName" în certificat. Această problemă a fost remediată prin includerea explicită a adresei IP în fișierul „mosquitto_openssl.cnf". De asemenea, a fost necesară convertirea corectă a certificatului (.crt) în format PEM compatibil cu ESP32-S2, cu terminatori de linie (\n) și caracter de terminare a șirului.

Sumarizând, configurarea brokerului MQTT cu TLS adaugă un nivel esențial de securitate sistemului IoT propus. Transmiterea datelor folosind un canal criptat protejează împotriva interceptării sau modificării conținutului și demonstrează aplicarea bunelor practici în proiectarea sistemelor conectate.

## 6.5 Script Python bridge

În cadrul arhitecturii propuse, interconectivitatea dintre nivelul de comunicații MQTT și backend-ul Django, care operează pe principii RESTful, este realizată printr-un script Python de tip bridge. Scriptul funcționează ca un client MQTT persistent, care ascultă topicurile publicate de placa ESP32-S2. Acesta preia datele și le transmite securizat către serverul Django, utilizând un endpoint API dedicat. Scriptul optimizează fluxul de date prin decuplarea componentei embedded de serverul web și permite extinderea ușoară a sistemului.

Pentru a stabili o sesiune cu brokerul Mosquitto local, scriptul utilizează biblioteca „paho-mqtt". Conexiunea se realizează pe portul 8883 și este securizată folosind TLS. Pentru validarea autenticității, este utilizat același certificat public (mosquitto.crt) ca cel folosit pentru modulul ESP32-S2. Acest fișier de certificat este specificat prin intermediul parametrului „ca_certs" în cadrul apelului funcției „client.tls_set()", ilustrat în Figura 16.


Figura 16. Configurarea clientului MQTT cu TLS în scriptul Python bridge

Odată ce clientul MQTT a stabilit conexiunea, acesta se abonează la cele trei topicuri relevante pentru datele senzorilor: „cosmin/temperature”, „cosmin/humidity” și „cosmin/air_quality”. Acest proces este ilustrat în Figura 17. La recepția fiecărui mesaj, callback-ul „on_message()” este apelat automat și stochează datele temporar într-un buffer dedicat.

```
# functie principala pornire client MQTT si ascultare permanenta a topicurilor
def run():
    client = connect_mqtt()
    client.subscribe([(topic_temp, 0), (topic_hum, 0), (topic_voc, 0)])
    client.on_message = on_message
    client.loop_forever() # asculta permanent
```

Figura 17. Abonarea la topicurile MQTT și ascultarea continuă a mesajelor

În cadrul funcției „on_message()”, datele primite sunt decodificate și introduse în buffer. Atunci când toate cele trei valori necesare sunt disponibile în bufferul respectiv, acestea sunt transmise către backend-ul Django printr-o solicitare HTTP POST, conform procesului ilustrat în Figura 18.

```
# daca toate cele 3 valori sunt complete, se trimite pachetul catre Django
if all (k in buffer for k in ('temperature', 'humidity', 'voc_index')):
  post_to_django(buffer['temperature'], buffer['humidity'], buffer['voc_index'])
  buffer.clear() # golire buffer
```

Figura 18. Funcție pentru prelucrarea mesajelor MQTT și transmitere către Django

Bridge-ul comunică cu backend-ul Django printr-un endpoint RESTful, disponibil la adresa „https://127.0.0.1:8000/api/data/”. În cadrul funcției „post_to_django()”, valorile colectate sunt convertite într-un payload JSON și transmise utilizând metoda „requests.post()”. Conexiunea este securizată prin utilizarea certificatului auto-semnat al serverului Django (server.crt). Certificatul este specificat explicit pentru validare. Această metodă, ilustrată în Figura 19, asigură protecția canalului de comunicații dintre bridge și serverul Django, validând identitatea serverului.

```
# functie care trimite datele catre API-ul Django prin POST
def post_to_django(temp, hum, voc):
    payload = {
        'temperature': temp,
        'humidity': hum,
        'voc_index': voc
    }
    try:
        # certificatul self-signed 'server.crt' folosit pentru validarea
identitatii serverului Django in timpul conexiunii TSL
        r = requests.post(url, json=payload,
verify=r'D:\Licenta\licenta_django\certificate\server.crt')
        print(f"POST to Django: {payload}, response: {r.status_code} {r.text}")
    except Exception as e:
        print(f"Error POST to Django: {e}")
```

Figura 19. Trimiterea datelor către endpoint-ul RESTful Django cu validarea certificatului serverului

În backend, cererea de tip POST este recepționată de clasa „SensorDataView”, definită în fișierul „views.py”. Fluxul acestui proces este prezentat în Figura 20. Această clasă preia payload-ul JSON și îl validează utilizând „SensorDataSerializer”. Odată validat, datele sunt salvate în baza de date SQLite.

```
    # primeste datele in format JSON, le valideaza si le salveaza in baza de
date
    def post(self, request):
        serializer = SensorDataSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Figura 20. Prelucrarea datelor primite prin API în backend-ul Django

În concluzie, scriptul „mqtt_to_django.py” permite integrarea eficientă a comunicațiilor IoT cu o aplicație web robustă. Integrarea TLS atât în cadrul MQTT, cât și în conexiunea HTTPS către API, asigură un nivel înalt de securitate pentru fluxul complet de date. Printre beneficiile arhitecturale aduse de acest script se numără:

- separarea responsabilităților între subsistemele embedded și cele web;
- stocarea locală a datelor în caz de deconectare temporară a serverului Django;
- facilitarea semnificativă a procesului de depanare prin existența unui punct de control logic între fluxul de date MQTT și persistența acestora în baza de date.

## 6.6 Backend Django

După transmisia datelor de la senzori prin scriptul bridge, acestea sunt direcționate către un backend Django. Rolul său este de a valida, stoca și expune datele printr-o interfață RESTful API. Tot acest sistem server-side funcționează ca o interfață intermediară între datele brute și aplicația web responsabilă cu vizualizarea acestora. Structura este simplă, dar suficient de flexibilă, permițând extensii și scalări viitoare.

Modelul de date „SensorData”, ilustrat în Figura 21, definește structura datelor colectate. Acesta conține trei câmpuri numerice: temperature, humidity și voc_index. De asemenea, conține un câmp „timestamp” care este completat automat la fiecare operațiune de inserare. Fișierul „models.py” reflectă direct formatul pachetelor JSON transmise de către bridge-ul Python.

```
# model pentru structura datelor primite de la ESP32
class SensorData(models.Model):
    temperature = models.FloatField()
    humidity = models.FloatField()
    voc_index = models.IntegerField()
    timestamp = models.DateTimeField(auto_now_add=True)
```

Figura 21. Model pentru stocarea valorilor primite de la senzori

Pentru un transfer eficient al datelor între client și server, se utilizează serializatorul „SensorDataSerializer”, ilustrat în Figura 22. Acesta este responsabil pentru conversia automată între obiectele Django și formatul JSON.

```python
# definirea clasei SensorDataSerializer pentru transformarea
# obiectelor modelului SensorData in format JSON si invers
# pentru utilizare in cadrul API REST
class SensorDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = SensorData
        fields = '__all__'
```

Figura 22. Serializerul folosit pentru conversia datelor senzorilor în/din JSON

Endpoint-ul principal, care gestionează solicitările POST provenite de la scriptul bridge, este definit în fișierul „views.py” și este ilustrat în Figura 20. Clasa „SensorDataView” moștenește „APIView” din Django REST Framework. Aceasta implementează două metode distincte: una pentru recepția datelor, prin metoda POST, iar cealaltă pentru returnarea acestora la cerere, prin metoda GET. În cazul unei cereri valide, datele sunt salvate automat în baza de date SQLite.

Înregistrarea acestui endpoint se realizează în fișierul „urls.py”, sub ruta „/api/data/”. Aceasta este și adresa către care scriptul „mqtt_to_django.py” trimite datele în mod periodic. În plus, aplicația oferă și alte rute, precum dashboard-ul HTML sau endpointul pentru actualizare AJAX.

În ceea ce privește baza de date, backend-ul folosește sistemul implicit oferit de Django, anume SQLite. Acesta este un motor de baze de date recunoscut pentru simplitatea, fiabilitatea și portabilitatea sa. Această alegere este potrivită pentru aplicații locale sau demonstrative, unde nu este necesar un server SQL complet. Structura bazei de date este generată automat prin procesul de migrare a modelului „SensorData”.

Fișierul „settings.py” centralizează toate configurațiile generale ale aplicației. Printre acestea se numără și fusul orar (Europe/Bucharest), aplicațiile instalate (rest_framework, senzori), precum și motorul bazei de date. Django REST Framework este folosit pentru expunerea datelor prin API, iar aplicația „senzori” este structurată modular pentru a grupa logic toate componentele relevante. Astfel, aceasta acoperă atât logica specifică senzorilor, cât și interfața web asociată.

Prin urmare, backend-ul Django constituie coloana vertebrală a sistemului. Oferă un punct central de stocare și un mod controlat de acces la date. Toate componentele se integrează simplu și eficient, fără procesări sau modificări intermediare.

## 6.7 Interfața web

Pentru a permite vizualizarea în timp real a valorilor provenite de la senzori, a fost implementată o interfață web structurată ca o pagină HTML cu actualizare automată. Aceasta oferă utilizatorului posibilitatea de a monitoriza temperatura, umiditatea și indicele VOC. Reîncărcarea manuală a paginii nu este necesară. Accesul către interfață este asigurat local, prin intermediul unui server Django.

Fișierul „sensor_dashboard.html” este integrat în cadrul aplicației Django. Randarea lui se face prin intermediul unui view dedicat, denumit „sensor_dashboard”, atunci când utilizatorul accesează ruta „/dashboard/”. Pagina este generată la nivel de server și este populată inițial cu cele mai recente 100 de înregistrări din baza de date. Fișierul HTML include o secțiune tabelară și un script JavaScript dedicat, alături de elemente de stilizare CSS. Scriptul gestionează atât actualizarea dinamică, cât și afișarea grafică a datelor.

Datele noi sunt inserate într-un element <tbody> al tabelului HTML, ilustrat în Figura 23. Fiecare înregistrare corespunde unui rând care conține informații despre temperatură, umiditate, indicele VOC și momentul înregistrării.

```html
<table id="sensor-table">
    <thead>
        <tr>
            <th>Timestamp</th>
            <th>Temperature (°C)</th>
            <th>Humidity (%)</th>
            <th>VOC Index</th>
        </tr>
    </thead>
    <tbody id="table-body">
        <!-- randuri generate din JS -->
    </tbody>
</table>
```

Figura 23. Structura HTML a tabelului pentru afișarea datelor senzorilor în interfața web

Periodic, la fiecare cinci secunde, scriptul execută o cerere AJAX către backend. Acesta trimite ultimul ID înregistrat, iar serverul furnizează doar datele cele mai recente. Ulterior, datele sunt integrate dinamic în structura tabelului existent, fără să afecteze rândurile deja afișate.

Mecanismul de actualizare asincronă, ilustrat în Figura 24, este implementat cu funcțiile native „fetch()” și „setInterval()”. El permite menținerea unei interfețe dinamice fără a necesita o reîncărcare completă a paginii. Întregul proces este gestionat la nivelul clientului, în browser, și nu sunt utilizate biblioteci externe.

Informațiile prezentate în dashboard sunt extrase dintr-o bază de date SQLite, prin intermediul unui view Django denumit „get_new_data”. Acesta filtrează înregistrările noi pe baza unui ID incremental. Mecanismul este adecvat scenariilor locale, unde latențele și nivelul de concurență sunt minime.

Interfața ilustrată în Figurile 25 și 26 prezintă un design minimalist, lipsit de elemente vizuale complexe. Rolul său este de a demonstra funcționarea corectă a sistemului și de a informa utilizatorul despre reacția senzorilor la modificări de temperatură, umiditate sau calitatea aerului.

```javascript
// AJAX Polling la fiecare 5 sec
setInterval(fetchNewData, 5000);

// actualizare stare ESP32
function updateESPStatus() {
    const now = Date.now();
    const statusDiv = document.getElementById("esp-status");

    if (lastDataTime === null) {
        statusDiv.textContent = "ESP32 Status: Offline";
        statusDiv.style.color = "#dc3545";
        return;
    }

    const elapsed = now - lastDataTime;
    if (elapsed <= 15000) {
        statusDiv.textContent = "ESP32 Status: Online";
        statusDiv.style.color = "#28a745";
    } else {
        statusDiv.textContent = "ESP32 Status: Offline";
        statusDiv.style.color = "#dc3545";
    }
}
// verificare status la fiecare 5 sec
setInterval(updateESPStatus, 5000;
```

Figura 24. Fragment JavaScript pentru actualizarea automată a datelor în interfața web

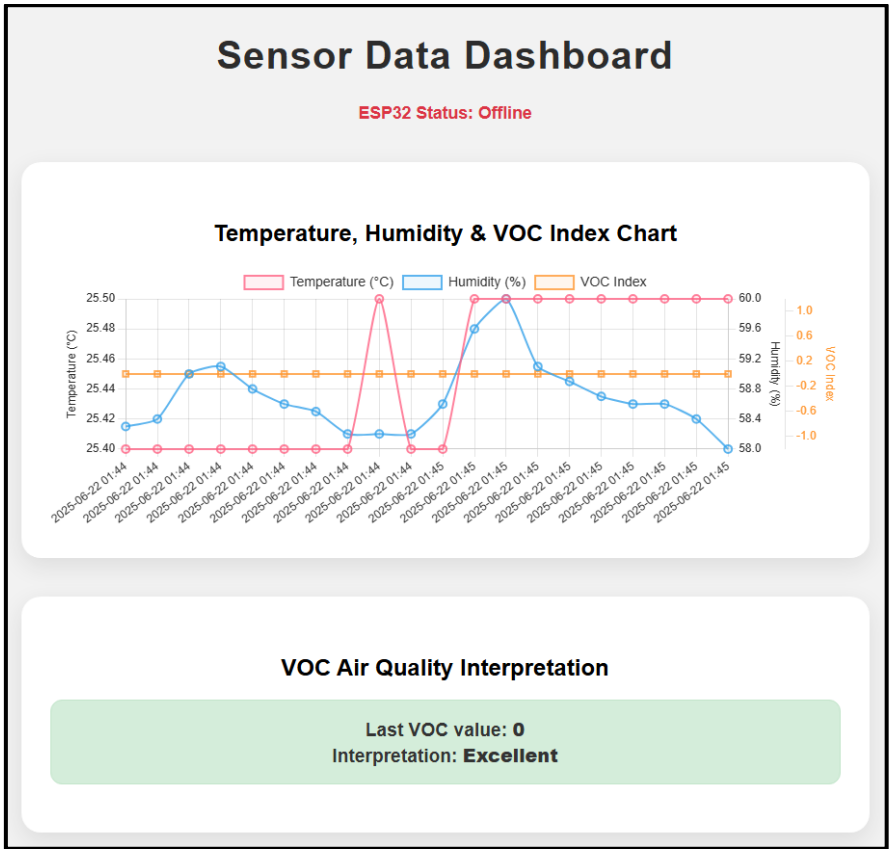



Figura 25. Aspectul interfeței web în timpul funcționării sistemului

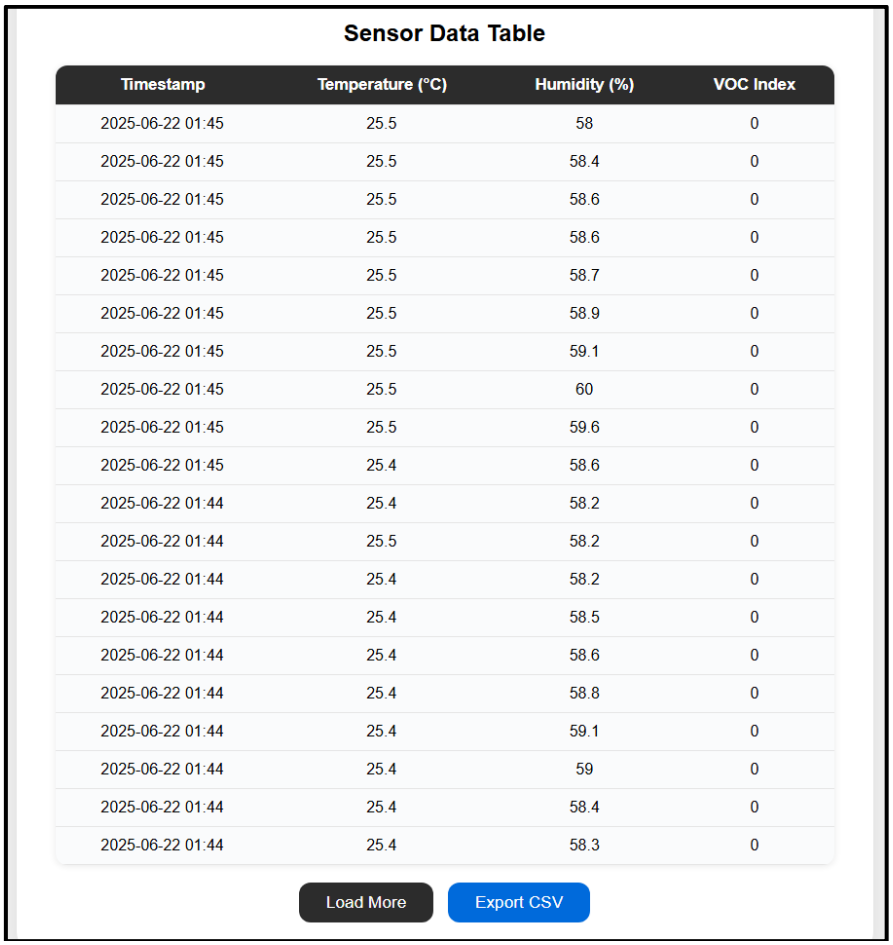



Figura 26. Aspectul interfeței web în timpul funcționării sistemului

## 6.8 Probleme întâmpinate și soluții aplicate

O primă observație a fost legată de comportamentul conexiunii Wi-Fi a modulului ESP32-S2. După o pierdere temporară a conexiunii, dispozitivul rămânea blocat într-o stare deconectată, iar reconectarea nu mai putea fi stabilită automat. În cadrul buclei principale „loop()" a fost implementată o verificare periodică a stării conexiunii, cu reapelarea funcției „connectWiFi()" pentru a preveni o întrerupere permanentă în fluxul de date. Această soluție este ilustrată în Figura 27.

```
void loop() {
  if (WiFi.status() != WL_CONNECTED) {
    connectWiFi();
  }
}
```

Figura 27. Verificarea și reapelarea conexiunii Wi-Fi în bucla principală

În etapa de configurare a protocolului TLS, cu toate că certificatul era generat corect, modulul ESP32-S2 refuza stabilirea conexiunii către brokerul Mosquitto. Eroarea era cauzată de lipsa extensiei „subjectAltName" în certificatul auto-semnat. Această extensie este esențială pentru validarea adresei IP a serverului în protocoalele TLS actuale. Soluția a fost modificarea fișierului de configurare (.cnf), ilustrat în Figura 28, utilizat la generarea certificatului prin includerea explicită a adresei IP a brokerului în câmpul „alt_names".

```
[ req ]
default_bits        = 2048
prompt              = no
default_md          = sha256
distinguished_name  = dn
x509_extensions     = v3_req

[ dn ]
C  = RO
ST = Cluj
L  = Cluj-Napoca
O  = ESPTest
OU = Dev
CN = 192.168.[illegible]

[ v3_req ]
subjectAltName = @alt_names

[ alt_names ]
IP.1 = 192.168.[illegible]
```

Figura 28. Fișier de configurare OpenSSL

O măsură de protecție suplimentară pentru senzorul SGP40 a fost implementarea unui mecanism de reîncercare a inițializării, ilustrat în Figura 29. Cu toate că funcționarea senzorului este stabilă, în cazul unei eventuale întreruperi, mecanismul este activat și declanșează o serie de

reinițializări automate după un interval predefinit. Această măsură contribuie la fiabilitatea sistemului, în special în scenariile de instabilitate pe magistrala I²C.

```
void loop() {
  // reinitializare SGP40 daca se deconecteaza
  if (!sgp40Initialized) {
    if (millis() - lastSGP40Retry > sensorRetryInterval) {
      Serial.println("Retrying SGP40 initialization...");
      sgp40Initialized = initSGP40();
      lastSGP40Retry = millis();
    }
  }
}
```

Figura 29. Reinițializare periodică a senzorului SGP40 în cazul pierderii conexiunii I²C

```
// functie export date in format CSV
function exportToCSV(dataArray) {
  if (!dataArray.length) return;

  const csvHeaders = ["Timestamp", "Temperature (C)", "Humidity (%)", "VOC
Index"];

  const csvRows = dataArray.map(row => [
    formatTimestamp(row.timestamp),
    row.temperature.toString().replace(".", ","),
    row.humidity.toString().replace(".", ","),
    row.voc_index
  ]);

  const csvContent = [
    csvHeaders.join(";"),
    ...csvRows.map(row => row.join(";"))
  ].join("\n");

  const blob = new Blob([csvContent], { type: "text/csv;charset=windows-1252;"
});
  const url = URL.createObjectURL(blob);

  const link = document.createElement("a");
  link.setAttribute("href", url);
  link.setAttribute("download", "sensor_data.csv");
  link.style.display = "none";

  document.body.appendChild(link);
  link.click();
  document.body.removeChild(link);
}

document.getElementById('export-csv').addEventListener('click', function () {
  exportToCSV(fullSensorData);
});
```

Figura 30. Funcție JavaScript pentru exportul datelor în format CSV

Tot ca măsură de prevenție, a fost implementată o validare în scriptul bridge „mqtt_to_django.py”, înainte de trimiterea datelor către API-ul Django. Mecanismul ilustrat în Figura 18 are rolul de a verifica dacă toate valorile sunt prezente. Din cauza naturii asincrone, mesajele MQTT pot ajunge în ordine aleatorie. Prin urmare, este esențial ca toate cele trei câmpuri pentru temperatură, umiditate și VOC să fie colectate înainte de a construi payload-ul destinat cererilor de tip POST. Această filtrare, realizată în cadrul bufferului, previne trimiterea unor date incomplete către server.

În primele implementări ale interfeței web, tabelul cu valorile senzorilor nu se actualiza corect. Variabila „lastId” nu era configurată corect la momentul încărcării paginii, iar interfața ignora datele noi returnate de backend. Soluția a constat în setarea variabilei „lastId” în funcție de primul rând de date afișat deja în tabel. După această ajustare, comportamentul a devenit stabil, asigurând o actualizare asincronă constantă.

Exportul datelor în format CSV (Comma-Separated Values) a fost particularizat pentru a asigura compatibilitatea cu sistemele locale. Codificarea implicită a fost înlocuită cu „windows-1252”. Separatorul zecimal a fost configurat să utilizeze virgula (,), iar separatorul de câmp a fost setat explicit la punct și virgulă (;). Acest proces este ilustrat în Figura 30. Astfel, fișierul CSV generat este compatibil cu aplicații precum Microsoft Excel configurat pentru setările regionale din Europa Centrală.

*Instrumentele de inteligență artificială au fost utilizate doar pentru corectarea gramaticală și stilizarea formală a textului. Conținutul lucrării nu a fost redactat cu ajutorul acestora. Intervențiile au vizat claritatea, lizibilitatea și coerența exprimării, fără a modifica fondul ideilor.*

# 7 Rezultate experimentale

## 7.1 Metodologia testării

În această etapă a fost vizată testarea funcționării întregului sistem într-un scenariu cât mai apropiat de utilizarea reală. S-a urmărit atât funcționarea individuală a componentelor, cât și integrarea dintre partea hardware și software, de la senzori până la afișarea valorilor în interfața web. Procesul de testare a contribuit semnificativ la modificările realizate pe parcursul dezvoltării proiectului, în special pentru a asigura comunicarea sigură între module și stabilitatea transmisiei.

Ansamblul format din placa ESP32-S2 și cei doi senzori a fost alimentat și amplasat într-o încăpere cu parametri ambientali relativ constanți. Pentru a observa și înregistra răspunsul sistemului la modificări, au fost realizate intervenții precum pornirea unui aparat de aer condiționat pentru a forța o variație detectabilă de către senzorul DHT22 sau apropierea unui marker de senzorul SGP40 pentru a modifica valoarea indexului VOC. Independent de sistem, a fost utilizat un termometru-higrometru digital de cameră pentru a compara valorile senzorului DHT22 cu cele afișate de dispozitivul de referință.

Pentru a testa funcționarea componentelor software, au fost inițializate serverul local, brokerul MQTT Mosquitto, bridge-ul Python și aplicația Django. Interfața web a fost accesată local, folosind un browser de pe un laptop conectat la aceeași rețea Wi-Fi. În acest mod, s-a putut urmări în timp real modul în care sunt colectate, transmise și afișate valorile de la senzori. În plus, au fost realizate exporturi de date în format CSV pentru analize ulterioare în Microsoft Excel.

Au fost vizate următoarele aspecte de funcționare: frecvența de actualizare a interfeței, stabilitatea conexiunilor și comportamentul aplicației în eventualitatea unor întreruperi. Aceste observații au contribuit la evaluarea scalabilității sistemului și la identificarea necesității unor modificări suplimentare pentru optimizarea performanței.

## 7.2 Colectarea datelor în regim normal

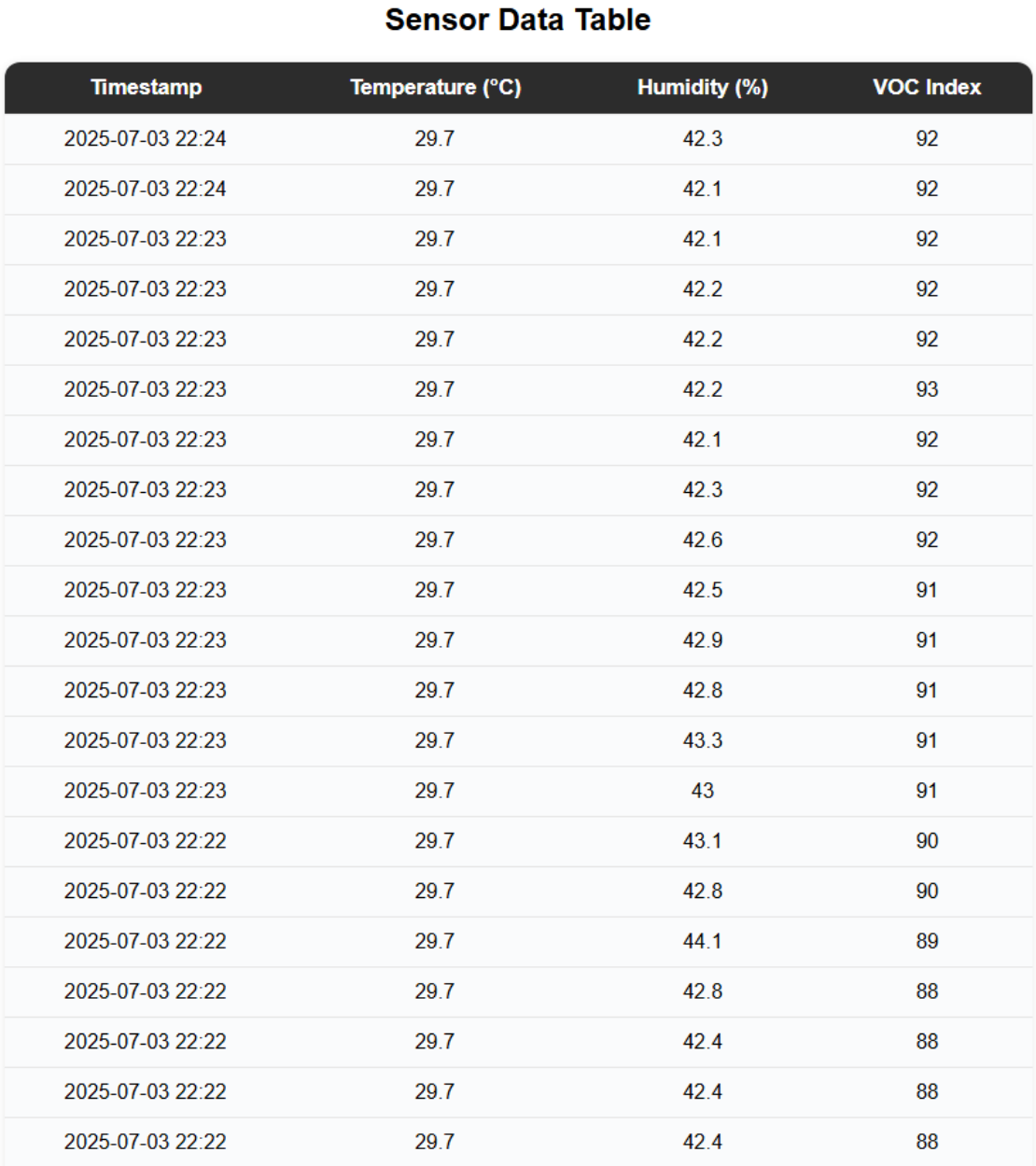
Pentru a evalua comportamentul sistemului în condiții normale, ansamblul hardware a fost pus în funcțiune într-un mediu stabil, fără oscilații intenționate. Senzorii DHT22 și SGP40 au fost lăsați să preia valori la intervale regulate de cinci secunde, conform configurației software. Sistemul a rulat continuu pentru o perioadă de aproximativ 20 de minute. Datele au fost colectate, transmise, stocate și afișate fără întreruperi.

Monitorizarea a fost realizată prin interfața web, iar comportamentul graficului și al tabelului a fost observat în timp real. Valorile au fost actualizate automat, conform mecanismului de AJAX polling implementat. Acest lucru a permis urmărirea continuă a sistemului, fără intervenții manuale. Nu au apărut blocaje, iar valorile au rămas constante în jurul unei referințe medii. În Tabelul 3 este prezentat un eșantion de 15 înregistrări colectate în regimul normal de funcționare. În Figura 31 este ilustrată interfața web în funcționare stabilă, iar în Figura 32 este reprezentat graficul valorilor senzorilor într-un interval de aproximativ 8 de minute.

Tabelul 3. Valori colectate de la senzori în regim de funcționare normală

| Nr. crt. | Timestamp | Temperatură (°C) | Umiditate (%) | VOC Index |
|---|---|---|---|---|
| 1 | 03.07.2025 22:24 | 29,7 | 42,3 | 92 |
| 2 | 03.07.2025 22:24 | 29,7 | 42,1 | 92 |
| 3 | 03.07.2025 22:23 | 29,7 | 42,1 | 92 |
| 4 | 03.07.2025 22:23 | 29,7 | 42,2 | 92 |
| 5 | 03.07.2025 22:23 | 29,7 | 42,2 | 92 |

| | | | | |
|---|---|---|---|---|
| 6 | 03.07.2025 22:23 | 29,7 | 42,2 | 93 |
| 7 | 03.07.2025 22:23 | 29,7 | 42,1 | 92 |
| 8 | 03.07.2025 22:23 | 29,7 | 42,3 | 92 |
| 9 | 03.07.2025 22:23 | 29,7 | 42,6 | 92 |
| 10 | 03.07.2025 22:23 | 29,7 | 42,5 | 91 |
| 11 | 03.07.2025 22:23 | 29,7 | 42,9 | 91 |
| 12 | 03.07.2025 22:23 | 29,7 | 42,8 | 91 |
| 13 | 03.07.2025 22:23 | 29,7 | 43,3 | 91 |
| 14 | 03.07.2025 22:23 | 29,7 | 43 | 91 |
| 15 | 03.07.2025 22:22 | 29,7 | 43,1 | 90 |



**Sensor Data Table**

| Timestamp | Temperature (°C) | Humidity (%) | VOC Index |
|---|---|---|---|
| 2025-07-03 22:24 | 29.7 | 42.3 | 92 |
| 2025-07-03 22:24 | 29.7 | 42.1 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.1 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.2 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.2 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.2 | 93 |
| 2025-07-03 22:23 | 29.7 | 42.1 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.3 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.6 | 92 |
| 2025-07-03 22:23 | 29.7 | 42.5 | 91 |
| 2025-07-03 22:23 | 29.7 | 42.9 | 91 |
| 2025-07-03 22:23 | 29.7 | 42.8 | 91 |
| 2025-07-03 22:23 | 29.7 | 43.3 | 91 |
| 2025-07-03 22:23 | 29.7 | 43 | 91 |
| 2025-07-03 22:22 | 29.7 | 43.1 | 90 |
| 2025-07-03 22:22 | 29.7 | 42.8 | 90 |
| 2025-07-03 22:22 | 29.7 | 44.1 | 89 |
| 2025-07-03 22:22 | 29.7 | 42.8 | 88 |
| 2025-07-03 22:22 | 29.7 | 42.4 | 88 |
| 2025-07-03 22:22 | 29.7 | 42.4 | 88 |
| 2025-07-03 22:22 | 29.7 | 42.4 | 88 |

Figura 31. Interfața web în regim de funcționare constantă

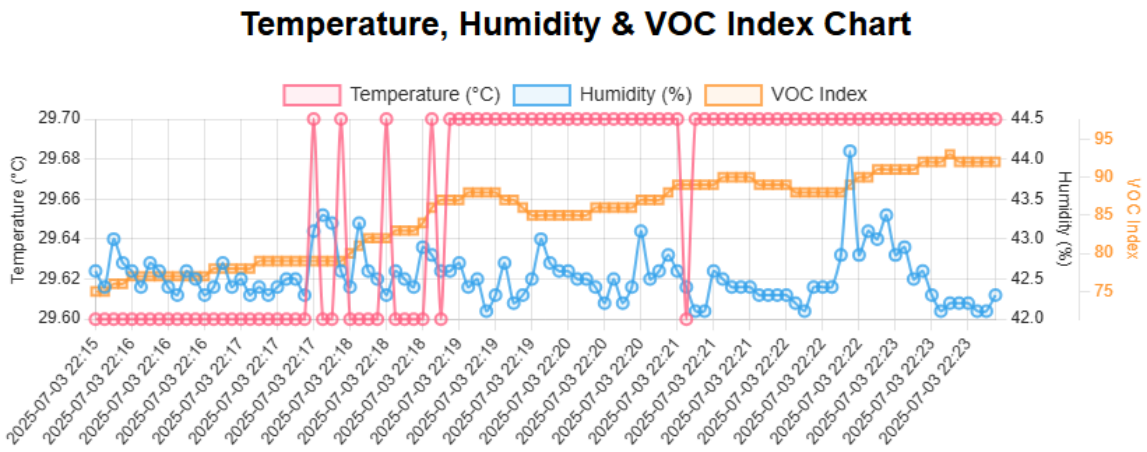



Figura 32. Graficul valorilor senzorilor în interval de aproximativ 8 minute

În urma testelor efectuate în regimul normal de funcționare, se poate concluziona că ansamblul hardware și fluxul de date funcționează stabil. Înregistrările au fost colectate fără discontinuități, demonstrând că arhitectura este potrivită pentru scenarii practice de monitorizare locală.

## 7.3 Compararea valorilor obținute cu un dispozitiv etalon

Pentru a evalua acuratețea senzorului DHT22, s-a folosit ca etalon un termometru-higrometru digital de cameră. Ambele instrumente au fost așezate unul lângă celălalt, în aceleași condiții de testare, fără existența altor factori perturbatori.

Ansamblul cu ESP32-S2 a fost lăsat să se stabilizeze timp de câteva minute. După stabilizare, au fost comparate și notate valorile citite de aparatul extern și cele transmise de sistemul testat prin interfața web. Datele au fost centralizate în Tabelul 4, în care se pot observa diferențele între măsurători.

Tabelul 4. Comparație între valorile înregistrate de senzorul DHT22 și cele ale dispozitivului etalon

| Nr. crt. | Timestamp | Temperatură DHT22 (°C) | Temperatură Etalon (°C) | Umiditate DHT22 (%) | Umiditate Etalon (%) |
|---|---|---|---|---|---|
| 1 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,1 | 37 |
| 2 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,2 | 37 |
| 3 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,2 | 37 |
| 4 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,2 | 37 |
| 5 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,1 | 37 |
| 6 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,2 | 37 |
| 7 | 2025-07-04 14:15 | 31,4 | 28,8 | 35,2 | 37 |
| 8 | 2025-07-04 14:14 | 31,4 | 28,8 | 35,1 | 37 |
| 9 | 2025-07-04 14:14 | 31,4 | 28,8 | 35,2 | 37 |
| 10 | 2025-07-04 14:14 | 31,4 | 28,8 | 35,2 | 37 |

Conform rezultatelor, senzorul DHT22 a înregistrat o temperatură constantă de 31,4 °C, în timp ce dispozitivul etalon a indicat o valoare de 28,8 °C pe întreaga durată a testului, rezultând o diferență de 2,6 °C. În cazul valorilor pentru umiditate, diferența a fost de aproximativ 1,8%, senzorul raportând valori de aproximativ 35,2%, în timp ce aparatul de referință a indicat constant valoarea de 37%.

Aceste abateri se justifică prin faptul că senzorul DHT22 a fost montat pe breadboard, în proximitatea modulului ESP32-S2, ceea ce induce un efect termic local. În plus, dispozitivul etalon este proiectat să funcționeze într-o carcasă închisă și este mai puțin influențat de fluctuațiile termice, în timp ce senzorul DHT22 este expus în mod direct curenților de aer.

Diferențele înregistrate nu afectează validitatea funcțională a sistemului propus pentru monitorizarea generală a parametrilor de mediu, dar indică faptul că pentru utilizări profesionale ar fi recomandată utilizarea unor senzori cu precizie superioară.

## 7.4 Testarea reacției senzorilor în anumite condiții

Pentru a verifica răspunsul senzorilor la diferite schimbări ale parametrilor de mediu, au fost aplicați stimuli direcți asupra componentelor de achiziție. Reacția senzorilor a fost observată în timp real prin interfața web.

În cazul senzorului DHT22 s-a utilizat un uscător de păr ca sursă de aer cald pentru a provoca o ușoară creștere a temperaturii. După câteva secunde, temperatura a crescut cu aproximativ 7,3 °C, de la 31,7 °C la 38 °C, iar umiditatea a scăzut de la aproximativ 35% la 26%, fapt ilustrat în Figura 33. Valorile au revenit la normal după aproximativ 5-6 minute. Alternativ, același senzor a fost testat prin expunerea la aerul rece al unui ventilator. După aproximativ 40 de secunde, s-a observat o scădere de temperatură de 3,5 °C, de la 31,9 °C la 28,4 °C, și o ușoară creștere a umidității, de la 35,4% la 42,7%. Valorile au revenit la normal în decursul a aproximativ 5-6 minute. Acest lucru este reprezentat în Figura 34.

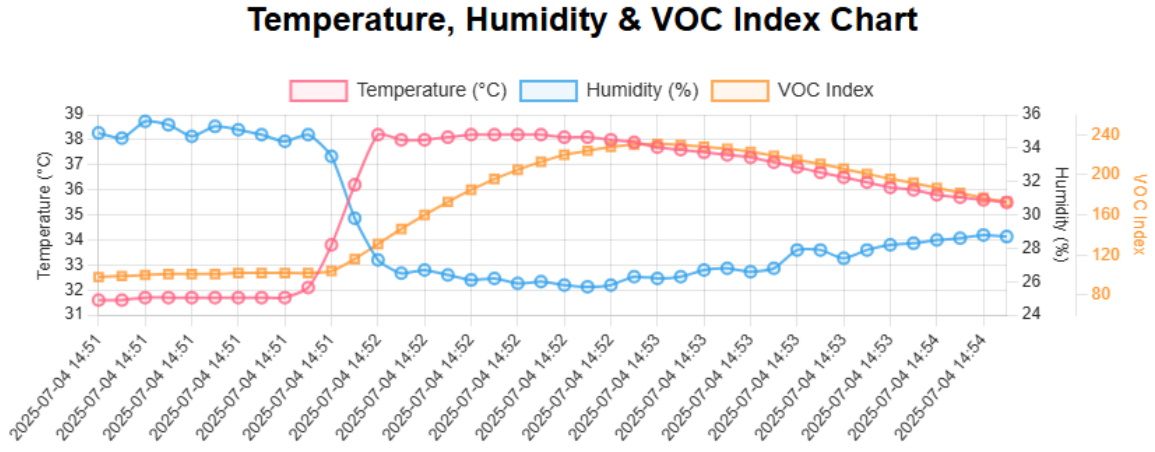



Figura 33. Reacția senzorului DHT22 la expunerea cu aer cald

Pentru manipularea senzorului SGP40 a fost utilizat un marker. Prin apropierea acestuia de senzor timp de câteva secunde, reacția a fost imediată. Valoarea VOC a început să crească brusc de la 100 la 245. Saltul a fost vizibil pe graficul interfeței web, observabil în Figura 35, iar interpretarea VOC s-a modificat în funcție de pragurile valorii. Valoarea a revenit la normal după aproximativ 3 minute de la depărtarea sursei perturbatoare.

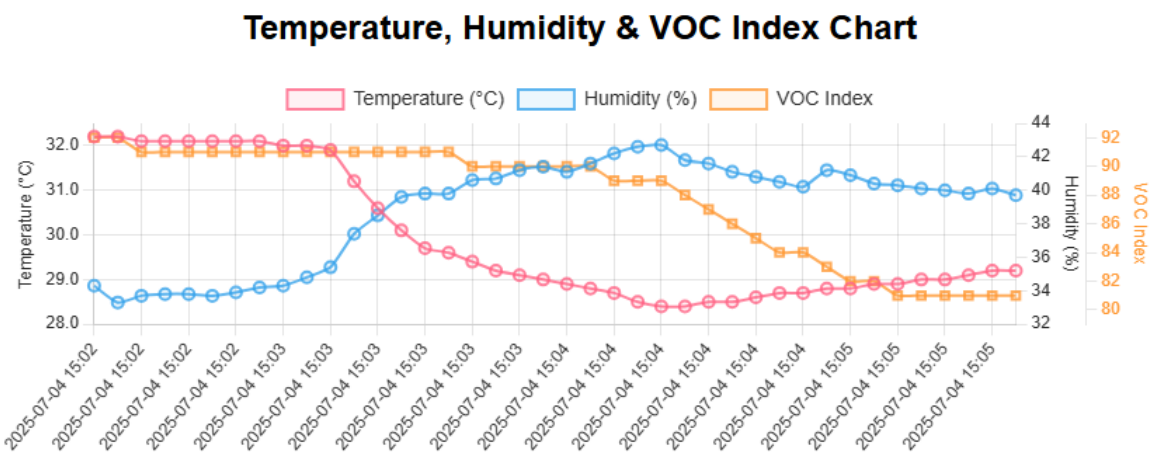



Figura 34. Reacția senzorului DHT22 la aer rece

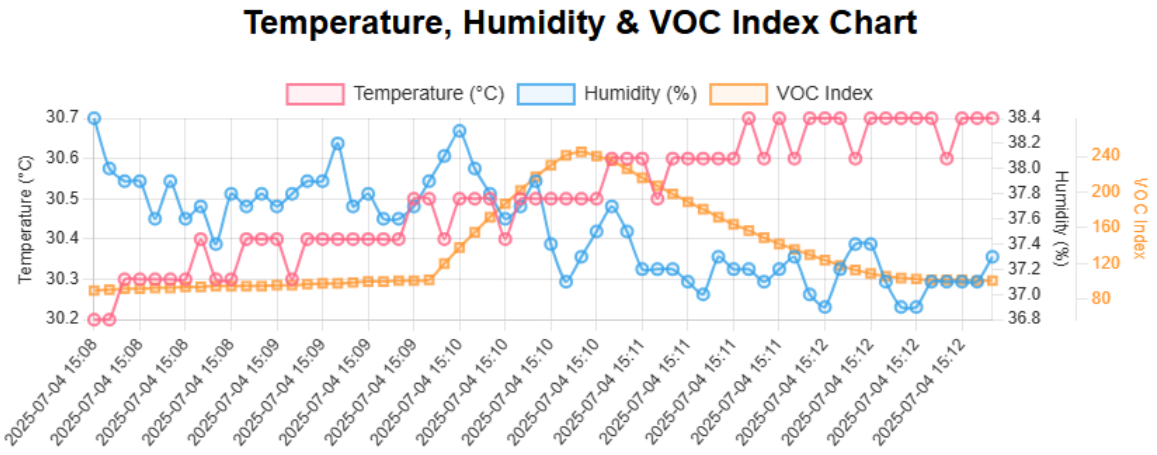



Figura 35. Creștere bruscă a valorii VOC la apropierea unui marker

Sistemul răspunde eficient la modificările mediului înconjurător, iar publicarea tuturor valorilor se face în mod normal și stabil. Atât partea de achiziție, cât și cea de interpretare vizuală a datelor corespund valorilor reale ale mediului ambiant, ceea ce oferă o bază solidă pentru aplicații de monitorizare.

## 7.5 Comportamentul sistemului în condiții limită

Aceste teste au avut ca scop evaluarea robusteții sistemului în fața unor situații neprevăzute. În urma simulării a mai multor scenarii, a fost observat comportamentul fiecărei componente și s-a validat faptul că sistemul poate reveni automat la o stare funcțională, fără intervenții semnificative.

Primul test efectuat a implicat întreruperea conexiunii Wi-Fi în timpul funcționării plăcii ESP32-S2. Imediat după pierderea semnalului, modulul a încercat automat reconectarea. Odată ce rețeaua a redevenit disponibilă, modulul ESP32-S2 a stabilit din nou conexiunea și a continuat transmiterea normală a datelor. Acest comportament este ilustrat în Figura 36, unde se observă secvența de mesaje din consola Arduino IDE.

```
--------------------------------
DHT22 - Temp: 31.60 °C, Hum: 34.30 %
SGP40 - VOC Index: 112
All sensor data published.
--------------------------------
Connecting to WiFi..E (6785008) wifi:sta is connecting, return error
.............. Connected.
ESP32-S2 IP address: 192.168.[illegible]
Connecting to MQTT broker with TLS... Connected.
DHT22 - Temp: 31.60 °C, Hum: 34.50 %
SGP40 - VOC Index: 112
All sensor data published.
--------------------------------
DHT22 - Temp: 31.60 °C, Hum: 34.40 %
SGP40 - VOC Index: 112
All sensor data published.
--------------------------------
```

Figura 36. Reconectarea automată a plăcii ESP32 la rețeaua Wi-Fi după pierderea semnalului

În cel de-al doilea scenariu, s-a verificat răspunsul sistemului la indisponibilitatea brokerului MQTT Mosquitto. Brokerul a fost oprit intenționat în timpul unei sesiuni active. Imediat după pierderea conexiunii au apărut mesaje de eroare la nivelul clientului MQTT în interfața Serial Monitor din Arduino IDE. Modulul ESP32-S2 a realizat în mod automat tentative repetate de reconectare, la intervale regulate. Acest mecanism este ilustrat în Figura 37.

```
--------------------------------
DHT22 - Temp: 31.70 °C, Hum: 34.20 %
SGP40 - VOC Index: 100
All sensor data published.
--------------------------------
Connecting to MQTT broker with TLS... Connection failed!, rc=-2
DHT22 - Temp: 31.70 °C, Hum: 34.30 %
SGP40 - VOC Index: 100
Failed to publish temperature
Failed to publish humidity
Failed to publish air quality
Connecting to MQTT broker with TLS... Connected.
DHT22 - Temp: 31.70 °C, Hum: 34.70 %
SGP40 - VOC Index: 100
All sensor data published.
--------------------------------
DHT22 - Temp: 31.70 °C, Hum: 34.70 %
SGP40 - VOC Index: 100
All sensor data published.
--------------------------------
```

Figura 37. Pierderea conexiunii cu brokerul MQTT și tentativa de reconectare

Pentru a testa funcționarea părții de backend, a fost întreruptă rularea serverului Django în timp ce bridge-ul Python era activ și primea date noi de la senzori. Imediat după oprirea serverului, scriptul bridge a întâmpinat o eroare la trimiterea cererii POST către endpoint-ul API. Comportamentul conexiunii HTTPS care nu a putut fi stabilită este ilustrat în Figura 38 și confirmă faptul că, în lipsa serviciului backend, bridge-ul gestionează corect eșecul conexiunii fără a se opri, afișând eroarea și continuând execuția în așteptarea reluării serverului.

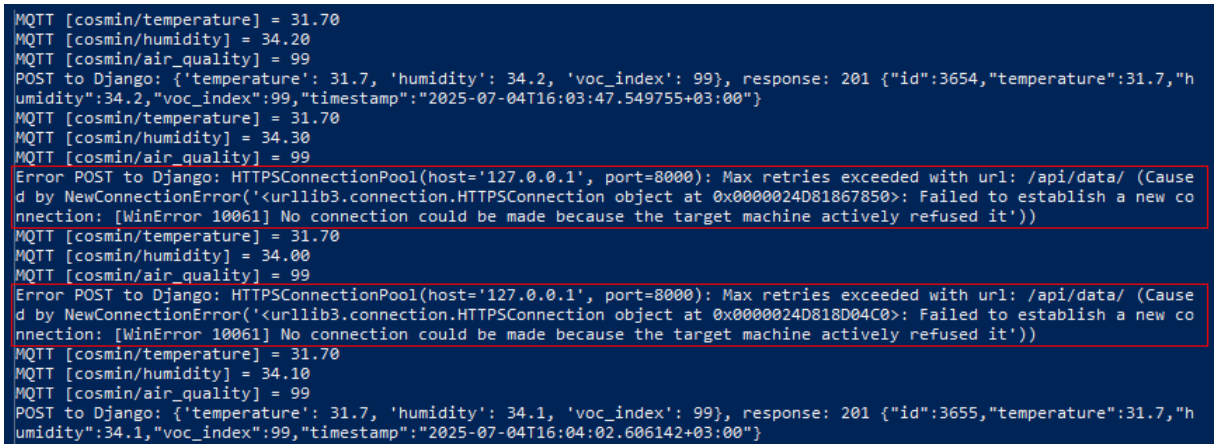


Figura 38. Eroare afișată de bridge-ul Python la oprirea serverului Django

Imediat după ce serviciile backend au fost repornite, întregul flux de date s-a restabilit automat. Bridge-ul a reluat transmiterea informațiilor, modulul ESP32-S2 a reînceput publicarea datelor, iar interfața web a revenit la afișarea în timp real a valorilor.

Toate aceste teste au demonstrat că sistemul este tolerant la erori și capabil să-și recupereze autonomia funcțională. Prin urmare, este un ansamblu adecvat pentru integrarea în aplicații de monitorizare în scenarii operaționale reale.

## 7.6 Exportul datelor și interpretarea acestora

Interfața web integrează o funcționalitate de export a datelor în format CSV. Acest lucru este destinat analizei ulterioare a valorilor colectate de la senzori. Funcția permite descărcarea ultimelor înregistrări din baza de date sub forma unui fișier compatibil cu aplicații precum Microsoft Excel.

Exportul se realizează prin apăsarea butonului „Export CSV”. Datele sunt organizate pe coloane, iar separatorul folosit este caracterul „;” (punct și virgulă). După descărcarea fișierului, valorile pot fi sortate, filtrate sau reprezentate grafic. În Figura 39 este ilustrat conținutul unui astfel de fișier CSV încărcat în Microsoft Excel. Ordinea apariției coloanelor este următoarea: timestamp, temperatură, umiditate și indicele VOC. Structura simplă permite analiza imediată pe un interval de timp mai lung.

În Figura 40 sunt ilustrate trei grafice care reprezintă evoluția celor trei indici măsurați, realizate pe baza valorilor din fișierul CSV exportat. Variațiile reprezentate pe graficele din Excel se corelează cu datele observate în timpul testării directe prin interfața web.

Această implementare nu este doar în scop tehnic auxiliar, ci are rolul de a extinde utilitatea sistemului dincolo de monitorizarea în timp real. Utilizatorul are posibilitatea de a salva datele și poate evalua un interval de timp extins pe un termen mai lung, în vederea observării anomaliilor sau pentru generarea rapoartelor de mediu pentru perioade extinse. Sistemul poate să devină astfel un instrument de înregistrare și analiză completă.

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Timestamp | Temperature (C) | Humidity (%) | VOC Index |
| 2 | 04.07.2025 17:08 | 31,7 | 34,2 | 105 |
| 3 | 04.07.2025 17:08 | 31,7 | 34,1 | 105 |
| 4 | 04.07.2025 17:08 | 31,7 | 34,1 | 105 |
| 5 | 04.07.2025 17:08 | 31,7 | 34,2 | 105 |
| 6 | 04.07.2025 17:08 | 31,7 | 34,2 | 105 |
| 7 | 04.07.2025 17:08 | 31,7 | 34 | 105 |
| 8 | 04.07.2025 17:08 | 31,7 | 34,3 | 106 |
| 9 | 04.07.2025 17:08 | 31,8 | 34 | 106 |
| 10 | 04.07.2025 17:07 | 31,7 | 34,2 | 107 |
| 11 | 04.07.2025 17:07 | 31,7 | 33,6 | 107 |
| 12 | 04.07.2025 17:07 | 31,7 | 33,9 | 108 |
| 13 | 04.07.2025 17:07 | 31,7 | 33,8 | 109 |
| 14 | 04.07.2025 17:07 | 31,8 | 33,6 | 109 |
| 15 | 04.07.2025 17:07 | 31,7 | 33,5 | 110 |

Figura 39. Fișier CSV exportat din interfața web

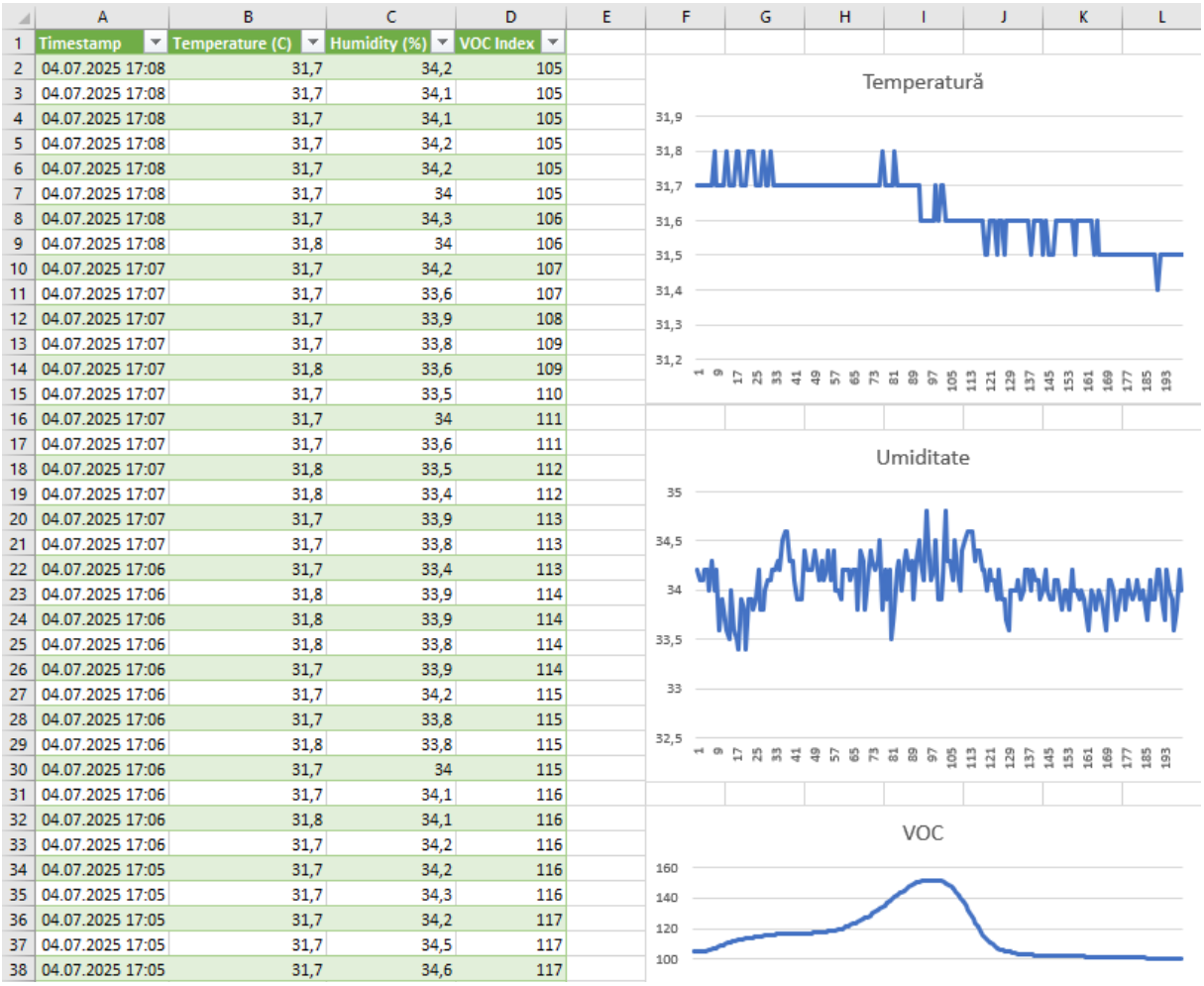



Figura 40. Grafic generat în Excel pe baza datelor exportate

*Instrumentele de inteligență artificială au fost utilizate doar pentru corectarea gramaticală și stilizarea formală a textului. Conținutul lucrării nu a fost redactat cu ajutorul acestora. Intervențiile au vizat claritatea, lizibilitatea și coerența exprimării, fără a modifica fondul ideilor.*

# 8 Concluzii

Prezenta lucrare de licență prezintă dezvoltarea unui sistem de monitorizare în timp real a parametrilor de mediu. Cei trei parametri vizați sunt temperatura, umiditatea și calitatea aerului. Sistemul dezvoltat integrează atât partea conexiunilor hardware, cât și cea software, incluzând firmware-ul modulului ESP32-S2, fluxul de transmitere a datelor și interfața web dedicată.

Implementarea a avut ca scop extinderea funcționalității unei drone comerciale prin adăugarea unui modul și senzori. Placa ESP32-S2 a fost aleasă pentru capacitatea sa de a gestiona conexiunile Wi-Fi și criptarea TLS. Pe partea software, implementarea a presupus dezvoltarea și testarea comunicației MQTT securizată, gestionarea brokerului Mosquitto și integrarea unui backend Django.

Aspectul esențial al acestui proiect a fost asigurarea securității comunicațiilor. A fost implementată autentificarea TLS bidirecțională, atât pentru conexiunea dintre modulul ESP32 și broker, cât și pentru cea dintre bridge-ul Python și interfața API Django. Această abordare asigură integritatea și confidențialitatea datelor transmise pe parcursul tuturor etapelor.

Arhitectura sistemului se bazează pe un principiu modular, în care datele sunt achiziționate la intervale de 5 secunde și publicate asincron prin protocolul MQTT. Ulterior, un script Python preia aceste date și le stochează într-o bază de date SQLite. Întregul sistem este susținut de o aplicație Django REST, care facilitează atât interfața web HTML, cât și endpoint-urile necesare pentru comunicațiile AJAX și persistența datelor. Designul minimalist al interfeței web facilitează monitorizarea în timp real a comportamentului sistemului și a fost un reper esențial pentru etapa de testare a ansamblului. În plus, funcția de export a datelor în format CSV permite utilizatorilor să efectueze analize externe, utilizând instrumente comune precum Microsoft Excel sau avansate cum ar fi Python, MATLAB, Tableau sau Power BI.

Din punct de vedere funcțional, răspunsul sistemului în condițiile de testare a fost consistent și recuperarea automată a funcționat în toate cazurile. Reacția senzorilor a fost monitorizată prin expunerea acestora la stimuli artificiali precum aer cald, aer rece și compuși volatili proveniți de la un marker. Valorile obținute au fost comparate cu cele ale unui dispozitiv etalon pentru a verifica abaterea senzorilor. Cu toate că senzorul DHT22 a raportat o valoare a temperaturii ușor superioară celei de referință, aceste abateri s-au încadrat în marjele de toleranță și nu afectează utilitatea generală a sistemului. În plus, a fost testat și comportamentul sistemului în cazuri limită legate de componenta software precum pierderea conexiunii Wi-Fi sau oprirea serverului. Rezultatul a fost unul favorabil, iar sistemul a încercat constant restabilirea conexiunilor fără a fi necesară intervenția manuală.

Proiectul a oferit o oportunitate valoroasă de a interacționa cu tehnologii reale, de la placa de dezvoltare ESP32-S2, senzori și protocoale de comunicație până la tehnologii de backend. Dificultățile întâmpinate au contribuit la înțelegerea aprofundată a importanței sincronizării componentelor și a implementării măsurilor de siguranță la nivel software.

În concluzie, sistemul demonstrează o funcționalitate stabilă și este scalabil, putând fi extins cu ușurință prin adăugarea de senzori suplimentari sau prin integrarea cu aplicații externe. Prezenta lucrare a vizat dezvoltarea unei soluții aplicabile fie în rol de extensie pentru o dronă de observație, fie ca nod autonom într-o rețea de senzori IoT.

*Instrumentele de inteligență artificială au fost utilizate doar pentru corectarea gramaticală și stilizarea formală a textului. Conținutul lucrării nu a fost redactat cu ajutorul acestora. Intervențiile au vizat claritatea, lizibilitatea și coerența exprimării, fără a modifica fondul ideilor.*