UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

CAPUEE: Final Project Report

Authors:
Luque Catena Raul
Mañé Fernandez Guillem
Sánchez Cuartas Miguel

Teachers:
Caprara Adriano
Micolau Puerto Marc

Academic year 2025-2026

Abstract

This project aims to raise awareness of the carbon dioxide (CO_2) emissions, measured in tonnes, generated by domestic electricity consumption, and to illustrate how this impact can vary throughout the day. To this end, the system is based on an ESP32 microcontroller and a personal computer, combined with a set of sensors and actuators, which enable or disable an electrical load depending on the CO_2 impact associated with the current energy mix. The project also includes a web-based interface for system control and historical data consultation. By linking real-time energy mix data with load control, the project demonstrates how electricity consumption decisions can be influenced by the temporal variability of carbon intensity.

Contents

List of Figures

1 Project Description and Workflow

1.1 Background and motivation

The objective of this project is to automatically optimize the energy consumption of a heating load based on its operating environment and the carbon dioxide (CO_2) emissions associated with the peninsular electrical system. The load consists of a resistive heating system designed to warm an indoor space using hot air. The software continuously monitors the room temperature in order to determine whether heating demand is critical, slightly below the target setpoint, or unnecessary. In addition, the system measures the electrical power consumption and takes into account the daily CO_2 emission levels of the electrical grid. This strategy allows the heating load to remain disconnected during periods of high carbon intensity when thermal comfort conditions are not critical, thereby contributing to a reduction in overall emissions.

The motivation behind this project is to raise awareness of energy consumption and its impact on the environment, while providing users with a convenient and automated solution to reduce their carbon footprint. By incorporating environmental indicators into the control strategy, the system enables end users to actively participate in emission reduction efforts without compromising comfort or requiring continuous manual intervention.

1.2 Project objectives and planing

For the development of the project two main thematic areas were defined: hardware and software. This separation enabled parallel development and a more efficient use of time. Project progress was monitored through periodic meetings and the use of a shared Git repository.

The validation of the proposed solutions was initially carried out individually and subsequently in a collective manner by integrating and evaluating the solutions developed by different team members. The system functionalities were added progressively following a modular approach, in which independent functional blocks were continuously incorporated and tested.

1.2.1 Hardware Layer (ESP32)

The ESP32 is responsible for sensor value acquisition, actuator control, and communication with the server. Its main functionalities are:

- Periodically read the current sensor value and transmit it via serial communication using the format CurrentSensor:XXX.
- Periodically read the temperature sensor value and transmit it via serial communication using the format TempSensor:XXX.
- Continuously monitor incoming serial commands to detect ACTIVATE_RELE or DESACTIVATE_RELE messages, activating or deactivating the relay accordingly.
- (Optional) Send relay state change notifications to a Telegram bot, such as *Relay activated* or *Relay deactivated*.
- (Optional) Receive commands from Telegram to manually change the relay state, generating a serial message so that the server-side software can detect the action.
- (Optional) Respond to Telegram requests for temperature and current measurements by acquiring and transmitting the corresponding sensor data.

1.2.2 Software Layer (Server)

The server is responsible for data processing, decision-making logic, and user interaction through dashboards and external services. Its main functionalities are:

- Read and parse serial data received from the ESP32, including `CurrentSensor:XXX` and `TempSensor:XXX` messages.
- Access the ESIOS API to obtain real-time information about the electrical grid and update a variable representing the current CO_2 emission level associated with the generation mix.
- Implement the decision-making logic for relay activation and deactivation based on temperature measurements, power consumption, and CO_2 emission levels.
- (Optional) Temporarily pause the automatic control logic for a predefined time interval (e.g., one minute) when the relay state is manually modified via Telegram.
- Send `ACTIVATE_RELE` or `DESACTIVATE_RELE` commands to the ESP32 through serial communication according to the control logic.
- Provide interactive dashboards for relay control, data visualization, and historical data analysis, implemented using Streamlit.

2 Project Development HW

The hardware architecture was designed to safely control and monitor a resistive heating load while acquiring environmental and electrical data. The system is composed of the following main elements:

2.1 ESP32 Microcontroller

The ESP32 acts as the central processing unit of the system. It is responsible for acquiring analog signals from the sensors through its ADC channels, executing the control algorithm, and driving the relay through a digital output. In addition, the ESP32 provides Wi-Fi connectivity, allowing the system to receive Telegram messages. Its computational capabilities and integrated network interface make it suitable for real-time control applications in smart energy systems.

Figure 1: ESP-32.

2.2 Heating Load

The controlled load consists of a resistive heating device that generates hot air to increase room temperature. From an electrical point of view, the load behaves predominantly as a resistive element, which simplifies power estimation from RMS current measurements. The load is connected to the mains supply through a Schuko socket and is electrically isolated from the low-voltage control electronics by the relay contacts.

Figure 2: Foto del calefactor.

2.3 Schuko Socket and Mains Wiring

A standard Schuko socket is used to connect the heating load to the mains supply. The phase conductor is routed through the relay contacts, allowing the ESP32 to enable or disable the power delivered to the load. Neutral and protective earth are directly connected to ensure safe operation. All mains wiring is physically separated from low-voltage electronics and mounted inside a protective enclosure. A fuse is placed in series with the phase conductor to protect against overcurrent conditions.

Figure 3: Socket used.

2.4 Relay Module

An electromechanical relay module is used as a low-voltage switching interface controlled by the ESP32. The relay coil is driven directly from a digital output using the onboard driver circuit, which provides sufficient current amplification for reliable actuation.

According to the manufacturer specifications, the relay is rated for 3.3 V DC and a maximum current of 10 A.

Mechanical separation between the control signal and the switched circuit provides basic electrical isolation, although the module is not intended for safety-critical mains applications. For real residential deployment, an additional certified power relay or solid-state relay rated for mains voltage would be required.

Figure 4: Relay DFR0473.

2.5 Current Sensor

Electrical current is measured using a split-core current transformer (CT) sensor, model CT013 with a nominal range of 30 A and an output of 1 V at rated current. The sensor clamps around the phase conductor of the heating load, allowing non-intrusive current measurement without electrical contact with the mains wiring.

The split-core design enables easy installation and preserves full galvanic isolation between the high-voltage power circuit and the low-voltage measurement electronics, which significantly improves safety during operation and testing.

This voltage-to-current proportionality allows the instantaneous current waveform to be sampled and processed in software to compute RMS current values over complete AC cycles.

Figure 5: Current sensor:CT013.

2.5.1 Current Sensor Calibration

To ensure accurate current measurements, the CT sensor was calibrated using a certified calibrated clamp ammeter as reference instrumentation. The clamp meter was placed on the same phase conductor as the CT sensor, allowing simultaneous measurement of the load current with both devices under identical operating conditions.

Figure 6: Clamp used to calibrate.

2.6 Temperature Sensor

Room temperature is measured using an analog temperature sensor placed close to the heating area but protected from direct airflow to avoid local overheating effects. The sensor output is directly connected to an ADC channel of the ESP32. Calibration was performed to compensate for sensor offset and to improve measurement accuracy near the operating temperature range of the system.

Figure 7: Sensor: DFR0024.

2.6.1 Temperature Sensor Calibration

To ensure accurate temperature measurements, the sensor was calibrated using a certified calibrated thermometer as reference instrumentation. The thermometer was measuring exactly the measuring point as the sensor.

Figure 8: Thermometer used to calibrate.

2.7 Analog Signal Conditioning Circuit

The CT sensor produces an AC signal centered around zero, which cannot be directly sampled by the ESP32 ADC. Therefore, a bias circuit is used to shift the signal to mid-supply level (approximately 1.65 V) using a resistive divider.

A burden resistor of 33 Ω converts the CT secondary current into a proportional voltage signal, scaled to approximately 1 V at nominal current. A decoupling capacitor of 33 μF stabilizes the reference voltage and reduces noise.

This conditioning allows safe and accurate sampling of the AC current waveform while preserving the galvanic isolation provided by the current transformer.

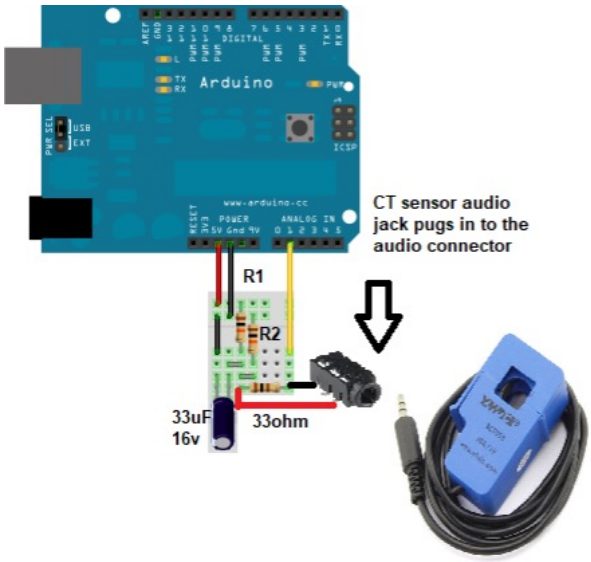


Figure 9: Circuit concept.

2.8 Power Supply

The control electronics are powered by an isolated low-voltage power supply that converts mains voltage to a regulated 5 V and 3.3 V suitable for the ESP32 and sensors. Electrical isolation between mains and low-voltage circuits is mandatory to ensure safe operation during testing and continuous use.

2.9 Personal Computer (PC)

A personal computer is used as an active component of the system to host the main control logic and the acquisition of real-time CO_2 emission data from online services. The PC executes the backend software responsible for processing environmental data, managing system states, and sending control commands to the ESP32.

Communication between the PC and the ESP32 is performed via serial or network interface, allowing the microcontroller to operate mainly as a sensor acquisition and actuation node. Therefore, continuous PC operation is required during system operation in the current prototype implementation.

2.10 System Integration and Safety Considerations

All components are integrated using a modular wiring approach to facilitate debugging and component replacement. Special attention is given to electrical safety, including proper wire gauge selection, insulation, grounding, and physical separation between high-voltage and low-voltage sections. These measures are essential when working with mains-powered loads in experimental laboratory environments.

3 Project Development SW

The complete source code developed for this project, including the ESP32 firmware, backend services, and frontend dashboard, is available in the following GitHub repository:

`https://github.com/gmane52/CAPUEE`

3.1 ESP32 Firmware

This subsection covers the complete firmware developed for the ESP32 microcontroller. It implements all the functionalities and objectives defined within the hardware layer of the project, including sensor data acquisition, actuator control, and bidirectional communication with the server-side software.

The main logic of this part of the software is shown in Figure10.

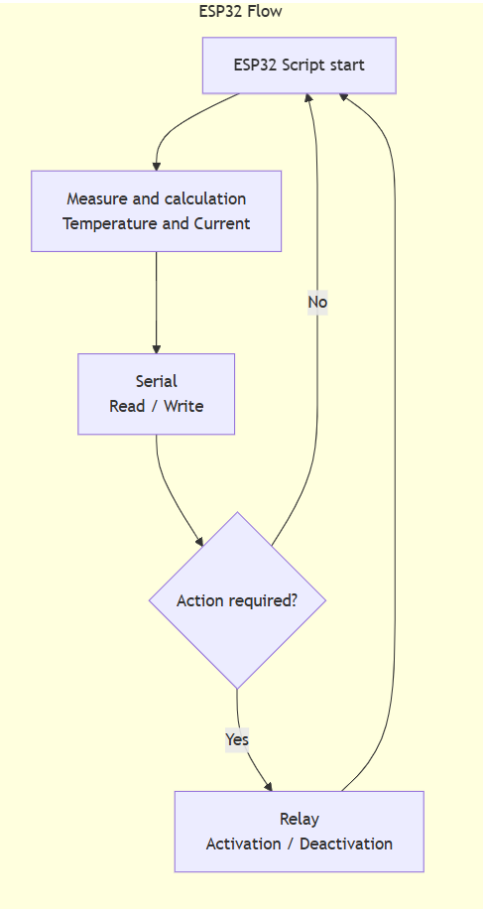


Figure 10: Clamp used to calibrate.

3.1.1 Communication and Connectivity

WiFi and Telegram Initialization

```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(500);
}

client.setInsecure();
bot.sendMessage(CHAT_ID, "ESP32 connected", "");
```

This block establishes the Wi-Fi connection required for remote communication and initializes the Telegram bot interface. Once the ESP32 is connected to the network, a confirmation message is sent to the user, indicating that the system is online and ready to receive commands.

3.1.2 Telegram Acquisition

Telegram Command Handling

```
void handleNewMessages(int numNewMessages) {
  for (int i = 0; i < numNewMessages; i++) {
    String text = bot.messages[i].text;

    if (text == "/on") {
      digitalWrite(relayPin, HIGH);
    }
    else if (text == "/off") {
      digitalWrite(relayPin, LOW);
    }
  }
}
```

This function processes incoming Telegram messages and allows the user to manually control the relay state using simple commands. This provides a remote override mechanism that can be used for testing and safety purposes.

3.1.3 Temperature Acquisition

Temperature Measurement

```
int adc = analogRead(sensorPin);
float voltaje = adc * (vref / 4095.0);
float temperatura = (voltaje * 100.0) + offset;
```

The temperature sensor is read using the ESP32 ADC with 12-bit resolution. The measured voltage is converted into temperature using a linear calibration model with an experimentally adjusted offset to compensate sensor inaccuracies.

3.1.4 Current Measurement and RMS Computation

Instantaneous Current Sampling

```
act_time = micros();
difTime = act_time - time_ant;

if (difTime >= 1000) {
  time_ant = act_time;
  double Vinst = read_voltage() - 1.65;
  double Iinst = Vinst * 30;
}
```

Current samples are acquired approximately every millisecond. The ADC voltage is first shifted to remove the DC bias and then converted into instantaneous current using the sensor scaling factor obtained during calibration. Then we calculate the RMS current value using the code learnt in class.

3.1.5 Control Logic and Output Management

Serial Communication with PC

```
if (Serial.available() > 0) {
  char command = Serial.read();

  if (command == 'H') {
    digitalWrite(outputPin, HIGH);
    Serial.println("GPIO 0 ON");
  }
  else if (command == 'L') {
    digitalWrite(outputPin, LOW);
    Serial.println("GPIO 0 OFF");
  }
}
```

This block enables bidirectional communication between the ESP32 and the PC through the serial interface. The microcontroller receives simple control commands from the backend software running on the PC and responds by activating or deactivating digital outputs.

This mechanism allows the external application to implement higher-level control logic, including environmental optimization and scheduling, while the ESP32 focuses on sensor acquisition and hardware actuation.

3.2 Server software

For this part of the project, the applied methodology played a particularly important role, as the coordination between the frontend and backend components and the use of multiple Python libraries required careful management. For this reason, documentation files and readme documents were created from the early stages of development in order to ensure proper coordination and traceability among all team members.

Additionally, Python virtual environments were used, along with systematic tracking of installed packages through a requirements file, in order to guarantee reproducibility and dependency consistency. The interactive dashboard and the core control logic were intentionally separated to improve modularity and maintainability.

To enable communication between both subsystems, a shared communication file was implemented, allowing real-time read and write operations by both components. Although more robust communication methods were considered given the limitations associated with simultaneous file access this approach was considered sufficient for the initial development phase of the project.

3.2.1 Backend

This subsection englobes the core system functionalities, including bidirectional communication with the ESP32, access to external APIs, and the decision-making logic responsible for the activation and deactivation of the electrical load. It also includes system configuration management and the control logic governing user interactions through the dashboard.

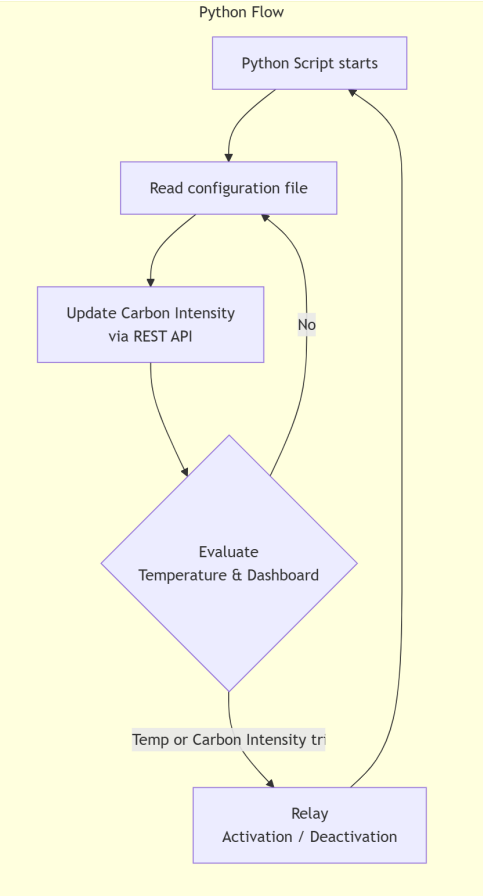


Figure 11: Backend logic scheme.

The main control logic of the backend is illustrated in Figure 11. It represents the decision-making process responsible for the activation and deactivation of the electrical load based on sensor measurements, grid emission data, and user-defined configuration parameters.

The diagram shows the presence of an infinite loop in which the system continuously operates. At the beginning of each iteration, configuration parameters and serial data received from the ESP32 are read and updated. Subsequently, the carbon intensity values are refreshed by connecting to the corresponding REST API.

The system then enters the logical evaluation stage, where decisions are made based on the measured temperature, the current carbon intensity level, and user actions performed through the dashboard. According to this evaluation, the relay is activated or deactivated, and the corresponding state-change command is sent to the ESP32 via the serial communication channel. Finally, the process repeats within the infinite loop.

The backend software can be divided into four main functional blocks: data acquisition, configuration management, control logic, and data logging, as illustrated in the diagram.

The data acquisition block is responsible for collecting information from multiple sources, including serial data received from the ESP32 and carbon intensity data obtained through a REST API service. The configuration block manages system parameters through a dedicated configuration file and allows switching between automatic and manual operating modes.

The control logic block implements the decision-making process based on temperature thresholds and carbon intensity levels, determining whether the electrical load should be activated or deactivated. Finally, the data logging block handles the storage of system data in CSV format, ensuring that all measurements and system states are recorded with appropriate timestamps for further analysis.

- `consulta_api_elecmap()`: Actualise the latest carbon intensity value from the Electricity Maps REST API and updates the internal state variable.
- `read_serial(ser)`: Parses incoming serial frames from the ESP32 (current and temperature) and updates the corresponding variables.
- `activate_relay(ser)` / `deactivate_relay(ser)`: Sends relay control commands to the ESP32 through the serial interface.
- `log_measurement()`: Appends timestamped measurements and system state to a CSV file for subsequent analysis.

The main part of the code is executed only once during initialization. This stage is responsible for opening the serial communication channel for subsequent use, updating the carbon intensity value, initializing and verifying the relay state, and checking for correct system activation. It also creates the logging file if it does not already exist.

Additionally, this section configures the scheduled tasks used throughout the execution of the program. Scheduling enables the definition of periodic tasks, which is essential for the efficient use of the external API and for ensuring proper synchronization of the logging processes.

Main part of the code

```
# ----- MAIN ----- #
## Open serial
ser = serial.Serial('COM7', 115200, timeout=1)
time.sleep(2)

## Set the initialy carbonIntensity value
Consulta_api_ElecMap()

## Configuration on ssheduled tasks
schedule.every(15).minutes.do(Consulta_api_ElecMap)
schedule.every(5).seconds.do(log_medida)

## Creates file for logginh
if not os.path.exists(csv_file):
    with open(csv_file, "w", newline = "") as f:
        writer = csv.writer(f)
        writer.writerow(["timestamp",
        "TempSensor", "CurrentSensor",
        "carbonIntensity", "ReleState"])

## Set the initial state of the rele:
DesactivarRele(ser)
time.sleep(2)
rele = False # False = open
```

Finally, the main loop of the code is presented. It can be observed that, at each iteration, the system first updates the parameters read from the `config.txt` file. Subsequently, the scheduled tasks are executed and the serial data received from the ESP32 is processed.

Afterwards, the control logic is evaluated, distinguishing between manual and automatic operating modes. In automatic mode, the temperature thresholds are first assessed to determine whether the relay should be activated or deactivated. Finally, the carbon intensity condition is evaluated. As shown in the code, the carbon intensity check is intentionally placed at the end of the loop and outside the temperature-based conditional logic, since it represents the primary criterion for interrupting power consumption. If this behavior is not desired, the relay can be manually controlled through the dashboard, or a configurable carbon intensity threshold can be defined to override the automatic disconnection logic.

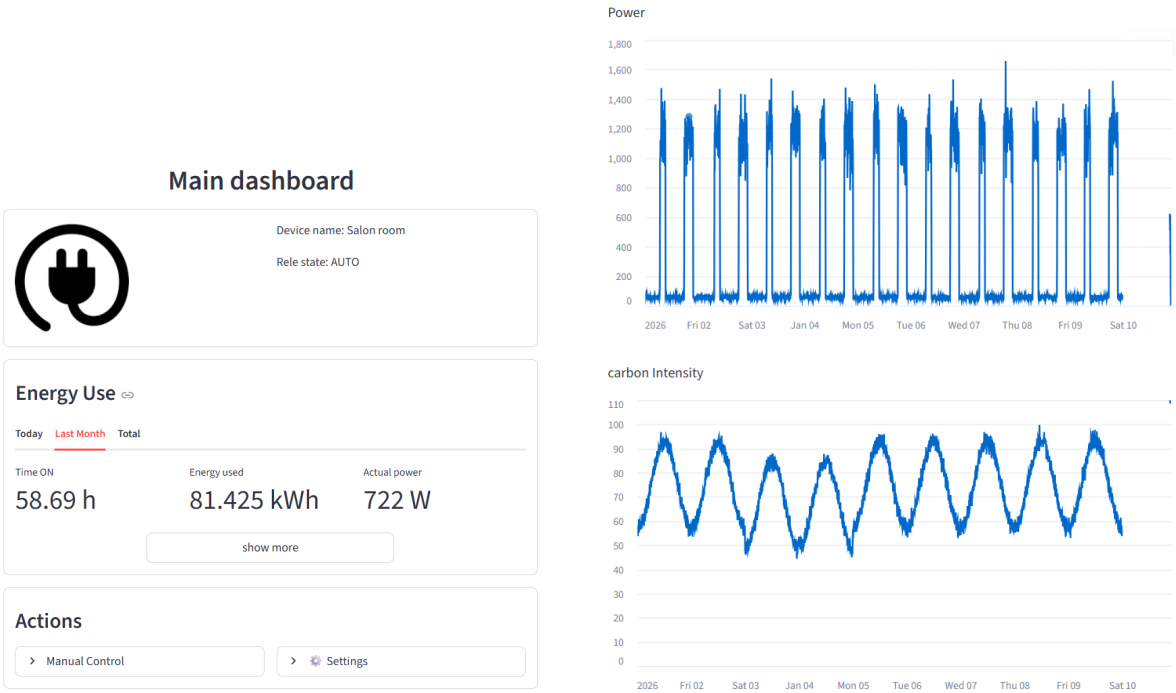
Simplified main control loop

```
## BUCLE
while True:
    try: # read CONFIG.txt
    except:
        pass

    schedule.run_pending()
    read_serial(ser)

    # manual control
    if CONTROL == "MANUAL":

    # auto control
    else:
        if TempSensor < TEMP_ON:
            ActivarRele(ser)
            rele = True

        elif TempSensor TempSensor > TEMP_OFF:
            DesactivarRele(ser)
            rele = False

        if carbonIntensity  < carbonIntensityMAX:
            DesactivarRele(ser)
            rele = False
    time.sleep(1)
```

3.2.2 Frontend

This section focuses on the interactive dashboard functionality, as well as the data processing and calculations required for the graphical representation of system variables within the web interface. It allows users to visualize real-time measurements, analyze historical data, and interact with the system in an intuitive and user-friendly manner.

As shown in Figure 12, the dashboard is divided into three main horizontal sections. The first section provides general information about the device, including its location, the current relay state, and a representative image. The second section is organized into three tabs corresponding to different time periods: current day, last month, and total accumulated data. The instantaneous power consumption is continuously displayed, and by selecting the *Show more* option, additional plots become available, including relay on/off time, power consumption over time, and carbon intensity values.



(a) General dashboard overview

(b) Historical data visualization

Figure 12: Frontend dashboard views

The final section of the dashboard, shown in Figure 13, corresponds to the actions and configuration panel. This section allows switching between manual and automatic control modes. Automatic operation follows the control logic described in previous sections, while manual mode enables direct relay actuation through the dashboard interface.

In addition, the settings panel allows modification of the `config.txt` file, enabling changes to parameters such as the device name, temperature thresholds, and carbon intensity limits, as well as the option to delete the stored dataset.

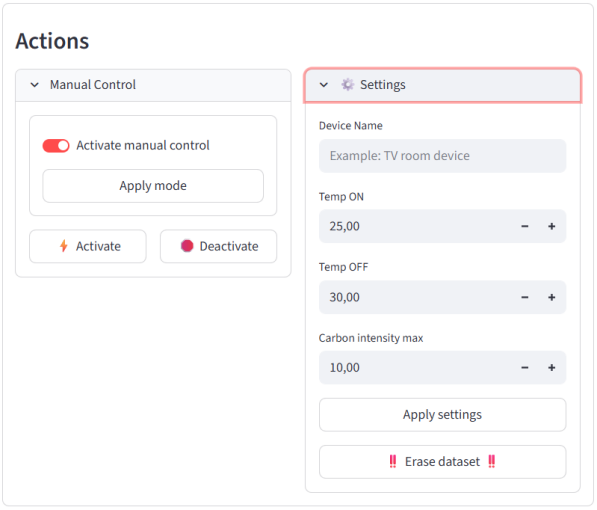


Figure 13: Frontend actions and configuration panel

In this part of the project, only the `pandas` library and the `Streamlit` framework are used. Pandas is employed to load the data obtained from the logging CSV file, compute instantaneous power values based on the most recent measurements, and perform the necessary data processing.

Using Streamlit in combination with pandas, all web interface functionalities are implemented, including data visualization and user interaction.

The following section presents an excerpt of the data processing required to calculate energy consumption and power profiles over different time horizons, such as daily, monthly, and total accumulated periods.

Calculation of energy for ploting in pandas

```
df = df.sort_values("timestamp").copy()
max_gap = pd.Timedelta("15min")

dt = df["timestamp"].diff()
on_mask = df["ReleState"].shift(1).fillna(False)

df["on_dt_calculado"] = dt.where((dt <= max_gap) & on_mask, pd.
    Timedelta(0)).dt.total_seconds().fillna(0)


# 3) Filtros
now = pd.Timestamp.now()
df_today = df[df["timestamp"].dt.date == now.date()]
df_month = df[(df["timestamp"].dt.year == now.year) & (df["
    timestamp"].dt.month == now.month)]

def resumen(d):
    time_on_h = d["on_dt_s_calc"].sum() / 3600
    energy_on_kwh = d["on_energy_Wh"].sum() / 1000
    return time_on_h, energy_on_kwh

time_total_h, energy_total_kwh = resumen(df)
time_today_h, energy_today_kwh = resumen(df_today)
time_month_h, energy_month_kwh = resumen(df_month)
```

4 Results

The system was tested with the heating load operating under different temperature and carbon intensity conditions. The obtained results can be observed in the recorded graphs.

As shown in Fig. 12, the heating load presents a clear ON/OFF behavior. When the relay is activated, the measured power and current rise to a constant value, which is expected for a mainly resistive load. When the relay is deactivated, the power drops to zero, confirming that the load is completely disconnected.

Fig. 12 shows the variation of carbon intensity throughout the day, with a clearly periodic profile. The influence of this parameter can be observed in same figure, where some heating cycles are skipped during periods of high carbon intensity, even if the temperature is close to the activation threshold.

Overall, the graphs confirm that the system correctly measures temperature and electrical consumption and adapts the operation of the heating load based on both temperature and carbon intensity.

5 Conclusion

This project successfully demonstrates the feasibility of implementing an energy-aware heating control system that combines environmental sensing with real-time external data sources. By integrating temperature measurements, electrical current monitoring, and CO_2 emission awareness, the prototype shows how control strategies can be adapted to reduce environmental impact while preserving acceptable user comfort.

The system architecture, based on a distributed approach using a PC backend and an ESP32 embedded node, proved to be effective for rapid development and testing. The ESP32 provided reliable sensor acquisition and actuation capabilities, while the PC handled higher-level logic, data processing, and user interaction. This separation simplified development and enabled flexible experimentation with different control strategies.

From a technical perspective, the project validated the accuracy of RMS current measurement using current transformer sensors and analog conditioning circuits, as well as the robustness of long-term averaging for noise reduction. The inclusion of manual override through Telegram commands ensured system usability and safety during operation.

However, several limitations were identified. The dependence on a continuously running PC, network connectivity issues, and the limited scalability of the prototype highlight the need for further system integration. In future versions, migrating the backend logic and CO_2 data acquisition directly to the ESP32 would significantly improve autonomy and reliability. Additional improvements could include a web-based user interface, support for multiple controlled loads, and long-term data logging for performance evaluation.

Overall, the project provided valuable practical experience in embedded systems, power measurement, communication protocols, and sustainable energy control. The obtained results confirm that environmentally aware control strategies are technically viable using low-cost hardware and can be extended toward real smart home energy management applications.