UNIVERSITÀ DI PISA

ENGINEERING DEPARTMENT

Internet of Things Project

Energy Saving for a Grain Dryer Furnace

Studenti:

Lorenzo Barci

Andrea Caporale

ANNO ACCADEMICO 2024/2025

Indice

1 Introduction

The *Energy Saving for Grain Dryer Furnace* project has been developed to optimize the operation of an electric grain-drying furnace in a small farm equipped with a photovoltaic system. Each day, following the harvest, the corn must be dried at a controlled temperature to ensure safe storage. This process takes several hours and represents a major share of the farm's energy consumption.

Initial analyses revealed that the furnace alone accounted for approximately 30% of the farm's total electricity use, making it a key target for energy savings.

To address this, the system integrates sensors that continuously monitor the photovoltaic output production, the furnace's power consumption, and ambient environmental conditions. A local processing unit analyzes this data in real time, combining it with short-term energy consumption forecasts to determine the optimal times to start or pause the drying cycle—favoring periods when energy demand are lower to avoid peaks and use solar energy in the best way.

All collected data and system events are stored in a database and visualized through an interactive dashboard, providing the user with a clear and immediate view of process status and past history.

1.1 Modularity and Scalability

The system is built from independent blocks. Sensors stream data and can be added or removed without any change to the core software, with only little modifications to the code. The decision logic hosts pluggable predictive models, while the user command interface allows to new machines to be substituted or coupled with minimal additional configuration. Database and dashboard remain separated from the field hardware, so the platform can scale to new environments, new assets, new algorithms, while keeping maintenance simple.

1.2 Other Use Cases

Small Enterprises —With or without solar power generation

With the system developed, a workshop, craft laboratory, or micro–industry equipped with a photovoltaic plant can have a furnace (or any other energy–hungry load machine) start during the hours of highest solar production, immediately lowering the energy drawn from the grid, and be always aware of the state of the total power consumption. Where photovoltaics are absent, the same controller leverages consumption to avoid peak and could be integrated with the tariff calendar: in that way, it shifts work cycles to the cheapest price bands and flattens power peaks with the prediction model, thereby optimizing energy use.

Apartment Buildings —Electric Vehicle Charging

In a condominium garage or parking lot, the system can be designed to allocates power to wallboxes according to two factors: the building's photovoltaic surplus (if available) or the most advantageous time slots. Cars begin charging when energy is cheapest or self–produced, and the algorithm could be adapted to distribute sessions fairly to ensure that every resident's battery is sufficiently charged, without exceeding the contracted power limit, generating peaks of consume and without manual intervention from the building manager.

Commercial Refrigeration —Optimization in Food Storage

In supermarkets, convenience stores, or cold storage facilities, refrigeration units represent a significant part of the electricity consumption. The system can dynamically adjust the operation of refrigeration compressors and cooling cycles based on the real-time availability of solar energy and predicted consumption peaks. When photovoltaic production is high, refrigeration units run at full capacity, storing cooling energy (exploiting thermal inertia), while during grid peak hours or tariff spikes, the system reduces compressor activity to avoid excessive costs and demand charges. This intelligent load shifting and peak shaving not only lowers energy bills but also extends the lifespan of refrigeration equipment by avoiding constant high loads, ensuring food safety without manual intervention.

2 Architecture of the Network

The designed system consists of an IoT network composed of six nodes running **Contiki-NG**: two sensors, two actuators, one intermediary node hosting the decision logic, and a *border router*. The border router exclusively connects the 6LoWPAN network to the cloud application via `tunslip`, acting as a bridge between the physical layer and the remote processing and visualization infrastructure.

The *sensor nodes* acquire simulated environmental information—temperature, humidity, solar production, and household energy consumption. One of the sensors runs a forecast of future solar production using a pre-trained neural model through **TensorFlow Lite Micro** executed directly on the embedded device. Both sensors periodically send their data to the intermediary node through CoAP messages formatted in JSON.

The *edge intermediary node* receives the aggregated data from the sensors and employs another neural model to estimate the expected energy consumption for the next hour. By combining the two forecasts, the node applies a decision logic that determines whether to activate the furnace; if so, it sends commands to the actuator node connected to it. It also informs the alarm node about the stability of the energy network based on the difference between forecast production and consumption. The edge node is additionally responsible for forwarding the collected data and the decisions taken to the CoAP server (**CoAPthon**) for persistence in the database.

The *actuator nodes* receive commands both from the intermediary node—via CoAP PUT to their resources—and from the cloud server, and perform simulated actions (turning the furnace or the alarm on/off) based on the received parameters.

The *remote control application* is built around the CoAPthon server. It is connected to the border router, which receives all data via CoAP and stores them in a MySQL database. The server also integrates a starvation-prevention logic for the furnace: if the furnace remains inactive for an extended period owing to unfavorable energy conditions, the server forces an activation to guarantee the minimum usage required by the farm. The remote control application additionally offers a **command-line interface (CLI)** that enables the user to change the state of the actuator nodes, update configuration parameters, and monitor the system in real time.

Finally, the data stored in the database are read and displayed through **Grafana**, providing the user with an intuitive graphical interface to analyze temporal trends of sensor data, furnace activations, and statistics on usage as well as forecasts of future production and consumption.

The following picture represents the scheme of the architecture:

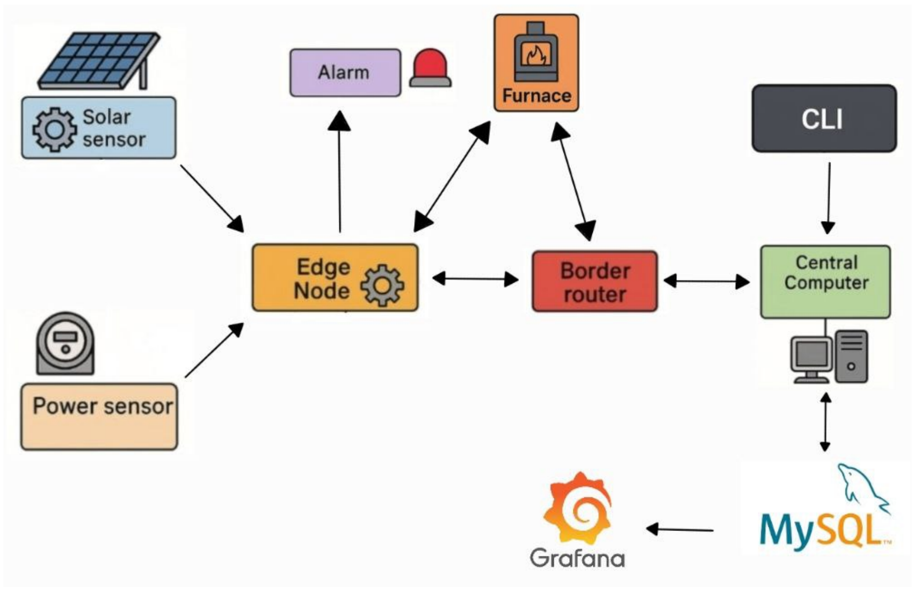


Figura 2.1: Scheme of the architecture

2.1 CoAP Network

At start-up each node performs a discovery phase to locate the *root* (in our case the border router) and registers its resources with the main server so that they can be discovered and reached by other nodes or by the server itself. Once registered, every node carries out specific actions.

2.1.1 Sensor Nodes

Roof Sensor Node Located on the roof, it measures atmospheric parameters (temperature and humidity) and the current photovoltaic power output. It executes a forecast of future solar production with an on-board regression model and sends the data to the edge node every hour via a CoAP PUT to the resource `/res_roof`.

Power Sensor Node Monitors the household's total energy consumption and transmits its measurements hourly to the edge node by issuing a PUT to the resource `/res_power`.

Edge Node

The edge node is the central element that receives sensor data, processes them with a pre-trained regression model, and takes control decisions. It exposes three CoAP resources:

- `/res_roof` – receives data from the roof sensor.

- `/res_power` – receives total electrical consumption data.
- `/res_threshold` – allows a remote client to update threshold parameters and to enable or disable the furnace's automatic control.

After obtaining readings from both sensors—or when the maximum waiting timer elapses—the edge node runs the regression, forwards raw data and results to the server, and executes the decision logic. It adjusts the state of the alarm and the furnace by sending PUT requests to `/res_alarm` and `/res_furnace` respectively. The node is also registered as an OBSERVER on `/res_furnace` and is notified whenever that resource changes owing to external events.

The resource `/res_threshold` conveys the configuration parameters `threshold_on` and `threshold_off` used in the decision logic. An additional parameter, `auto_furnace_ctrl`, indicates whether the node may autonomously change the furnace state; its value can be altered via remote server requests or by pressing the on-board button (a short press turns it on, whereas a three-second press turns it off).

2.1.2 Actuator Nodes

Furnace Actuator Node Receives on/off commands via CoAP PUT from the edge node, based on decisions taken by the internal model or the remote application. The furnace can also be turned on by pressing the physical button on the device (a short press activates it; a press of three seconds deactivates it).

Alarm Actuator Node Receives energy status notifications from the edge node and changes the color of the alarm light and, in a real-world scenario, triggers an acoustic signal in case of overload or blackout risk.

Border Router Connects the device network to external access, enabling communication with the Python-based application via CoAPthon.

All communication between nodes occurs via the CoAP protocol. This architecture allows for a clear division of responsibilities: sensors collect data, the edge node makes intelligent decisions, actuators execute physical actions, and the border router handles data transmission to the server and database.

Node LED Indicators

Each node is equipped with two programmable LEDs. Upon power-up, the little yellow LED lights up to indicate correct power supply. During root discovery the red LED blinks. Then, until the node has registered its resources to the main server, that RGB led remains off. Once registered:

- Sensor and edge nodes light the green LED steadily, indicating operational status.

- Actuator nodes light the LED that corresponds to their current status.

Furnace Actuator Node LED States

- **Off**: Purple LED
- **On**: Green LED

Alarm Actuator Node LED States

- **Optimal conditions for furnace activation**: Blue LED
- **Moderate electrical load**: Green LED
- **Near maximum load**: Red LED
- **Emergency (risk of cut-off)**: Purple LED

Additionally, sensor nodes and the edge node turn on the blue LED (instead of green) while performing an outgoing data transmission.

3 ML and Decision Logic

3.1 Data Preparation and Model Training

The HomeC dataset [1] was split into blocks, cleaned of any inconsistent values, and converted to an hourly resolution through statistical aggregations (sum for energy consumption, average for meteorological parameters, and mode for textual variables). Following this temporal normalization, one-step-ahead prediction labels were created, and basic temporal attributes (month, day, and hour) were added. After feature normalization, the dataset was split into training and test sets. A compact Multilayer Perceptron (MLP) was selected for its good trade-off between accuracy and model size.

Two separate MLPs were trained:

- The first takes as input: Solar [kW], Month, Hour, Temperature, and Humidity to predict `next_Solar [kW]`. It is executed on the Roof node after conversion into a C header using Emlearn, where it estimates solar irradiance for the next time step.
- The second takes as input: Difference [kW], Month, Hour, Temperature, and Humidity to predict `next_Difference [kW]`. It runs on the Edge node and forecasts the available residual power. Difference is the power consumption of the farm without the Furnace, to study and predict the trend of the rest of the energy consumption and decide when to turn on the Furnace.

Both models are quantized, occupy only a couple hundred kilobytes, and operate in real time directly on their respective nodes, reducing latency and traffic to the cloud.

3.2 Decision Logic on the Edge Node

After executing the model to predict `next_power` and receiving `next_solar` from the Roof sensor node, the Edge node performs a check to determine whether to activate the furnace and which alarm level to trigger.

- If $\texttt{next\_solar} - \texttt{next\_power} < \texttt{threshold\_on}$, conditions are optimal for turning on the furnace using solar energy and avoiding power peaks. The furnace is turned **on** (if automatic control is enabled), and the alarm is set to **Blue**.
- If $\texttt{threshold\_on} < \texttt{next\_solar} - \texttt{next\_power} < \texttt{threshold\_off}$, conditions are not optimal for maximum savings. The furnace is not turned on automatically, but if it is already on, it remains on to avoid frequent toggling. The alarm is set to **Green**.

- If `next_solar` $-$ `next_power` $>$ `threshold_off`, the conditions are unfavourable, and continuing to use the furnace might lead to overload. The furnace is scheduled for **shutdown**, and the alarm is set to **Red**.
- If `next_solar` $-$ `next_power` $>$ `threshold_cut`, the audible alarm is activated, and the alarm colour switches to **Purple**. The risk of overload in the next hour is high, so the furnace is forcibly turned **off**.

This logic enables optimal scheduling for activating the furnace—or any device connected to the IoT network—helping the company reduce energy consumption, maximize use of self-produced renewable energy, and prevent overloads before they cause blackouts.

The server-side logic further ensures that the furnace operates the required number of hours each day, preventing incomplete drying cycles by the time of unloading or loading new material the following day.

4 CoAPthon and Database

4.1 Python CoAPthon Server

4.1.1 General Structure and Service Launch

The server process defines a subclass `CoAPServer` of `CoAP`. Upon launch (in `__main__` mode), the server binds to IPv6 address `"::"` on port 5683 to be reachable in the network, and publishes five resources: `res_data/`, `res_prediction/`, `register/`, `lookup/`, and `starvation/`. In parallel, it starts two threads that observe the resources `/res_furnace` and `/res_threshold` on remote nodes to handle notifications. Finally, it enters the listening loop `listen(10)` until termination via terminal.

4.1.2 MySQL Integration

The module imports the file `db.py` which provides `connect_db()`. Each resource, in its constructor, creates the required table if it does not exist and, in the `render_*` methods, it opens a temporary connection, executes the INSERT/SELECT query, and closes the connection. This avoids persistent connections and ensures consistency between memory and the database.

4.1.3 Node Directory – `/register`

`RegisterResource` allows devices to self-register:

- It checks if the node is already present; otherwise, it stores its IP and declared resources in the database.
- It maintains an in-memory list (`registered_nodes`) and populates the `nodes` table with `node_id`, `node_ip`, and `resource`.
- It provides a GET endpoint that returns the current epoch timestamp for time synchronization of remote nodes at connection.

4.1.4 Data Collection from Nodes – `/res_data`

`ResData` receives sensor data aggregated by the Edge node (solar, power, temperature, etc.). It converts the timestamp to epoch seconds using `to_epoch_seconds()` and inserts the records into the `res_data` table. At the end, it invokes `avoid_starvation()` to apply the decision logic described in section 7.

4.1.5 Prediction Collection – `/res_prediction`

`ResPrediction` receives future forecasts (`next_power`, `next_solar`) and a `missing` flag from the Edge predictor node and stores them in the `res_prediction` table.

4.1.6 Discovery Service – `/lookup`

`LookupResource` receives queries in the format `?res=/resource_name` and searches the `nodes` table for the IP address of the device exposing that resource. If found, it returns a JSON object `"ip": "<addr>"`; otherwise, it responds with code 4.04.

4.1.7 Furnace Management and Anti-Starvation Logic

The server implements smart control logic for the electric furnace:

- **Remote Observation:** Using `HelperClient`, the server subscribes to notifications from `/res_furnace` (on/off status) and `/res_threshold` (automatic control enabled/disabled), so it is always aware of the furnace's status and configuration.

- **Periodic Logger:** A background thread logs the furnace status every 15 seconds into `furnace_log`, enabling real-time monitoring via Grafana and time-series visualization of furnace activity.

- `avoid_starvation` **Algorithm:** Maintains a sliding window vector of the past 24 hours to track furnace runtime, ensuring it is on at least `min_on` hours and does not exceed `max_on` hours before the `load_hour`. It sends a `PUT` to disable automatic control on the Edge node and forcibly turns the furnace on or off when needed: a counter determines how many hours remain to be scheduled and plans forced activation before the load hour. If the maximum is exceeded, it disables automatic control until the furnace is reset.

4.1.8 Configuration Endpoint – `/starvation`

`StarvationResource` provides GET/PUT methods to read or update the parameters `max_on`, `min_on`, and `load_hour` remotely. Changes take effect immediately in the above logic without restarting the server.

4.1.9 Utility Functions

- `to_epoch_seconds()`: Converts string-formatted dates to epoch seconds.

- `send_put()`: General-purpose function to send CoAP PUT requests to target nodes.

- `get_ip()`: Retrieves a node's IP address from the database given the resource name.

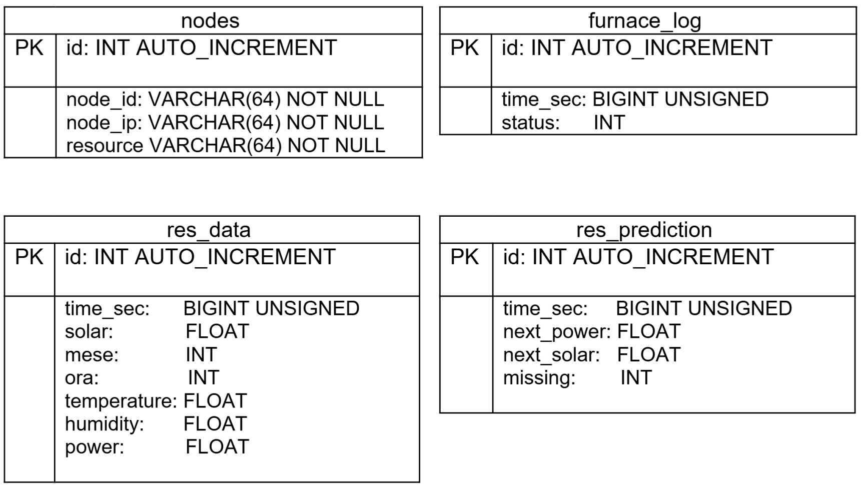
4.2 Command Line Interface: CLI

The functions made available to the user through the CLI are the following:

1. **Turn on furnace**: Immediately forces furnace activation, temporarily ignoring automatic logic.
2. **Turn off furnace**: Immediately forces furnace deactivation.
3. **Set activation energy threshold**: Allows setting a threshold value (in Watts) below which the furnace may activate in Auto mode.
4. **Set shutdown energy threshold**: Sets a power value (in Watts) above which, in Auto mode, the furnace is turned off to avoid energy peaks.
5. **Enable Auto Control**: Activates automatic control: the edge node continuously evaluates thresholds and use predicted values to decide ON/OFF.
6. **Disable Auto Control**: Disables automatic control; the furnace then responds only to manual ON/OFF commands, or forced activation/shutdown from the anti-starvation logic.
7. **Set max daily runtime**: Sets the maximum number of hours the furnace can operate within a 24-hour window.
8. **Set min daily runtime**: Sets the minimum number of hours the furnace must operate during the day.
9. **Set load time**: Saves the time the furnace is loaded; from this hour, the min/max settings from points 7 and 8 will be used by the server side logic.
10. **System Info**: Displays a complete summary: furnace status, Manual/Auto mode, thresholds in Watts, max/min hours for the day, load/unload time.
11. **Exit**: Closes the CLI and the Client process.

4.3 Database

The following picture shows a simple diagram of the DB schema:



nodes	
PK	id: INT AUTO_INCREMENT
	node_id: VARCHAR(64) NOT NULL node_ip: VARCHAR(64) NOT NULL resource VARCHAR(64) NOT NULL

furnace_log	
PK	id: INT AUTO_INCREMENT
	time_sec: BIGINT UNSIGNED status: INT

res_data	
PK	id: INT AUTO_INCREMENT
	time_sec: BIGINT UNSIGNED solar: FLOAT mese: INT ora: INT temperature: FLOAT humidity: FLOAT power: FLOAT

res_prediction	
PK	id: INT AUTO_INCREMENT
	time_sec: BIGINT UNSIGNED next_power: FLOAT next_solar: FLOAT missing: INT

Figura 4.1: Scheme of the architecture

As shown, we will have four tables:

- The `nodes` table is used during registration to insert the name of the CoAP resources and IP of the node having it. Here we register IP addresses and names of nodes that don't have resources too. For each resource, a record will be created even if a node has more than one resource.
- The `res_data` table is used to save the measurements done by the sensors. In particular, the temperature, humidity, total consumption, solar production, and the date and time of sampling will be recorded here. This table also includes `time_sec`, the timestamp coming from the Edge node, calculated by adding the seconds elapsed from the start of the process on the dongle to the real UNIX time sent by the server during registration. This value is used by Grafana.
- The `res_prediction` table stores the predictions made by the two Machine Learning models: one for the next solar production and one for the next total power consumption. The `time_sec` column links each record to the corresponding entry in `res_data`, and it is used by Grafana to display both actual and forecasted values on the same graph.
- The `furnace_log` table continuously saves the state of the furnace along with its corresponding timestamp, stored as `time_sec` for Grafana.

In each table, the `id` field serves as the primary key.

5 Design Decisions

5.1 Communication Protocol: CoAP

For communication within our IoT device network, we chose to adopt only CoAP (Constrained Application Protocol). This decision was motivated by architectural needs and advantages over alternatives like MQTT.

Using MQTT would have introduced a dependency on a central broker, requiring all pub/sub messages to pass through the main server acting as collector and parse manager. This approach results in redundant data flow:

- Sensor data would be sent to the broker on the main server.
- The server would then forward them to the Edge node hosting the regression model.
- The predicted value would be sent back to the server before being stored in the database.

This cycle introduces unnecessary latency, increases routing logic complexity, and reduces responsiveness.

Conversely, CoAP allows direct, lightweight communication between nodes, following a RESTful model similar to HTTP but optimized for low-power devices. Nodes can expose resources and perform GET, POST, PUT,or Observable actions simply, without needing an intermediate broker.

Advantages include:

- Minimal protocol and payload overhead, ideal for constrained networks.
- Direct device addressing, reducing hops and improving efficiency.
- Native support in Contiki-NG, including resource management and the observation model.

Thus, the Edge node can directly receive data from sensors, run regression locally using TensorFlow Lite, and send decisions to actuators without intermediate steps through a broker.

5.2 Data Format: JSON

For data representation and transmission among network nodes and to the server , we chose JSON as the unified format.

This choice was driven by practical and architectural factors: flexible and human-readable structure, easy parsing, and system-wide uniformity (all messages—sensor inputs, actuator outputs, CLI parameters—use the same format).

XML was not chosen due to its verbosity and overhead, which are unsuitable for resource-constrained IoT devices. XML parsing requires more memory and CPU compared to JSON and is less efficient over 6LoWPAN, especially alongside a lightweight protocol like CoAP.

Furthermore, for example, the CLI client interacting with `/res_threshold` or `/starving` resources can send updates such as:

```
{ "min_on": 4, "max_on": 10, "load_hour": 1 }
```

to configure parameters. Thanks to JSON's structure, it's also possible to specify a single parameter and use the same parser to detect what parameter has been changed.

6 Grafana

The Grafana dashboard is organized into five main widgets:

- **Solar:** Top-left shows real photovoltaic power (yellow line) alongside next hour forecast (green line).
- **Power:** To the right, a similar chart compares measured electrical consumption (blue line) with predicted demand (purple line).
- **Weather Conditions:** On the left, plots record environmental conditions: outside temperature (orange line) and humidity (pink line).
- **Use Furnace:** On the right, a pie chart summarizes the percentage of time the furnace was on (24% green slice) versus off (76% yellow slice), showing the total duty cycle.
- **State Furnace:** At the bottom, a binary ON/OFF graph tracks the furnace's state over time.

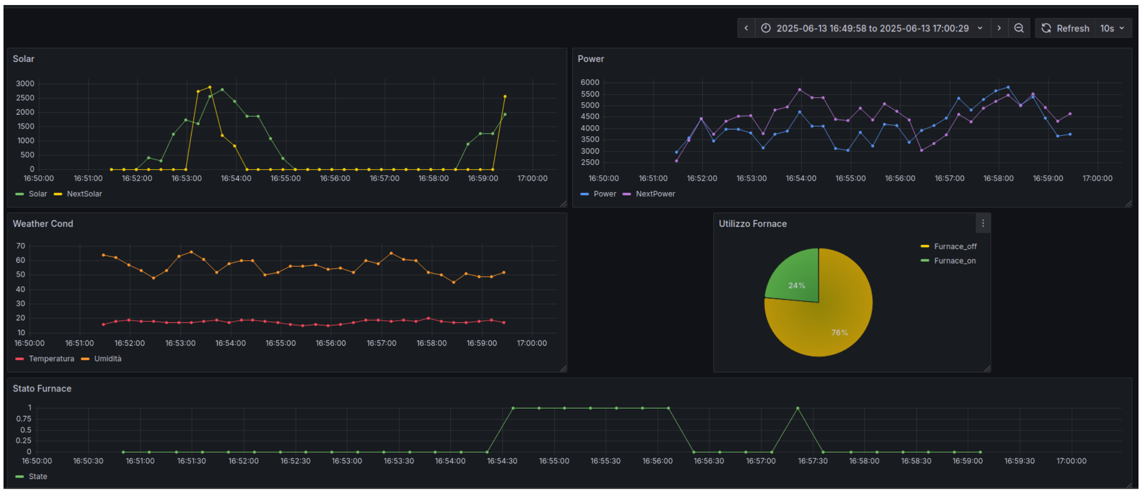


Figura 6.1: Dashboard of Grafana

Bibliografia

[1] Taranvee Singh. Smart home dataset with weather information, 2022. https://www.kaggle.com/datasets/taranvee/smart-home-dataset-with-weather-information.