Smart Irrigation System

Github Repository

https://github.com/BryantSarabia/SE4AS

Contents

System Description 4
Functional requirements 4
Real time monitoring 4
Automated watering 4
Weather integration 4
User interface 4
Data storage 4
Non-functional requirements 4
Reliability 4
Scalability 4
Performance 5
Compatibility 5
System components 5
Sensors 5
Irrigation devices 5
Monitor 5
Analyzer 5
Planner 5
Executor 6
Adaptation goals 6
Decision function 6
User Interface 7
Data storage 7
System architecture 8
Knowledge 8
Monitor 8
Analyzer 9
Planner 9
Executor 9
Simulator 9
Sensors 9
Actuators 9
Mosquitto 9
MongoDB 9
Initializer 9

Grafana 9
Data model 10
Addressing requirements 11
Real time monitoring 11
Automated watering 13
Weather integration 13
User interface 14
Data storage 15
System Technologies 16

System Description

Smart Irrigation System is a solution designed to optimize water usage in agricultural applications. By using a combination of sensors, data analytics and automated control systems, this smart system will ensure that plants receive the right amount of water at the right time.

The Smart Irrigation System continuously monitors environmental conditions through its network of sensors. When the soil moisture level drops below a certain threshold, the system triggers the irrigation devices to water the plants. Conversely, if the weather forecast predicts rain, the system can cancel watering to conserve water.

Functional requirements

Real time monitoring

The system continuously monitor environmental conditions through sensors, including soil moisture sensors, temperature sensors, humidity sensors and light sensors.

Automated watering

When the soil moisture level falls below a certain threshold, the rule-based decision function triggers irrigation devices to water the plants.

Weather integration

The system integrates with weather forecasts to adjust watering plans. If rain is predicted, the system may cancel watering based on rule-defined conditions.

User interface

The user interface provides real-time status updates of the system, allows users to adjust system settings, offer water usage consumption and influence the rule-based decision-making process.

Data storage

The system stores sensor data for further analysis and the system configuration settings. e.g., moisture level threshold.

Non-functional requirements

Reliability

The system must operate under varying environmental conditions.

Scalability

The system should be able to accommodate additional sensors or irrigation devices if necessary.

Performance

The system should be able to respond promptly to changes in environmental conditions.

Compatibility

The system should be compatible with a variety of sensors and irrigation devices.

System components

The smart irrigation system is composed of several key components such as:

Sensors

These are crucial for **monitoring** and gathering real-time data about environmental conditions. The main sensors are:

- Soil moisture sensor
- Temperature sensor
- Humidity sensor
- Light sensor

Irrigation devices

The irrigation devices are the physical components that deliver water to the plants. The executor instructs them when to start or stop watering.

Examples of irrigation devices:

- Sprinklers
- Drip irrigation lines

Monitor

The monitor receives data from sensors and actuators and stores it on the knowledge.

Analyzer

The Analyzer is responsible for **analyzing** the data gathered by the sensors, it also considers the weather forecast to determine the need for watering. It calculates the right amount of water in liters that is needed in each field.

Planner

Based on the analysis, the system creates a **plan** adhering to rule-based decisions**.** If the soil needs watering and no rain is forecasted, the plan will include turning on the irrigation system. If rain is predicted, the plan might cancel watering. The goal is to devise a plan that ensures the plants receive the right amount of water at the right time optimizing water usage.

Executor

The system then **executes** the plan based on rule-defined conditions. If the plan involves turning on the irrigation system, it sends a command to the irrigation devices to start watering. If the plan involves canceling watering, the system cancels the irrigation.

Adaptation goals

Goal	Description	Evaluation metric
Trigger irrigation devices	Trigger irrigation devices when the soil moisture level drops below a certain threshold and rain is not predicted in the next X hours (adjustable by the user).	$(Sml \leq Smt) \wedge \neg Rp$
Stop irrigation	Stop irrigation when the soil moisture level has reached the soil moisture threshold or rain is predicted rain in the next X hours (adjustable by the user) to avoid overwatering.	$(Sml > Smt) \vee Rp$

- *Sml*: Soil moisture level
- *Smt*: Soil moisture threshold
- *Rp*: Rain prediction

Decision function

The Smart Irrigation System employs a rule-based decision-making approach to determine optimal watering plans. The decision logic is governed by a set of predefined rules that consider environmental conditions, weather forecasts and user-defined settings.

Currently, there is an evaluation rule only for the soil moisture sensors. The current evaluation rules are the following:

- Soil moisture rule:
    - If the average soil moisture level is less than the soil moisture threshold and rain is not predicted, calculate the amount of water required to increase soil moisture to the soil moisture threshold and send the analysis to the planner.
    - Otherwise, if the average soil moisture level is greater than the soil moisture threshold, send a message to the planner to stop irrigation.

```python
def _determine_irrigation_action(
    self,
    soil_moisture_threshold: float,
    soil_moisture_threshold_avg: float,
    rain_prediction: bool,
    field: Field
) -> Optional[dict]:
    logger.info(f"Analyzing irrigation action: Smt: {soil_moisture_threshold}, Sml: {soil_moisture_threshold_avg}, Rp: {rain_prediction}")
    try:
        if soil_moisture_threshold_avg ≤ soil_moisture_threshold and not rain_prediction:
            water_need = self.calculate_water_required(
                soil_moisture_threshold_avg,
                soil_moisture_threshold,
                self.soil_capacity,
                field.soil_depth,
                field.area
            )
            return {
                "water_need": water_need,
                "action": "trigger_irrigation",
                "reason": "(Sml ≤ Smt) ∧ ¬Rp"
            }
        elif soil_moisture_threshold_avg > soil_moisture_threshold or rain_prediction:
            return {
                "action": "stop_irrigation",
                "reason": "(Sml > Smt) ∨ Rp"
            }
        return None
    except Exception as e:
        logger.info(f"Error determining irrigation action {e}")
```

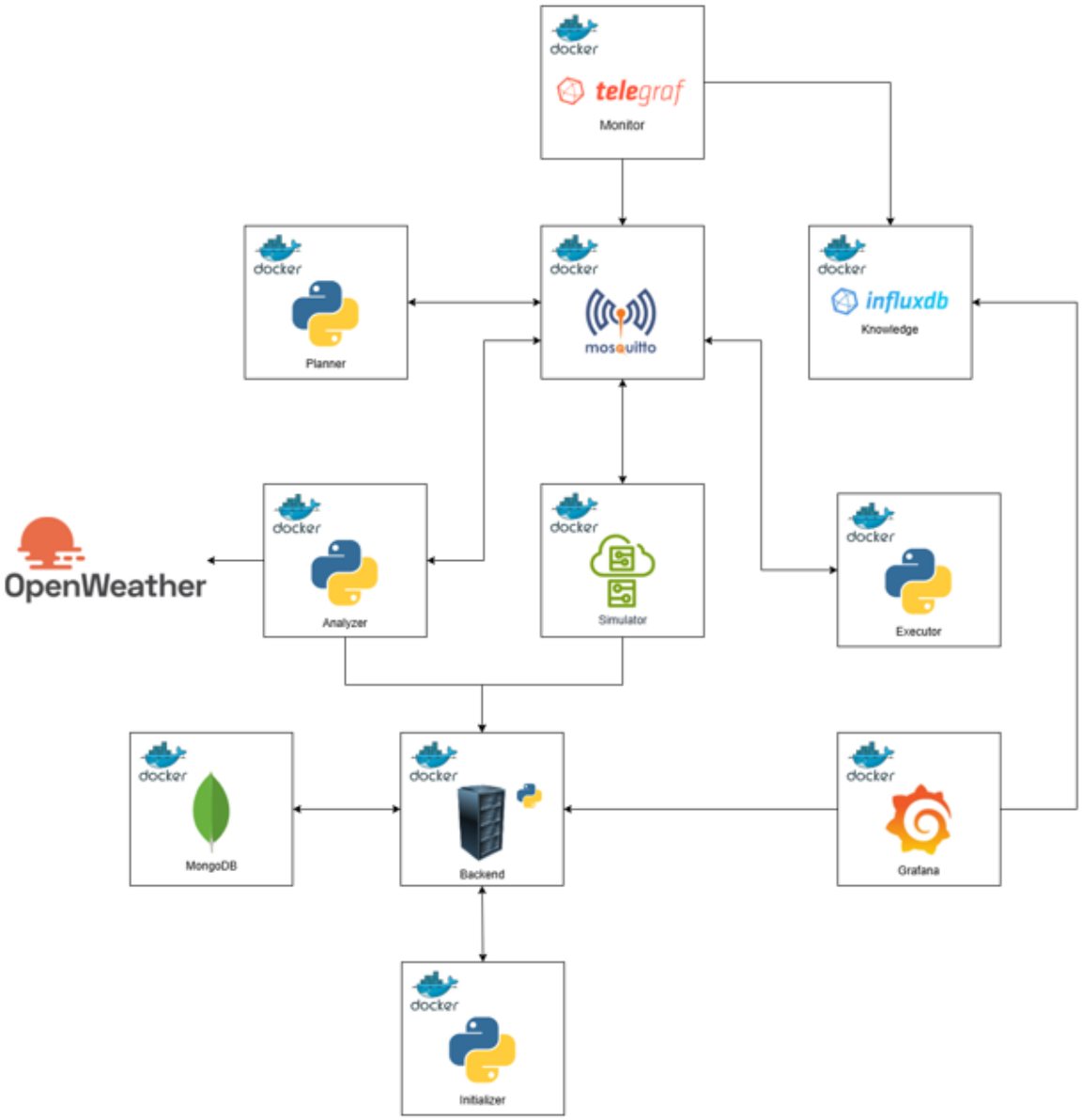
User Interface

The user interface allows users to monitor the system's status, adjust settings, and view water usage reports.

Data storage

The data collected by the sensors and actuators a will be stored on the knowledge database and the system configuration will be stored on MongoDB.

System architecture



Knowledge

For the knowledge, we opted for the InfluxDB database, which is a timeseries type and meets our performance and data modeling requirements for the data coming from the monitoring component.

Monitor

The monitoring component in our system is represented by a Docker container that includes the Telegraf plugin. This plugin observes the various MQTT communications sent from the zones, extracts the relevant data, and subsequently stores them in the knowledge base. The choice of this component stems from its perfect synergy with influxdb.

Analyzer

The analyzer is a python application that receives messages from the sensors and actuators via MQTT and sends. It analyzes the data considering the system settings, the current values of the sensors and the weather conditions of the zone, the output is then sent to the Planner via MQTT.

Planner

The Planner is a python application that runs on a container. It receives from the analysis performed by the Analyzer; it applies a policy and then sends it to the Executor.

Executor

The executor is a python application running on a container. It receives commands from the Planner and sends them to the actuators.

Simulator

The Simulator is a python application running on a container. It gets the system settings from the Backend and starts the simulation. It simulates the sensors and actuators.

Sensors

The sensors publish to MQTT the values read by them. These values are simulated if the actuators are not sending data, otherwise the value is calculated considering the values sent by the actuators.

Actuators

The sensors are managed by the executor. They perform actions based on the commands sent by the executor.

Mosquitto

Mosquitto is used to create the MQTT broker for the system.

MongoDB

MongoDB is used to store the system information such as zones, fields, sensors, actuators and settings.

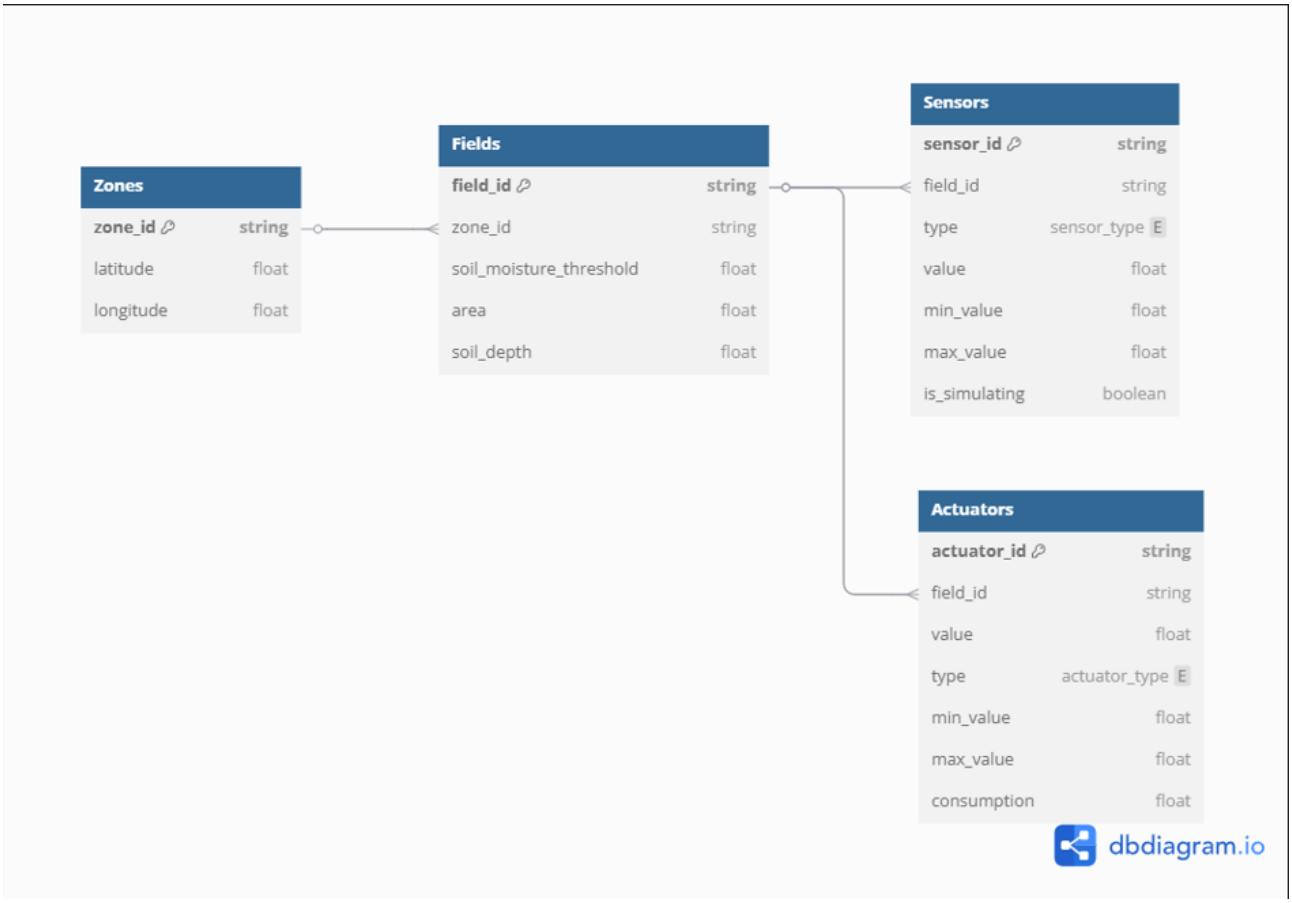
Initializer

The initializer is a python application running on a container. It follows the init container pattern. Its function is to populate the system configuration if it has not been loaded yet.

Grafana

Grafana is used to monitor the zones, fields, sensor and actuators data. It also is used to update the soil moisture threshold.

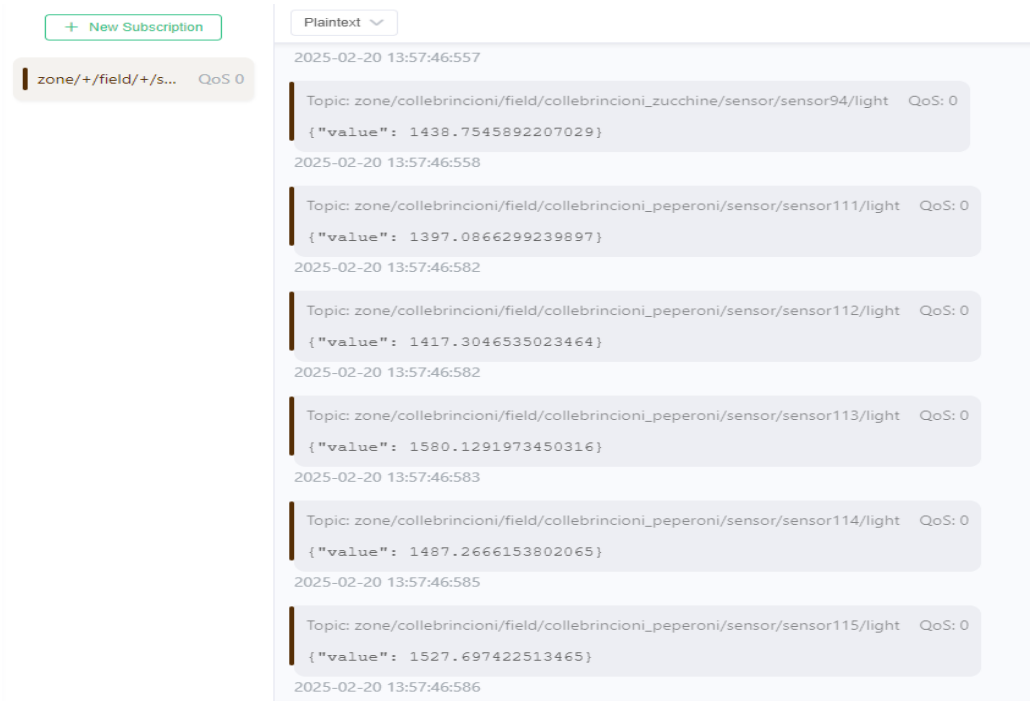
Data model


Zones
zone_id string
latitude float
longitude float
Fields
field_id string
zone_id string
soil_moisture_threshold float
area float
soil_depth float
Sensors
sensor_id string
field_id string
type sensor_type E
value float
min_value float
max_value float
is_simulating boolean
Actuators
actuator_id string
field_id string
value float
type actuator_type E
min_value float
max_value float
consumption float
dbdiagram.io

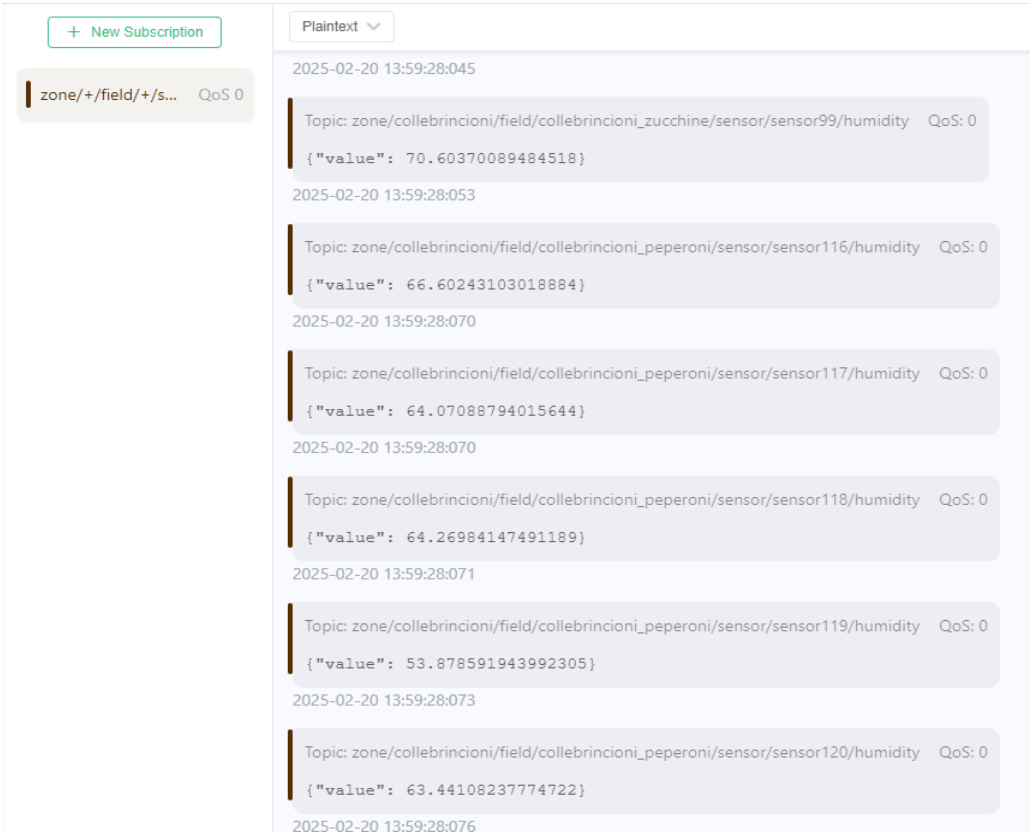
Addressing requirements

Real time monitoring

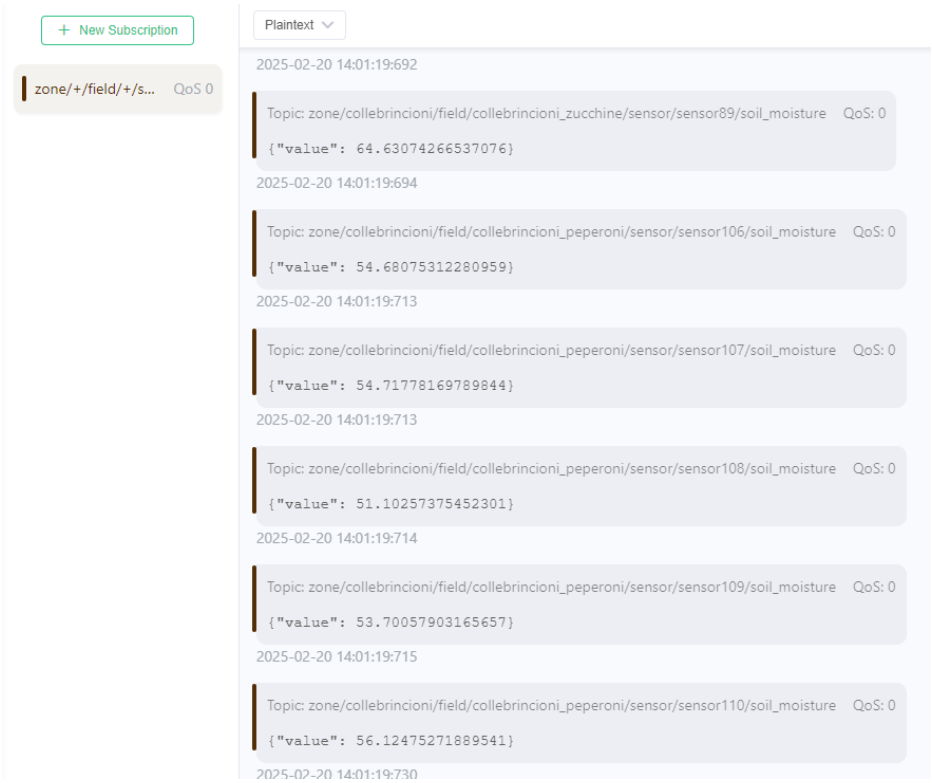
The figure below depicts the values of light intensity (lux) in the field which have been published by the sensor to the Mosquitto MQTT Broker.



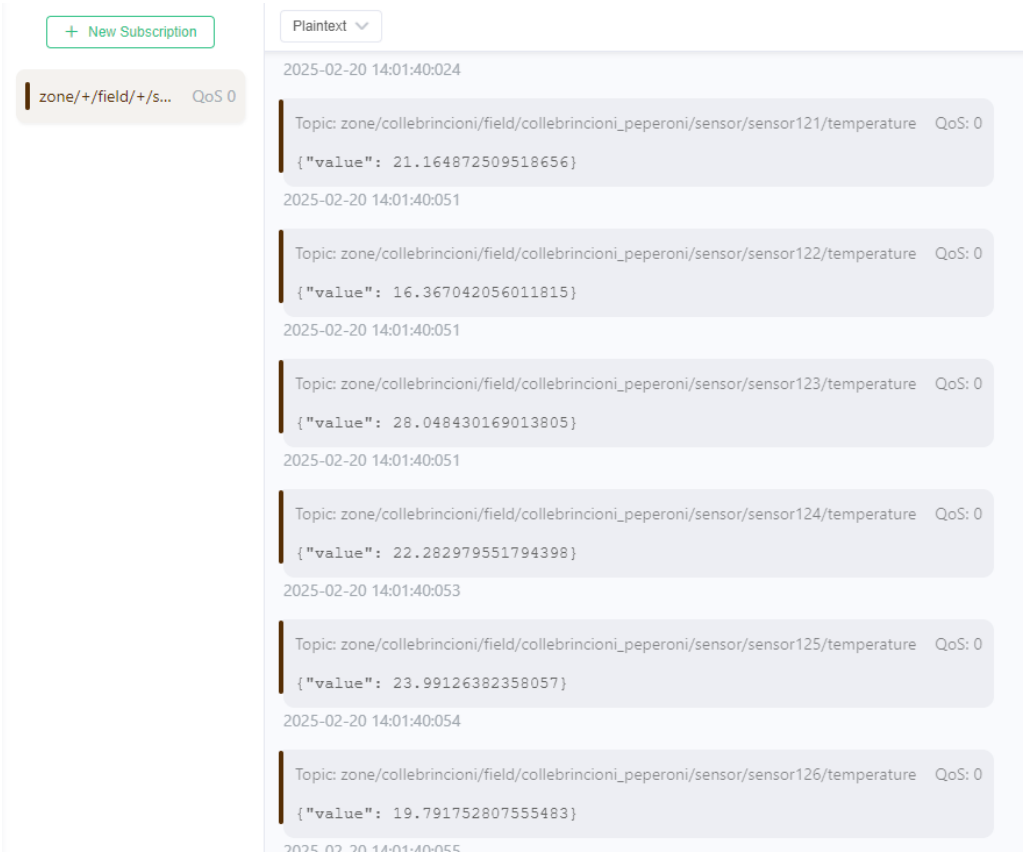
The figure below depicts the values of humidity (%) in the field which have been published by the sensor to the Mosquitto MQTT Broker.



The figure below depicts the values of soil moisture (%) in the field which have been published by the sensor to the Mosquitto MQTT Broker.

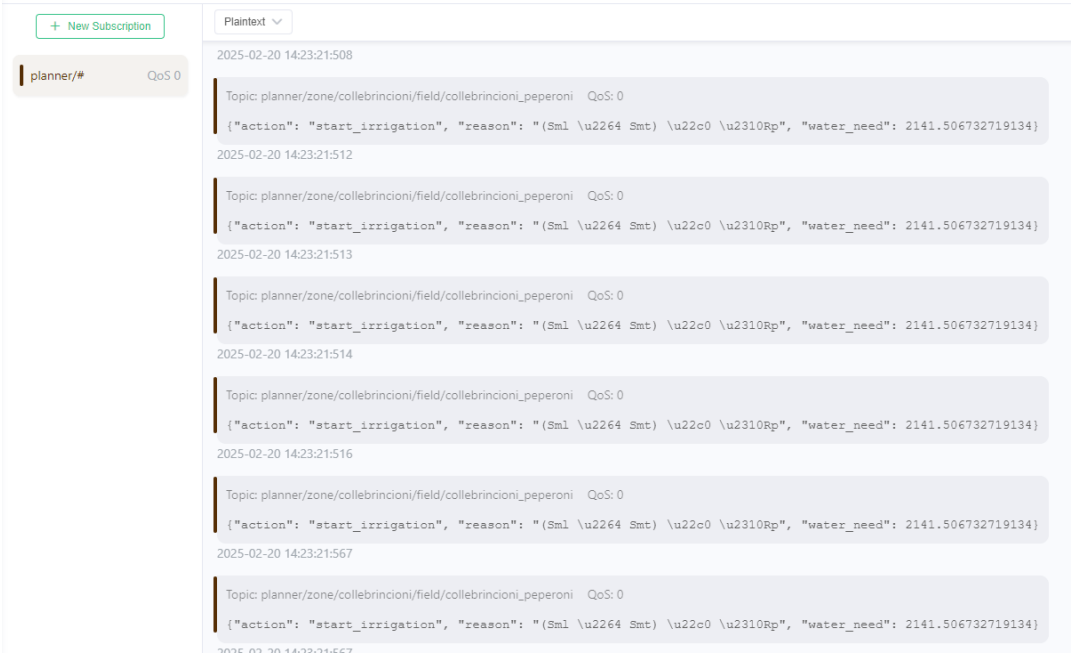


The figure below depicts the values of temperature (°C) in the field which have been published by the sensor to the Mosquitto MQTT Broker.



Automated watering

The figure below depicts the actions taken by Planer after it has evaluated the analysis from the Analyzer.



Weather integration

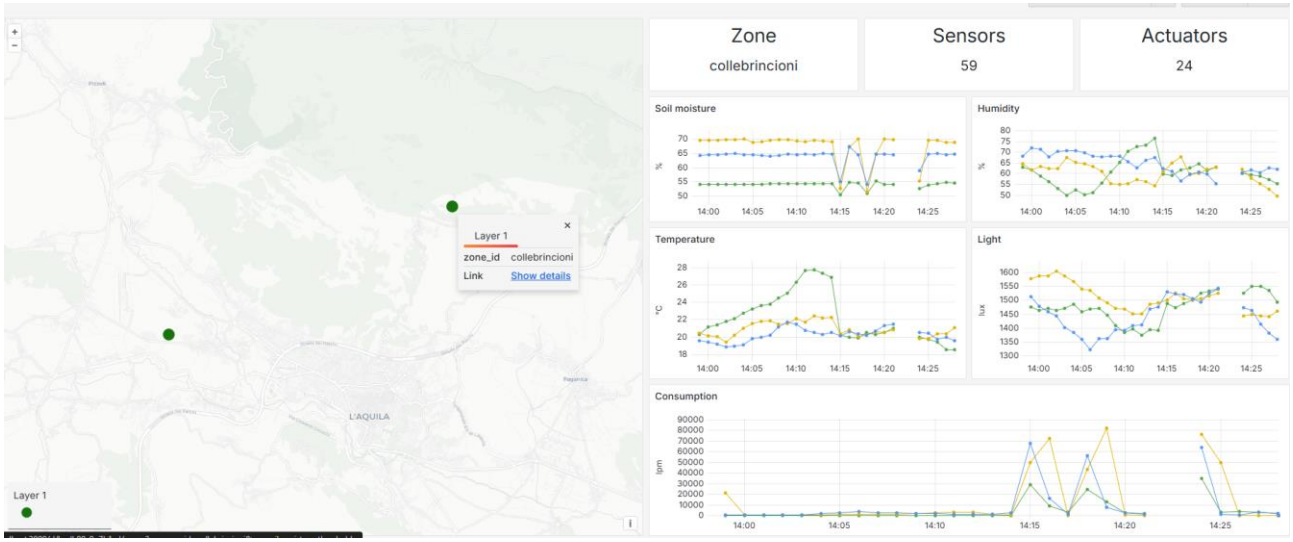
The analyzer gets weather data from OpenWeather by using the WeatherFetcher class as shown below. The WeatherFetcher class uses a cache with a max size of 1024 items and a time to live of 600s.

```python
def _is_rain_predicted(self, lat: float, lon: float) -> bool:
    weather_data = self.weather_fetcher.get_weather(lat, lon)
    if not weather_data:
        return False
    try:
        offset = min(self.config.NEXT_HOURS, len(weather_data['list']) - 1)
        return 'rain' in weather_data['list'][offset]['weather'][0]['description'].lower()
    except Exception as e:
        logger.error(f"Error checking rain prediction: {e}")
        return False
```

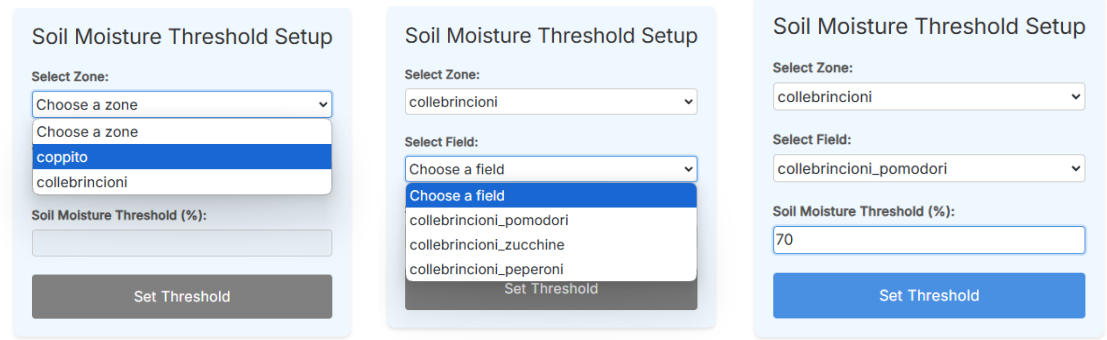
```python
BASE_URL = "http://api.openweathermap.org/data/2.5/forecast"

You, 21 hours ago | 1 author (You)
class WeatherFetcher:

    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = BASE_URL

    def fetch_weather(self, lat, lon):
        params = {
            'lat': lat,
            'lon': lon,
            'appid': self.api_key,
            'units': 'metric'
        }
        response = requests.get(self.base_url, params=params)
        if response.status_code == 200:
            data = response.json()
            return data
        else:
            print(f"Failed to fetch weather data: {response.status_code}")
            return None

    # Cache the weather data for 10 minutes
    @cached(cache=TTLCache(maxsize=1024, ttl=600))
    def get_weather(self, lat, lon):
        return self.fetch_weather(lat, lon)      feat: enhance rain prediction log
```

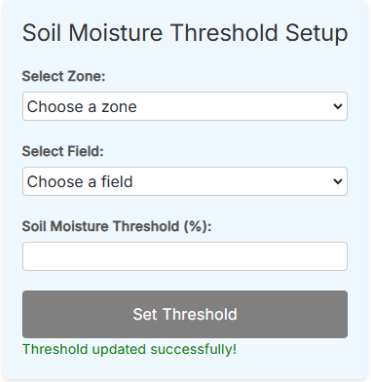
User interface

The UI provides real-time status updates, enables users to monitor the system, adjust soil moisture thresholds and access water consumption information. The zones are displayed on a map and when user clicks on “show details” the data of the charts is updated automatically.



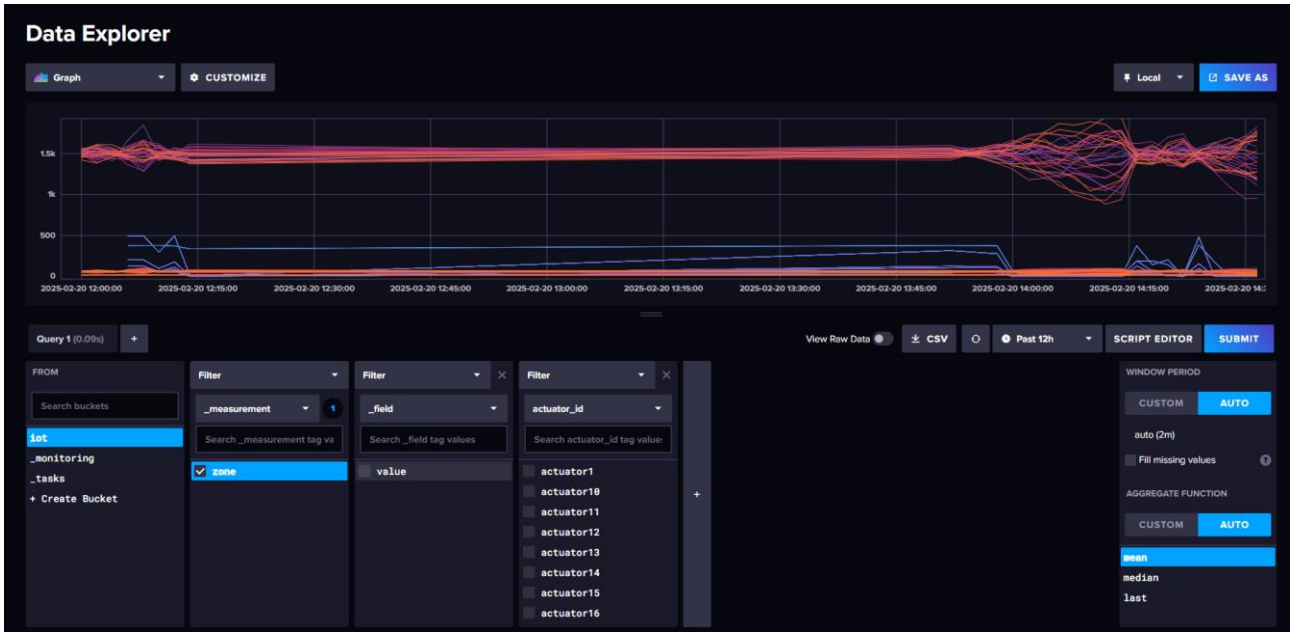
Users can update the soil moisture threshold from the user interface.

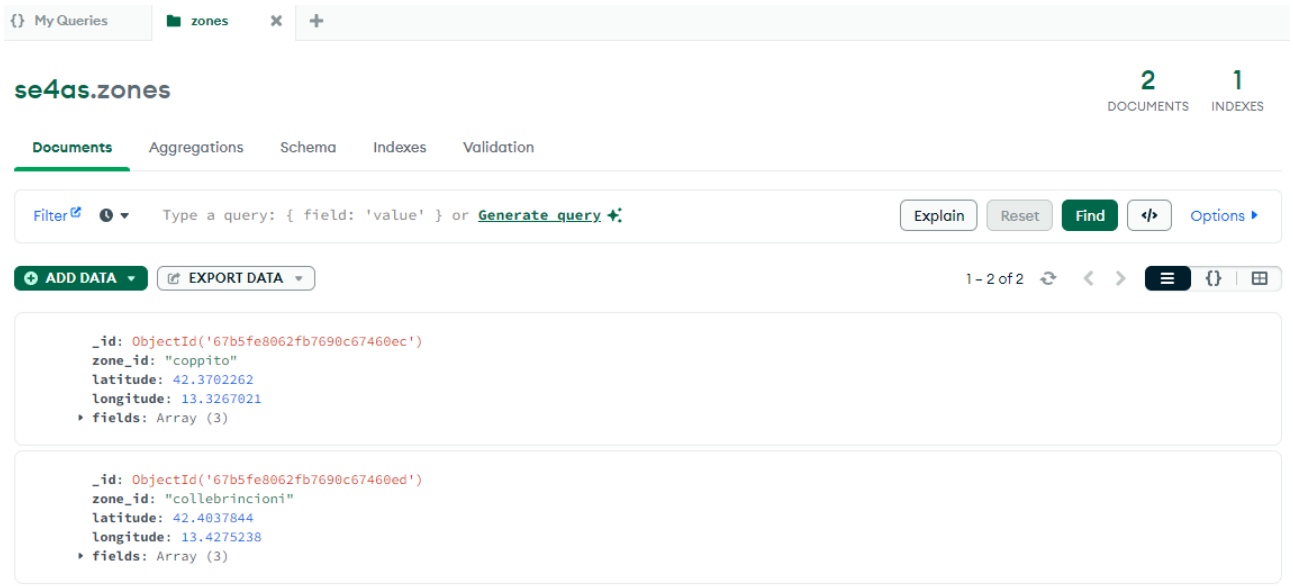




Data storage

The system stores the sensors and actuators data on Influxdb and system settings on MongoDB.





System Technologies