IoT-Based Decision Support System for Room Selection

Group 3

Fedor Chikhachev
Tarik Tornes
Thuc Kevin Nguyen
Vladyslav Siulhin

Introduction to IoT
February 2, 2025

1 Project Objective

The project involves designing and implementing a decision support system to help select the most suitable room in the building. The selection is based on:

- **Environmental factors:** Temperature, CO_2 levels, air quality, sound, light, VOC and humidity.
- **Room facilities:** Availability of equipment (e.g., projectors, seating capacity, microphone, blackboard).
- **User preferences:** Customized priorities set by the end-user.

2 System Overview

The system follows a modular architecture to ensure scalability and maintainability. It consists of various components that interact through well-defined communication channels. Docker is used to containerize services and manage dependencies efficiently.

Docker-Based Architecture To ensure modularity, scalability, and simplified deployment, the system is built using Docker containers. The key components include:

- **Mosquitto (MQTT Broker):** Manages communication between IoT sensors and backend services.
- **InfluxDB (Time-Series Database):** Stores real-time sensor data for processing.
- **PostgreSQL (Relational Database):** Stores static room-related data such as equipment availability.
- **MQTT to InfluxDB Service:** Subscribes to MQTT topics, processes sensor data, and stores it in InfluxDB.

- **Grafana (Monitoring Dashboard):** Provides real-time visualization of sensor data.
- **REST API for Room Data:** Exposes endpoints for retrieving and managing room data.
- **Booking System:** Handles room reservations and integrates with Google Calendar.
- **Dashboard UI:** Allows administrators to monitor room conditions and system health.
- **Static Data Service:** Initializes PostgreSQL with pre-defined room information.
- **Serial MQTT Bridge:** Bridges serial data from sensors to the MQTT broker.

3 Containerized Database Initialization and Setup

3.1 Overview

To ensure a reproducible and efficient database setup, we use Docker to containerize the database initialization process. The Dockerfile defines how dependencies are installed and manages the execution of the init_db.py script, which creates and populates tables in PostgreSQL.

3.2 Workflow and Implementation

1. The Dockerfile sets up a Python-based environment with PostgreSQL dependencies.
2. The init_db.py script connects to the PostgreSQL database and initializes tables for rooms, equipment, and sensors.
3. A configuration file (`config.json`) provides predefined room metadata, available equipment, and sensor details.
4. The database container runs automatically when launched, ensuring all components are initialized correctly.

3.3 Database Initialization Process

The **init_db.py** script is responsible for creating and populating the PostgreSQL database:

- Connects to the PostgreSQL database using SQLAlchemy.
- Creates tables for rooms, equipment, and sensors.
- Reads configuration data from a JSON file to initialize the database with predefined room attributes.
- Ensures atomic transactions, preventing partial database updates.

3.4 Example JSON Configuration

The initialization script reads from `config.json`, containing room and sensor details:

```
{
    "rooms": [
        {
            "name": "MSA3500",
            "equipment": [
                {"name": "Projector", "value": true, "type": "boolean"}
            ],
            "sensors": [
                {"name": "Temperature", "range": [18, 28]}
            ]
        }
    ]
}
```

3.5 Key Benefits

- Automated Deployment: Ensures consistent database setup in any environment.
- Scalability: Allows database configuration to be modified via `config.json` without altering the script.
- Reproducibility: Docker ensures that dependencies and scripts run consistently across deployments.

4 MQTT Client and InfluxDB Integration

4.1 Overview

To handle time-series data from sensors transmitted via the MQTT protocol, we use InfluxDB, a time-series database optimized for large volumes of real-time data. InfluxDB offers efficient data storage and retrieval capabilities, making it suitable for IoT applications. Additionally, it integrates smoothly with monitoring tools and platforms like Grafana, enabling advanced data visualization and analysis.

4.2 Module Description

This module functions as a subscriber within the publish-subscribe model of MQTT. It listens to sensor data generated by our simulation framework and stores the measurements in an InfluxDB instance for further analysis.

4.3 Data Flow and Processing

4.3.1 Subscription to MQTT Topics

The module subscribes to MQTT topics following the convention:

```
+/sensors/#
```

This pattern matches all sensor data topics defined by the simulator (e.g., `MSA1601/sensors/temp`, `MSA3500/sensors/co2`).

4.3.2 Data Extraction

Each received message contains:

- The topic, which follows the format {`room_id`}`/sensors/`{`sensor_type`}.
- The message payload, which includes:
  - **Value:** The sensor reading.
  - **Timestamp:** The time when the data point was generated.

The module parses the topic to extract:

- **room_id:** The unique identifier for the room.
- **sensor_type:** The type of sensor.

4.3.3 Data Storage

Parsed data is inserted into the `room_data` measurement in the InfluxDB instance.

The data schema includes:

- **Tag:** room_id.
- **Fields:** A key-value pair where the key is sensor_type and the value is the sensor reading.
- **Timestamp:** Derived from the message payload.

4.3.4 Continuous Operation

The subscriber runs indefinitely to ensure real-time data processing. This is achieved using the `loop_forever()` method from the `paho.mqtt` Python package. The method keeps the MQTT client in a continuous listening state, enabling it to handle incoming messages without interruption.

4.3.5 Technical Implementation

A single-file Python script continuously listens for updates on subscribed topics. The following Python packages are used:

- **influxdb_client:** For managing data insertion into InfluxDB.
- **paho.mqtt:** For MQTT client operations and message handling.

5 Dashboard for Equipment and Sensor Simulation

5.1 Overview

This module is designed to manage and monitor room equipment while simulating sensor device behavior in an IoT environment. The main objective is to simulate real-world conditions, enabling testing and validation of other project components for feasibility and applicability. It offers both API and Streamlit-based user interfaces to handle room data and equipment information, structured into several submodules.

5.2 Sensor Simulator

The Sensor Simulator is responsible for emulating sensor devices. Originally, the project required integration with Arduino-based devices to measure room conditions. However, due to hardware constraints, a simulation framework was developed to generate sensor data based on predefined decision logic. The simulated data mimics real-world conditions for the following parameters:

- Temperature
- Humidity
- Light intensity
- CO_2 levels
- PM2.5 and PM10 particle concentrations
- VOC (Volatile Organic Compounds) levels

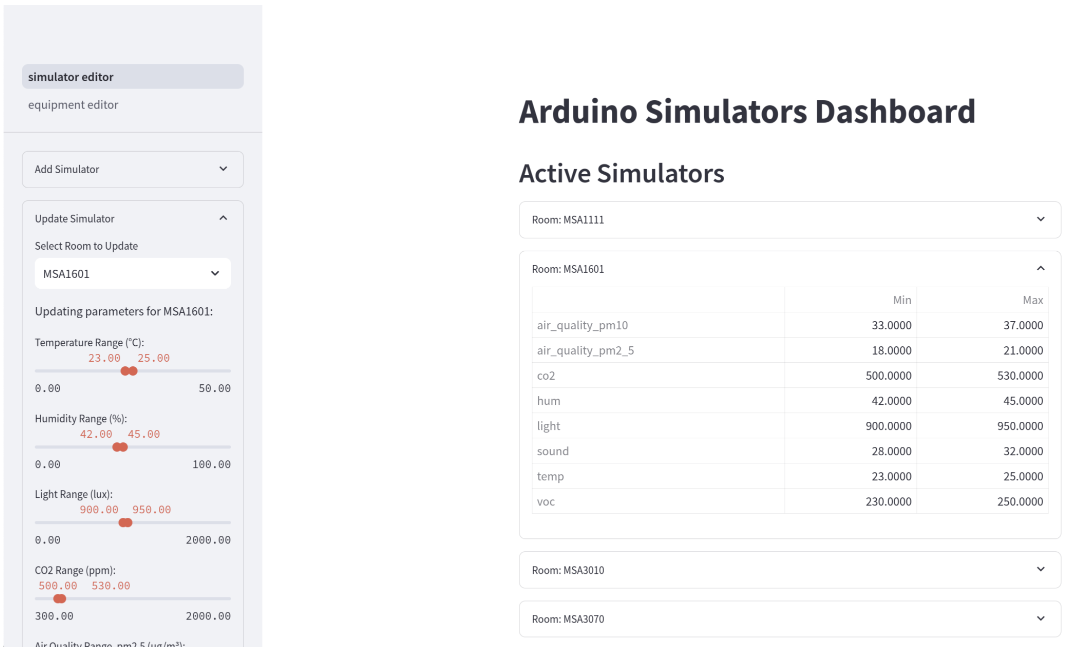
5.3 How It Works

- **Multithreading:** Each simulated device runs asynchronously on its own thread to enable concurrent sensor operations.
- **Configuration:** Initial parameters for the simulations are retrieved from a PostgreSQL database through an auxiliary REST API.
- **Data Transmission:** Simulated data is published to a Mosquitto MQTT broker, mimicking how physical microcontrollers or embedded systems would transmit real sensor data.

5.4 Why MQTT?

MQTT was chosen due to its efficient publish-subscribe model. The topic structure follows:

```
{room_id}/sensors/{sensor_name}
```



Select Room to Update: MSA1601

Updating parameters for MSA1601:

Temperature Range (°C): 23.00 25.00 (0.00 – 50.00)

Humidity Range (%): 42.00 45.00 (0.00 – 100.00)

Light Range (lux): 900.00 950.00 (0.00 – 2000.00)

CO2 Range (ppm): 500.00 530.00 (300.00 – 2000.00)

Air Quality Range, pm2.5 (ug/m³):

Arduino Simulators Dashboard

Active Simulators

Room: MSA1111

Room: MSA1601

	Min	Max
air_quality_pm10	33.0000	37.0000
air_quality_pm2_5	18.0000	21.0000
co2	500.0000	530.0000
hum	42.0000	45.0000
light	900.0000	950.0000
sound	28.0000	32.0000
temp	23.0000	25.0000
voc	230.0000	250.0000

Room: MSA3010

Room: MSA3070

Figure 1: Simulators dashboard: on the left you can choose to add a new simulator, to delete one, or to adjust the ranges of simulation for selected room.

5.5 Simulation Logic

- **Gaussian Distribution:** Sensor values are generated using a Gaussian (normal) distribution centered around a predefined mean.
- **Cosine Adjustment:** Some sensors use a time-based cosine function to simulate daily fluctuations:
  - Temperature decreases at night and peaks during the day.
  - Humidity raises over the night and lowers during the day.
  - Light intensity peaks during the day and drops at night.
  - CO_2 is lower during nights and higher during daytime.
  - Sound level is lower during nights and higher during the days.

5.6 Example Sensor Logic

```
if 'temp' in ranges:
    temp = round(random.gauss(mean(ranges['temp']), sigma_3(ranges['temp'])), 2)
    temp += -1.5 * cos((time_float - 1) / 12 * pi)
```

5.7 Equipment Editor

In addition to dynamic sensor data, each room is associated with static equipment like:

- Number of available seats
- Blackboard and whiteboard availability
- Microphones
- Smart systems for Webex meetings

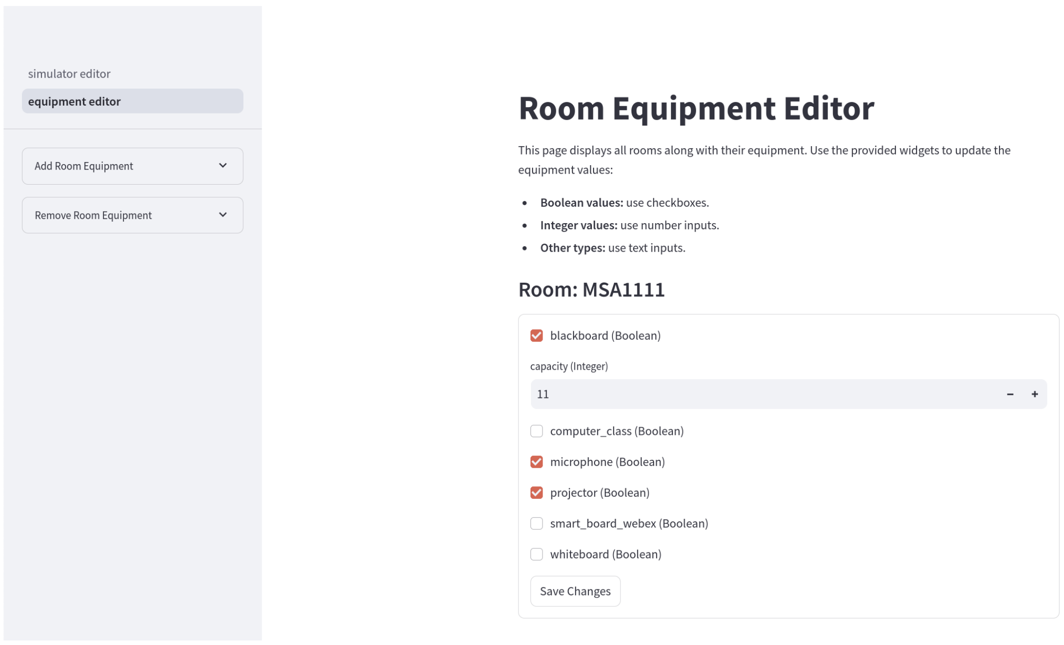
- Projectors
- PCs for students

5.8 REST API for Database Management

The REST API serves as a critical interface between the PostgreSQL database and the visual components of the simulation module. It supports standard CRUD (Create, Read, Update, Delete) operations and can be extended to handle additional IoT components in the future.

5.9 User Interface

The simulator's interface and equipment editor's page are built using the Streamlit framework (cf. Figure 1 and 2), which offers a lightweight, intuitive solution for data visualization and control. The backend REST API provides access to the data stored in the hosted database, ensuring real-time updates and easy configuration.



Room Equipment Editor

This page displays all rooms along with their equipment. Use the provided widgets to update the equipment values:

- **Boolean values:** use checkboxes.
- **Integer values:** use number inputs.
- **Other types:** use text inputs.

Room: MSA1111

☑ blackboard (Boolean)

capacity (Integer)

11 − +

☐ computer_class (Boolean)

☑ microphone (Boolean)

☑ projector (Boolean)

☐ smart_board_webex (Boolean)

☐ whiteboard (Boolean)

Save Changes

Figure 2: Equipment editor: on the left you can choose to add a new room with default equipment, to delete one. On the page you can update the equipment accordingly.

5.10 Technical Implementation

- The API is built with Flask, chosen for its simplicity and lightweight nature, making it suitable for small-to-medium-scale web services.
- SQLAlchemy, a popular Python ORM (Object-Relational Mapper), is used to interact with the PostgreSQL database, providing efficient and maintainable database management.

6 Grafana Dashboard for Admin Monitoring

The Grafana dashboard implementation provides administrators with a centralized, real-time view of room metrics by integrating time-series sensor data (via InfluxDB) and static room metadata (via PostgreSQL). Designed for scalability and usability, the dashboard facilitates anomaly detection and granular room-specific analysis through a structured two-tier design.

6.1 Design Choices & Key Features

The system employs dual data source integration, leveraging InfluxDB for dynamic sensor metrics (e.g., temperature, CO2 levels) and PostgreSQL for static room attributes (e.g., equipment availability, capacity). This separation optimizes query efficiency, ensuring real-time performance for sensor data and reliable access to structural room details.
The dashboard architecture consists of two primary views:

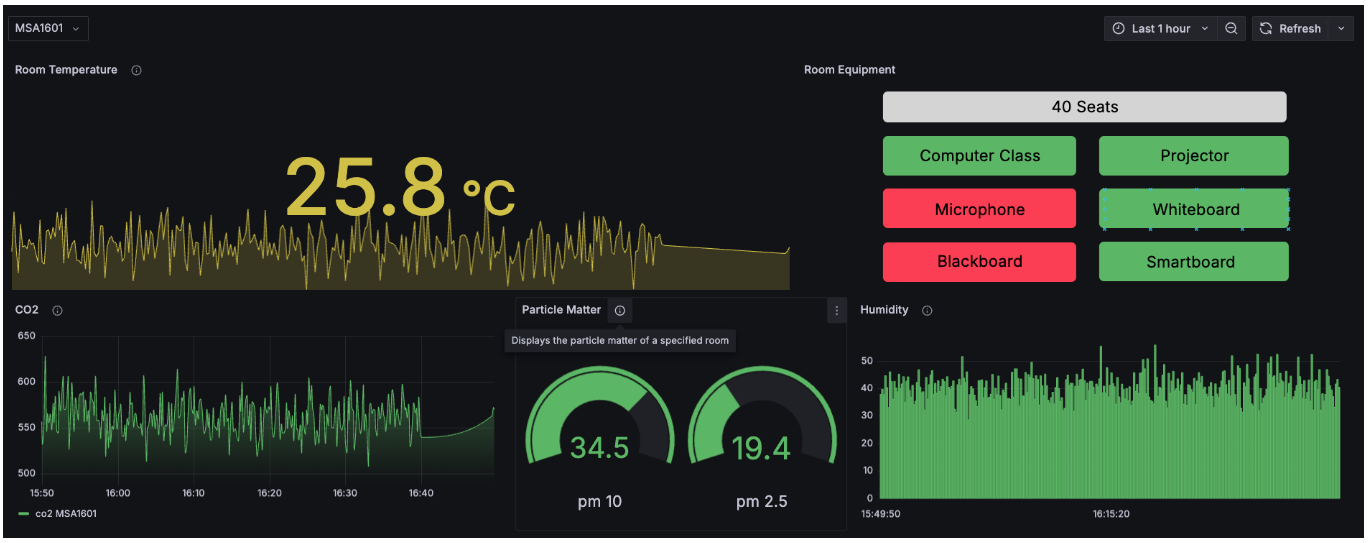
- Overall Metrics: Aggregates averages (e.g., mean temperature) and identifies anomalies (e.g., rooms exceeding CO2 thresholds) using InfluxDB functions like MEAN(), MAX(), and MIN().
- Room-Specific Metrics: Features a dropdown menu, which retrieves and sorts available rooms from PostgreSQL in ascending order. The dropdown menu queries PostgreSQL dynamically, ensuring new rooms or sensors added to the simulation are instantly available in Grafana without manual configuration. This view combines real-time sensor graphs (e.g., humidity trends) with static equipment tables, updating automatically as new rooms or sensors are added to the system (Fig. 6.2).

Administrators benefit from visual tools such as color-coded anomaly alerts (e.g., PM2.5 thresholds) and flexible time-range selection to analyze trends across hours, days, or weeks.

6.2 Key Challenges

A primary challenge involved synchronizing time-series (InfluxDB) and relational (PostgreSQL) data within Grafana, requiring careful alignment of queries by room identifiers. Additionally, to accommodate future sensor additions, reusable panel templates were designed, simplifying the integration of new metrics like noise or VOC levels without overhauling existing dashboards.

This implementation ensures scalability while maintaining clarity for administrators, aligning with the project's goal of efficient, data-driven room management.



7 Arduino Device as an IoT Data Collector

As part of the project, an Arduino Uno R3 was provided to demonstrate the integration of IoT devices in a sensor network. This microcontroller serves as a data collector, capable of connecting to sensors from the GroVe starter kit. While the device was useful as a proof-of-concept, certain limitations and challenges influenced its role in the project.

7.1 Device Overview

The Arduino Uno R3 is a basic microcontroller that supports analog and digital sensor inputs. However, it lacks built-in Wi-Fi or other direct networking capabilities, making it difficult to connect to MQTT or other IoT protocols without additional hardware.

7.2 Challenges Encountered

Sensor Applicability and Calibration Many sensors in the GroVe kit were not fully aligned with the project requirements. In addition, some sensors returned relative rather than absolute values, making them unsuitable for real-world measurements without calibration.

Communication with MQTT Due to the absence of Wi-Fi or Ethernet capabilities, direct communication between Arduino and MQTT broker was not possible. Instead, a serial communication bridge was implemented.

7.3 Demonstration Setup

For the proof-of-concept, we connected:

- Temperature sensor (analog input)
- Light sensor (analog input)
- LCD Display (for real-time feedback)

The Arduino reads data from the sensors, performs basic calibration, and outputs sensor values to the serial port as a JSON object.

8 Bridge from Serial Port to MQTT Publisher

8.1 Overview

This module acts as a bridge between serial port communication (e.g., with Arduino devices) and MQTT publishing. Its purpose is to enable Arduino-based or similar devices to behave like MQTT publishers by receiving data over serial connections, parsing the data, and transmitting it to an MQTT broker.

The module supports asynchronous handling of multiple serial ports and devices simultaneously, ensuring scalability and efficient real-time data transfer. It is designed to work within an IoT architecture that requires sensor data to be transmitted over the MQTT protocol.

The script runs continuously in a loop, actively monitoring for newly connected devices. This design enables the system to dynamically handle multiple Arduino devices simultaneously, rather than being limited to a single device. This approach allows the system to scale efficiently, supporting the extension to several Arduino boards or other compatible devices.

8.2 Data Flow and Processing

Device Discovery The module uses `serial.tools.list_ports` to scan for devices connected to serial ports. It looks for devices with descriptors or names containing ”Arduino,” ”ttyACM,” ”usbmodem,” or ”usbserial.”

Serial Data Handling

- For each detected device, the module starts an asynchronous task to read from the serial port.
- Incoming data is read line-by-line and expected to be valid JSON. An example JSON payload might look like:

```
{
    "room": "MSA3500",
    "temp": 22.5,
    "light": 300
}
```

- The data is parsed, and sensor values are extracted.

MQTT Publishing

- Sensor values are published to MQTT topics using `paho.mqtt.publish.single()`.
- Example MQTT topics for the data above would be:

```
room_101/sensors/temp (with payload {"value": 22.5, "timestamp":
"2025-02-02T14:15:00Z"})

room_101/sensors/light (with payload {"value": 300, "timestamp":
"2025-02-02T14:15:00Z"})
```

- The broker address and port are configurable through environment variables:
  - **MQTT_BROKER**: Defaults to Mosquitto.
  - **MQTT_PORT**: Defaults to 1883.

8.3 Technical Implementation

- **AsyncIO**: Enables asynchronous handling of multiple serial devices.
- **Serial Communication** (`serial_asyncio, serial.tools.list_ports`):
  - Used to scan for and read data from serial devices.
- **MQTT** (`paho.mqtt`): Handles communication with the MQTT broker.
- **JSON**: Data format for sensor messages.

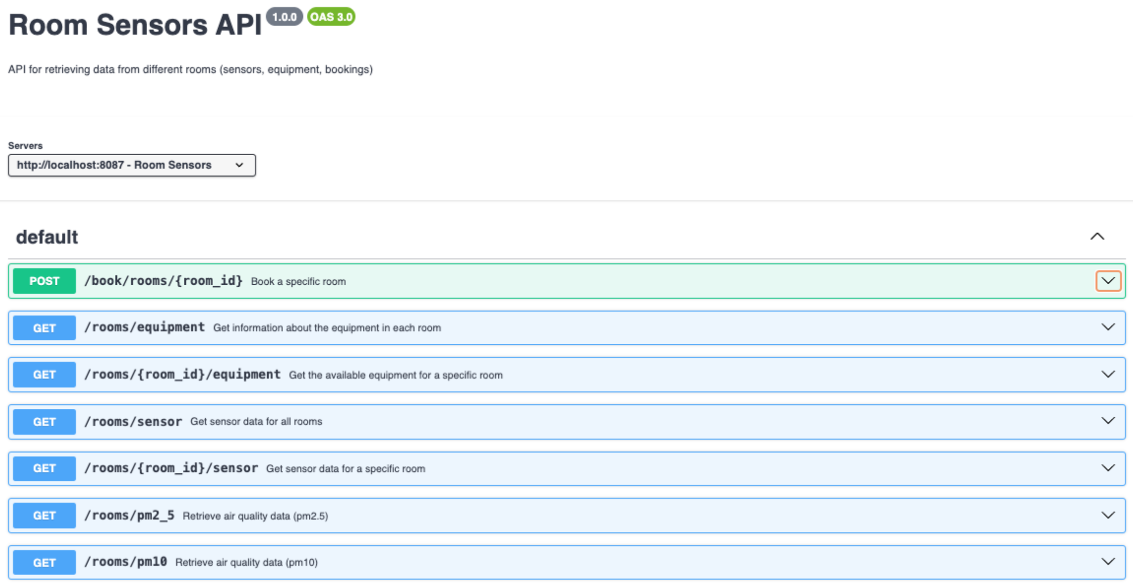
8.4 REST API 1 (Input for Decision System)

8.4.1 Overview

The REST API serves as the primary interface for retrieving sensor data, managing room bookings, and accessing equipment metadata within the IoT ecosystem. Implemented using Flask and documented via Swagger (OpenAPI 3.0), it integrates with InfluxDB for timeseries sensor storage, PostgreSQL for information retrieval about static data (e.g. equipment) and Google Calendar for reservation management.

8.4.2 Design Choices & Key Features

The API adopts a modular endpoint structure, providing both global endpoints (e.g., `/rooms/sensor` for aggregated sensor data) and room-specific endpoints (e.g., `/rooms/{room_id}/bookings`) to balance scalability with granular data access. Responses follow a standardized JSON schema defined in Swagger, ensuring consistency across endpoints. For instance, temperature data adheres to a structure where each entry includes a timestamp-value pair nested under a `RoomTemperature` object, enabling predictable parsing by both humans and systems. Booking functionality is centralized at `/book/rooms/{room_id}`, which enforces 30-minute time slots, validates timestamps, and synchronizes with Google Calendar while detecting conflicts (e.g., returning `HTTP 409` for overlapping reservations). Static room attributes, such as equipment availability and capacity, are exposed via endpoints like `/rooms/equipment`, directly supporting the decision system's ranking logic. Robust error handling ensures clarity, with descriptive status codes (e.g., `404` for missing data, `500` for server errors) to streamline debugging.


Room Sensors API 1.0.0 OAS 3.0

API for retrieving data from different rooms (sensors, equipment, bookings)

Servers
http://localhost:8087 - Room Sensors

default

POST `/book/rooms/{room_id}` Book a specific room

GET `/rooms/equipment` Get information about the equipment in each room

GET `/rooms/{room_id}/equipment` Get the available equipment for a specific room

GET `/rooms/sensor` Get sensor data for all rooms

GET `/rooms/{room_id}/sensor` Get sensor data for a specific room

GET `/rooms/pm2_5` Retrieve air quality data (pm2.5)

GET `/rooms/pm10` Retrieve air quality data (pm10)

8.4.3 Integration & Development Highlights

The API's structured output is optimized for the decision system, enabling efficient ranking based on sensor metrics (e.g., CO_2 levels), equipment metadata, and booking availability. Swagger's interactive documentation played a pivotal role in development, offering auto-generated specifications, testability via Swagger UI, and schema enforcement to maintain endpoint consistency. Temporal filtering for bookings, supported through optional `startDate` and `days` parameters, reduces data overhead by allowing customized time windows.

8.4.4 Key Challenges

The most significant challenge involved designing a data format that balanced machine-readability with human intuition. This required enforcing consistency across sensor types (e.g., uniform timestamp-value pairs), ensuring accessibility for developers, and maintaining extensibility for future sensors. By leveraging Swagger's schema reuse and hierarchical modeling, the API aggregates multi-sensor data under unified schemas (e.g., `RoomData`) while isolating metrics in dedicated endpoints (e.g., `/rooms/temperature`), achieving both clarity and efficiency.

9 Room Booking System

9.1 Overview

The room booking system ranks available classrooms based on two key criteria. The first is Environmental Sensor Data, which includes CO_2 levels, PM2.5, PM10, noise, lighting, humidity, VOC, and temperature. Each sensor's data is validated against EU and international compliance standards. The second is Equipment Data, which considers room features such as seating capacity, projector availability, smartboards, PCs, whiteboards, and other relevant equipment.

User preferences for environmental conditions and room amenities are provided through a Streamlit-based web interface. The system processes sensor and equipment data using the TOPSIS (Technique for Order Preference by Similarity to Ideal Solution) method to rank rooms. REST API endpoints, documented with Swagger, enable seamless integration and testing.

9.2 Execution Flow and Function Order

9.2.1 Data Retrieval and Preprocessing

Data Fetching The system retrieves real-time room data through API calls to the room database. Functions include:

- Manages GET requests with retries and exponential backoff for robust communication with REST API 1.
- Fetches sensor-specific JSON data, including CO_2 levels, temperature, and humidity.
- Additional functions to gather booking details and equipment availability.

9.2.2 Primary Room Validations

Before ranking, rooms are checked if they are available for the given time and if they have enough seats.

9.2.3 Compliance Checking

Each sensor type undergoes a compliance validation process, ensuring conformity to EU and German regulatory standards. The system:

- Computes key statistics such as average, maximum, and compliance percentage.
- Evaluates compliance against strict German/EU standards, as German regulations are well-documented and stricter than Luxembourgish policies
- Introduces tolerance margins to mitigate outliers caused by sensor inaccuracies.

Compliance Standards

1. **CO_2:** German Committee on Indoor Air Guide Values
   - Good air quality: CO_2 levels should be below 1000 ppm.
   - Acceptable threshold: CO_2 levels should not exceed 1500 ppm.

2. **PM2.5 & PM10:** EU Air Quality Standards
   - PM2.5: 24-hour average should not exceed 25 µg/m^3.
   - PM10: 24-hour average should not exceed 50 µg/m^3.

3. **Acoustic:** EN ISO 3382-2:2008
   - Noise Limit: 85 dB.

4. **Light:** EN 12464-1
   - Minimum Light Intensity: 500 lux.

5. **Humidity:** DIN EN 16798
   - Minimum Humidity: 30%.
   - Maximum Humidity: 70%.

6. **VOC:** German Umweltbundesamt
   - VOC Limit: 400 ppb.

7. **Temperature:** German ASR (Arbeitsstätten Regeln)
   - Minimum Temperature: 19°C.
   - Maximum Temperature: 26°C.

9.2.4 Building TOPSIS Decision Matrix

The TOPSIS matrix is essential for the TOPSIS algorithm calculation. Each column represents a criterion, which includes various environmental sensor data and equipment attributes. Each row corresponds to a specific room, allowing for a structured comparison of all available options.

9.2.5 TOPSIS MCDM Algorithm

The TOPSIS algorithm ranks rooms based on user preferences, equipment availability, user-defined weights, and environmental compliance.

1. **Penalty-Based Normalization**: To standardize sensor readings:

$$\text{adjusted value} = -\left|\frac{x - \text{user\_pref}}{\sigma}\right|$$

   where x is the average room's sensor reading, user_pref is the user-defined target, and σ is the standard deviation across all rooms. $x -$ user_pref measures the distance between the sensor data and the user preference which is scaled by the standard deviation. By negating the the absolute value of this, it makes penalizes higher deviations as it leads to more negative values. This is not part of the original TOPSIS algorithm and a slight modification to respect user preferences.

2. **Normalization across the matrix:** Using Euclidean norm operations with a small constant:
$$V_{ij} = \frac{X_{ij}}{\sqrt{\sum X_{ij}^2} + 10^{-9}}$$

3. **Weighting:** User-defined weights (1-9) are normalized and applied on the normalized matrix.

4. **Ideal and Negative-Ideal Solutions:** Computed for each criterion:

   - Ideal Best (PIS): Maximum value.
   - Ideal Worst (NIS): Minimum value.

   Attributes where lower values are preferable are adjusted, such that the PIS is the minimum value from each criterion and the NIS the maximum value from each criterion

5. **Distance Calculation and Ranking:** Using Euclidean distances:
$$D_i^+ = \sqrt{\sum (V_{ij} - A_j^+)^2}, \quad D_i^- = \sqrt{\sum (V_{ij} - A_j^-)^2}$$

6. **Final Ranking:** Closeness coefficient:
$$C_i = \frac{D_i^-}{D_i^+ + D_i^-}$$

   Higher C_i score indicate better room suitability and the rooms are ranked based on this value.

9.2.6 Why TOPSIS Was Chosen

Among various Multi-Criteria Decision-Making (MCDM) methods (AHP, ELECTRE, PROMETHEE), **TOPSIS was selected** due to:

- **Simplicity & Intuitiveness**: Calculates Euclidean distances for simple ranking.
- **Computational Efficiency**: Handles large datasets efficiently, which is suitable for the various sensor data and equipment data.
- **Effective Normalization & Weighting**: The modificated TOPSIS algorithm respects user preferences and balances diverse attributes.
- **Transparent Decision-Making**: Produces a closeness coefficient, ensuring interpretable rankings.

9.2.7 Motivation Behind the Weight Selection

- **Priority to Equipment Data**: Higher weights for projectors, smartboards, and PCs since they impact usability. Seating capacity is penalized if significantly overestimated.
- **Sensor Data Considerations**: VOC levels are less relevant in standard classrooms but critical in specialized environments.
- **Balanced Ranking Strategy**: A room is only highly ranked if it meets both **functional (equipment) and environmental (sensor) requirements**.

9.2.8 REST API Integration:

A GET request to the REST API endpoint (documented via Swagger) of the Room Booking System triggers the decision logic to retrieve the room ranking

10 Graphical User Interface for Room Bookings

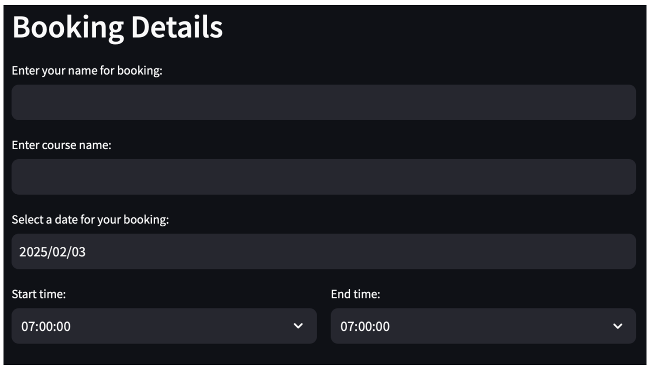
The following section describes the user interface of the **Room Booking System**, which allows users to book rooms based on environmental conditions, equipment availability, and scheduling constraints.

10.1 Room Booking System – User Input Form

The first interface displays the **Room Booking Form**, where users provide input regarding their booking requirements.

10.1.1 Booking Details

The user enters their name and course name, select a date and choose the start and end times.



Booking Details

Enter your name for booking:

Enter course name:

Select a date for your booking:
2025/02/03

Start time:
07:00:00

End time:
07:00:00

Figure 3: Booking Details

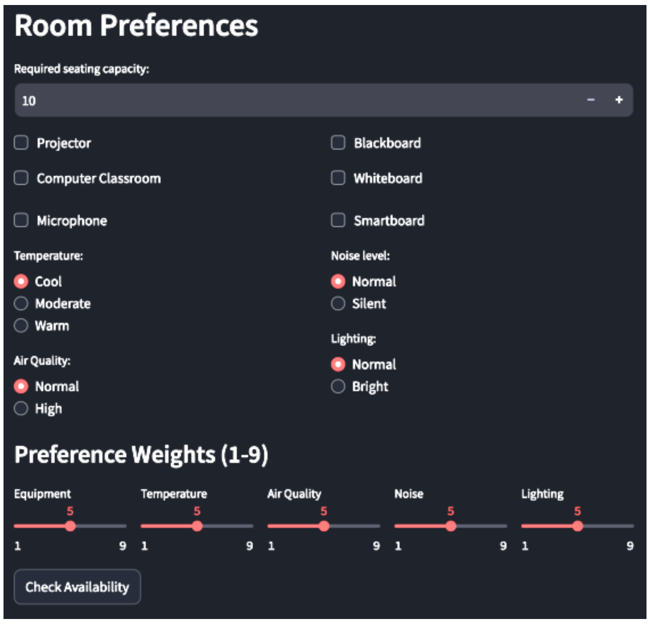
10.1.2 Room Preferences

The user can specify the required seating capacity and select necessary equipment, such as a projector, microphone, blackboard, whiteboard, smartboard, or computer classroom. Additionally, users can define environmental preferences, including air quality, noise level, light intensity, and temperature preference. The categories are written in natural language and mapped to values in the back-end, as the end-user can interpret those more easily than specified values.

10.1.3 Preference Weighting System

To refine the selection process, the interface introduces a preference weighting system that allows users to assign weights to the different criterias. The weight of each criteria-Equipment, Temperature, Air Quality, Noise Level, and Lighting—can be adjusted by as slider using the Saaty scale.

By clicking on the ”Check Availability” button, the interface sends a GET request to the REST API of the booking system to retrieve the room ranking for the specified preferences and time window.



Room Preferences

Required seating capacity: 10

- [ ] Projector
- [ ] Blackboard
- [ ] Computer Classroom
- [ ] Whiteboard
- [ ] Microphone
- [ ] Smartboard

Temperature: [✓ Cool] Moderate Warm

Noise level: [✓ Normal] Silent

Lighting: [✓ Normal] Bright

Air Quality: [✓ Normal] High

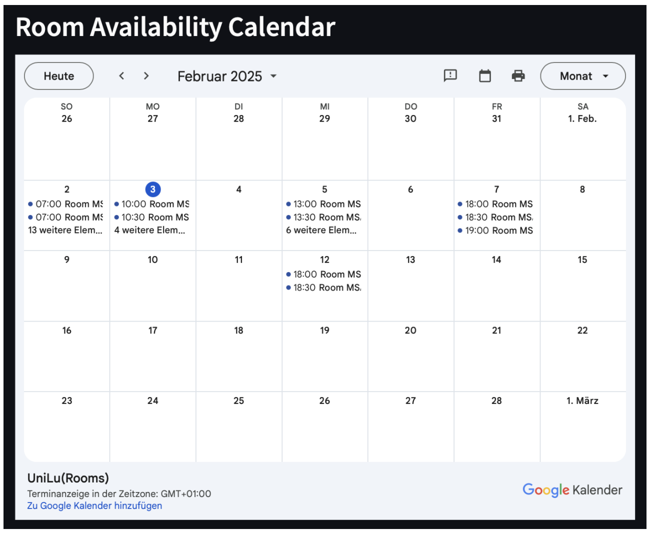
Preference Weights (1-9)

Equipment	Temperature	Air Quality	Noise	Lighting
5	5	5	5	5

Figure 4: Caption

10.1.4 Room Availability Calendar

The user interface provides a Room Availability Calendar, displaying available and booked time slots through Google Calendar Integration. Users can see room bookings by date and time, helping them choose an available slot efficiently.



Room Availability Calendar

SO	MO	DI	MI	DO	FR	SA
26	27	28	29	30	31	1. Feb.
2 ● 07:00 Room MS ● 07:00 Room MS 13 weitere Elem...	3 ● 10:00 Room MS ● 10:30 Room MS 4 weitere Elem...	4	5 ● 13:00 Room MS ● 13:30 Room MS 6 weitere Elem...	6	7 ● 18:00 Room MS ● 18:30 Room MS ● 19:00 Room MS	8
9	10	11	12 ● 18:00 Room MS ● 18:30 Room MS	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	1. März

UniLu(Rooms)
Terminanzeige in der Zeitzone: GMT+01:00
Zu Google Kalender hinzufügen

Figure 5: Room Availability Calendar

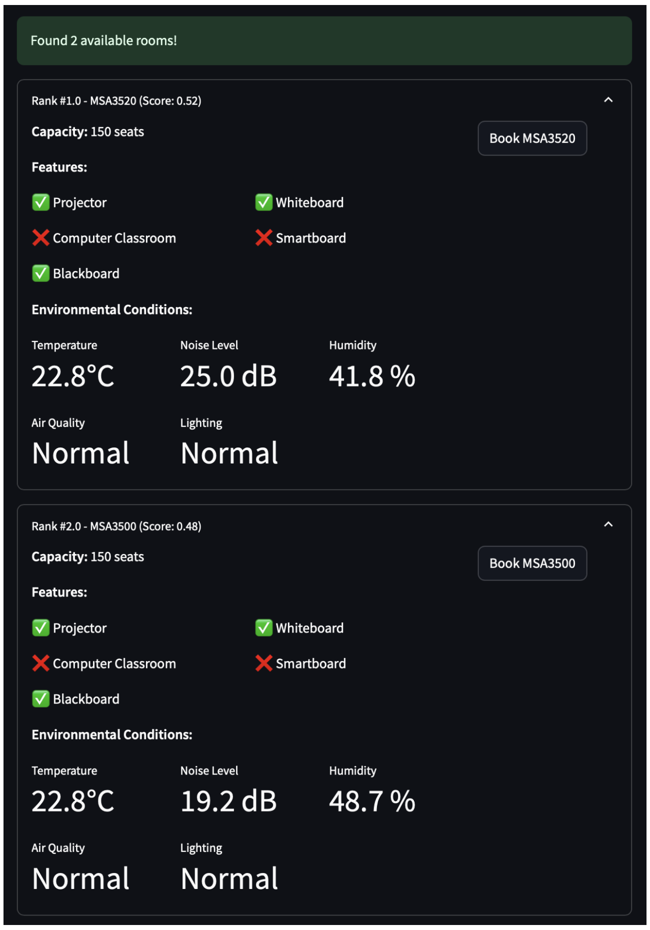
10.1.5 Ranked Room Suggestions Based on User Preferences

When retrieving the rooms from the booking system, the interface displays the calculated ranking of the rooms in ascending order.

Each room listing contains the Room ID, a ranking position score, and its capacity. The availability of equipment is visually indicated, with checkmarks denoting available features and crosses for unavailable ones.

The environmental conditions include temperature, noise level, humidity, air quality, and light intensity conditions. Users can review these details before selecting the preferred room.

Each room listing has a “Book” button, allowing users to confirm their reservation with one click.



Found 2 available rooms!

Rank #1.0 - MSA3520 (Score: 0.52)

Capacity: 150 seats

Book MSA3520

Features:

✅ Projector ✅ Whiteboard
❌ Computer Classroom ❌ Smartboard
✅ Blackboard

Environmental Conditions:

Temperature	Noise Level	Humidity
22.8°C	25.0 dB	41.8 %

Air Quality	Lighting
Normal	Normal

Rank #2.0 - MSA3500 (Score: 0.48)

Capacity: 150 seats

Book MSA3500

Features:

✅ Projector ✅ Whiteboard
❌ Computer Classroom ❌ Smartboard
✅ Blackboard

Environmental Conditions:

Temperature	Noise Level	Humidity
22.8°C	19.2 dB	48.7 %

Air Quality	Lighting
Normal	Normal

Figure 6: Ranked Room Suggestions

11 Team Contributions

Each team member contributed to distinct aspects of the project, ensuring a well-structured and functional system. The development of this project leveraged Git and GitHub for version control, enabling structured collaboration across the team. Through this process, we gained practical experience in managing workflows, resolving merge conflicts, and maintaining code integrity in a shared repository. By adopting branching strategies and regular commits, the team coordinated contributions while minimizing integration issues.

Fedor Chikhachev was responsible for developing the serial-to-MQTT bridge, which enables communication between Arduino devices and the MQTT broker. He also worked on the Arduino-based data collection system, integrating sensor readings into the IoT network. Additionally, he contributed to the simulation framework for sensor-based data, as well as to control system for managing available equipment in the rooms.

Tarik Tornes focused on the Google Calendar integration, allowing seamless room booking synchronization with external scheduling systems. He also developed the Grafana-based monitoring dashboard, which visualizes sensor data and system metrics for administrators. Furthermore, he played a key role in designing the REST API, ensuring structured data retrieval and system interoperability.

Thuc Kevin Nguyen was responsible for the booking interface, which allows users to select rooms based on environmental and equipment criteria. He also worked on the booking system logic, ensuring smooth reservation management and integration with the decision-making algorithms.

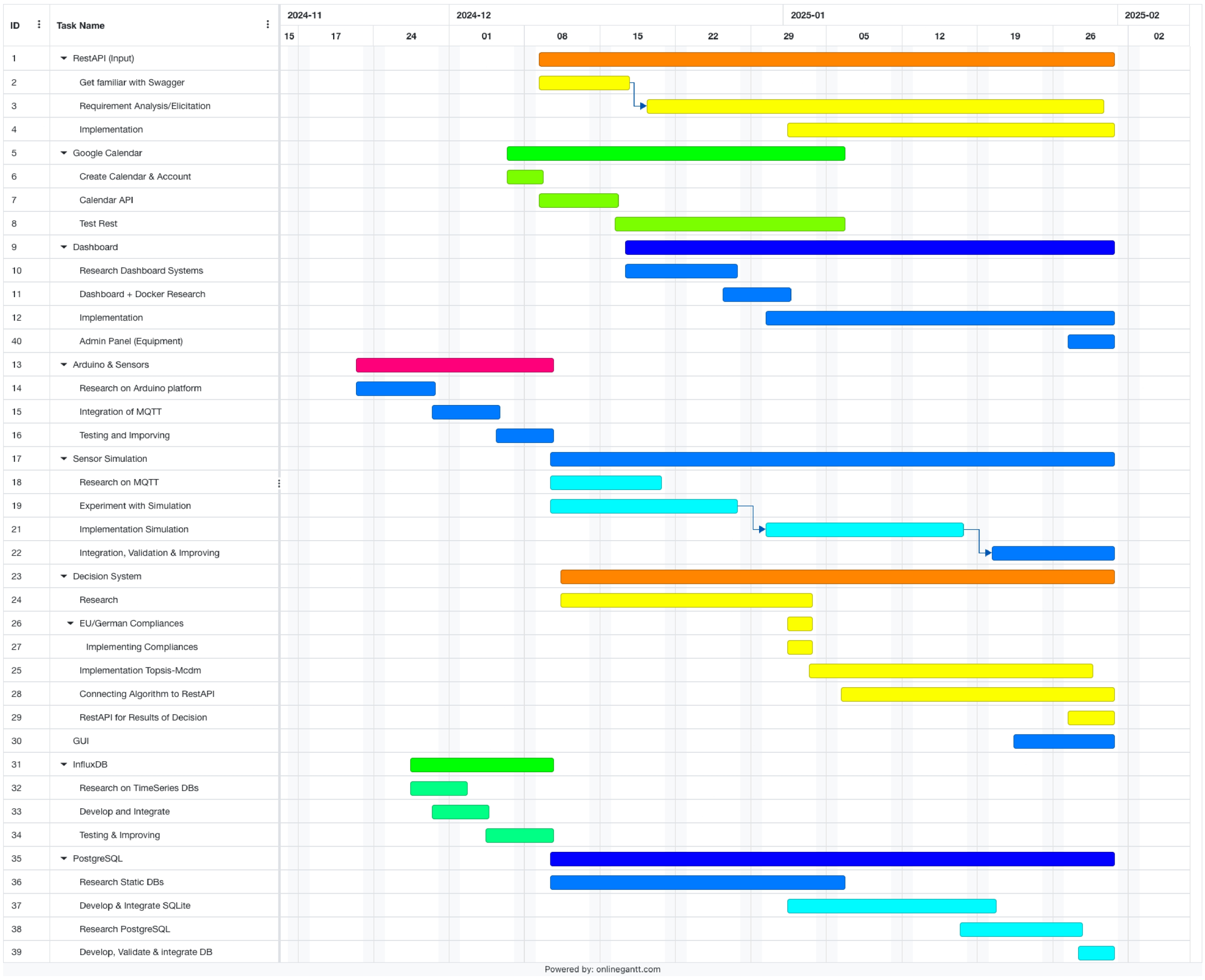
Vladyslav Siulhin handled static data management, setting up and maintaining structured storage for room data and equipment details. Additionally, he implemented the MQTT-to-InfluxDB service, ensuring real-time ingestion and structured storage of sensor data for further analysis.

This structured collaboration allowed the team to efficiently integrate various system components, ensuring a scalable and reliable IoT-based decision support system. A detailed breakdown of each contribution and the corresponding timeline can be found in the Gantt chart in Section 13.

12 Conclusion

The project successfully developed an IoT-based decision support system for room selection, integrating real-time sensor data, user preferences, and advanced decision-making algorithms. It demonstrates the potential of IoT technologies in smart building management.

13 Appendix



ID	Task Name
1	RestAPI (Input)
2	Get familiar with Swagger
3	Requirement Analysis/Elicitation
4	Implementation
5	Google Calendar
6	Create Calendar & Account
7	Calendar API
8	Test Rest
9	Dashboard
10	Research Dashboard Systems
11	Dashboard + Docker Research
12	Implementation
40	Admin Panel (Equipment)
13	Arduino & Sensors
14	Research on Arduino platform
15	Integration of MQTT
16	Testing and Imporving
17	Sensor Simulation
18	Research on MQTT
19	Experiment with Simulation
21	Implementation Simulation
22	Integration, Validation & Improving
23	Decision System
24	Research
26	EU/German Compliances
27	Implementing Compliances
25	Implementation Topsis-Mcdm
28	Connecting Algorithm to RestAPI
29	RestAPI for Results of Decision
30	GUI
31	InfluxDB
32	Research on TimeSeries DBs
33	Develop and Integrate
34	Testing & Improving
35	PostgreSQL
36	Research Static DBs
37	Develop & Integrate SQLite
38	Research PostgreSQL
39	Develop, Validate & integrate DB