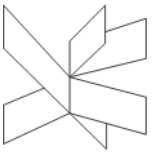


GHA – Project Report

Alin-Gabriel Abdallh (331979)

Dániel Hock (331276)

Name of supervisor(s):

Jens Cramer Alkjærsig (JCA)

[Number of characters: 61802]

Software Technology Engineering

7th semester

05.02.2026

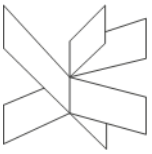
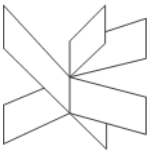
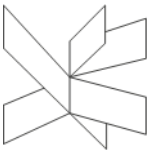


Table of content



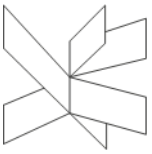


1. Abstract

The aim of this project was to create a system where a greenhouse owner can monitor the environment inside their greenhouse(s) with visualisation.

The project is using a Mosquitto server to communicate with the IOT devices and Filebeat through MQTT(S), Filebeat is communicating with Logstash through Beats, and Logstash, Opensearch and Grafana are communicating with each other through the Opensearch HTTP API. Furthermore, the website developed with React is embedding Grafana's panels and storing data using Pocketbase, while Keycloak makes sure everything stays secure regarding authentication.

The project has fulfilled its goal and successfully completed all the defined requirements within the set scope.

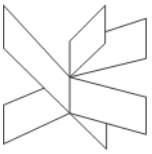


2. Introduction

The agricultural industry has changed massively over the course of the last few years, largely due to the massive advancements in technology, which all major companies now use to their advantage. Due to this, some traditional farmers feel like they have been left behind as they do not possess enough knowledge to use the technology on their own in an efficient way. One such farmer named Kim decided to buy some sensors to monitor the environment inside his greenhouses, but he did not know how to use the incoming information from them, so he approached the team and presented his problems so that the team could help him get up to date with the newest tools in the industry.

This project has been made to help maximise crop growth in greenhouses by utilising IoT devices. While in a real-world context, this is important for farmers to know how the environment changes daily in their greenhouses, the technologies which this project has required to solve the problem were also of great importance academically since the members of the team have learnt a lot from it. Especially because all of the used technology was new to the development team. Even though there would be numerous ways to improve the product, in its current state, it is very usable for the purpose of monitoring environmental data.

Improving the state of the greenhouses would benefit the economy, since farmers would be able to sell more high-quality crops and create more jobs, as working in agriculture would become socially more popular due to its success. Therefore, improving greenhouses would bring a positive impact to Denmark in the bigger picture.



This report will highlight the entire process of developing the project, from deriving the first requirements, all the way to creating dashboards and visualising the sensor data.

In the next section, the analysis stage of the project will be detailed.

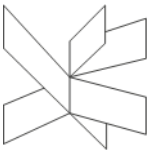
3. Analysis

After the project idea was approved by the supervisor, the team moved on to derive the requirements needed in order to complete the project, which were ordered by how important they were.

The first five requirements represent the most essential parts of the system. After their completion, the user would be able to monitor their greenhouses in an efficient way. As the number of requirements increases, their priority lowers, but they still remain important to the system. The requirements were deduced after a number of discussions that the team had with the user.

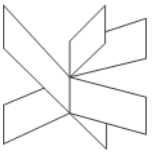
The requirements revealed the fact that the system consists of multiple actors and their respective responsibilities, which is being shown later in this chapter on the Use Case diagram.

The analysis helped the team by highlighting the system’s core through the essential requirements as well as additional features, and by defining the actors involved and their roles in the system.



3.1. Functional Requirements

1. As a user, I want to have a secure connection from the sensors to the visualisation, so I can access it online.
2. As a user, I want to have a platform where I can see what's going on in my greenhouses, so that I can maximise their efficiency.
3. As a user, I want to have a login system so that I can use different accounts.
4. As a user, I want to see a graph of how much time the temperature has spent in its respective range in percentages.
5. As a user, I want to see a graph of how much time the humidity has spent in its respective range in percentages.
6. As a user, I want to see a graph of how much time the light has spent in its respective range in percentages.
7. As a user, I want to see a graph of the collected temperature in the past 24 hours, so that I can monitor how the situation changes in the greenhouses.
8. As a user, I want to see a graph of the collected humidity in the past 24 hours, so that I can monitor how the situation changes in the greenhouses.
9. As a user, I want to see a graph of the collected light in the past 24 hours, so that I can monitor how the situation changes in the greenhouses.
10. As a user, I want to have a button which allows me to add a new Greenhouse.
11. As a user, I want to have another page where I can see the details of my Greenhouse.



12. As a user, I want to have a button which allows me to add a Grafana panel to the Greenhouse.
13. As a user, I want to have a button which lets me delete Greenhouses.
14. As a user, I want to have a button which lets me edit Greenhouses.
15. As a user, I want to have a button which lets me delete Panels.
16. As a user, I want to have a button which lets me edit Panels.
17. As a user, I want to be able to compare the data inside of my greenhouse with the outside weather, so that I know how isolated my crops are from the outside.
18. As a user, I want to be able to receive alerts when the values go outside of set ranges, so that I know that something is wrong.

3.2 Non-functional requirements

1. The database must be made using Opensearch.
2. The first server that communicates with the sensors has to use MQTTS.
3. The system must be accessible by multiple users concurrently.

3.3 Use case diagram

The system consists of one actor, the user, who interacts with the system in different ways. The user monitors the sensor data through visualisations on the platform, by adding Grafana panels to their already existing Greenhouse, or they can also create new Greenhouses with their own respective panels, title and description. Both Greenhouses and panels can be edited or deleted individually by pushing their respective buttons. Additionally, the users have

their own accounts now, so only those who have the username and password can now view this page.

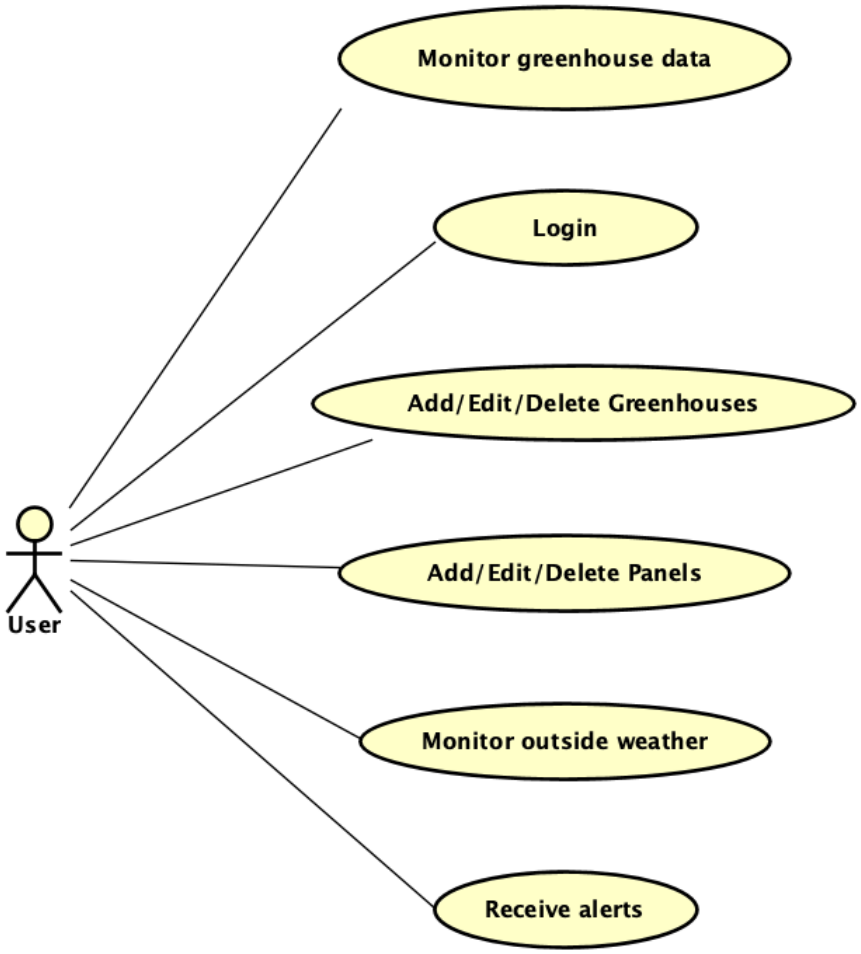
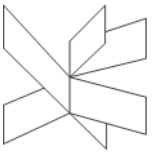


Figure 1. Can be found in the Appendices/project_appendices/diagrams/usecase_diagram/UseCase Diagram.png

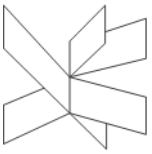


3.4 Use Case Descriptions

Login

The Login use case description showcases how the system allows the user to log into it, by opening the website, inputting their username and password and pressing the login option. Afterwards, the system checks the validity of the information typed by the user and then logs them in if it is correct and displays an error message if the information is wrong. The table below shows how everything works including the actors involved, the base sequence and the alternate sequence.

Use case	**Login**
Summary	The system allows the user to login into the platform to view their greenhouses' info.
Actor	User
Postcondition	The user is logged in and can see the information on the platform.
Base sequence	1. Open the website 2. Input the information Required: *email* - user's email *password* - user's password 3. Choose login option 4. Check if credentials are correct 5. Login into account
Alternate sequence (branch	0 Process can be cancelled at any step 4.1 Show error message

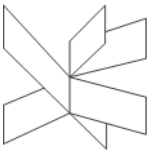


or exception)	*Invalid email or password* The user can repeat steps 2 and 3 until they are logged in or the process is cancelled.
Note	

Add/Edit/Delete Panels

The Add/Edit/Delete Panels use case deals with the possible scenarios regarding the panels of each greenhouse. The user has three possible options, to add, edit or delete, and the use case description below showcases how each different scenario is dealt with. In the case of adding, the user must then fill in the fields correctly and then press the “Add panel” button, for editing the user has to change the fields and then press the “Save” button, and for deleting, he has to press “OK” in order to complete the deletion. For adding and editing, there is an alternate scenario shown in case the required fields are left incomplete.

Use case	**Add/Edit/Delete Panels**
Summary	The system allows the user to add, edit or delete panels.
Actor	User
Postcondition	User adds, edits or deletes a panel from a specific greenhouse.



<table>
<tr><td>Base sequence</td><td>1. Open Grafana panels view
2. Click on one of the options
IF Add panel [A] (Go to A.1)
IF Edit [B] (Go to B.1)
IF Delete [C] (Go to C.1)
<table><tr><td>Scenario [A]
Add panel</td><td>1. Fill in the required information
iframe src URL
2. Choose add panel option
3. Check if conditions are met
4. Add panel</td></tr><tr><td>Scenario [B]
Edit</td><td>1. Fill in the required information
panel name
panel URL
panel size (width and height)
2. Choose save panel option
3. Check if conditions are met
4. Change panel</td></tr><tr><td>Scenario [C]
Delete</td><td>1. System asks if the user wants to delete panel
2. Choose OK
3. Delete panel</td></tr></table></td></tr>
<tr><td>Alternate sequence (branch or exception)</td><td>0 Process can be canceled at any step
[A] 4.1 Show error message
Field is required
[B] 4.1 Show error message
Field is required</td></tr>
<tr><td>Note</td><td></td></tr>
</table>

3.5 Activity Diagrams

Login

The activity diagram better illustrates the login workflow and the decision points involved in the process. It shows the actions made by the user and system as well as the conditional branching that happens when the credentials entered by the user are validated. This way, the diagram clarifies both the successful path and the error handling path and how the process is completed in both cases.

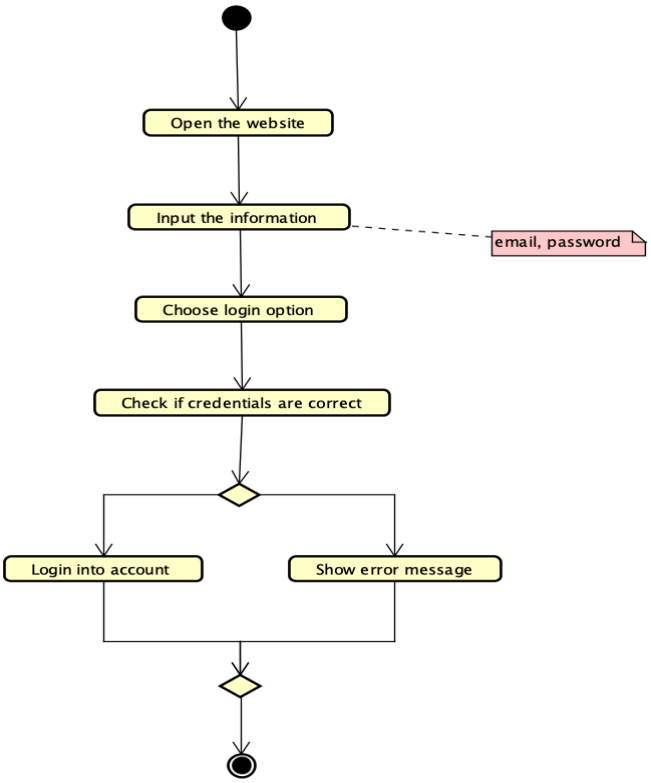


Figure 2. Can be found in the Appendices/project_appendices/diagrams/activity_diagrams/login_activity_diagram.png

Add/Edit/Delete Greenhouses

The activity diagram below illustrates all of the scenarios present while adding, editing and deleting greenhouses from the website. Following a logic similar to the Add/Edit/Delete Panel use case, it highlights the system's validation steps and clarifies how errors are handled and how all scenarios return to a common end.

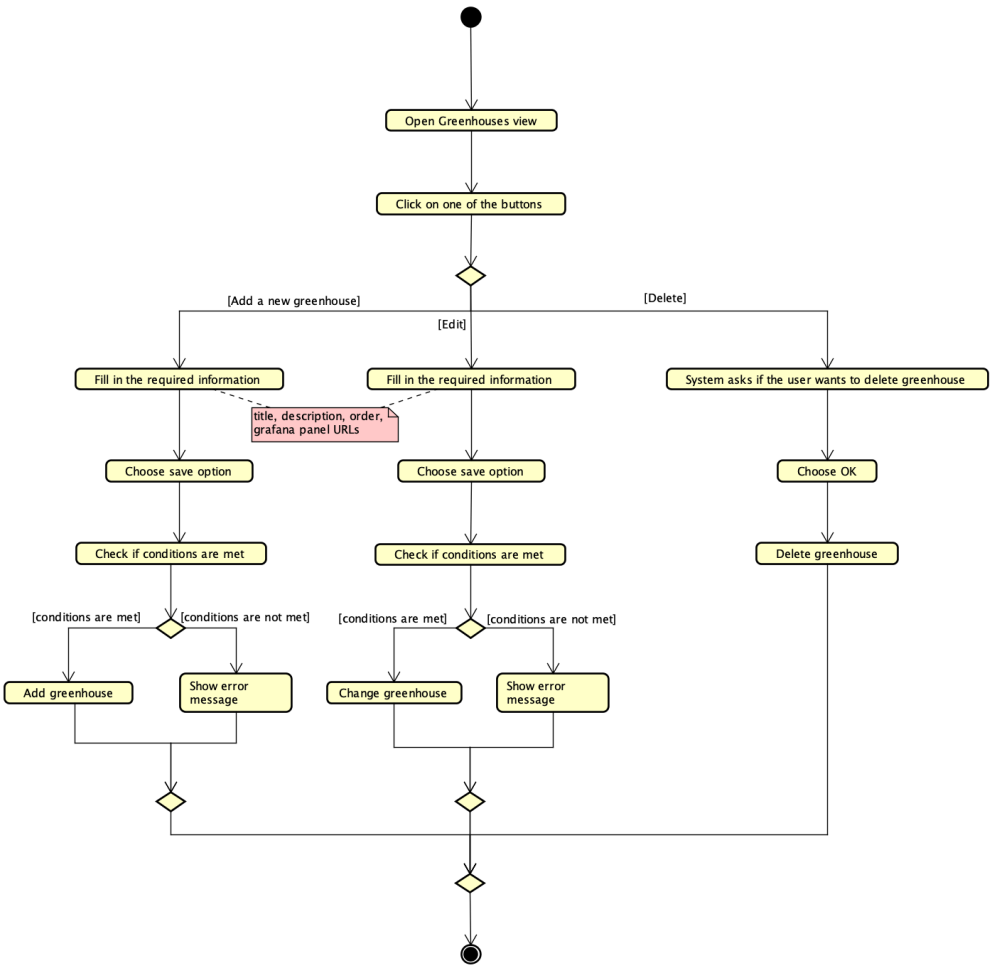


Figure 3. Can be found in the Appendices/project_appendices/diagrams/activity_diagrams/add_edit_delete_greenhouse_activity_diagram.png

4. Design

After the analysis was completed, the team advanced with the project to the design phase. In it, an architectural diagram was developed in order to create a clear plan, as well as some sequence diagrams to show how certain workflows look to an outsider.

4.1 Architecture diagram

Below is the Architecture diagram, it shows how it goes from gathering data up until visualising it, it also shows every technology and protocols which the team has used for the project.

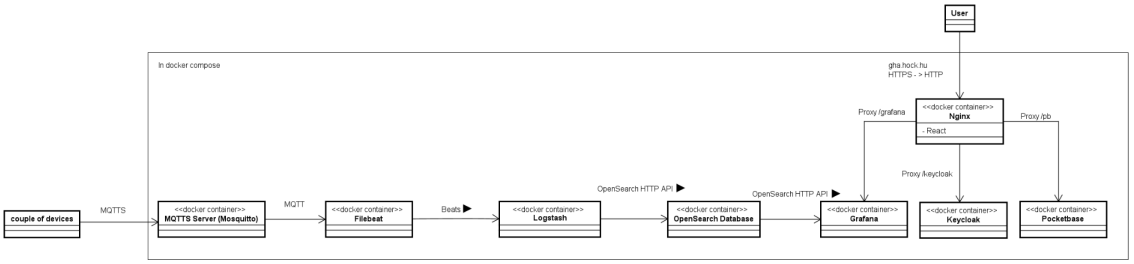
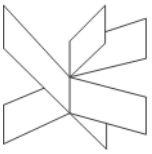


Figure 4. Can be found in the Appendices/project_appendices/diagrams/architecture_diagram/Architecture Diagram.png

The flow starts from a `Couple of Devices`, which are `IOT sensors`, that are communicating with a `Mosquitto server` through `MQTTS`, which then transfers the data to another server called `Filebeat` also through `MQTT`. Here the data receives its timestamp, which will be used for its storage. `Filebeat` then deposits the data in a data processing pipeline called `Logstash` through Beats. The data will then be moved into `OpenSearch Database` through the



OpenSearch HTTP API, which will then be available for the developers also through the OpenSearch HTTP API to use in Grafana to visualise the data. The User can access the React application through HTTPS and the Nginx integrates the four services together to one domain, and they are separated by URI prefixes. Finally, it's worth mentioning that all of the servers are in their respective Docker containers composed together with Docker compose.

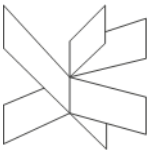
4.2 Technologies used

In this section, the technologies used will be detailed one by one, as well as the reason for using each one and its benefits. Before that, the team would like to clarify that the true goal of this project for the team was to learn about telemetry collection, store and visualisation. Because of funding, the team was constrained to using **open-source** services in order to complete the project, since other options, such as Power BI and ElasticSearch, require a budget.

While the team would like to discuss every technology individually, some of the reasons for using them, which have to be considered, make it so they are related to each other and have to be talked about together.

Below is the list of technologies used in the project:

- Mosquitto 2.0
- Filebeat 8.15.0
- Logstash 8.15.0
- OpenSearch 2.15.0



- Grafana v12.3.2
- Nginx
- Pocketbase v0.36.2
- Keycloak v26.5.1
- React 19.2
- Docker images
- Docker Compose
- ESP32-S2 + ESPHome

4.2.1 Grafana

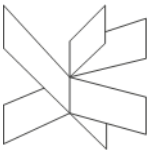
The project required a tool, which is widely used in the industry, to visualize and display telemetry data, with the possibilities to expand it by a large margin if it’s necessary, meaning in a scenario, when there are millions of time-series data coming in.

Grafana is specifically built for time-series data. It has multiple advanced panels, a query and transformation engine, alerting, annotations, and thresholds. It is ideal for operational monitoring, which the project is about.

On the other hand, it is much more difficult to use as a beginner in this technology.

However, the team has looked into other alternatives, such as PowerBI, or InfluxDB’s UI. PowerBI is really easy to use, especially in the business intelligence area, and has strong executive reporting.

On the other hand, it has an ingestion, row and column limit which restricts high-volume streams; its real-time updates are also fairly limited, just as its time-series optimisations. Furthermore, for the free version of PowerBI,



there would be no options to publish reports to share and collaborate, as well as no advanced dataflows, no advanced datamarts, and the model memory size limit would have been 0, just as the refresh rate for the datasets. So to even make this work, the team would have had to pay a monthly subscription.

InfluxDB's UI has a native flux support, which is a powerful time-series language, it's good for direct explorations of time-series data.

However, it is strongly tied to its own DB, limiting the possibilities to explore between more databases. Also, it only has basic panels such as line, bar and table, and its UI is fairly outdated.

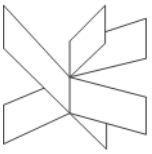
In conclusion, Grafana seemed to be the most suitable tool from the researched variants at the time of choosing.

4.2.2 OpenSearch

OpenSearch is open source and has an official Grafana data source, making it very easy to fetch the data from it. It has a rich query DSL such as full-text, filters and aggregations. It also has flexible schemas e.g: JSON from MQTT maps naturally to documents, very easy to add fields later if necessary.

On the other hand, it uses more disk than compressed Time-Series Databases would have, while OpenSearch's query style is great for "search/aggregate over time ranges", it is less optimised than a TSDB for "raw series at max resolution over long windows."

When making the decision, the team also looked at other options, which will be detailed below, combined with the explanation of why the team chose not to use them:



ElasticSearch is the paid version of OpenSearch, and while it has more functionality and more mature integration, it also requires paid licenses to use, which meant that, therefore, it was out of the question from the start.

InfluxDB was purposely built for time-series ingestion, has native support for influx line protocol, and it integrates well with IoT telemetry, MQTT plugins and Grafana.

However, it doesn’t handle multiple users on a large scale. Nevertheless, InfluxDB remains a strong candidate.

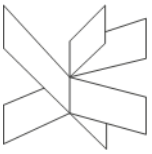
4.2.3 Filebeat + Logstash

The main reason for requiring another service is because the database can not communicate with MQTTS. The team chose Filebeat, because:

Filebeat is designed to run in containers, it has a broad input support, which allows it to ingest telemetry-like events easily. It has a reliable delivery thanks to the built-in buffering and backpressure handling, and it retries on failure. It supports structured parsing, which is very useful when the incoming data needs to be timestamped.

On the other hand, Filebeat can’t connect to OpenSearch without Logstash.

Telegraf is putting metrics-first, unlike Filebeat, which puts log-first. It supports a huge variety of inputs. Telegraf is designed for numeric metrics, time-series points, tags, fields and so on. Telegraf also supports advanced processors, such as aggregation, filtering, etc. Just like Filebeat, it has built-in buffering and backpressure handling, and it retries on failure. It doesn’t require Logstash to communicate with OpenSearch.



Overall, Telegraf is made for metrics and telemetry, rather than logs, it is a viable option.

Fluent Bit was the last technology which the team considered. It has broad input support, though it doesn't support MQTTS directly, it is very fast and has a low memory usage since it's written in C.

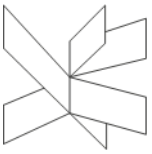
However, Fluent Bit is not optimised for numeric metrics, it also doesn't have native metric types or time-series math. So, for this project, it is not suitable.

4.2.4 Mosquitto

The sensor based on ESPHome firmware is generally designed to communicate with Home Assistant, but it also extends to share the sensor data over MQTTS, and while there were three server implementations of this protocol, both RabbitMQ and ActiveMQ are having message handling problems as of the time when this report is being written, therefore the only possible solution was the Mosquitto.

4.2.5 ESP32-S2 + ESPHome

Even though the sensors were given to us by the user, whom the team mentioned in the Project Description, the team still felt the need to explain the choice of using a more modern approach regarding sensor development and implementation.



While there are major hardware differences between ESP32-S2 with its board Lolin S2 Mini(Together they are called GHANode for the project) vs Arduino and ESP-01, listing them would be very hard to read, so the team will keep it brief and simple. GHANode has Native WiFi on ESP32-S2, it supports MQTTS, has full TLS support, CA pinning and hostname check. Thanks to this, it can check both username and password, and the whole node can be configured in one YAML file.

While Arduino can’t do any of the said things above, because it doesn’t have the hardware capabilities to support TLS, the most it can do is raw TCP/HTTP or non-TLS MQTT due to the RAM/flash limits on ESP-01. While a developer could try to implement their own encryption, it is a complex task which takes a lot of time.

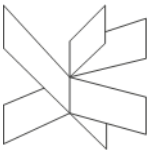
4.2.6 Nginx

The team required a proxy to do the reverse proxy work in a docker-compose environment, there were three candidates, Nginx, Traefik and Caddy. Traefik is great for routing, Caddy is the simplest to implement and Nginx is the best for debugging.

The team valued the debugging option more over the others as well as the fact that they had previous experience with Nginx.

4.2.7 Pocketbase

While there may be better alternatives, due to time constraints the team had no chance to develop a backend by themselves, therefore they went with a technology that was the easiest to set up, but still could handle collections and



has proper authentication protocols, such as SSO. Pocketbase is used in this project, to store greenhouses and user preferences for the website.

4.2.8 Keycloak

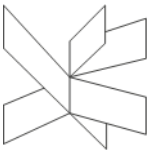
This project is using keycloak to authenticate users. It has full support of OAuth 2.0, OpenID Connect (OIDC), SAML 2.0, Single Sign-On (SSO), and Single Logout (SLO) , and it can run in docker. Besides these supports, it also has security features such as 2FA, password policies, Brute-force protection, session management and token revocation. Furthermore it is open-source, which was perfect for this project, even though it was quite difficult to set it up with the database, webpage, and Grafana.

The team has looked at other alternatives such as Auth0, but it wasn't open-source, and just as with Pocketbase the time constraints didn't allow the team to build an authentication system from scratch.

4.2.9 Docker & Docker-compose

Besides the Docker images, Docker Compose and Nginx, the rest of the technologies mentioned above were completely new to the team, which made it difficult but intriguing to work with.

The team had already been working with docker images and compose in previous projects, where they have been proven reliable and suitable for



handling work such as the one required by this project and, therefore, the team decided to bring them on board.

4.2.10 React

Having to make a website the natural choice was React since the team has worked with it over multiple semesters, therefore, the team did not consider any other alternatives.

4.3 Class diagram

Below is the class diagram of the webpage, which shows the overall structure and the relations of the “classes” between each other. It illustrates how the application is divided into sub-parts, such as pages, contexts, utils, etc. and their dependency on one another. Each component type has a different colour to make it easier to understand the logical structure of the system.

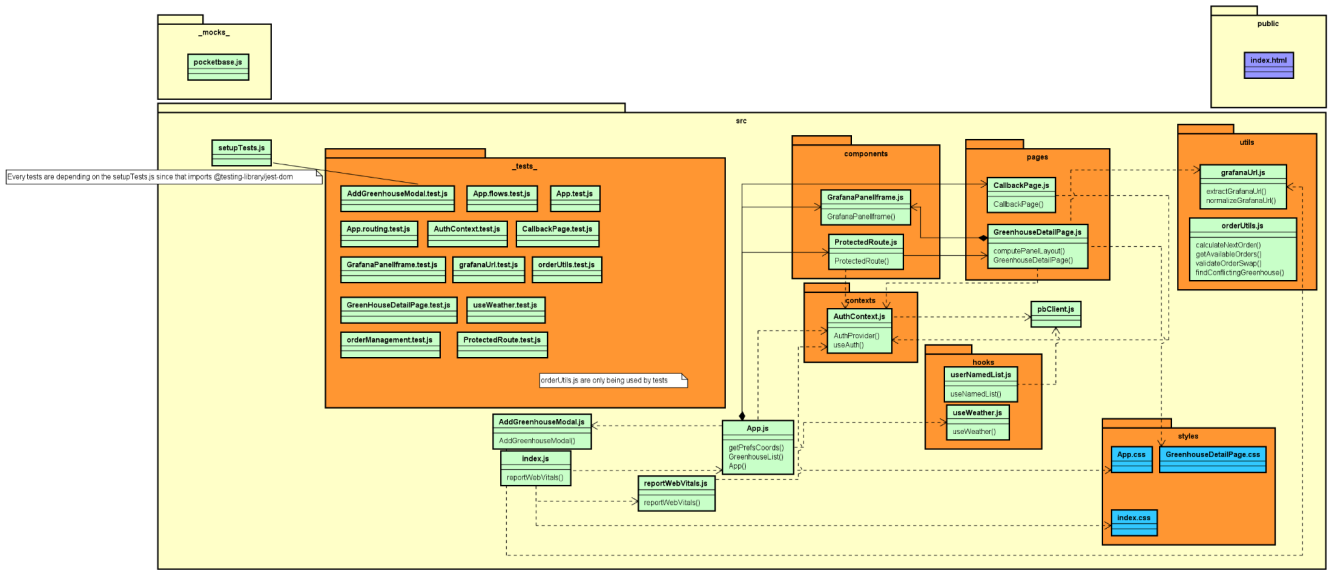


Figure 5. Can be found in the Appendices/project_appendices/diagrams/class_diagram/ghafrontend_class_diagram.png

4.3 Sequence diagrams

In this sub-chapter, the team will discuss two sequences in the project. As said in the beginning of the Design section, these sequence diagrams have been made for the readers to understand better how certain aspects of the system are functioning without looking into the implementation.

4.3.1 Data Travelling Sequence Diagram

The sequence diagram below shows how the data travels from the IOT devices until the database.

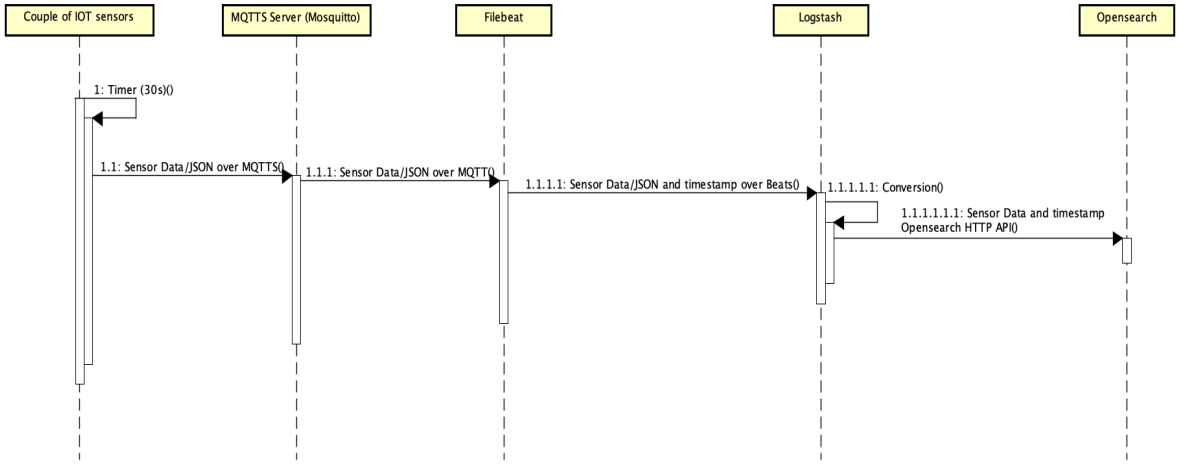


Figure 6. Can be found in the Appendices/project_appendices/diagrams/sequence_diagram/Data Travelling SD.png

In the sensors' a timer is triggered every 30 seconds, which will send the Sensor Data/JSON (H., 2025) as a message to the MQTTS Server (Mosquitto) topic, which wi ll pass the message to subscribed Filebeat connection using MQTT. (the s means secure, which is not necessary after reaching the private servers.) In Filebeat the message gets its timestamp and gets sent over with Beats to Logstash for the last time in Sensor Data/JSON format, because from Logstash it will be converted into Sensor Data and timestamp, which will be then sent to the Opensearch database using Opensearch HTTP API, where it will finally be stored.

4.3.2 New Dashboard Sequence Diagram

The other sequence diagram showcases the entire process of how a new diagram is created, from the user pressing the buttons up until what happens

inside the database and also how the data is then returned and displayed back to the user.

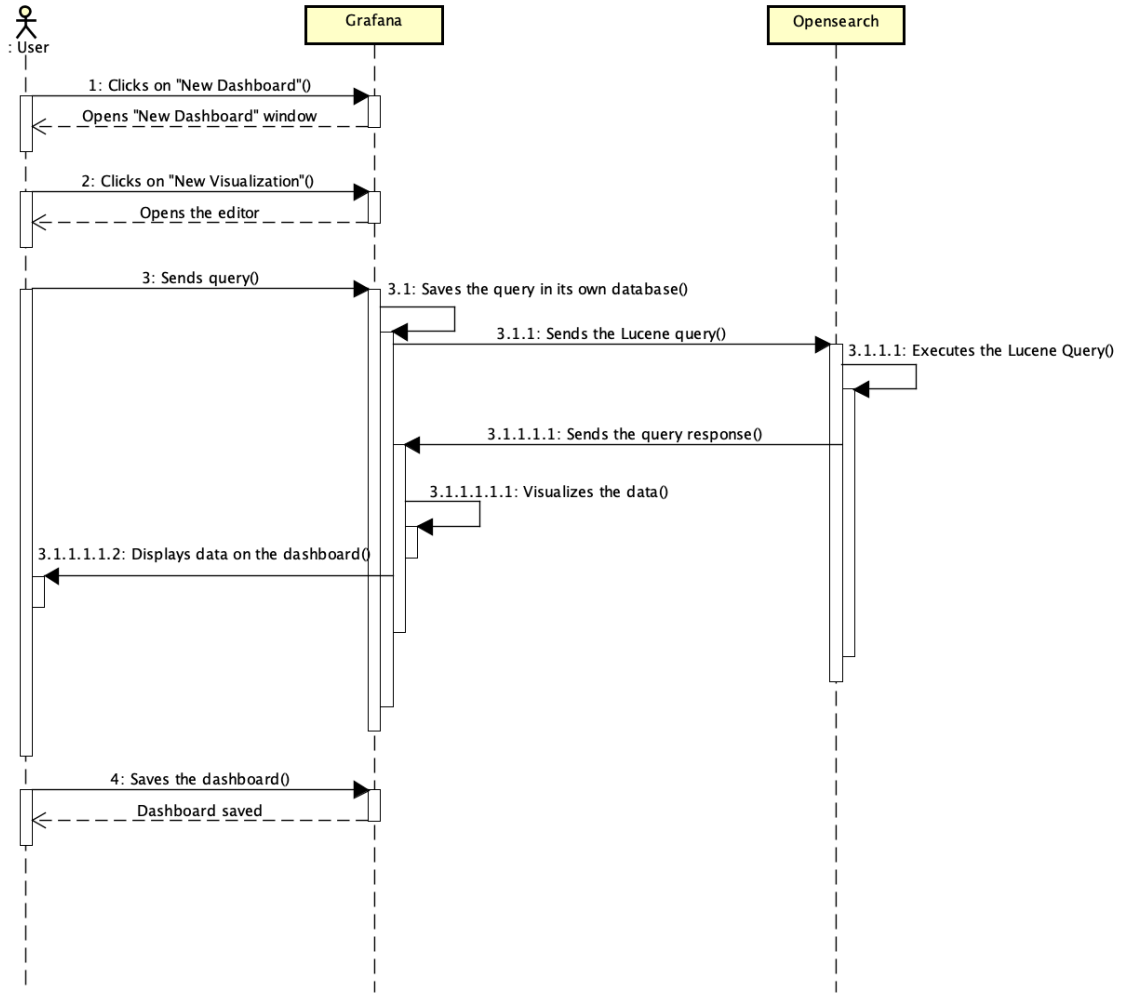
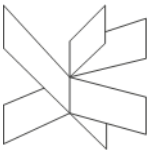


Figure 7. Can be found in the Appendices/project_appendices/diagrams/sequence_diagram/New Dashboard SD.png

The user begins the sequence by clicking on the “New Dashboard” button inside the Grafana UI. This will open a new window in which he will press on the “New Visualisation” button in order for the editor to open up. He will then write and send a new query to Grafana, which will then save it to its own database and then convert it to Lucene if needed and send it to Opensearch.



There, OpenSearch will execute the Lucene query and send back the query response to Grafana. From there, Grafana will visualise the data and display it on the new dashboard. The final step that is left of the sequence is for the user to press the “Save” button in order to save the dashboard, which will then save it inside Grafana.

4.4 Security

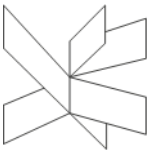
This chapter will talk about the security of the system. Since the services are composed together, the communication between them is, by default, very secure. However, the user side and the sensor communication required additional tweaks to ensure everything is protected.

In order to ensure a safe passage between the sensors and the Mosquitto server, the server has a valid SSL certificate, which is validated by the sensor, then the sensor creates an encrypted communication channel to the server TLS 1.3 and authenticates itself with a username and passwords against the server.

On the user side, the HTTPS is terminated by the haproxy, which contains the let’sencrypt certificate, and forwards it to the Nginx webserver, which is the entry point for the user traffic on the system.

The React and the Keycloak login page are public. The Pocketbase user data and the greenhouse list are only accessible after a JWT validation. This token is being created by Keycloak after successful login.

The Grafana is also being set up as a client to the Keycloak system and its dashboards are being assigned to Keycloak users by the Administrator. Each user can only access their own dashboard, which contains all of the panels that



could be embedded in the react application. Grafana maintains its own HTTP session, so its auto-login feature is on, so the dashboards can load automatically in the login flow.

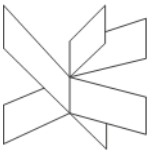
For the logout the application redirects to the Grafana's logout page so the Grafana session can be cleared. Then the Keycloak redirects back to the index page, which shows the login page again and the Pocketbase token is just being cleared from the browser.

5. Implementation

This section of the report will talk about the implementation of the system, focusing on the crucial parts of the project and how everything works together. It will explain the Docker Compose as well as its components, other important files and configurations that help the data pass smoothly between all parts of the system. Afterwards, some of the queries which are related to a number of requirements will be explained here, as well as other features implemented. In the end, it will demonstrate a feature from the website both on GUI and code.

5.1 Docker Compose

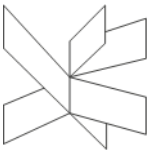
The Docker Compose file is used in order to run all parts of the system together in a smooth manner. It defines every service that is used in the GHA



system and makes sure that they start and work in the correct order, communicate the data between them and restore it if they are restarted.

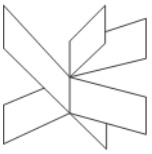
The compose file created consists of Grafana for visualisation and OpenSearch as the database where the data received from the sensors is stored. Moreover, the OpenSearch dashboard gives another interface where the type of data received can be observed and managed before anything is done inside Grafana. The data from the sensor is received at first by the Mosquitto MQTT, which is then collected by Filebeat, forwarded to Logstash, where it is processed and then sent to Opensearch. This sequence is made possible by the Docker Compose, as it ensures that the correct order is being kept at all times and that there is no data being lost anywhere. Additionally, Keycloak is added in order to provide centralized authentication, and Grafana is configured to use this identity provider for user login. An Nginx reverse proxy is added in front of the web-based services in order to provide clean URLs and simplify external access. Lastly, Pocketbase is included as a lightweight backend service that can be used for data storage and configuration.

Below is the docker-compose.yml made for the system.



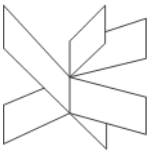
```
services:
  grafana:
    image: grafana/grafana-oss
    container_name: grafana
    restart: unless-stopped
    environment:
      - GF_LOG_LEVEL=info
      - GF_SERVER_ROOT_URL=https://gha.hock.hu/grafana/
      - GF_SERVER_SERVE_FROM_SUB_PATH=true
      - GF_SECURITY_ALLOW_EMBEDDING=true
      - GF_AUTH_GENERIC_OAUTH_AUTO_LOGIN=true
      - GF_AUTH_GENERIC_OAUTH_ENABLED=true
      - GF_AUTH_GENERIC_OAUTH_CLIENT_ID=grafana
      - GF_AUTH_GENERIC_OAUTH_CLIENT_SECRET=${GF_OAUTH_CLIENT_SECRET}
      - GF_AUTH_GENERIC_OAUTH_SCOPES=openid profile email
      - GF_AUTH_GENERIC_OAUTH_AUTH_URL=https://gha.hock.hu/keycloak/realms/gha/protocol/openid-connect/auth
      - GF_AUTH_GENERIC_OAUTH_TOKEN_URL=https://gha.hock.hu/keycloak/realms/gha/protocol/openid-connect/token
      - GF_AUTH_GENERIC_OAUTH_API_URL=https://gha.hock.hu/keycloak/realms/gha/protocol/openid-connect/userinfo
      - GF_AUTH_GENERIC_OAUTH_SIGNOUT_REDIRECT_URL=https://gha.hock.hu/keycloak/realms/gha/protocol/openid-connect/logout
      - GF_SMTP_ENABLED=true
      - GF_SMTP_HOST=172.19.2.1:25
      - GF_SMTP_FROM_ADDRESS=ghagrafana@hock.hu
      - GF_SMTP_STARTTLS_POLICY=NoStartTLS
    ports:
      - "3000"
    volumes:
      - grafana-data:/var/lib/grafana
    depends_on:
      opensearch:
        condition: service_healthy
    networks:
      default:
        ipv4_address: 172.19.2.3

  opensearch:
    image: opensearchproject/opensearch:2.15.0
    container_name: opensearch
    restart: unless-stopped
    environment:
      - discovery.type=single-node
      - bootstrap.memory_lock=true
      - plugins.security.disabled=false
      - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${PASSWORD_OPENSEARCH}
      - OPENSEARCH_JAVA_OPTS=-Xms1g -Xmx1g
    ulimits:
      memlock:
        soft: -1
        hard: -1
    ports:
      - "9200"
    volumes:
      - os-data:/usr/share/opensearch/data
    healthcheck:
      test: ["CMD-SHELL","curl -k -u admin:${PASSWORD_OPENSEARCH} https://localhost:9200 >/dev/null 2>&1 || exit 1"]
      interval: 10s
      timeout: 10s
      retries: 30
    networks:
      default:
        ipv4_address: 172.19.2.4

  dashboards:
    image: opensearchproject/opensearch-dashboards:2.15.0
    container_name: opensearch-dashboards
    restart: unless-stopped
    environment:
      OPENSEARCH_HOSTS: '["https://opensearch:9200"]'
      OPENSEARCH_USERNAME: admin
      OPENSEARCH_PASSWORD: ${PASSWORD_OPENSEARCH}
      OPENSEARCH_SSL_VERIFICATION: "false"
      SERVER_BASEPATH: "/opensearch"
      SERVER_REWRITEBASEPATH: "true"
      SERVER_PUBLICBASEURL: "https://gha.hock.hu/opensearch"
    ports:
      - "5601"
    depends_on:
      opensearch:
        condition: service_healthy
    networks:
      default:
        ipv4_address: 172.19.2.5
```

Figure 8



```
mqtt:
  image: eclipse-mosquitto:2-openssl
  container_name: mqtt
  restart: unless-stopped
  ports:
    - "1883"
    - "8883"
  environment:
    DEBUG: "true"
  volumes:
    - ./mosquitto/config:/mosquitto/config
    - ./mosquitto/data:/mosquitto/data
    - ./mosquitto/log:/mosquitto/log
  networks:
    default:
      ipv4_address: 172.19.2.6

logstash:
  build: ./logstash
  container_name: logstash
  restart: unless-stopped
  environment:
    LS_JAVA_OPTS: "-Xms512m -Xmx512m"
    PASSWORD_OPENSEARCH: ${PASSWORD_OPENSEARCH}
  ports:
    - "5044"
  depends_on:
    opensearch:
      condition: service_healthy
  volumes:
    - ./logstash/pipeline/logstash.conf:/usr/share/logstash/pipeline/logstash.conf:ro
    - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro
  networks:
    default:
      ipv4_address: 172.19.2.7

filebeat:
  image: docker.elastic.co/beats/filebeat:8.15.0
  container_name: filebeat
  restart: unless-stopped
  depends_on:
    logstash:
      condition: service_started
  volumes:
    - ./filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro
    - ./filebeat/logs:/usr/share/filebeat/logs
    - ./filebeat/data:/usr/share/filebeat/data
  networks:
    default:
      ipv4_address: 172.19.2.8

nginx:
  image: openresty/openresty:bookworm-fat
  container_name: nginx
  depends_on:
    - grafana
  volumes:
    - ./nginx/conf.d:/etc/nginx/conf.d:ro
    - ./nginx/html:/usr/share/nginx/html:ro
  ports:
    - "80:56340"
  networks:
    default:
      ipv4_address: 172.19.2.9

pocketbase:
  image: ghcr.io/muchobien/pocketbase:latest
  restart: unless-stopped
  volumes:
    - pb-data:/pb_data
  ports:
    - "8090"
  command: ["serve", "--http=0.0.0.0:8090"]
  networks:
    default:
      ipv4_address: 172.19.2.10
```

Figure 9



```
  keycloak:
    image: quay.io/keycloak/keycloak:26.5.1
    environment:
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: ${PASSWORD_OPENSEARCH}
    command:
      - start-dev
      - --http-relative-path=/keycloak
      - --proxy-headers=xforwarded
      - --hostname=https://gha.hock.hu/keycloak
    ports:
      - "8080"
    networks:
      default:
        ipv4_address: 172.19.2.11

networks:
   default:
     name: "network-gha_gha"
     external: true

volumes:
  os-data:
  grafana-data:
  pb-data:
```

Figure 10

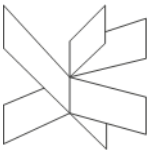
As mentioned above, many of the services depend on Opensearch to be up and running for them to start running as well. Similarly, the Filebeat service depends on Logstash since it needs to forward its data there. All of these are made possible by using the depends_on health check as it can be seen below:

```
depends_on:
  opensearch:
    condition: service_healthy
```

Figure 11

```
depends_on:
  logstash:
    condition: service_started
```

Figure 12



The compose file also makes use of volumes for all of the services, which ensures that the data will not be lost when the container restarts and that it will remain safe. For example, the dashboard data is stored in a grafana-data volume, which stores it securely.

```
volumes:
  - grafana-data:/var/lib/grafana
```

Figure 13

Additionally, the compose file ensures that the services are deployed to a server where they can be accessed externally. This is done especially for the Grafana and Opensearch services since they are crucial for managing the data as well as creating the subsequent dashboards. The file also assigns static internal IPs, making sure it will always have the same address inside the network instead of receiving a random address at every restart. These can be seen in the code below for the Grafana service.

```
environment:
  - GF_LOG_LEVEL=info
  - GF_SERVER_ROOT_URL=https://hga.hock.hu/grafana/
  - GF_SERVER_SERVE_FROM_SUB_PATH=true
```

Figure 14

```
networks:
  default:
    ipv4_address: 172.19.2.3
```

Figure 15

This docker compose is being deployed on the team's server, the opensearch dashboard and the grafana can be accessed at: https://gha.hock.hu/opensearch/ and https://gha.hock.hu/grafana/

5.2 Gha.conf

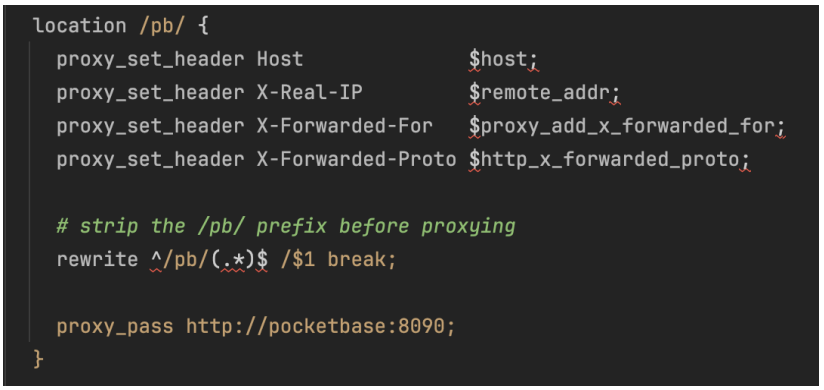
The GHA system uses Nginx reverse proxy to expose multiple internal services through a single entry point. Nginx acts as a gateway which forwards incoming HTTP requests to the appropriate service based on the requested URI prefix. This is better than having to publish each container directly to the internet, and it improves security, as well as avoiding CORS issues.

The configuration file gha.conf made by the team defines one virtual server that listens on port 80 and contains multiple location blocks.

The first line of the file defines the virtual server that listens on port 80 and receives user traffic from the haproxy that terminates the HTTPS. All the location blocks that come afterwards apply to this server.

Figure 16

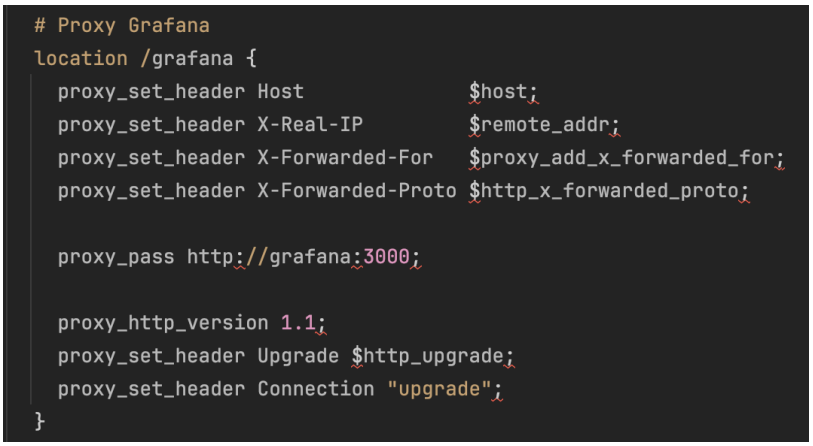
The PocketBase block forwards all requests starting with /pb/ to the PocketBase container running on port 8090. The rewrite rule removes the /pb/ prefix before forwarding the request.



```
location /pb/ {
  proxy_set_header Host              $host;
  proxy_set_header X-Real-IP         $remote_addr;
  proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $http_x_forwarded_proto;

  # strip the /pb/ prefix before proxying
  rewrite ^/pb/(.*)$ /$1 break;

  proxy_pass http://pocketbase:8090;
}
```

Figure 17

The Grafana block exposes Grafana through the /grafana path. Requests are forwarded to the Grafana container on port 3000. WebSocket related headers, namely “Upgrade” and “Connection”, are required for live dashboards, updates.



```
# Proxy Grafana
location /grafana {
  proxy_set_header Host              $host;
  proxy_set_header X-Real-IP         $remote_addr;
  proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $http_x_forwarded_proto;

  proxy_pass http://grafana:3000;

  proxy_http_version 1.1;
  proxy_set_header Upgrade $http_upgrade;
  proxy_set_header Connection "upgrade";
}
```

Figure 18

Grafana calls API endpoints using absolute paths such as /apis/… . The block below deals with a bug in Grafana 12.3.0 in which some Grafana features don't use the configured /grafana prefix. This bug is better documented here: https://github.com/grafana/grafana/issues/114213

Without this, some dashboard features and API calls would fail. This bug is fixed in Grafana 12.3.2, which came out recently at the time of writing.

```
# Grafana "new API structure" endpoints (Grafana may call these as /apis/... even with subpath UI)
location ^~ /apis/ {
  proxy_set_header Host              $host;
  proxy_set_header X-Real-IP         $remote_addr;
  proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $http_x_forwarded_proto;

  proxy_pass http://grafana:3000;
}
```

Figure 19

The Keycloak block is straightforward, as it exposes the Keycloak authentication server. Keycloack provides centralised authentication and is used by Grafana and PocketBase. Routing with Nginx makes sure that authentication traffic and the platform use the same domain.

```
location /keycloak {
  proxy_set_header Host              $host;
  proxy_set_header X-Real-IP         $remote_addr;
  proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $http_x_forwarded_proto;

  proxy_pass http://keycloak:8080;
}
```

Figure 20

This block forwards requests starting with /weather to the external Open-Meteo API used to retrieve real-world weather data for comparisons. Clients can call weather/v1/forecast?... and Nginx forwards the request to api.open-meteo.com. This ensures that any CORS issues are avoided.

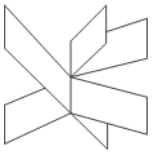
```
# /weather/v1/forecast?latitude=47.406525&longitude=18.938877&current=temperature_2m,relative_humidity_2m,shortwave_radiation
 location /weather/ {
   proxy_ssl_server_name on;

   proxy_set_header Host api.open-meteo.com;
   proxy_set_header X-Real-IP         $remote_addr;
   proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
   proxy_set_header X-Forwarded-Proto $http_x_forwarded_proto;

   proxy_pass https://api.open-meteo.com/;
 }
```

Figure 21

The last location block has the purpose of serving the React webpage for the project. The try_files attempts to serve the requested file and if it doesn't exist, the request falls back to index.html.

```
# Serve your webpage
location / {
  root /usr/share/nginx/html;
  try_files $uri $uri/ /index.html;
}
```

Figure 22



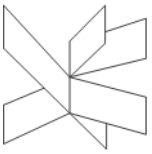
5.3 Filebeat.yml

The Filebeat.yml file outlines how Filebeat collects data from the MQTT and then sends it to Logstash where it will be processed. The entire Filebeat.yml can be viewed below.

```
strict.perms: false
logging.level: info
logging.to_files: false

filebeat.inputs:
  - type: log
    paths:
      - /usr/share/filebeat/logs/*.ndjson
  - type: mqqt
    enabled: true
    hosts:
      - "mqtt://mqtt:1883"
    username: "ghasensor"
    password: "*****"
    topics:
      - "ghanode/sensor"
    data_format: plain
    tags: ["mqtt", "ghanode"]
    fields:
      app: ghanode_sensor
    fields_under_root: true

output.logstash:
  hosts: ["logstash:5044"]

setup.ilm.enabled: false
setup.template.enabled: false
```

Figure 23

The data is received from MQTT, but before the connection to the IOT devices was established, the system was tested using JSON files which were read from the Filebeat log directory, as seen below:



```
filebeat.inputs:
  - type: log
    paths:
      - /usr/share/filebeat/logs/*.ndjson
```

Figure 24

This ensured an easy and quick way of testing the workflow of the system in the beginning of implementation.

The other and more important input is the MQTT input which makes sure that the Filebeat listens directly to an MQTT, connects to the container correctly using its hostname and port, and authenticates with the correct username and password. The data format is set to plain, and it also creates and adds a field name in the root called “app: ghanode_sensor”.

The output is then forwarded to Logstash, using its hostname and port, as it can be seen below.

```
output.logstash:
  hosts: ["logstash:5044"]
```

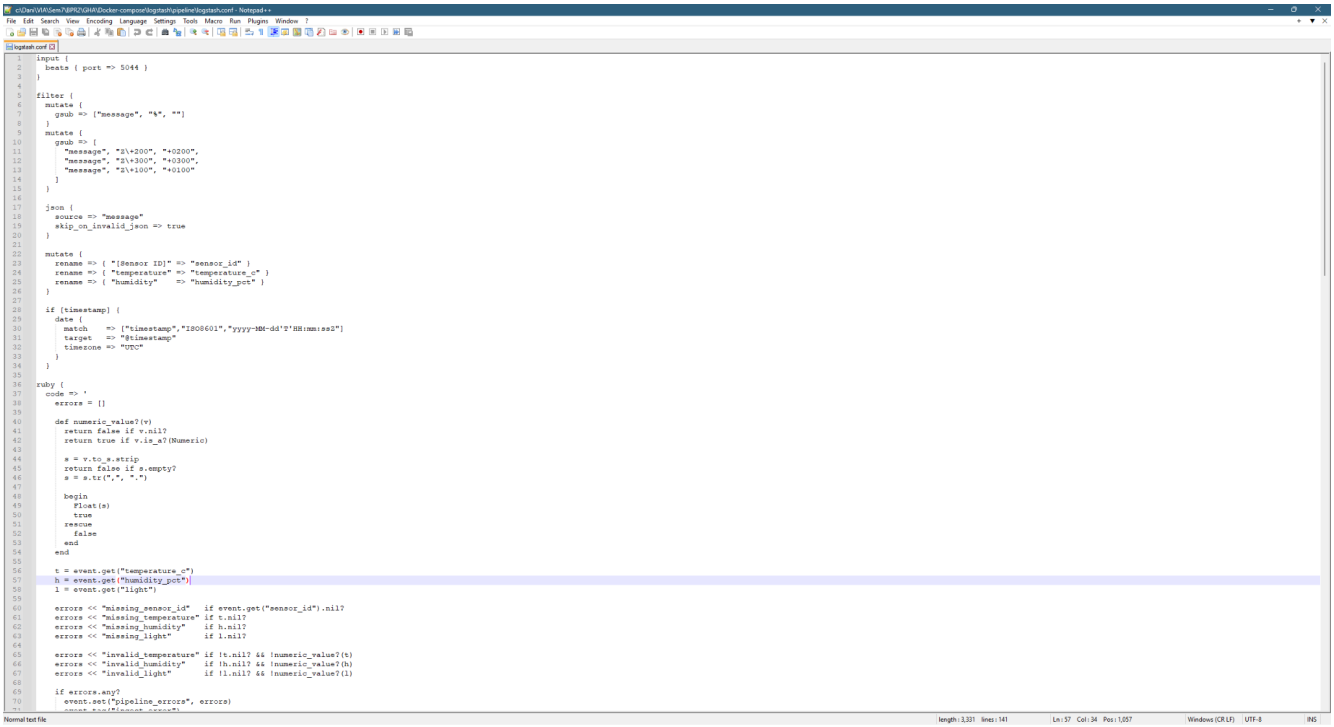
Figure 25

5.4 Logstash.conf

The Logstash configuration file defines how the incoming sensor data will be processed before being sent to Opensearch. It receives the events from

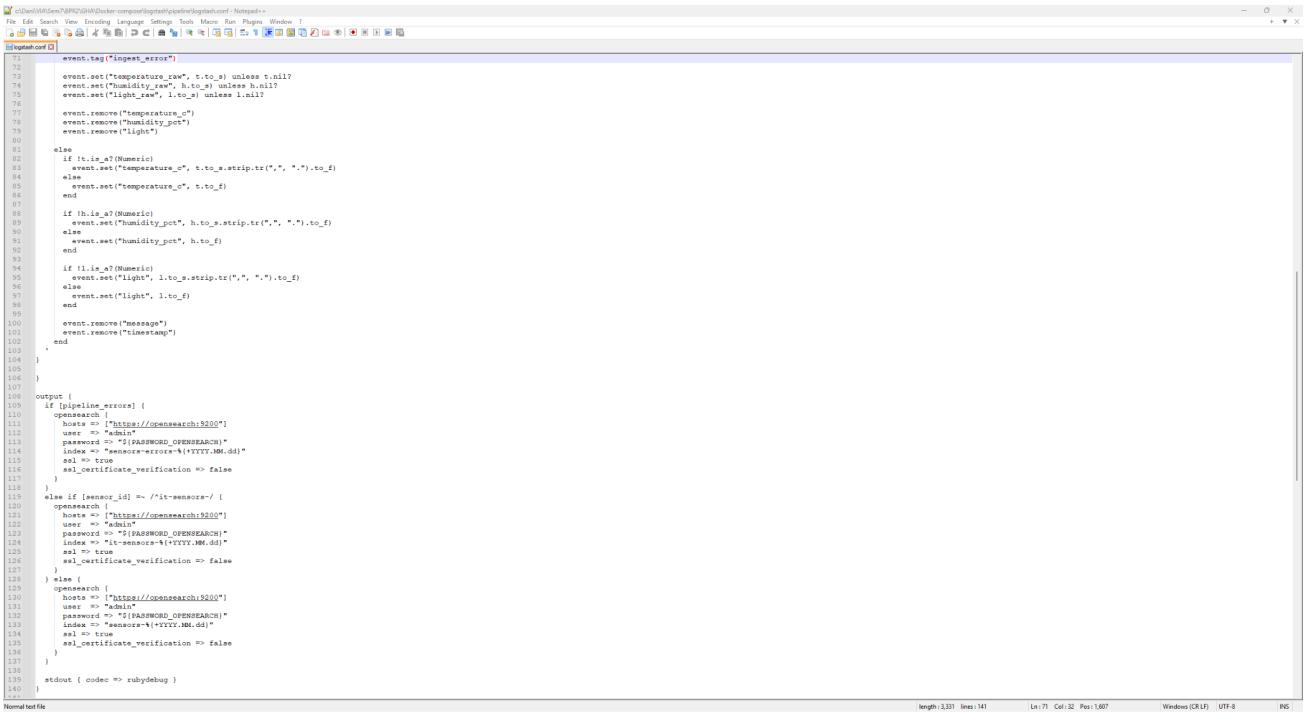
Filebeat, as mentioned above, and then performs validations and conversions, and then routes either valid data or error data to different OpenSearch indices. The design of the file supports later integration tests, by making sure correct payloads end up in the expected index and invalid or missing payloads appear in a dedicated error index with their respective error reasons.

Below is the logstash.conf file developed for the GHA system.



```
input {
  beats { port => 5044 }
}

filter {
  mutate {
    gsub => ["message", "%", ""]
  }
  mutate {
    gsub => [
      "message", "Z\+200", "+0200",
      "message", "Z\+300", "+0300",
      "message", "Z\+100", "+0100"
    ]
  }

  json {
    source => "message"
    skip_on_invalid_json => true
  }

  mutate {
    rename => { "[Sensor ID]" => "sensor_id" }
    rename => { "temperature" => "temperature_c" }
    rename => { "humidity"    => "humidity_pct" }
  }

  if [timestamp] {
    date {
      match    => ["timestamp","ISO8601","yyyy-MM-dd'T'HH:mm:ss2"]
      target   => "@timestamp"
      timezone => "UTC"
    }
  }

ruby {
  code => '
    errors = []

    def numeric_value?(v)
      return false if v.nil?
      return true if v.is_a?(Numeric)

      s = v.to_s.strip
      return false if s.empty?
      s = s.tr(",", ".")

      begin
        Float(s)
        true
      rescue
        false
      end
    end

    t = event.get("temperature_c")
    h = event.get("humidity_pct")
    l = event.get("light")

    errors << "missing_sensor_id"   if event.get("sensor_id").nil?
    errors << "missing_temperature" if t.nil?
    errors << "missing_humidity"    if h.nil?
    errors << "missing_light"       if l.nil?

    errors << "invalid_temperature" if !t.nil? && !numeric_value?(t)
    errors << "invalid_humidity"    if !h.nil? && !numeric_value?(h)
    errors << "invalid_light"       if !l.nil? && !numeric_value?(l)

    if errors.any?
      event.set("pipeline_errors", errors)
```

Figure 26



```
      event.tag("ingest_errors")

      event.set("temperature_raw", t.to_s) unless t.nil?
      event.set("humidity_raw", h.to_s) unless h.nil?
      event.set("light_raw", l.to_s) unless l.nil?

      event.remove("temperature_c")
      event.remove("humidity_pct")
      event.remove("light")

    else
      if !t.is_a?(Numeric)
        event.set("temperature_c", t.to_s.strip.tr(",", ".").to_f)
      else
        event.set("temperature_c", t.to_f)
      end

      if !h.is_a?(Numeric)
        event.set("humidity_pct", h.to_s.strip.tr(",", ".").to_f)
      else
        event.set("humidity_pct", h.to_f)
      end

      if !l.is_a?(Numeric)
        event.set("light", l.to_s.strip.tr(",", ".").to_f)
      else
        event.set("light", l.to_f)
      end

      event.remove("message")
      event.remove("timestamp")
    end
  '
}

}

output {
  if [pipeline_errors] {
    opensearch {
      hosts => ["https://opensearch:9200"]
      user  => "admin"
      password => "${PASSWORD_OPENSEARCH}"
      index => "sensors-errors-%{+YYYY.MM.dd}"
      ssl => true
      ssl_certificate_verification => false
    }
  }
  else if [sensor_id] =~ /^it-sensors-/ {
    opensearch {
      hosts => ["https://opensearch:9200"]
      user  => "admin"
      password => "${PASSWORD_OPENSEARCH}"
      index => "it-sensors-%{+YYYY.MM.dd}"
      ssl => true
      ssl_certificate_verification => false
    }
  } else {
    opensearch {
      hosts => ["https://opensearch:9200"]
      user  => "admin"
      password => "${PASSWORD_OPENSEARCH}"
      index => "sensors-%{+YYYY.MM.dd}"
      ssl => true
      ssl_certificate_verification => false
    }
  }

  stdout { codec => rubydebug }
}
```

Figure 27

The first few lines of the configuration file show how the Logstash is set up to listen for events from Filebeat on a port. The data is being sent from Filebeats over Beats, as seen below.

```
input {
  beats { port => 5044 }
}
```

Figure 28

The next parts of the file have the role of processing and polishing the data and removing unwanted characters, such as the percentage symbol that is

received with the humidity value. Other transformations included changing timezone offsets that were undesired with more traditional values, and renaming fields to have more meaningful names which include the metric of the sensor value. All of these were done using the mutate filter, as it can be observed below.

```
filter {
  mutate {
    gsub => ["message", "%", ""]
  }
```

Figure 29

```
mutate {
  gsub => [
    "message", "Z\+200", "+0200",
    "message", "Z\+300", "+0300",
    "message", "Z\+100", "+0100"
  ]
}
```

Figure 30

```
mutate {
  rename => { "[Sensor ID]" => "sensor_id" }
  rename => { "temperature" => "temperature_c" }
  rename => { "humidity"    => "humidity_pct" }
}
```

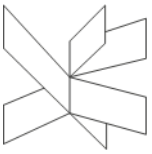
Figure 31

Afterwards, it checks whether the “timestamp” field from the JSON exists and parses it into an actual timestamp value. The end result is then written into @timestamp to be used by Opensearch for the time-series.

```
if [timestamp] {
  date {
    match    => ["timestamp","ISO8601","yyyy-MM-dd'T'HH:mm:ssZ"]
    target   => "@timestamp"
    timezone => "UTC"
  }
}
```

Figure 32

After that, the next part of the file consists of a ruby validation block which contains the core logic of the logstash configuration file. It first creates an array “error” to collect multiple validation errors in one event, such as, if there



was a payload with multiple missing fields and some invalid types, the error answer will contain all of the errors together, not just one of them.

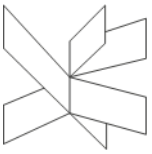
```
ruby {
  code => '
    errors = []
```

Figure 33

Afterwards, a function was made that returns true if a value can be interpreted as a number, even if it arrives as a string.

```
def numeric_value?(v)
  return false if v.nil?
  return true if v.is_a?(Numeric)

  s = v.to_s.strip
  return false if s.empty?
  s = s.tr(",", ".")

  begin
    Float(s)
    true
  rescue
    false
  end
end
```

Figure 34



The reason why the team used Ruby here instead of a much simpler “mutate convert” was because that can silently turn invalid values into something misleading or create mapping conflicts. On the other hand, Ruby allows more explicit logic, and it was used to detect invalid numeric fields before conversion and make a clear decision about routing.

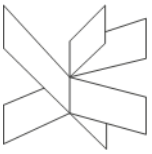
The next batch of code deals with missing and invalid checks. Firstly, the key sensor fields are extracted. They can either be “nil”, which means they are missing, numeric or strings.

```
t = event.get("temperature_c")
h = event.get("humidity_pct")
l = event.get("light")

errors << "missing_sensor_id"   if event.get("sensor_id").nil?
errors << "missing_temperature" if t.nil?
errors << "missing_humidity"    if h.nil?
errors << "missing_light"       if l.nil?

errors << "invalid_temperature" if !t.nil? && !numeric_value?(t)
errors << "invalid_humidity"    if !h.nil? && !numeric_value?(h)
errors << "invalid_light"       if !l.nil? && !numeric_value?(l)
```

Figure 35

Then, the system adds missing eros for every missing field. No else-if was used so that multiple fields get collected. Afterwards, the system checks invalidity only if the value exists. This way, a missing field won’t cause an invalid flag.

The code then deals with the two possible scenarios: error and valid. For the error case, it adds pipeline_errors as an array of strings as well as a



“ingest_error” tag. It also stores the original values for troubleshooting by saving them in “_raw” fields. Finally, it removes the numeric fields from error events. This is crucial in order to avoid OpenSearch mapping errors. If, for example, temperature_c is mapped as float, sending a string such as “NOT_A_NUMBER” would break indexing.

```
if errors.any?
  event.set("pipeline_errors", errors)
  event.tag("ingest_error")

  event.set("temperature_raw", t.to_s) unless t.nil?
  event.set("humidity_raw", h.to_s) unless h.nil?
  event.set("light_raw", l.to_s) unless l.nil?

  event.remove("temperature_c")
  event.remove("humidity_pct")
  event.remove("light")
```

Figure 36

For the valid event case, it starts converting the temperature, humidity and light values to float safely. Afterwards, it removes raw fields from valid events to keep the indexed document clean.

```
  else
    if !t.is_a?(Numeric)
      event.set("temperature_c", t.to_s.strip.tr(",", ".").to_f)
    else
      event.set("temperature_c", t.to_f)
    end

    if !h.is_a?(Numeric)
      event.set("humidity_pct", h.to_s.strip.tr(",", ".").to_f)
    else
      event.set("humidity_pct", h.to_f)
    end

    if !l.is_a?(Numeric)
      event.set("light", l.to_s.strip.tr(",", ".").to_f)
    else
      event.set("light", l.to_f)
    end

    event.remove("message")
    event.remove("timestamp")
  end
'
```

Figure 37

The last part consists of the output block, which checks if pipeline_errors exist and if affirmative, routes to the error index “sensors-errors-*”. If it is valid and the sensor_id matches the test prefix “it-sensors-xx” it routes to it-sensors-*”. If neither of those is the case, the valid events go to the default production index “sensors-*”. Finally, the full events are printed to the logs for debugging in a readable format.

The result of this can also be seen inside the Opensearch dashboard, as in the image below. In all of the indexes, the mappings reflect the conversions that were done in the config file.

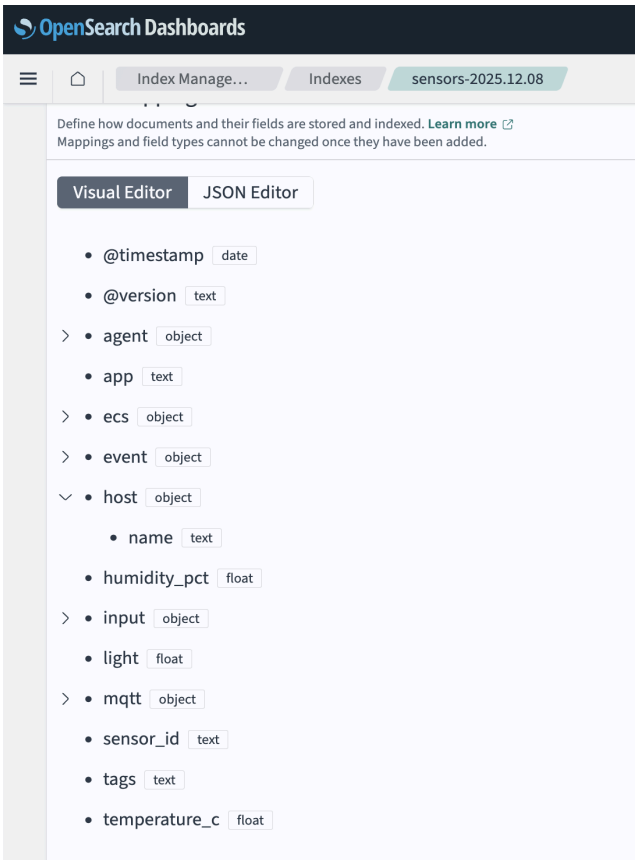


Figure 38

The created indexes according to the pattern for each day can then be observed in the Opensearch Dashboard, as can be seen below, along with the total number of entries for each day, which is around 2880 documents per day, since the device sends data every 30 seconds.

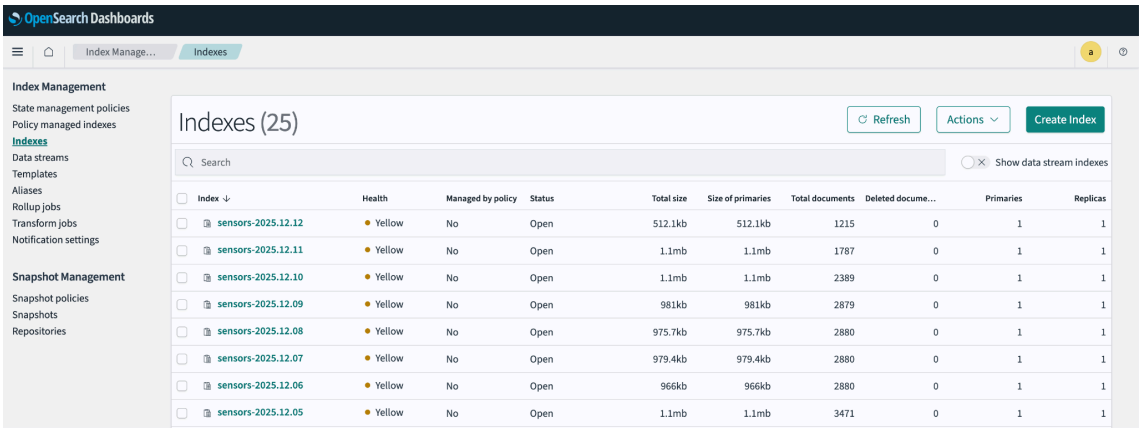


Figure 39

5.5 Grafana querying

5.5.1 Daily values in % dashboard

One of the first dashboards that the team created in Grafana was the “Daily values in %” dashboard, which served the role of calculating what % of the last 24 hours did the temperature, humidity and light sensor values remain in a specific range which was deemed the optimal range. This was done using queries from Grafana to the Opensearch database.

For example, in order to see whether the temperature was in the optimal range for the last 24 hours, the first query was to find out the total number of temperatures stored in the last 24 hours. The sensor sends data every 30 seconds, so the total number will always be around 2880 values of each field. To get this result, the team first set the time range of the dashboard to the last

24 hours and then queried the database for the total count of “temperature_c” values recorded in all of the indexes. The result was stored in a field named “total”, while the query was written in PPL as it can be seen below.

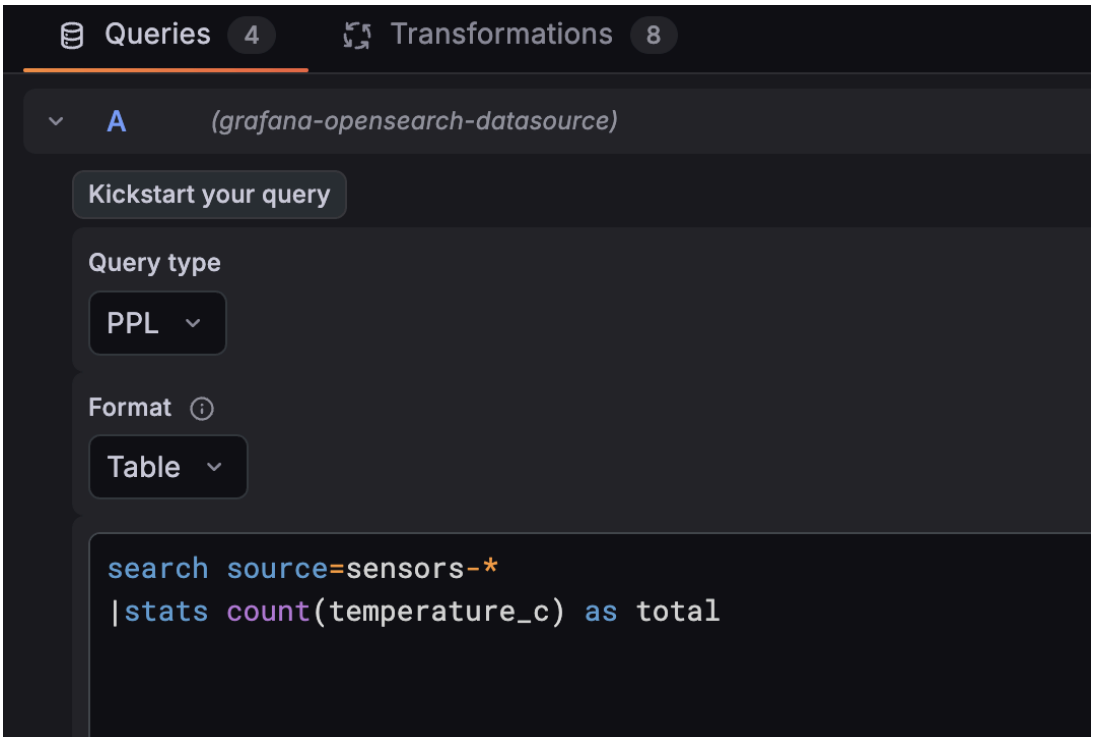


Figure 40

Afterwards, a similar query was written this time to get the count of the total times the temperature was recorded between the optimal range, which was between 10 and 30 degrees Celsius. The result was kept in a field called “in_range_temp” to be used for the next transformations.

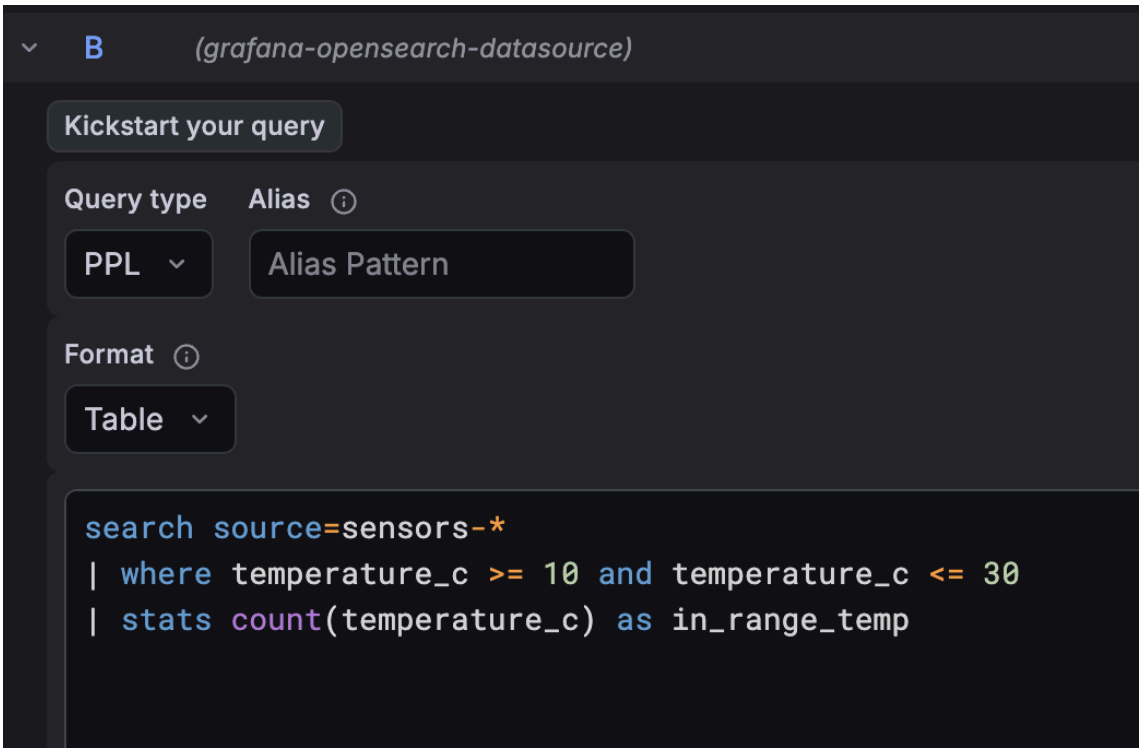


Figure 40

After the team had the new fields, it moved onto the transformations. Firstly, it needed to concatenate them so that they appeared on the same row inside the table and they can be used in future calculations. Grafana allowed for this easily as seen below, by just copying the frame name to the field name.

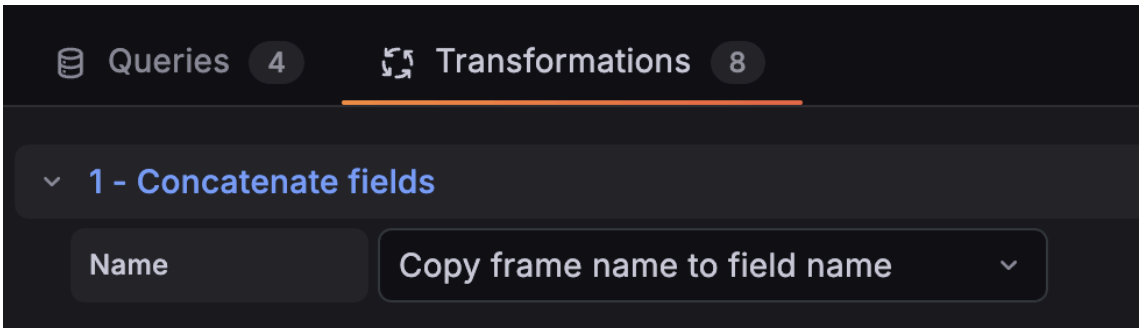


Figure 41

Afterwards, the last parts of resolving the daily value in % for the temperature were completed by first performing a calculation of binary operation mode in which the “in_range_temp” is divided by the “total” value and stored in a “fraction_temp” field. The last step was to multiply this value by 100 in order to get the percentage. These two calculations can be seen below.

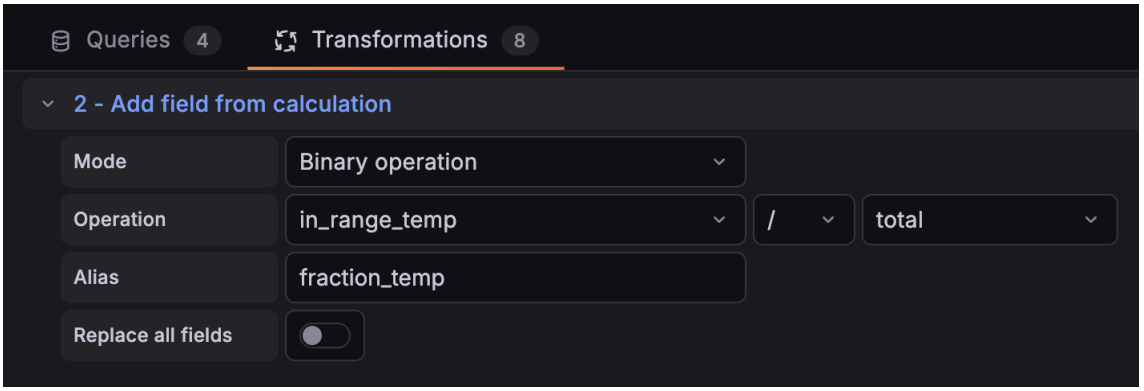


Figure 42

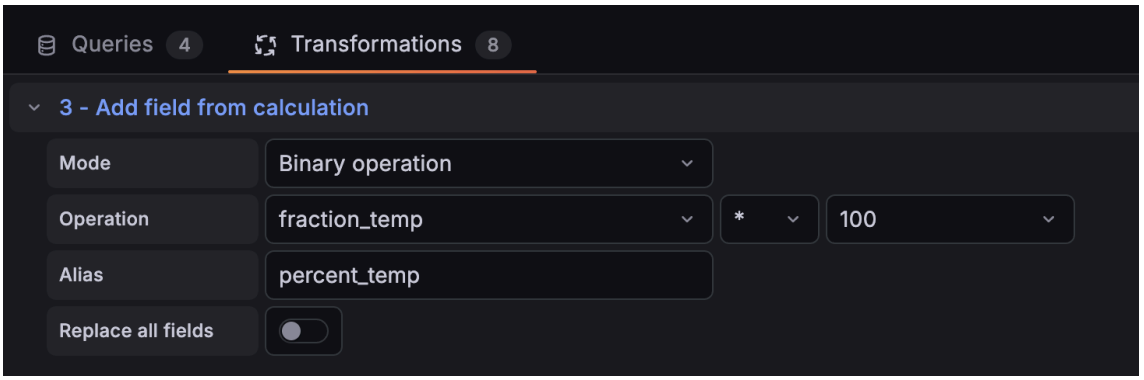


Figure 43

This entire process was then repeated for the humidity and light values with success, and then the team chose to display only the percentages on the dashboard as they represent the important part of the dashboard, and not display the other calculations made along the way.

Figure 44

Above can be seen the completed dashboard, which shows that the temperature and humidity never went out of range, while the light only was in the optimal range for around 40% of the time. This is largely due to the fact that the light sensor wasn't set up in the correct place from the beginning.

5.5.2 Environmental Data dashboard

A more simple dashboard created by the team was the "Environmental Data" dashboard. Since the data was stored efficiently using indexes inside the Opensearch, the team could easily retrieve the values from there and then apply the "Average" metric to each one for the last 24 hours and then group by the @timestamp date histogram, as it can be seen below.

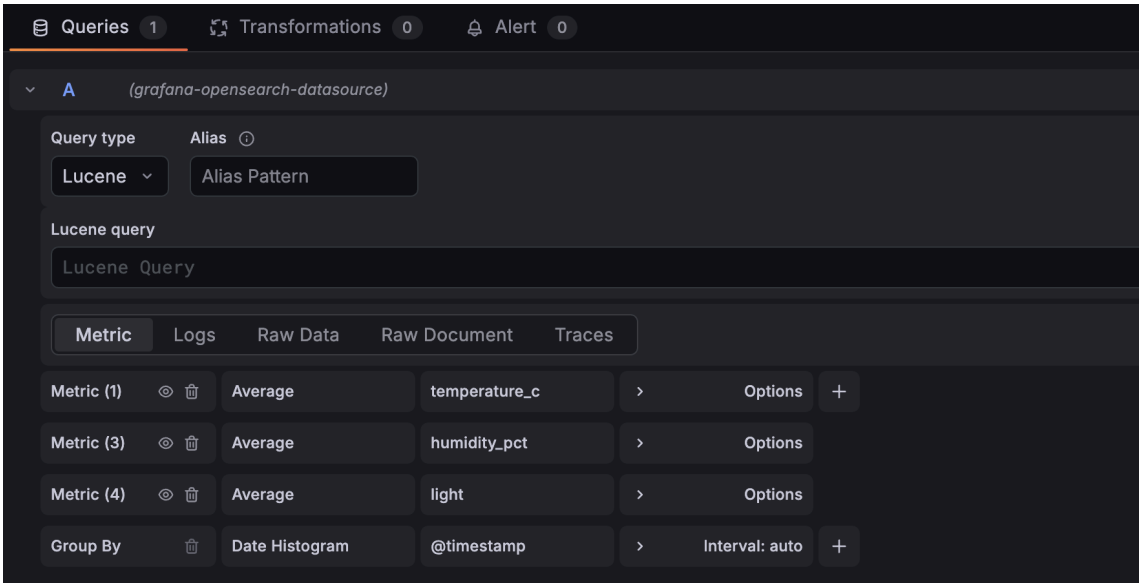


Figure 45

The result is an instant, easy-to-read dashboard that the user can look at and understand the current values of each field for his greenhouses.

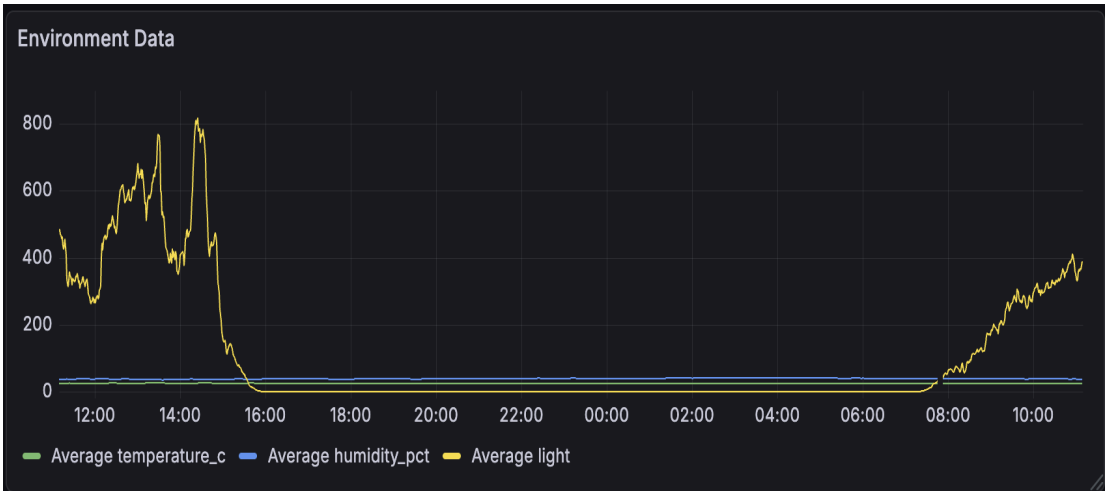


Figure 46

5.6 Grafana alerts

Finally, the last implementation was the alerts. The team wanted to make sure that the owner would be notified in case either of the values would be outside of their defined range. These are called alert rules, which determine when an alert should fire based on specific ranges set up by the team.

For example, the humidity alert rule was created first by defining the query that it should monitor, which is the average humidity retrieved the same way it was done in the “Environmental Data” dashboard. Afterwards, the expressions were made which are used to manipulate the data returned from the queries and set up the alert condition. For the humidity, the first expression takes the last numeric value from the query. The second expression then takes this value and checks if it is outside the range between 20 and 50. This

expression is set as the “alert condition” which means that when it is triggered, the alert will be sent to the user. The last two expressions can be observed below.

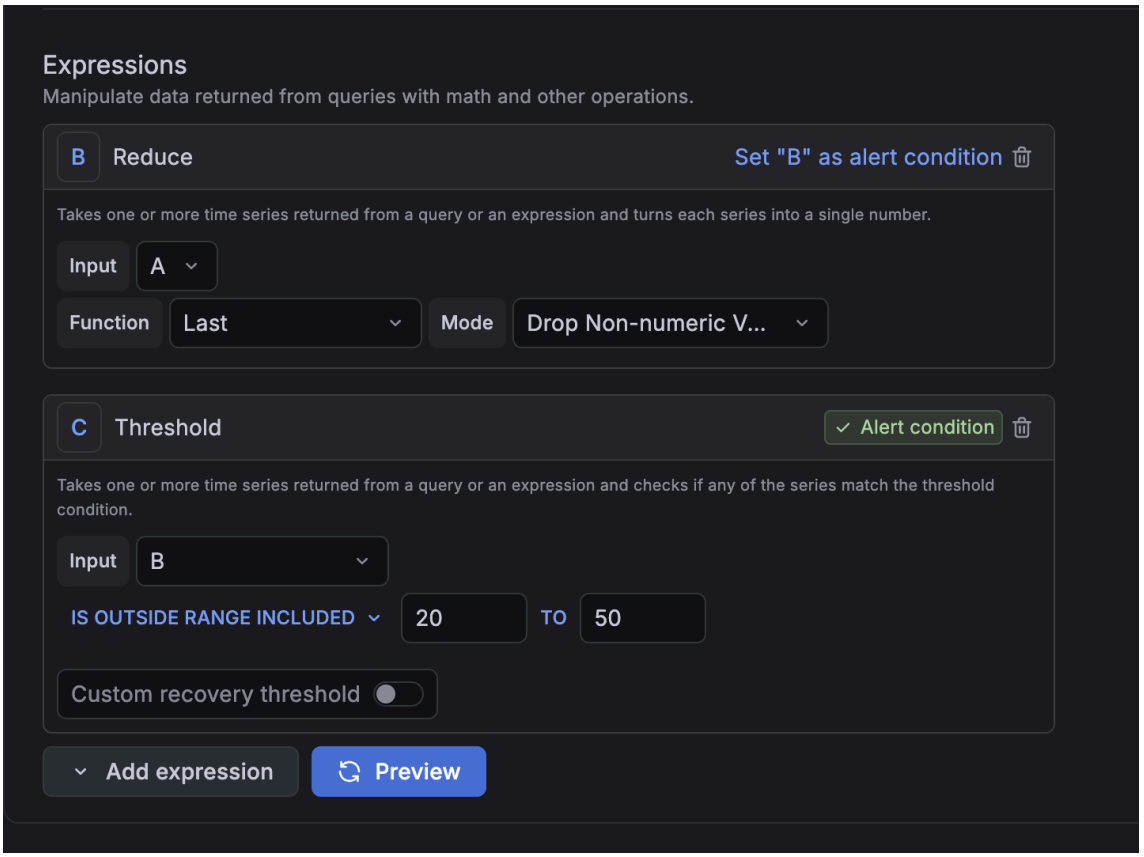


Figure 47

Afterwards, the alert is added to a folder where other future alerts will be added, such as the one for temperature, which follows the same method. Then the evaluation behaviour is set which, for both humidity and temperature, makes sure that the rules are evaluated every hour and that it only sends the alert

when the value has been outside of the range for more than one hour. It will also send a “Resolved” alert when the values are back to normal.

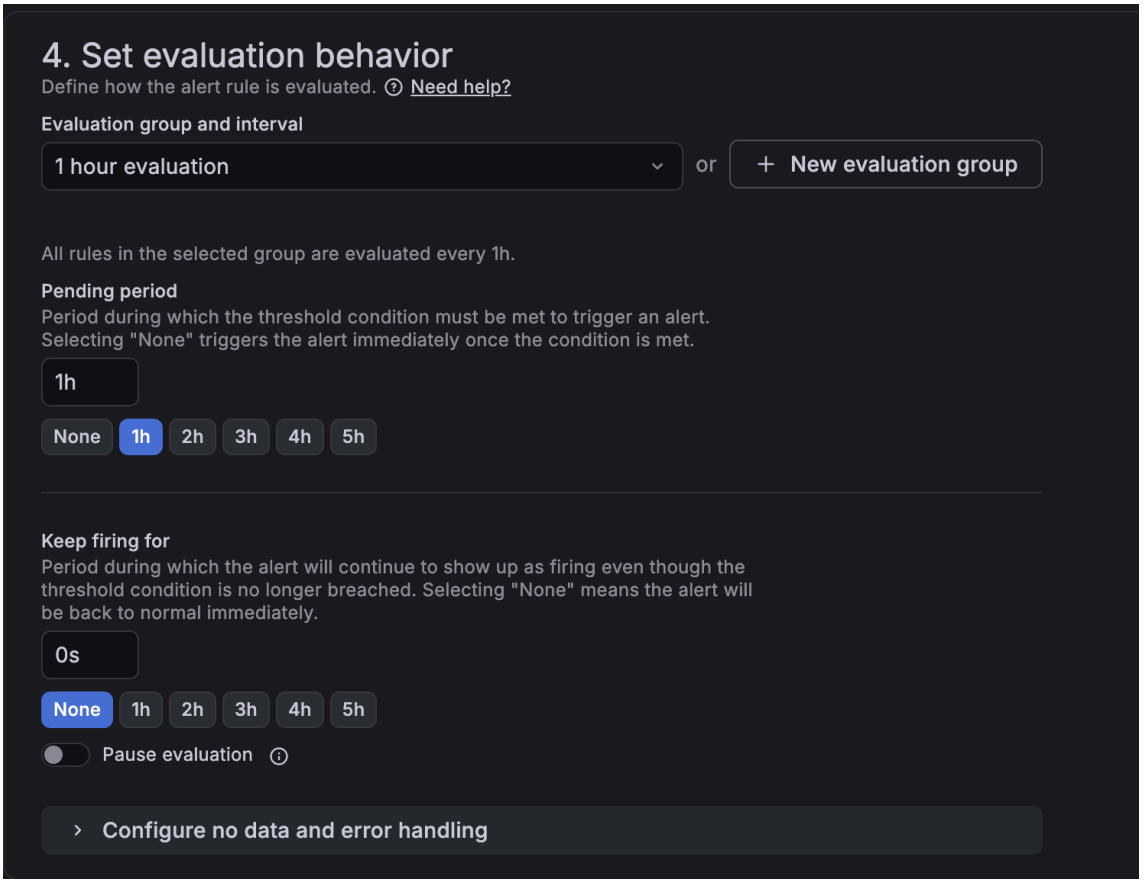


Figure 48

The next step was to configure the notifications. The contact point was set to a discord server in order for the team to test it more efficiently. It can also be sent through email which is what the user will want.

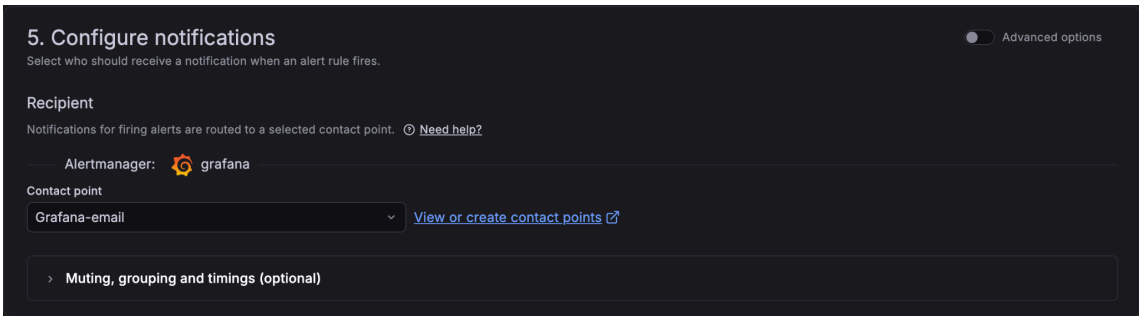


Figure 49

The last part of the alert set up was to configure notification messages which were kept to a simple but informative manner in order for the user to know exactly what is wrong in an efficient way.

The finished alert in function works without issues and sends alerts when the values are outside of the range under “Firing” state and then under “Resolved” state when the alert is no longer triggered. Both situations can be seen in the images below.

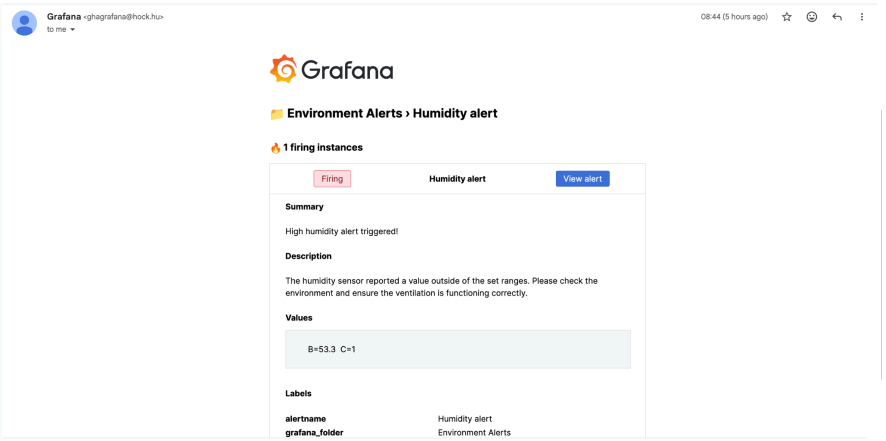


Figure 50

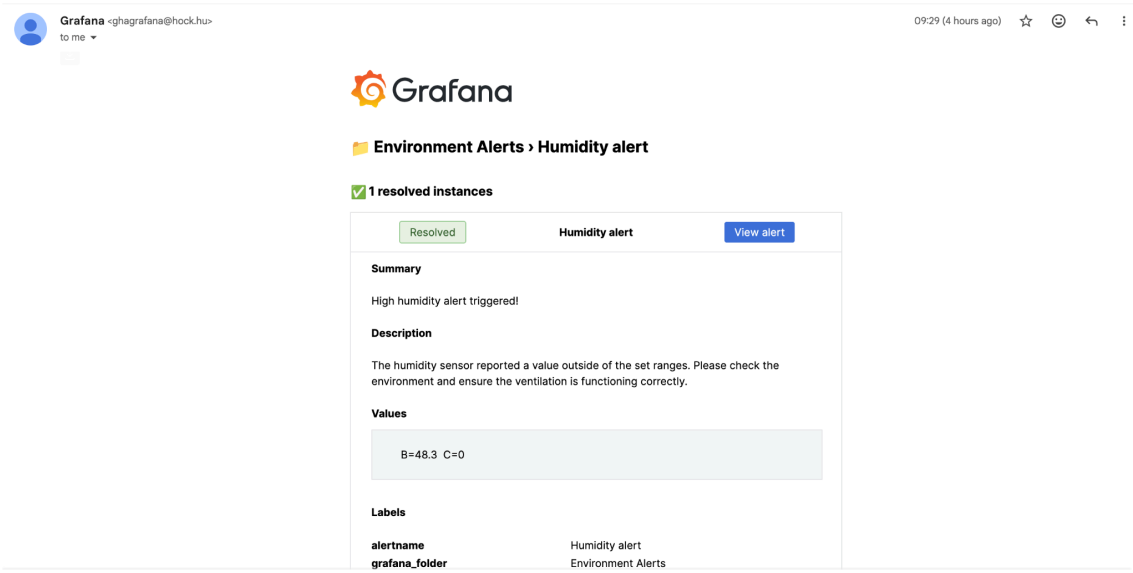


Figure 51

The temperature alert rule is set up in the same manner, with its ranges being different from humidity, but with the same structure and logic behind it.

5.7 Frontend

While there are many interesting features on the Frontend, being able to add a new greenhouse is one of the most important ones, so let's talk about how it works. On the UI, the user just pushes the "Add a new greenhouse" button, a form appears, they fill it out, click on save and that's it, they have a new greenhouse on their page. Seems simple, but how does it look in the code?

Below is the UI presentation of adding a greenhouse.

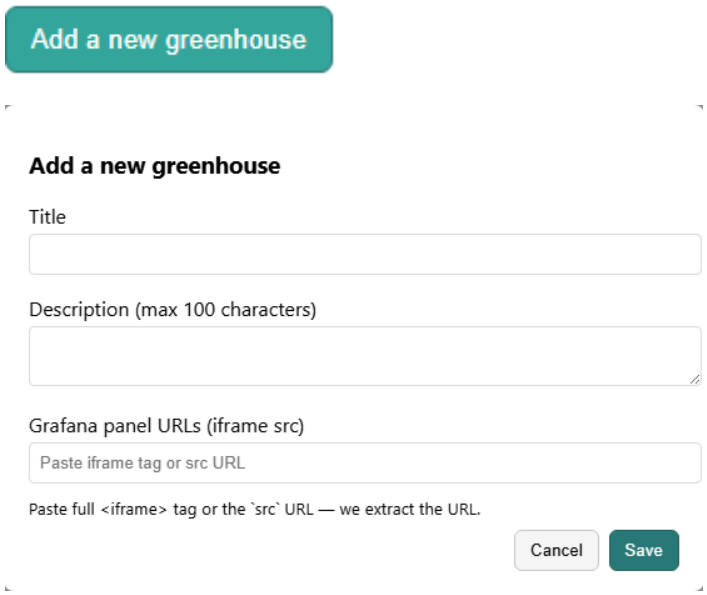


Figure 52

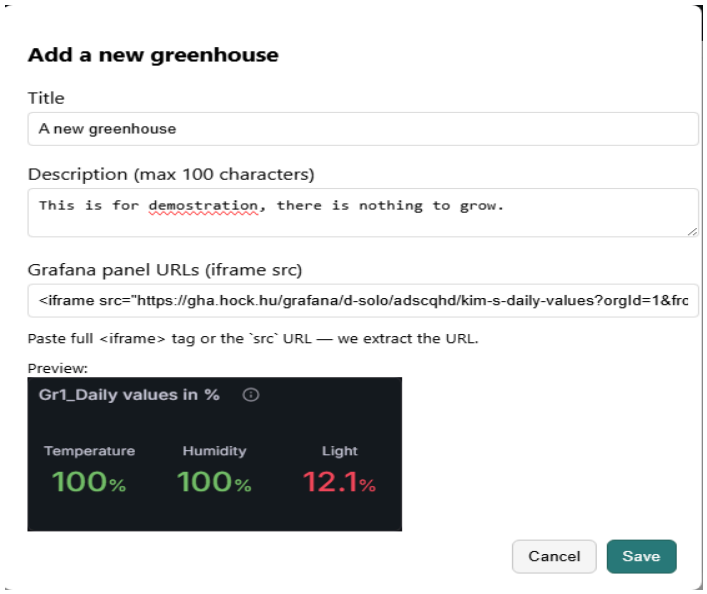


Figure 53

Figure 54

```
<div className="header-center">
  <button className="add-btn" onClick={handleAddClick}>
    Add a new greenhouse
  </button>
</div>
```

Figure 55

When the user clicks on the button the “handleAddClick” is being called. handleAddClick is just to set the editing item to null and the modal to true, which opens the AddGreenhouseModal.js.

```
function handleAddClick() {
  setEditingItem(null);
  setModalOpen(true);
}
```

Figure 56

```
useEffect(() => {
  if (open) {
    setTitle(initial.title || '');
    setDescription(initial.description || '');
    setOrder(initial.order ?? 0);
    const gd = initial.grafanadata;
    if (gd && Array.isArray(gd)) {
      setGrafanaUrl(gd[0].grafanaURL);
    } else {
      setGrafanaUrl('');
    }
    setErrors({});
  }
}, [open, initial]);

function validate() {
  const out = {};
  if (!title || !title.toString().trim()) out.title = 'Title is required.';
  if (grafanaUrl && grafanaUrl.toString().trim()) {
    const extracted = extractGrafanaUrl(grafanaUrl);
    if (extracted && !normalizeGrafanaUrl(extracted)) {
      out.grafanaUrl = 'Invalid URL. Paste the iframe src or a full URL.';
    }
  }
  return out;
}
```

Figure 57

Since the modal is in the if open = true, the modal is waiting for a given “title”, “description”, “order” and “grafanadata”, which it either will get from the initial (that is if the user actually fill out the form as it was intended), or it gives a default value, such as empty string, or just 0 for the order. Then before this could be submitted, it goes through validation to make sure that the title is there, it extracts the grafanaURL from the iframe if it was pasted like that and if it’s incorrect, it will throw an error.

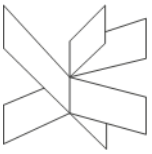
```
function handleSubmit(e) {
  e.preventDefault();
  const v = validate();
  if (Object.keys(v).length) {
    setErrors(v);
    return;
  }
  setErrors({});

  // Build grafanadata: [{ grafanaURL, panel_width: 350, panel_height: 200 }, ...]
  let grafanadata = [];
  if (grafanaUrl && grafanaUrl.toString().trim()) {
    const normalized = normalizeGrafanaUrl(grafanaUrl, { addTimeRange: true });
    grafanadata.push({
      grafanaURL: normalized || extractGrafanaUrl(grafanaUrl),
      panel_width: 350,
      panel_height: 200,
    });
  }
  onSave({ title: title || 'Untitled', description, order, grafanadata });
}
```

Figure 58

If the validate didn't give any errors, it proceeds to edit the grafanaUrl again to include the correct time range, since upon pasting the iframe it always pastes with the time when it was pasted, not the original 24 hours. Once that's done, it normalizes the size of the panel to be 350 wide and 200 tall and if everything could have been completed, it saves in the database and returns to the App.js among the rest of the loaded greenhouses.

6. Test

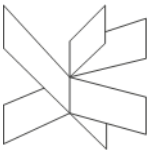
The project was tested both for its pipeline of handling data and for its frontend webpage. As for the pipeline testing, it used grey box integration testing, while for the webpage, it used white box testing with grey box style behavioral assertions. Below, the results of all requirements are showcased and



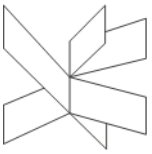
afterwards, the report will continue first with the integration testing and then with the webpage testing.

6.1 Test specifications

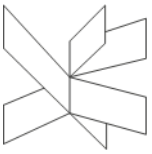
Requirement	Results	Comments
As a user, I want to have a secure connection from the sensors to the visualisation, so I can access it online.	Completed	
As a user, I want to have a platform where I can see what's going on in my greenhouses, so that I can maximise their efficiency.	Completed	
As a user, I want to have a login system so that I can use different accounts.	Completed	
As a user, I want to see a graph of how much time the temperature has spent in its respective range in percentages.	Completed	



As a user, I want to see a graph of how much time the humidity has spent in its respective range in percentages.	Completed	
As a user, I want to see a graph of how much time the light has spent in its respective range in percentages.	Completed	
As a user, I want to see a graph of the collected temperature in the past 24 hours, so that I can monitor how the situation changes in the greenhouses.	Completed	
As a user, I want to see a graph of the collected humidity in the past 24 hours, so that I can monitor how the situation changes in the greenhouses.	Completed	
As a user, I want to see a graph of the collected light in the past 24 hours, so that I can monitor how the situation changes in the greenhouses.	Completed	
As a user, I want to have a button which		



allows me to add a new Greenhouse.	Completed	
As a user, I want to have another page where I can see the details of my Greenhouse.	Completed	
As a user, I want to have a button which allows me to add a Grafana panel to the Greenhouse.	Completed	
As a user, I want to have a button which lets me delete Greenhouses.	Completed	
As a user, I want to have a button which lets me edit Greenhouses.	Completed	
As a user, I want to have a button which lets me delete Panels.	Completed	
As a user, I want to have a button which lets me edit Panels.	Completed	
As a user, I want to be able to compare the data inside of my greenhouse with the outside weather, so that I know how isolated my crops are from the outside.	Completed	



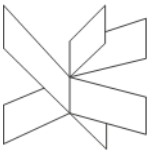
As a user, I want to be able to receive alerts when the values go outside of set ranges, so that I know that something is wrong.	Completed	

6.2 Integration testing

The testing implemented is best described, as mentioned above, as grey-box integration testing. The tests only interact through external interfaces, publishing sensor payloads to an MQTT broker and verifying the result in Opensearch, which is a sign of black-box testing. At the same time, the tests also use knowledge from the pipeline such as knowing certain behaviours and rules. The assertions check for specific internal error codes in the pipeline_errors field, such as missing_temperature, and verify that valid events are transformed into normalized field names (such as humidity to humidity_pct) before indexing. This internal awareness makes it so that the tests are not just black-box.

The goal of the tests was to validate the pipeline end-to-end, from the MQTT publisher all the way to Opensearch indexing. This ensured that the pipeline behaved correctly under realistic conditions.

The integration tests run in a containerized environment using a minimalized version of the Docker Compose, which includes only the services required to test the pipeline: MQTT, Filebeat, Logstash, Opensearch, and the test container. The full Docker Compose contains additional services, but they are excluded here to reduce runtime and isolate the pipeline which is validated.



The tests each follow the same structure. They create a unique identifier (sensor_id) to avoid collisions, they publish a JSON message to the MQTT topic using a real MQTT client, query Opensearch until the expected condition is met. For valid payloads, the document appears in the correct index, which was named “it-sensors-*” for valid tests, and has the correct schema and types. As for invalid payloads, the document doesn’t appear in the valid indices but instead appears in the error index with the correct attached pipeline_errors. After the tests are done, the test documents are deleted by query from both the valid and error indices. Below are helper functions for publishing, negative assertions and cleanup.

```python
def publish_mqtt(host, port, username, password, topic, payload, timeout=10):
    done = {"ok": False, "err": None}

    def on_connect(client, userdata, flags, rc):
        if rc != 0:
            done["err"] = RuntimeError(f"MQTT connect failed rc={rc}")
            return
        client.publish(topic, payload, qos=1)
        done["ok"] = True
        client.disconnect()

    client = mqtt.Client()
    client.username_pw_set(username, password)
    client.on_connect = on_connect

    client.connect(host, int(port), keepalive=30)
    client.loop_start()

    t0 = time.time()
    while time.time() - t0 < timeout and not done["ok"] and not done["err"]:
        time.sleep(0.1)

    client.loop_stop()

    if done["err"]:
        raise done["err"]
    if not done["ok"]:
        raise TimeoutError("MQTT publish did not complete in time")
```

Figure 59

```python
def assert_not_indexed(client, index_pattern, query, timeout_seconds=20):
    deadline = time.time() + timeout_seconds
    last_err = None
    while time.time() < deadline:
        try:
            res = client.search(index=index_pattern, body=query)
            hits = res.get("hits", {}).get("hits", [])
            if hits:
                hit = hits[0]
                raise AssertionError(
                    f"Expected NO document, but found one in index {hit.get('_index')} id={hit.get('_id')}"
                )
            return
        except Exception as e:
            last_err = e
        time.sleep(2)
    if last_err:
        raise AssertionError(f"Search kept failing for {index_pattern}. Last error: {last_err}")


def delete_test_docs_everywhere(client: OpenSearch, sensor_id: str) -> None:
    for index_pat in ["it-sensors-*", "sensors-errors-*"]:
        try:
            resp = client.delete_by_query(
                index=index_pat,
                body={"query": {"match_phrase": {"sensor_id": sensor_id}}},
                conflicts="proceed",
                refresh=True,
            )
            print(f"\n[cleanup] index={index_pat} sensor_id={sensor_id}")
            print(f"[cleanup] Response: {resp}\n")
        except Exception as e:
            print(f"\n[cleanup] WARNING index={index_pat} sensor_id={sensor_id}: {e}\n")
```

Figure 60

6.2.1 Valid payload: test_end_to_end_mqtt_to_opensearch

This test verifies that a valid sensor payload is successfully ingested, transformed, and indexed into the correct index (it-sensors-*). It asserts that the normalized fields exist and that their types are correct. This confirms correct parsing, renaming, conversion and routing behaviour when no errors occur.

```python
@pytest.mark.integration
def test_end_to_end_mqtt_to_opensearch():
    mqtt_host = os.getenv("IT_MQTT_HOST", "mqtt")
    mqtt_port = os.getenv("IT_MQTT_PORT", "1883")
    mqtt_user = os.getenv("IT_MQTT_USER", "ghasensor")
    mqtt_pass = os.getenv("IT_MQTT_PASS", "*****")
    mqtt_topic = os.getenv("IT_MQTT_TOPIC", "ghanode/sensor")

    os_host = os.getenv("IT_OS_HOST", "opensearch")
    os_port = os.getenv("IT_OS_PORT", "9200")
    os_user = os.getenv("IT_OS_USER", "admin")
    os_pass = os.getenv("IT_OS_PASS", os.getenv("PASSWORD_OPENSEARCH", ""))

    sensor_id = f"it-sensors-{uuid.uuid4().hex[:10]}"
    now_iso = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())

    payload_obj = {
        "Sensor ID": sensor_id,
        "temperature": 21.5,
        "humidity": 45.2,
        "light": 123.0,
    }

    client = make_opensearch_client(os_host, os_port, os_user, os_pass)

    index_pattern = "it-sensors-*"
    query = {
        "query": {"match_phrase": {"sensor_id": sensor_id}},
        "sort": [{"@timestamp": {"order": "desc"}}],
        "size": 1,
    }

    try:
        print("\n===== MQTT payload sent =====")
        pprint(payload_obj)
        print("============================\n")

        publish_mqtt(mqtt_host, mqtt_port, mqtt_user, mqtt_pass, mqtt_topic, json.dumps(payload_obj))

        deadline = time.time() + 60
        last_err = None

        while time.time() < deadline:
            try:
                res = client.search(index=index_pattern, body=query)
                hits = res.get("hits", {}).get("hits", [])
                if hits:
                    hit = hits[0]
                    doc = hit["_source"]

                    print("\n===== OpenSearch document found =====")
                    print(f"Index: {hit.get('_index')}")
                    print(f"Document ID: {hit.get('_id')}")
                    pprint(doc)
                    print("=====================================\n")

                    assert doc["sensor_id"] == sensor_id
                    assert isinstance(doc["temperature_c"], float)
                    assert isinstance(doc["humidity_pct"], float)
                    assert isinstance(doc["light"], float)
                    return
            except Exception as e:
                last_err = e

            time.sleep(2)

        raise AssertionError(f"Document not found in OpenSearch within timeout. Last error: {last_err}")

    finally:
        delete_test_docs_everywhere(client, sensor_id)
```

Figure 61

6.2.2 Missing field: test_payload_without_temperature

This test validates the pipeline's ability to detect missing values and to prevent bad payloads from entering valid indices. This specific test omits the temperature. The tests then makes sure that the document appears in the sensors-errors-* indice with pipeline_errors containing missing_temperature. This proves the pipeline enforces the fields which are required and also ensures that production indices will contain only valid data.

```python
@pytest.mark.integration
def test_payload_without_temperature():
    mqtt_host = os.getenv("IT_MQTT_HOST", "mqtt")
    mqtt_port = os.getenv("IT_MQTT_PORT", "1883")
    mqtt_user = os.getenv("IT_MQTT_USER", "ghasensor")
    mqtt_pass = os.getenv("IT_MQTT_PASS", "*****")
    mqtt_topic = os.getenv("IT_MQTT_TOPIC", "ghanode/sensor")

    os_host = os.getenv("IT_OS_HOST", "opensearch")
    os_port = os.getenv("IT_OS_PORT", "9200")
    os_user = os.getenv("IT_OS_USER", "admin")
    os_pass = os.getenv("IT_OS_PASS", os.getenv("PASSWORD_OPENSEARCH", ""))

    sensor_id = f"it-sensors-{uuid.uuid4().hex[:10]}"
    now_iso = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())

    payload_obj = {
        "Sensor ID": sensor_id,
        "humidity": 45.2,
        "light": 123.0,
    }

    client = make_opensearch_client(os_host, os_port, os_user, os_pass)

    normal_query = {"query": {"match_phrase": {"sensor_id": sensor_id}}, "size": 1}

    error_index_pattern = "sensors-errors-*"
    error_query = {
        "query": {
            "bool": {
                "must": [
                    {"match_phrase": {"sensor_id": sensor_id}},
                    {"match_phrase": {"pipeline_errors": "missing_temperature"}},
                ]
            }
        },
        "size": 1,
        "sort": [{"@timestamp": {"order": "desc"}}],
    }

    try:
        print("\n===== MQTT invalid payload sent (missing temperature) =====")
        pprint(payload_obj)
        print("==========================================================\n")

        publish_mqtt(mqtt_host, mqtt_port, mqtt_user, mqtt_pass, mqtt_topic, json.dumps(payload_obj))

        assert_not_indexed(client, "it-sensors-*", normal_query, timeout_seconds=25)
        assert_not_indexed(client, "sensors-*", normal_query, timeout_seconds=25)

        deadline = time.time() + 30
        last_err = None
        while time.time() < deadline:
            try:
                res = client.search(index=error_index_pattern, body=error_query)
                hits = res.get("hits", {}).get("hits", [])
                if hits:
                    doc = hits[0]["_source"]
                    print("\n===== Error document found in sensors-errors-* =====")
                    pprint(doc)
                    print("=====================================================\n")
                    assert doc["sensor_id"] == sensor_id
                    assert "pipeline_errors" in doc
                    assert "missing_temperature" in doc["pipeline_errors"]
                    assert "message" in doc or "event" in doc
                    return
            except Exception as e:
                last_err = e
            time.sleep(2)

        raise AssertionError(
            f"Expected error doc not found in {error_index_pattern}. Last error: {last_err}"
        )

    finally:
        delete_test_docs_everywhere(client, sensor_id)
```

Figure 62

6.2.3 Invalid numeric field: test_payload_with_invalid_temperature

This test sends a payload where temperature is a string. The purpose is to verify the handling of invalid types without breaking Opensearch indexing. The pipeline must detect the invalid value and add invalid_temperature to pipeline_errors, route the event to sensors-errors-*, and avoid indexing invalid type fields, such as avoid leaving temperature_c=”NOT_A_NUMBER” as a fields mapped as float in Opensearch, because it will cause a mapping error.

```python
@pytest.mark.integration
def test_payload_with_invalid_temperature():
    mqtt_host = os.getenv("IT_MQTT_HOST", "mqtt")
    mqtt_port = os.getenv("IT_MQTT_PORT", "1883")
    mqtt_user = os.getenv("IT_MQTT_USER", "ghasensor")
    mqtt_pass = os.getenv("IT_MQTT_PASS", "*****")
    mqtt_topic = os.getenv("IT_MQTT_TOPIC", "ghanode/sensor")

    os_host = os.getenv("IT_OS_HOST", "opensearch")
    os_port = os.getenv("IT_OS_PORT", "9200")
    os_user = os.getenv("IT_OS_USER", "admin")
    os_pass = os.getenv("IT_OS_PASS", os.getenv("PASSWORD_OPENSEARCH", ""))

    sensor_id = f"it-sensors-{uuid.uuid4().hex[:10]}"

    payload_obj = {
        "Sensor ID": sensor_id,
        "temperature": "NOT_A_NUMBER",
        "humidity": 45.2,
        "light": 123.0,
    }

    client = make_opensearch_client(os_host, os_port, os_user, os_pass)

    normal_query = {"query": {"match_phrase": {"sensor_id": sensor_id}}, "size": 1}

    error_query = {
        "query": {
            "bool": {
                "must": [
                    {"match_phrase": {"sensor_id": sensor_id}},
                    {"match_phrase": {"pipeline_errors": "invalid_temperature"}},
                ]
            }
        },
        "size": 1,
        "sort": [{"@timestamp": {"order": "desc"}}],
    }

    try:
        print("\n===== MQTT invalid payload sent (invalid temperature) =====")
        pprint(payload_obj)
        print("============================================================\n")

        publish_mqtt(mqtt_host, mqtt_port, mqtt_user, mqtt_pass, mqtt_topic, json.dumps(payload_obj))

        assert_not_indexed(client, "it-sensors-*", normal_query, timeout_seconds=25)
        assert_not_indexed(client, "sensors-*", normal_query, timeout_seconds=25)

        deadline = time.time() + 30
        last_err = None
        while time.time() < deadline:
            try:
                res = client.search(index="sensors-errors-*", body=error_query)
                hits = res.get("hits", {}).get("hits", [])
                if hits:
                    doc = hits[0]["_source"]

                    print("\n===== Error document found in sensors-errors-* =====")
                    pprint(doc)
                    print("=====================================================\n")

                    assert doc["sensor_id"] == sensor_id
                    assert "pipeline_errors" in doc
                    assert "invalid_temperature" in doc["pipeline_errors"]
                    return
            except Exception as e:
                last_err = e
            time.sleep(2)

        raise AssertionError(f"Expected invalid_temperature error doc not found. Last error: {last_err}")

    finally:
        delete_test_docs_everywhere(client, sensor_id)
```

Figure 63

As it can be seen below, the tests pass without errors, ensuring that all possible scenarios are handled.

Figure 64

Furthermore, all integration tests are executed automatically using Github Actions whenever code is pushed to the main branch or a pull request is opened. The CI pipeline builds the minimalised Docker Compose stack, and then runs all the integration tests inside a dedicated test container. This setup ensures that any change to the Logstash configuration, Docker environment, or test code is validated immediately. If a change produces an error in the tests, the CI job will fail and block the merge. Below is showcased a successful CI run where the Docker environment is built and all integration tests pass.

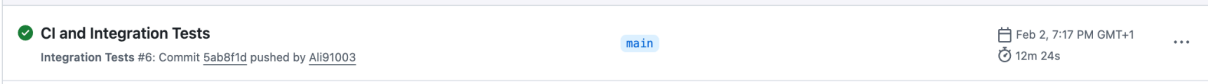
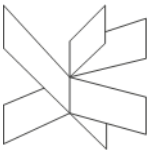


Figure 65

6.3 Webpage testing

The frontend used mostly white box testing with grey box style behavioral assertions. The goal was to achieve complete coverage in the system. All the



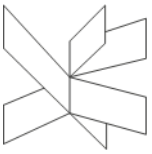
functionality has been tested in a separated environment in the form of unit or component tests with mocks, there are also tests where the team is testing integration in the form of asserting to UI and callbacks.

Below is a test case that is part of the **AuthContext.test.js.** This test file just like every other in the system is using **@testing-library/jest-dom.** The test case in particular is about making sure that the user does not have to log in again, even after reloading the page, since they did not logout.

A mocked version of the authstore was used, to set authstore's session to be valid, which means that they are already logged in, the authstore is given a dummy user, then the component tree is rendered, where the team is wrapping a small component *TestConsumer* that uses a function called *useAuth()* which reads the value that *AuthProvider* has put in the context. TestConsumer also shows if the user has loaded in, if it's authenticated and the user's id. At the end of the render, the DOM is updated and the test can finally query it. Since the original authentication is async, the tests also have to be async, the team is "awaiting" here until the conditions are true or they get timed out.

The first "expect" is looking for an element with *data-testid="loading"*, and asserts that its text value is *'false'*, indicating that the user has finished its initial load.

Finally, the team asserts that the user's id is displayed, confirming the user was restored from the auth store.



```
describe('AuthProvider', () => {
  test('sets user from authStore when valid on mount', async () => {
    mockAuthStore.isValid = true;
    mockAuthStore.record = { id: 'user1', email: 'u@test.com' };

    render(
      <AuthProvider>
        <TestConsumer />
      </AuthProvider>
    );

    await waitFor(() => {
      expect(screen.getByTestId('loading')).toHaveTextContent('false');
    });
    expect(screen.getByTestId('authenticated')).toHaveTextContent('true');
    expect(screen.getByTestId('user-id')).toHaveTextContent('user1');
  });
```

Figure 66

The next example shown is a component test case from the **CallbackPage.test.js**

The test case below is asserting that the authenticating UI of the callback page is rendered. This test case is synchronous, because it only checks that the right text is on the screen right after rendering.

It uses a mocked useAuth(), so when the CallbackPage calls it, it gets an object with “handleOAuthCallback” , a mock that resolves to true and “loading: false”. So the team can simulate that the auth isn’t loading on the page. Though the team isn’t asserting on “handleOAuthCallback” since they only need the page to render without throwing. The team then renders the component. In the render call, “MemoryRouter with initialEntries” is used so that the current URL is “/callback”, the team then

define a route so that “/callback” renders the “<CallbackPage/>”, once the render is over the “CallbackPage” has mounted, run its effect, which calls “handleOAuthCallback()”, and the dom is updated so it can be queried. This test does not wait for the “handleOAuthCallback” to resolve or for navigation, it only checks the initial UI.
Then it asserts that the exact text “Authenticating…” is present in the document, if “CallbackPage” didn’t render this text, this assertion would fail. The next assertion is also about making sure that the text in the document matches the regex: “Please wait while we complete your login”

```
test('renders Authenticating message', () => {
  useAuth.mockReturnValue({
    handleOAuthCallback: jest.fn().mockResolvedValue(true),
    loading: false,
  });

  render(
    <MemoryRouter initialEntries={['/callback']}>
      <Routes>
        <Route path="/callback" element={<CallbackPage />} />
      </Routes>
    </MemoryRouter>
  );

  expect(screen.getByText('Authenticating...')).toBeInTheDocument();
  expect(screen.getByText(/Please wait while we complete your login/i)).toBeInTheDocument();
});
```

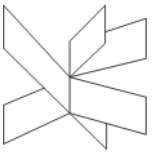
Figure 67

Finally the last test shown is an integration test in the “**App.flows.test.js”** This test case is about when the user has a lat/lon and the weatherAPI is working, the city and the local weather are shown. The test case is async, since the UI needs to load in order to see this data. The test starts by rendering the App wrapped in the mocked

AuthProvider, which supplies a user with the preferences: {lat:47.5, lon: 19.0, city: ‘Budapest’} and the mocked useWeather returns: {temperature:22, humidity: 65, lightLux: 11000} so the GreenhouseList function inside the App will render the header with weather. After the render, the DOM will be updated, then the test waits until the callback’s expectations pass or “waitFor” times out and then asserts that an element with an accessible label “Local weather” is in the document. This implies that the main content and header have been rendered. After “waitFor” completes, the test asserts that the “Weather data: Budapest” is visible and then the test asserts each of the mocked values that was defined in earlier lines.

```
test('header shows weather when user has lat/lon and useWeather returns data', async () => {
  render(
    <AuthProvider>
      <App />
    </AuthProvider>
  );

  await waitFor(() => {
    expect(screen.getByLabelText('Local weather')).toBeInTheDocument();
  });
  expect(screen.getByText(/Weather data: Budapest/)).toBeInTheDocument();
  expect(screen.getByText(/Temp: 22°C/)).toBeInTheDocument();
  expect(screen.getByText(/Humidity: 65%/)).toBeInTheDocument();
  expect(screen.getByText(/Light: 11000 lux/)).toBeInTheDocument();
});
```

Figure 68



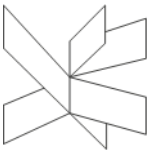
7. Results and Discussion

The finished project managed to fulfill all of the planned requirements that were derived in the analysis period. The project's main requirements, alongside most of the additional features planned, were completed in due time.

The main focus and scope of the project was to create a hosted visualisation solution which uses various top-of-the-line technologies, which gathers data from the IOT sensors, passes them securely, processes them and stores them in a secure database. The complete project also included the creation of various visualisations for the user to better understand the data that he has and to be able to act according to it, and also for him to receive alerts based on it. The project also had a webpage, with a multi user authentication, where the user could login and be able to add new greenhouses, with title, description, order and visualizations. All values can be edited, and the order of the greenhouses can be swapped with each other to ensure priority, it also sorts it by the 1st element. Additionally Grafana panels made for visualization now can be added on the details page and they also can be resized and renamed.

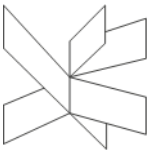
Furthermore, the user can now compare their greenhouses' isolation, by seeing the local weather data on the said webpage by the usage of a weatherAPI, for the future, additional sensors are suggested to be placed directly outside of the greenhouses to ensure maximum accuracy.

Another important point is the team's focus on current popular trends and frameworks used in similar systems. This information was well researched by the team in order for them to create a competent project that satisfies the user and is also in line with other similar existing systems.



Even though the system manages to meet and satisfy the primary requirements of the project, there are a few limitations which should be observed. One of them would be the fact that the evaluation is based on a limited deployment scope, and it depends on the accuracy of the sensor data. Also, how the system might perform under a larger amount of data wasn't taken into consideration. All of these elements might influence how well the system performs in other circumstances and should be kept in mind in future iterations.

All in all, the system does what it was supposed to do based on the requirements that were set up. It kept true to the requests of the user while making sure that it leaves room for it to be further developed in the future. Based on it, the subsequent conclusion has been developed.



8. Conclusion

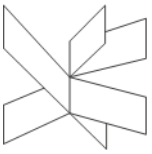
The conclusion section of this report will gather all of the most important points of the previous sections and create a picture of the whole system from start to finish.

The introduction presents the origin of the project, what its main aim is, and what are some key information about the user, as well as some background information about the situation.

Afterwards, the analysis section focuses on everything that happened during the analysis period. It is here that the discussions that were made with the user were transformed into requirements for the project, and they were ranked in order of their priority. Then, the requirements which go together were made into use cases for the use case diagram for the idea of the system, as well as its actors, to be easier to understand. From there, the team could move on to creating the design of the system.

In the design section, the team focused on defining why each technology was chosen and how they were used to interact with each other and form a scheme of the architecture of the system. Shortly after, other diagrams were created, which were related to how data is handled in different parts of the system, how the user will interact with it, as well as how the frontend's class structure will look like. This was a crucial point, as without a clear and logical design, the implementation would have been much more difficult to do.

The next section is the implementation section, in which all of the steps that were followed to deliver the system were explained, as well as the reason



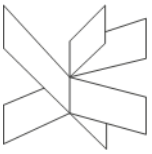
for choosing certain methods in developing it. It contains code snippets for explaining the logic of the system, as well as some of the queries and transformations used in the visualisation side of implementation. Moreover, at the end of the section, the technologies used were listed to provide a clear view of the stack used throughout the project. In the end, all requirements were completed except one, with the project reaching its original goal that was set.

Afterwards the testing section followed, in which was showcased how the system was tested for errors, both regarding the integration pipeline and regarding the webpage.

To conclude, the team sees the finished project as a success, as all the new technologies and frameworks were implemented as intended, creating an advanced system which can also be further developed in the future.

9. Project future

For the future of this project, there are numerous aspects where improvements could be added. During our implementation phase, the team found out that Filebeat cannot connect to Opensearch without Logstash, which greatly solved the issue; however, in a much later iteration, the team found out that there is another way to avoid having extra unnecessary servers if the team switches from Filebeat + Logstash + Opensearch to Telegraf + Opensearch. It seems like Telegraf can be configured to directly connect to the Opensearch Database, without the need for any extra services. Naturally, the user would not experience any differences between the two ways of transporting data, but from

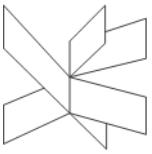


the backend's perspective, it would make a huge difference, since there would be fewer places to encounter errors.

Furthermore, if the collaboration with the user could have been expanded for a longer period of time, a new set of sensors could have been installed outside of the greenhouse to measure how well the crops are isolated from the outside world. In addition to having sensors on the outside, another sensor could be added on the inside. A soil humidity sensor is missing from the equation to properly monitor the plants.

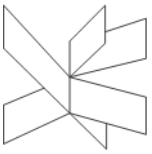
Lastly, the current user interface is using Grafana embedded to display visualizations, which works well, up until the point when more complex queries have to be made upon the user's request, which, in Lucene, only can be done by using too much compilation, that is constantly throwing an error from Opensearch. Therefore, learning from this outcome, for the future, the webpage will visualize the data directly queried from the OpenSearch Dashboard, making it more efficient and designable on the frontend.

In conclusion, some parts of the system can be developed further. The IoT devices are lacking in sensor types and numbers, while the backend is a bit overcomplicated at the moment, and the frontend is too difficult to use for a non-experienced user, which can be solved by making a front page with only the dashboards on it.



10. References

H., S. (2025, December 10). *ghanode*. Github. https://github.com/Szabka/ehcomp/blob/master/examples/ghanode.yml



11. Appendices

Diagrams can be found under:

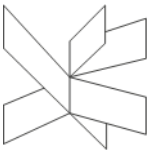
BPR - GHA/Appendices/project_appendices/diagrams/

The project description can be found at:

BPR - GHA/Appendices/project_appendices/project_description/

The use case descriptions can be found at:

BPR - GHA/Appendices/project_appendices/use_case_descriptions/



GHA – Process Report

Alin-Gabriel Abdallh (331979)

Dániel Hock (331276)

Name of supervisor(s):

Jens Cramer Alkjærsig (JCA)

[Number of characters: 25812]

Software Technology Engineering

7th semester

17.12.2025

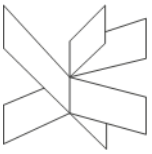
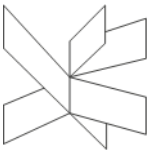


Table of content



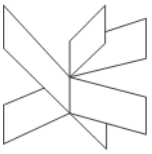
1. Introduction

The Greenhouse Analysis project has been made to give us the opportunity to work with a real user. Having received constant feedback was very helpful, and we could learn a lot from the meetings we had with him. However, it was also stricter, due to the fact that it had to fit our customer's needs.

The challenge of the project was to implement a system which would help the user better visualise their greenhouses. This required us to use technologies we were not familiar with and to learn how to implement and use them efficiently to deliver a successful product. We wanted to use the most popular tools that are used in the industry, so that the project would be modern and of professional standard.

The project was developed by two 7th semester students, which meant that because of their low number, they had to be focused and on point at all times. To make sure that is the case, we decided to use SCRUM combined with Kanban and Agile Unified Process. This helped us organise everything efficiently and stay on top of tasks while working on the project throughout the semester alongside other courses' assignments.

Lastly, this report will talk about the process of developing the project, from Inception all the way to submission. It will also talk about how we worked as a group and the background of the members, the beginning of the project, its execution, including the SCRUM logbooks of our daily meetings, followed by our reflections on the entire process of developing the final project, as well as our reflections on working with the supervisors. Finally, the results of the entire process will be summed up in a conclusion.



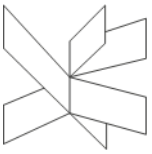
2. Group Work

The task of completing the project was fulfilled by two students in their final semester, both of them being from two different countries, which meant that their cultures and approach to working on a project would be slightly different. Despite this difference, we didn't really have any conflicts or any issues while working in a team. Dániel comes from Hungary, while Alin is from Romania, which, despite the small amount of differences in our cultures, we found many similarities between each other in work style and work ethic.

Both of the team members are high context and direct, therefore it was very easy to understand each other. For example, deciding on who is doing a specific part in the system, for us, it was like a walk in the park. We both had parts, which we were interested in, so dividing the workload was very straightforward. Regarding conflicts, we didn't experience any throughout the entire process, which meant that we have experience in avoiding unnecessary conflicts, and we have also worked together in previous semesters. Therefore, neither of us had any problem sticking with the group contract and not deviating from it.

Regarding our personal profiles, Alin is a mixture between red and blue with very little green and yellow, which means he prefers being more direct when communicating with other people, while Dani is a mixture of green, blue and yellow with a very little red, which might look like our communication norms are different, but thanks to the fact that the both of us can turn to each other in a professional manner, directness has been a great advantage to use rather than it being a painful needle.

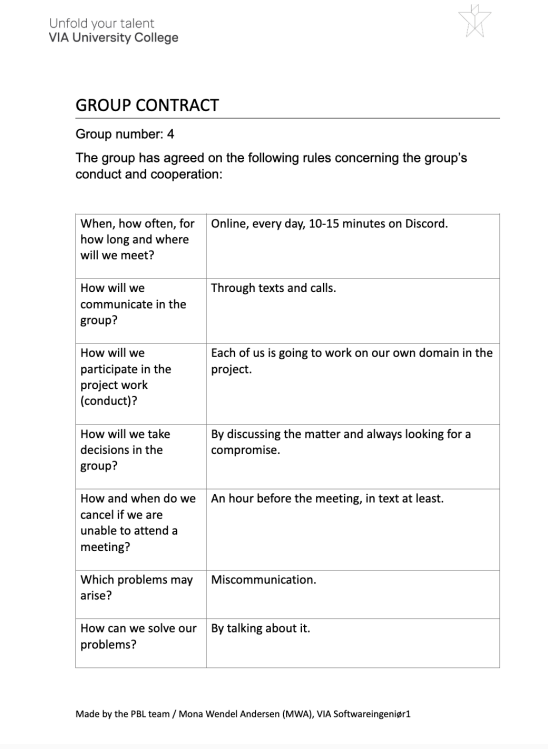
In conclusion, despite having different dominant colours and cultures, we have worked together perfectly.



3. Project Initiation

The bachelor's project started from the sixth semester with the BPR1 course, in which we first had to choose what our teams were going to be. We decided then to work in the same group as we had teamed up before in the fourth semester SEP4 group project, even though we were in different subteams of that project. We decided to keep the team as two people because the project consisted of two components, and having one person for each would be more than enough.

The next step after group formation was to set up a group contract in order to define the boundaries of how we wanted to work as a team. We discussed how often we should meet, for how long, how we will communicate and how we will make decisions as a group. Also, we talked about how we should behave with each other in order to avoid offending one another and how we should act if any issues arise while working on the project. Since we had worked together before, we already knew how each other likes to work, so we made sure to take that into account in our group contract. We decided to use SCRUM in order to keep track of meetings, as it is the most widely used framework in the industry.



Unfold your talent
VIA University College

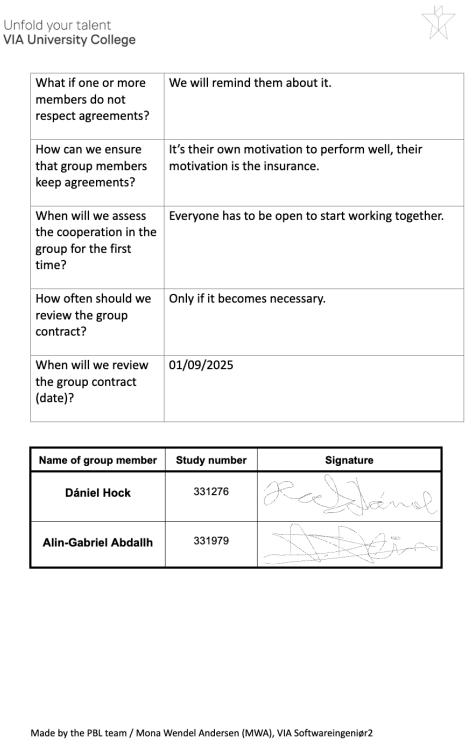
GROUP CONTRACT

Group number: 4

The group has agreed on the following rules concerning the group's conduct and cooperation:

When, how often, for how long and where will we meet?	Online, every day, 10-15 minutes on Discord.
How will we communicate in the group?	Through texts and calls.
How will we participate in the project work (conduct)?	Each of us is going to work on our own domain in the project.
How will we take decisions in the group?	By discussing the matter and always looking for a compromise.
How and when do we cancel if we are unable to attend a meeting?	An hour before the meeting, in text at least.
Which problems may arise?	Miscommunication.
How can we solve our problems?	By talking about it.

Made by the PBL team / Mona Wendel Andersen (MWA), VIA Softwareingeniør1



Unfold your talent
VIA University College

What if one or more members do not respect agreements?	We will remind them about it.
How can we ensure that group members keep agreements?	It's their own motivation to perform well, their motivation is the insurance.
When will we assess the cooperation in the group for the first time?	Everyone has to be open to start working together.
How often should we review the group contract?	Only if it becomes necessary.
When will we review the group contract (date)?	01/09/2025

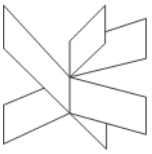
Name of group member	Study number	Signature
Dániel Hock	331276	
Alin-Gabriel Abdallh	331979	

Made by the PBL team / Mona Wendel Andersen (MWA), VIA Softwareingeniør2

Figures 1 & 2

Afterwards, we met a few times in order to discuss ideas for the project. We first thought about trying to find a company that could give us a bachelor's project idea to complete, but afterwards, we found our current client, who could give us the project idea as well as feedback on our project. We then inquired if the idea would be good, and after we got an affirmative answer, we started thinking about the next part of the process, which was the project description.

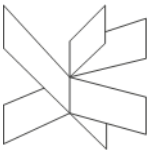
The project description had two iterations: a first draft and a final version. The first draft focused on defining our project idea and what problem it would solve, as well as being our first project proposal to the supervisor. To do that, we had a couple of meetings with the client in which he laid out what the project would be about. The problem revolved around a farmer who wanted to



modernise but could not do so due to him not being proficient with the newest tools in the tech industry. We started by writing the background description, in which we described the user, his background, the problem that he had, what other solutions are currently available, how we would solve it and also defined what the main goal of the project would be. We also discussed how relevant this problem and subject are in day-to-day life as well as the ethics involved. Our project deals with making crop growth more efficient and eliminating waste, so when taking into account the UN's Sustainable Development Goals, our project manages to hit several of them. From them are the second goal, which is to end hunger and promote sustainable agriculture, the third goal, which focuses on ensuring healthy lives and well-being for everyone, and the eighth goal which is to promote sustainable economic growth as well as full and productive employment.

After the background description, the next step was to create a problem statement as well as following subquestions. We singled out the farmer's inability to use modern sensors and tools as the main problem, and how we can solve it by creating visualisations for him so that he can operate faster and better. Then, we defined some of the project's delimitations, such as the decision of what kind of technology will be used for the data transportation between the sensors and the first service is not in our scope, as that was imposed by the type of sensors that the client had.

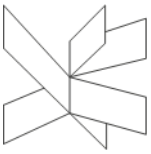
We decided that for methods we should use a mixture of SCRUM, which we have had many experiences with in previous projects, and Kanban, with AUP. Since we were two people, Dani was the Product Owner, while Alin was the Scrum Master. We defined sprint duration as being 1 month, since having shorter sprints felt inefficient. Then, the technologies used in the project were laid out according to the decision of the team.



Once that was done, we proceeded to make our expected time schedule for the project, where we agreed that the final day of our bachelor's project would be the 18th of December, which is before the deadline. Setting this up gave us a bigger motivation to finish the project successfully.

The next and last part of the project description was to take into account the possible risks that come with the project. Some of the risks include equipment, such as sensors breaking and not showing the correct data, as well as the new technologies not being known to us, which had the risk of slowing down the development process and making us not have a finished product before the deadline.

After the first draft of the project description was submitted, we received feedback that it was a great start. However, we still had to extend the introduction as well as make our user into a persona inside the project description. After those changes were made, the final version was submitted and approved before the end of BPR1, which meant we could start working on developing the project.



4. Project Execution

Regarding the execution of the project, we decided to keep a similar structure to the ones from previous semesters. We used Agile Unified Process alongside ScrumBan, which is a mixture between SCRUM and Kanban. The optimal sprint duration was set to one month, since having shorter sprints felt unsuitable, because of our having to work on other courses' assignments, which sometimes meant that we couldn't work on this project for extended periods of time.

In order to keep track of tasks and everything SCRUM related, we used Jira Software, as we had experience with it during previous semester projects, as well as working during internships. This tool is widely used by most companies in the industry, and it felt like a must to use it for this project, as it makes the job of the Scrum Master much easier, as he can distribute tasks as well as monitor ongoing ones.

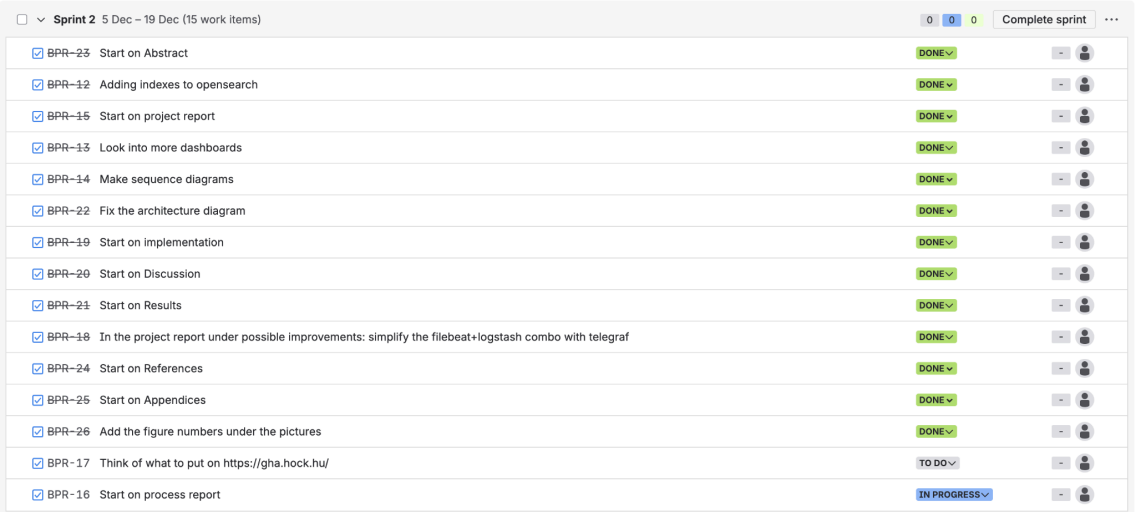
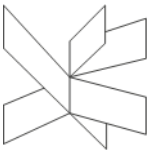


Figure 3



As it can be seen from an example of the 2nd sprint backlog above, the tasks can be easily seen by the Scrum Master so that he can monitor them and add new tasks when necessary. This way, he could also see the Kanban of each task, whether it is in “to do”, “in progress” or “done”.

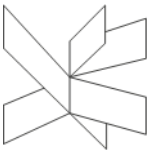
In order to keep track of what each member does, and what was done and will be done, daily scrum meetings were held each time we met, from the elaboration phase all the way to the end of the transition phase. This way, we made sure that we always stayed on track, and we each knew what we needed to do and when. We kept a logbook with all of the daily meetings, so that we could compare what was done between them as well as monitor what was done during an entire sprint.

Daily scrum meeting:

Date: 09.12.2025
Time: 18:20 -18:35
Duration: 15 Min
Who’s here?
- Alin and Daniel

	What was done?	What is next?	Any problems?
Alin	- Added the percentage dashboards - Fixed issue with Opensearch installation	- Start on documentation	None
Dani	- Made Sequence diagrams - Finished the hosting of the app	- Start on documentation	None

Figure 4



After each sprint was completed, we had a short sprint retrospective in which we discussed what we did in the previous sprint and reviewed whether the overall sprint goal was reached or not. Additionally, we would talk about what we needed to start doing, stop doing, as well as what we needed to keep doing.

Sprint 1 Retrospective

Observations: The sprint goal was hit as all of the technologies were implemented on localhost, and the first bunch of dummy data could be sent the whole way through the whole system.

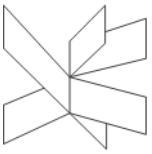
Start doing	Stop doing	Keep doing
Make dashboards Documentation Deploy the application	Implementing technologies	Working hard

Figure 5

4.1 Phase 1 - Inception

The start of the inception phase was when the teams were set up in the 6th semester, and we knew that we were working together. The main goal of the inception phase was to complete our project description and get it approved, as it was the first project-related assignment that we had.

During the project description, we wrote a lot about the idea of the project and who the main actors are, since this was also the first time we could pitch our project idea to the supervisor.



Following the approval of the project from the supervisor, we then moved on to deriving the requirements that we needed to complete in order to be able to deliver a satisfactory product for the user. To do this, we had to interact with the user and find out what the user really needs and how. We also decided what technologies we were going to use in order to develop the project.

Besides the project description, we took part in multiple lessons during BPR1 in which we learned important information about requirements and how to write them, about stakeholders and who they are in our specific project, in which the main stakeholder is the client. We also learned about the One Rule and the importance of keeping an overview at all times. Additionally, we learned more about the philosophy of engineering and science as well as engineering ethics. These helped us tremendously when making sure our project conforms to all of these standards that we learned about during this course. We also wrote an interpersonal report, where we reflected on everything we have learnt throughout the semesters about the process, such as critical thinking, motivation, team dynamics, ethics, and personal profiles etc.

4.2 Phase 2 - Elaboration

After the Inception phase was over and we had our project description approved, we moved on to Elaboration. Here, we first decided our SCRUM roles, with Dani being the Product Owner and Alin being the Scrum Master. In Elaboration, we wanted to set up the technologies that were going to be used by the system. We started in the beginning of the 7th semester, before we got any assignments from other courses, so that we could get ahead of time from the beginning. The first step, before implementing anything, was to create an

architecture diagram to better understand how the data passes throughout the system, as well as which technologies need each other to work together and which of them we should start implementing first, alongside their protocols. It was here that we finally realised what challenges await us when making the mentioned diagram, which can be seen below.

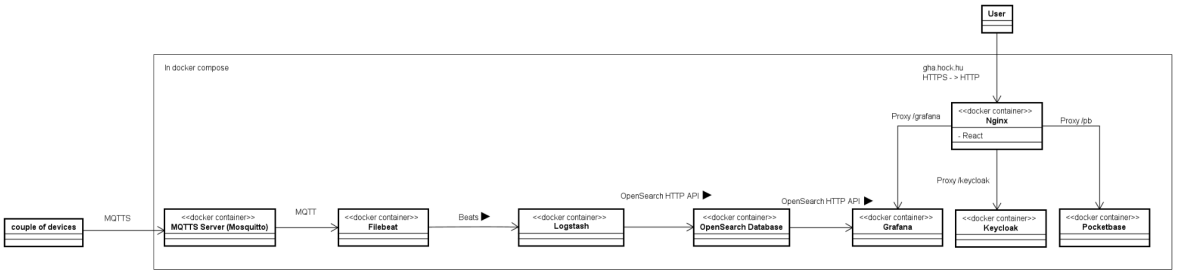
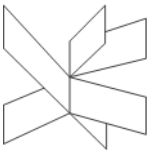


Figure 6. Can be found in the Appendices/project_appendices/diagrams/architecture_diagram/Architecture Diagram.png

Our stack had multiple technologies, all of which were the first time either of us used them. Even though this was a challenge and a risk at first, we decided to do it since they are tools used widely in data visualisation, and we wanted to make sure our project used the most popular technologies in the industry.

As mentioned, our goal for the Elaboration phase was to get all the technologies working on localhost. We wanted to do this since this would act as a proof of concept for our system that we can then build upon in the next phases. Once everything worked smoothly and we could see our dummy data passing throughout the system, we took a large break before continuing with the project. This was expected, and this is the reason why we started elaboration and creating our proof of concept in September, since we knew that with the assignments coming from other courses as the semester advanced, we would not be able to have as much time as we had in the beginning of the semester. We also got encouraged to do this by our supervisor as well, since he told us



that taking care of other courses' requirements was as important as working on the bachelor's project.

When the assignments slowed down, we returned to the project and picked up from where we left off.

4.3 Phase 3 - Construction

The Construction phase represented the part of the process in which we worked on the rest of the implementation of the project, as well as on creating diagrams. After we were done with course assignments at other courses, we had again time to work on the bachelor's project. We met daily to work on the project, with the sprint being 1 month long.

The amount of work expected to be done increased as this phase had the goal of finishing everything related to the system and its diagrams, and leaving only the documentation to be done. We deployed our system and created visualisations for the user.

While making visualisations we worked with Grafana which was something new for both of us. At the beginning, this showed, as we didn't know how to make any smart visualisations or how the Lucene or PPL query languages work, but, as we learned more and more, we managed to solve these issues and use queries and transformations to better visualize our data.

Figure 7

Besides the implementation and visualisation, we also made diagrams during this phase to better illustrate how the system works and what its logic is.

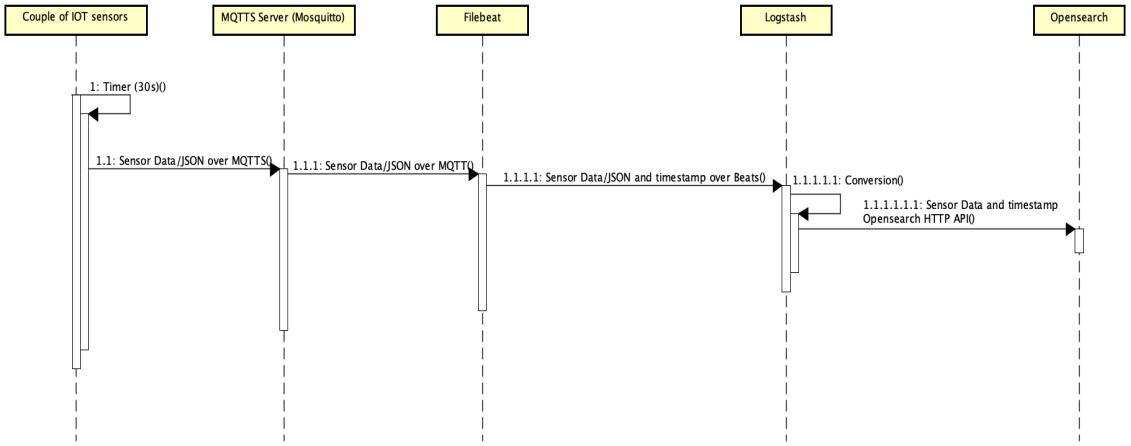


Figure 8. Can be found in the Appendices/project_appendices/diagrams/sequence_diagram/Data Travelling SD.png

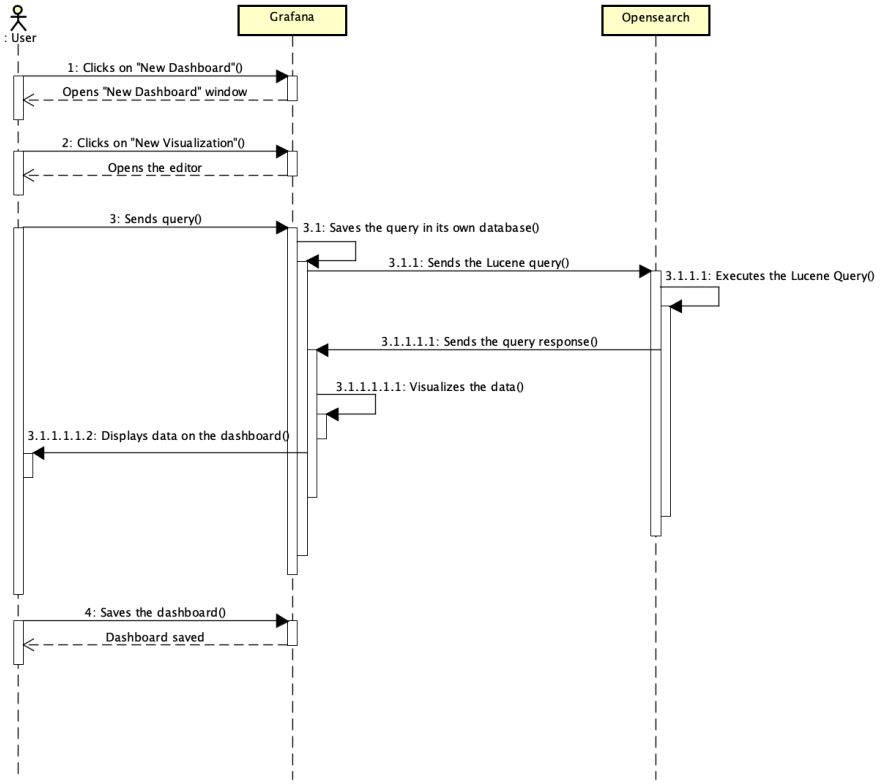
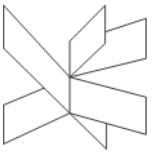


Figure 9. Can be found in the Appendices/project_appendices/diagrams/sequence_diagram/New Dashboard SD.png

The diagrams above are sequence diagrams made to show how the data passes through our system in different scenarios.

After the tuition period was finished on the 28th of November, the project period started. We immediately decided to meet on a daily basis, from 10 am to 2 pm, and then again in the evening from 5:30 pm to 9 pm. On some days, we worked even more as we wanted to finish the project ahead of the deadline without having any issues or worries regarding timing.

While working on the project, we also encountered a few errors that took time in order to solve. One of such errors was that on one of the dashboards which was supposed to show the percentage each measurement stayed between a set range during the last 24 hours, the dashboard displayed the



correct value and then, during the next refresh, it displayed “No Data” and stayed like this for a few minutes before displaying correctly again and then repeating the same error. We spent a lot of time trying to figure out the issue with this, and on that particular day, we stayed much after the normal time to find the solution. In the end, the issue was the fact that we received new sensor data every half a minute while the dashboard refreshed every 10 seconds, which meant that when it would search for new values, nothing would be there, and then it would appear as “No Data”. This represents one example of an issue which arose, but none of them managed to overcome us and our will to make everything work correctly and create a successful product. With everything running correctly and visualisations and diagrams being created successfully, we only had the documentation of the project left to do, which was worked on in the last phase of AUP.

4.1 Phase 4 - Transition

The last phase of our project was the Transition phase. It represented the period between the completion of our system to the submission of the project as a whole. It included checking if all existing diagrams were correct and if any needed any changes, verifying the dashboards if they work accordingly and making the documentation of the project.

While looking back at the work done throughout the project in the previous phases, we figured that a good way to visualise that work was to look at GitHub’s generated diagrams.

Figure 10. Can be found in the Appendices/process_appendices/Commits_over_time.png

As it can be seen above, the diagram showcases a part of the commits done by us from the beginning of the semester until the end. We can observe the gap left from October to the end of November, which represented the break that we took because of other courses. This diagram doesn't represent the entire picture, however, since much work was also done inside Grafana and Opensearch themselves, which is not included by GitHub. This also explains why on GitHub it looks like a big difference in lines committed, while in reality, we split the work, one of us worked on the code which is on GitHub, while the other worked with Opensearch and Grafana.

The burndown chart shows that rather than putting all the tasks from the beginning, we only added them briefly and added details later. Such as "Start on the project report" was one of the originally added tasks in the sprint, but later on, we felt like adding new tasks to specify where we are currently in documentation would help us keep track, which made the chart look as it does at the end of the sprint.

We are aware that this is not the right way of using SCRUM, and we will make sure to agree on a way of noting down tasks for the future to avoid sudden increases.

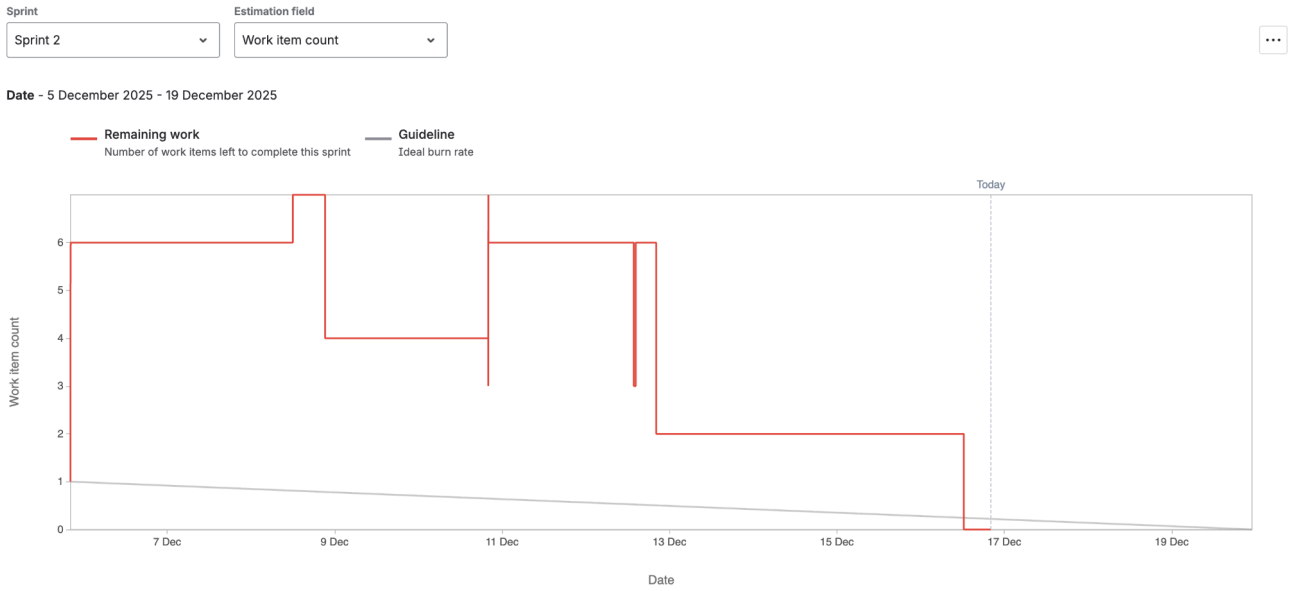
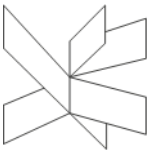


Figure 11. Can be found in the Appendices/process_appendices/Burndown_chart.png

In conclusion we think we used our time well and even though we didn’t apply all the methodology 100% accurately, we still learnt a lot from our mistakes and successfully delivered the product.



5. Personal Reflections

Daniel

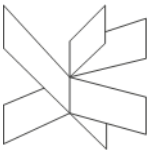
Looking back at what we have done throughout this project, I can say I have learnt a lot of practical knowledge in all of these new technologies, which we had the chance to use, as well as academic knowledge regarding philosophy of engineering, stakeholders and many more.

However, even with the success of delivering the product, I can't ignore the overwhelming feeling of possibilities which I could have done if we knew the tech better. I barely scratched the surface and for the future I would like to broaden my knowledge regarding these technologies, by applying them in new projects.

On the other hand I failed to use SCRUM completely as it was intended.

We had sprints, with defined tasks before starting the sprint, but we kept adding to it as the sprint was already going on, making it look weird. For the future I will pay attention to the rules more and I will make sure all the tasks will be added before the start of the sprint.

In conclusion I am satisfied with this project, I learnt a lot, I had a nice teammate to work with, and we also managed to finish the project on time. I won't ever forget the experience I had learnt by working on this project and I will bring this knowledge with me for the future.



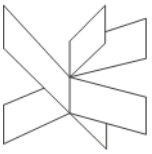
Alin

This bachelor's project has helped me learn a lot and gain valuable experience from a technical standpoint, as well as academically. As mentioned many times before, the system had a stack with technologies which were all new to me, having never used them before. My main tasks centred around implementing some of the technologies and then making dashboards and other visualisations for the user. Therefore, I had to take responsibility for going through my own learning process of understanding how each technology worked so that I could complete my tasks. I would consider working with unfamiliar technology, and managing to learn them and use them at the same time, a very important learning outcome of the bachelor's project.

Having a good collaboration with Dani was essential in order to make sure that the project was completed correctly. While on some parts I could work independently, on others I had to coordinate myself with him so that we could solve the requirement or specific part together. I also learned the importance of sharing the documentation that I found online, so that we would both know how everything worked inside our system.

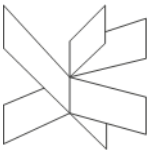
Looking back at the process itself, we didn't encounter any large issues, but one issue that we had was our wrong use of SCRUM, especially in Jira. While we had defined general system requirements, we didn't create the subtasks at the beginning of each sprint, and instead created them as we went, resulting in a strange burndown chart. In future works, I will definitely pay more attention to SCRUM, making sure everything is done by the book.

For the future, I would say that from now on, when I encounter new technologies, I will focus on being more structured in my learning. Additionally, based on our SCRUM issues, I plan to make sure that I apply SCRUM and



Agile more carefully in the future, especially in planning sprints and creating subtasks.

Overall, this project has strengthened my ability as a software developer, gaining knowledge that I didn’t have before it. At the beginning, I thought it would be very risky and difficult to work with entirely unfamiliar technology, but, with great collaboration with Dani and a structured learning process, we managed to create a successful project that achieved our goals.

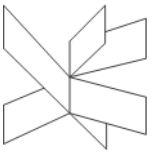


6. Reflect on Supervision

Jens Cramer Alkjærsig (JCA) was our Bachelor supervisor, who always made sure that we are doing a great job, when we were confused about documentation and diagrams, he gave us guidance of how to continue forward, he also ensured we are not falling behind with other course assignments, and he emphasized how important these assignments are to see us depart in the end. We met him once in the 6th semester, and regularly every 3 weeks from the 7th semester. He was very flexible, he let us organize meetings at the last minute and we could also meet him online.

On the other hand, most of these meetings were organized by the supervisor, because it was a requirement to check up on us regularly, so for the future we will make sure we collaborate with our supervisor more.

We are very thankful for his supervision and we will never forget his guidance.



7. Conclusion

Going into the project, we were following a plan since day 1 of BPR1, which we have followed through these two semesters, and regardless of how many challenges we have had to face, we stood our ground, solving any possible issues, and in the end, we accomplished our set goals.

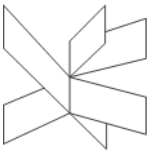
While working on the project, we tried to follow a plan to ensure everything went well, and, as we gained more experience, we thought of some other things which we could do and not do in order to achieve the desired outcome. In our opinion, a good list of what to do while working on a project in a team would be the following:

- Communicate with your team members.
- Be direct, but respectful
- Cooperation is important
- Try to aim for a compromise.
- Be independent, but don’t hesitate to ask for help.

However, there are things that someone should not do in any circumstances:

- Don’t be a couch potato
- Don’t be a hitchhiker
- Don’t hide crucial information from the team.
- Don’t argue with emotions.
- Don’t ignore deadlines or assume others will cover for you.

In conclusion, this bachelor’s project was a perfect final preparation before the end of university. We used all the knowledge gained over all of the semesters to complete the project, and, in doing so, became perfectly prepared to work professionally after the end of our studies.



8. References

9. Appendices

Burndown chart can be found at:
BPR - GHA/Appendices/process_appendices/Burndown_chart.png

Commits over time can be found at:
BPR - GHA/Appendices/process_appendices/Commits_over_time.png

Diagrams can be found under:

BPR - GHA/Appendices/project_appendices/diagrams/

The project description can be found at:

BPR - GHA/Appendices/project_appendices/project_description/

The scrum logbook can be found at:

BPR - GHA/Appendices/process_appendices/SCRUM_Logbook.pdf