2023

SAXION
UNIVERSITY OF
APPLIED SCIENCES

Profitflow.

Digital Vertical Farming

BUILDING A RASPBERRY PI-BASED VERTICAL FARM WITH MULTI-FARM ADMINISTRATION DASHBOARD

YUNUS ELMAS (430361)
HBO-ICT/Software Engineering
Graduation Report

DIGITAL VERTICAL FARMING - BUILDING A RASPBERRY PI-BASED VERTICAL FARM WITH MULTI-FARM ADMINISTRATION DASHBOARD

SAXION UNIVERSITY OF APPLIED SCIENCES

DEVENTER, NETHERLANDS

Thesis:
Digital Vertical Farming – Building a Raspberry Pi-based Vertical Farm with Multi-Farm Administration Dashboard

by **Yunus Elmas** (430361)

Graduation teacher: **Peter Ebben**
Saxion University of Applied Sciences

Graduation company: **Joey Teunissen**
ProfitFlow B.V.
Deventer, Netherlands

Abstract

This graduation report describes the implementation of a Proof-of-Concept vertical farm system and an accompanying custom admin dashboard client with multi-farm management capabilities. The system allows for the monitoring and control of connected vertical farms through a single client, providing a centralized management solution for potentially large-scale vertical farming operations.

Vertical farming is an innovative approach to plant cultivation that involves stacking layers of plants in a controlled indoor environment, optimizing land usage. This technique employs Controlled-Environment Agriculture (CEA) technology to control environmental factors such as air quality, irrigation, lighting, and soil conditions. The ability to monitor and control multiple vertical farms allows manual and/or automated fine-tuning of the system's behavior for specific crops by adjusting actuators and physical outputs based on measured environmental factors. The implementation of such a system can significantly boost productivity, efficiency, and sustainability in the vertical farming industry.

Table of Contents

1. Introduction

In recent years, there has been a growing interest in smart agricultural solutions to address challenges in food production and resource management. As part of this trend, vertical farms have been emerging as a potential solution for sustainable urban agriculture. One key aspect of vertical farming systems is the integration of technology for monitoring and control, which enables precise management of the environment and resources.

Developing a dashboard for managing vertical farms can help optimize their performance by providing real-time information and control over various parameters, such as temperature, humidity, and light intensity. This project aims to design and implement a custom vertical farm management system, including a user-friendly dashboard that interacts with a Raspberry Pi-driven vertical farm.

1.1. Project Background and Context

Escari is a startup company that specializes in the production, sales, and distribution of microgreens. They currently cultivate microgreens using a traditional vertical farming setup without any digital integration. Microgreens are young, edible plants that are harvested just a few weeks after germination, when the first set of true leaves has emerged. They are smaller than baby greens and are often used as a garnish or as an ingredient in salads, sandwiches, and other dishes at culinary restaurants.

However, Escari recognizes the potential benefits of integrating an IoT-based solution into their vertical farming setup. The company wants to build a prototype that will allow them to integrate a Raspberry Pi into their vertical farming setup and an admin dashboard to monitor, control, and automate many Raspberry Pi's acting as vertical farms.

1.2. Problem definition

Escari's vertical farming setup is currently being operated through manual labor to cultivate crops (such as microgreens) and selling these crops to culinary restaurants in Zwolle to generate a decent side-income. However, if Escari wants to expand its operations and deploy multiple vertical farming setups, the resources required for this endeavor will increase in proportion to the number of vertical farms and at some point, become impossible to manage individually. Existing vertical farm management solutions may not fully address the unique requirements and constraints of a custom vertical farm system. Furthermore, integrating various technologies and tools can be challenging, particularly when considering compatibility, scalability, and overall practicality.

To scale their vertical farming operations, Escari requires a comprehensive system that can monitor, control, and automate their vertical farm setups. The project's challenge is to research, design and implement a custom vertical farm management system that effectively integrates the chosen technology stack, meets the specific needs of the target users, and can be seamlessly integrated into Escari's existing vertical farming infrastructure.

The vertical farm currently operated by Escari is used for cultivating crops without any hardware or software integrations. The entire process relies on hydroponics knowledge, time, and manual labor. The setup includes basic components such as 12V fans, a heating element and white LED lights, while water supply to the microgreens is handled manually based on the specific needs of the crops.

Figure 1: current setup of the vertical farm used by Escari

In addition, Escari monitors essential environmental factors like air temperature, humidity, and soil moisture using consumer-grade electronic devices. The system is housed within a cube-like insulated structure, which makes it easier to regulate the internal environment. This cube contains three racks, each with multiple layers, and is equipped with necessary lighting and ventilation systems.

1.3. Objectives

The primary objective is to design and implement a system that will provide Escari with these capabilities (monitor, control, automate) through the accompanying admin dashboard.

Figure 2: industrial-grade vertical farm

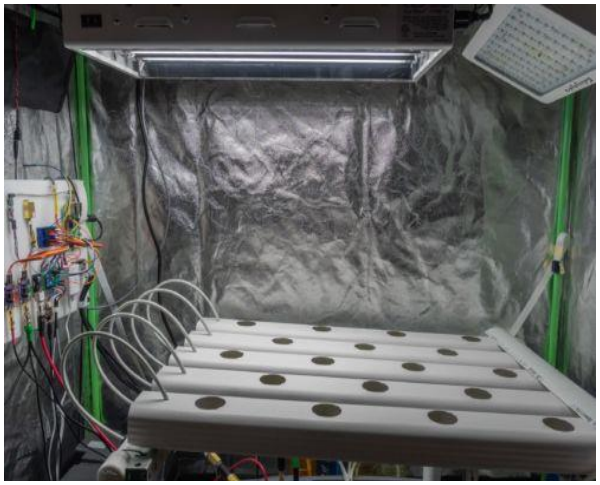

Figure 3: single-layer hobby grade vertical farm

The dashboard will also provide insight on key metrics such as temperatures, humidity, water levels, and other relevant data. The dashboard should provide capabilities to operate/control the vertical farms as well as some degree of automating the system. These functionalities will enable Escari to gain insight into their vertical farming operations and allow for real-time adjustments to optimize growth conditions on a larger scale and with less user resources.

The above figures are some examples of such systems to give an idea of the physical aspects of the system. Accompanying this setup would be an administration dashboard that ideally allows to monitor, control and automate each deployed vertical farm individually in the system.

1.4. Scope and Limitations

The scope of this project encompasses the design and implementation of a custom vertical farm management system, with a particular focus on an administration dashboard that interfaces with one or many Raspberry Pi-driven vertical farms. Additionally, the project involves setting up a physical prototype to test and demonstrate the system on.

Additionally, the system's development is influenced by several factors, which must be taken into consideration to ensure that the system meets the needs of the customer:

- Firstly, the customer has requested the use of a Raspberry Pi instead of a microcontroller. While microcontrollers are typically more suited for this type of application, the customer's preference for the Raspberry Pi must be accommodated in the design of the system.
- Secondly, the customer has requested that the hosting of the system be kept local for the time being. This means that the system must be designed to at least operate within a local network, rather than being accessible over the internet. Considerations might have to be made to accommodate a remote deployment in the future.
- Third, the customer has expressed a desire not to use wireless sensing/actuator technologies in the system, despite the advantages of using such technologies for data transmission as well as interoperability. While this requirement may limit the implementation of the system, it is a requirement that must be accommodated in the system's design.
- Lastly, to make the development of the system feasible – besides the sensor/actuator connects, anything more towards the "physical" implementation of the vertical farm is to be considered **out-of-scope**. For example, the positioning of the layered crops, water supplication & distribution, light placement etc. will not be considered within the system. The customer has agreed to this, if there is some flexibility provided in the implementation to accommodate for changing sensor and actuator setups.

2. Methodology

In this chapter, we will describe the methodology that was chosen for this project and how it was applied. The project management methodology chosen was a SCRUM-based approach, which is an Agile methodology used for software development projects. This methodology was chosen because of its flexibility and adaptability.

2.1. Scrum

Sprint reviews will be held with the customer after every sprint timeframe. During this sprint review, development of the last sprint will be discussed and feedback from the customer will be processed into further development.

Scrum Rule	How?
Sprint length	Fixed 2-week sprint length
Definition of Done	See below

Table 1: scrum rules for this project

For a user story to be considered "done" according to the abovementioned *Definition of Done*, the following criteria must be met on a user story:

1. The user story has been integrated with the existing codebase, and there are no conflicts or issues.
2. The user story has been documented in the appropriate documentation.
3. The user story has been approved by all stakeholders.

3. Research framework

In this chapter, we will delve into the research questions outlined in *Table 2* and discuss the approaches taken to address them. For each question, the relevance and context will be explained, highlighting how it is connected to the overall development of the project.

The research questions delve into various aspects of the project, including theoretical requirements such as potential system architectures, domain-specific knowledge of vertical farming, and an analysis into the theory behind complex system requirements.

Main question	
How can a system be realized that allows for remote monitoring, control and automations for raspberry pi-based vertical farms?	
Sub questions	
1.	**What architectural patterns can be used to implement a system that allows for remote monitoring & control as well as automations on vertical farm(s)**
Exploring possible architectures enables the identification of the most appropriate design for the system, considering aspects such as scalability, maintainability, and performance.	
2.	**Which processes from typical vertical farming systems can be translated into an IoT-based solution and what is required to accommodate these features?**
A deeper understanding of the vertical farming domain helps in tailoring the system to address the specific challenges and requirements unique to this sector. This question will look to answer the question on how the chosen hardware helps our system's functioning as a vertical farm.	
3.	**What are the possible ways to integrate user-defined automations with varying levels of complexity into the vertical farm management system?**
This sub-question explores the integration of user-defined automations. An automation boils down to triggering some action in the system, such as turning an actuator on/off and integrated into the system through an "automations" service. Automations can be divided into three separate levels. ➢ **Level 1** Users can manage automations on a fixed schedule, such as turning on Actuator X at 08:00 AM and turning it off at 09:00 AM. ➢ **Level 2** Users can manage automations based on sensor data, which is considered a "dumb" feedback loop. An example would be, "if Sensor X exceeds a certain threshold, activate Actuator Y until Sensor X returns to an acceptable range." ➢ **Level 3** Users can manage automations that consider multiple sensor values, actuator states, and user-defined thresholds. This level represents a "smart" feedback loop, ideally providing self-correcting behaviors within the vertical farm. This level	

expands on Level 2 by providing a more extensive and more autonomous automation system.

Table 2: research questions

3.1. Methodologies

For each research question, a specific methodology or set of methodologies will be employed. These methodologies may include, but are not limited to:

- Literature review: Gathering and analyzing information from relevant publications, articles, and documentation to understand the current state of knowledge in the domain.
- Comparative analysis: Examining and comparing different technologies, tools, and design approaches to determine their suitability for the project.
- Prototyping and experimentation: Building and testing prototypes to evaluate the feasibility and effectiveness of various design approaches and technologies.

3.1.1. 1 - Architectural Patterns for Remote Monitoring, Control, and Automations

3.1.1.1. Approach

To identify the most suitable architectural patterns, a analysis will be done into existing, similar systems. Additionally, an examination of the architecture patterns and their applicability to vertical farming will be carried out. This analysis of different architectures will help identify key features and patterns that may be implemented in the implementation of this project.

3.1.1.2. Monolithic Architecture

In a monolithic architecture, the entire system is built as a single, tightly coupled unit. For a vertical farm, this means that all the components, such as sensor data processing, actuator control, and managing automation rules, are part of the same codebase and run within the same process. While this approach can simplify development and deployment, it may limit scalability and make it difficult to adapt to changing requirements. Additionally, a failure in one component may affect the entire system.

Pros:

- Simplicity: Developing, testing, and deploying the entire system as a single application reduces complexity and simplifies integration between the server-side components and dashboarding.

- Consistency: A monolithic architecture provides a consistent environment for development, which can streamline the process of building and updating the system.

Cons:

- Scalability: Scaling a monolithic architecture can be challenging, as any change to the system requires the entire application to be redeployed. This can be particularly problematic for vertical farming systems that need to accommodate varying workloads and adapt to changing requirements.
- Maintainability: As the system grows in complexity, maintaining and updating a monolithic architecture can become increasingly difficult. The tight coupling of components makes it harder to isolate and fix issues, and updates to individual components may necessitate a complete redeployment of the system.

3.1.1.3. Microservices Architecture

A microservices architecture decomposes the system into multiple small, loosely coupled, and independently deployable services. In the context of a vertical farm, each service could be responsible for a specific aspect of the system, such as sensor management, actuator control, or automation rule processing. This approach allows for flexibility, scalability, and easier maintenance, as each service can be developed, deployed, and scaled independently. However, the increased complexity of managing multiple services, ensuring proper communication between them, and maintaining consistency across the system could be challenging.

Pros:

- Scalability: Microservices can be easily scaled by deploying more instances of a specific service or using container orchestration platforms like Kubernetes, without affecting the rest of the system.
- Flexibility: Individual services can be developed, tested, and deployed independently, allowing for rapid development, updates, and the use of different technology stacks for each service as needed.
- Technology-agnostic: Microservices enable the use of diverse technologies, programming languages, or frameworks for each service, as they communicate through well-defined interfaces.

Cons:

- Complexity: Microservices architecture requires complex coordination and communication between services, typically using APIs and messaging systems. This can increase development and maintenance efforts, as well as introduce new potential points of failure.
- Consistency: Ensuring data consistency across multiple services can be challenging, particularly when implementing distributed transactions or managing shared resources.

- Security: Inter-service communication can introduce potential security vulnerabilities, requiring careful implementation of authentication and authorization mechanisms, such as using OAuth 2.0 or JSON Web Tokens (JWT).

3.1.1.4. Serverless Architecture

In a serverless architecture, the backend logic is broken down into individual functions that are executed in response to specific events or triggers. For a vertical farm, this could mean having separate functions for processing sensor data, controlling actuators, or sending alerts. These functions are managed by a cloud provider when deployed, allowing for automatic scaling and reduced operational overhead. However, latency might be an issue due to the stateless nature of serverless functions and reliance on external services.

Pros:

- Cost-Effective: Serverless architecture only charges for the compute time used, reducing costs for idle resources.
- Scalability: The cloud provider automatically scales the resources based on demand, allowing the system to handle variable workloads.
- Simplified Operations: No need to manage servers, as the cloud provider takes care of infrastructure management, enabling developers to focus on writing code.
- Fine-Grained Implementation: Serverless functions allow for precise implementation of features without having to worry about infrastructure, leading to more modular and maintainable code.

Cons:

- Vendor Lock-In: Serverless architecture relies on cloud provider services, making it difficult to switch providers or move to an on-premises environment. Depending on the cloud provider service, additional configurations might have to be done to make the cloud to Raspberry Pi function properly in a deployed environment.
- Cold Start: Initial function invocations may experience increased latency due to the time required to provision resources.
- Limited Customization: The serverless environment may impose restrictions on runtime environments, available resources, and execution time, potentially limiting the customization options of the vertical farming system.

3.1.1.5. Event-Driven Architecture

An event-driven architecture is centered around the concept of events being produced, detected, and consumed by various components within the system. In the context of a vertical farm, this could involve sensor data updates, actuator commands, or automation rule triggers being treated as events that are propagated throughout the system. Components can subscribe to specific events and react accordingly, enabling a highly decoupled and flexible architecture. However, managing and debugging event-driven systems can be

complex, and ensuring proper event handling and consistency across a complex system may introduce its own challenges.

Pros:

- Scalability: Components can be scaled independently, allowing the system to handle varying workloads efficiently. This can be achieved using techniques such as partitioning and replication in the event processing implementation.
- Resilience: Decoupling event producers and consumers enables the system to handle component failures without affecting the entire system, improving fault tolerance and error isolation.

Cons:

- Complexity: Implementing an event-driven architecture requires complex coordination and handling of asynchronous communication. It may involve implementing concepts such as message brokers, managing backpressure, event ordering, and delivery guarantees (at-least-once, at-most-once, or exactly-once).
- Debugging: Asynchronous communication can make debugging and tracing issues more challenging, as it requires understanding the flow of events and their dependencies.

3.1.1.6. Comparing Architectural Patterns for the Vertical Farming System

1. *Monolithic Architecture*

Monolithic architectures offer simplicity and consistency, making them relatively easy to develop, test, and deploy. However, they may face issues in scalability and maintainability, particularly as the system grows in complexity. For a vertical farming system that may require continuous updates and improvements, monolithic architectures may not be the most suitable choice.

2. *Microservices Architecture*

Microservices architectures provide flexibility and scalability, with individual services that can be developed, tested, and deployed independently. However, they introduce complexity in terms of coordination and communication between services and may require additional security measures. This architecture may be suitable for a vertical farming system that demands a high degree of adaptability and the ability to scale services independently.

3. *Serverless Architecture*

Serverless architectures offer cost-effectiveness, scalability, and fine-grained implementation of features without worrying about infrastructure management. However, they may introduce latency issues, vendor lock-in, and cold start problems. For a vertical farming system that requires low-latency communication between components and may

need to operate on-premises, serverless architecture might not be the most appropriate choice.

4. *Event-Driven Architecture*

Event-driven architectures provide scalability and resilience through decoupled components that communicate asynchronously. However, they may introduce complexity in coordination and handling asynchronous communication, as well as challenges in debugging. For a vertical farming system that needs to handle many events and respond to various triggers, this architecture could be suitable, provided that the associated challenges are managed effectively.

Each architectural pattern offers its own set of advantages and disadvantages. When selecting an architecture for the vertical farming system, it is essential to consider the specific requirements and constraints of the project. Based on the analysis, a microservices or event-driven architecture may be the most suitable for the vertical farming system, given their inherent scalability and flexibility.

3.1.2. 2 - Translating the Domain into IoT-based Sub-systems

3.1.2.1. Approach

The approach for this research question entails studying the vertical farming domain to comprehend its key processes and requirements to facilitate these processes.
The research question will try to identify as many processes as possible within a vertical farming system and based on these processes, the sensors and actuators will be deduced with corresponding requirements that arise for the implementation of the system to facilitate support for the identified processes. Literature reviews and case studies will be examined to gain insights into the operational aspects of vertical farming. This analysis will aid in identifying the sub-systems of these processes and provide an overview of the functional domain.

3.1.2.2. Functional Processes within a Vertical Farm (VF)

Processes within a vertical farm can be considered feature-focused, (hardware-related) sub-systems of the vertical farm which are required to facilitate operations for these processes. These processes of the vertical farm will be identified and analyzed in this section.

According to an article about the perfect environment for vertical farming (Edinburgh Sensors, 2018), Temperature, light, humidity, water supply, nutrient content, and atmosphere are all critical factors in vertical farming that must be carefully monitored and controlled to ensure optimal crop yields and minimal resource consumption.

3.1.2.2.1. Lighting

Lighting is a crucial factor in vertical farming, as it directly influences plant growth and development. To optimize lighting, it is essential to choose the right type of lighting system that can be dynamically controlled for factors such as *spectrum*, *intensity*, and *duration*, allowing for customization according to specific plant requirements. This sub-system can be broken down into several sub-processes:

Sub-process	Context
Light intensity monitoring	The available light intensity needs to be monitored to ensure that the plants receive the required amount of light for their growth stage and species.
Light spectrum control	Different plant species and growth stages have specific light spectrum requirements. In vertical farms, it is crucial to adjust the light spectrum to meet these needs and promote optimal growth.
Light intensity control	Adjusting light intensity is essential in a vertical farm to ensure that plants receive the necessary amount of light without wasting energy or causing photoinhibition.

Table 3: lighting sub-system

3.1.2.2.2. Climate Control

Climate control is vital for maintaining optimal growing conditions in a vertical farm, where temperature and humidity fluctuations can negatively impact plant growth and yield. Ensuring stable and efficient climate control requires a combination of sensors and actuators that monitor and regulate temperature and humidity. The system should be adaptable and capable of maintaining a suitable environment across various plant species and growth stages. This process can be broken down into several sub-processes:

Sub-process	Context
Temperature monitoring	Monitoring the air temperature is essential for maintaining optimal growing conditions in a vertical farm, as temperature fluctuations can negatively impact plant growth, development, and yield.
Temperature control	Regulating temperature is crucial in a vertical farm to ensure that plants are not exposed to extreme temperatures that can negatively affect their growth and development.
Humidity monitoring	Humidity plays a vital role in vertical farming, as it affects plant transpiration rates, nutrient uptake, and the risk of diseases. Monitoring humidity levels is necessary to ensure a suitable environment for plant growth.
Humidity control	Controlling humidity in a vertical farm is crucial for maintaining a balance between sufficient moisture for plant growth and minimizing the risk of diseases related to excessive humidity.
Gas concentration monitoring	Monitoring gas concentrations, such as CO2, is crucial in a vertical farm, as these gases directly impact plant growth and development.

Gas concentration control	Regulating gas concentrations in a vertical farm is essential for maintaining optimal growing conditions and preventing the buildup of harmful gases such as CO2.

Table 4: climate control sub-system

3.1.2.2.3. Water and Nutrient Management

Water and nutrient management in vertical farming systems directly impact plant health and resource utilization efficiency. Traditional irrigation methods, such as flood or drip irrigation, may be unsuitable for multi-layered vertical farms, requiring alternative approaches. Efficient water and nutrient management systems must precisely monitor and control the delivery of water, nutrients, and dissolved oxygen. This involves selecting appropriate sensors and actuators to maintain optimal levels while minimizing waste and reducing environmental impact. This process can be broken down into several sub-processes:

Sub-process	Context
Water and nutrient monitoring	Monitoring the quality and availability of water and nutrients in a vertical farm is essential for maintaining optimal growing conditions and minimizing waste.
Water and nutrient delivery	Delivering the right amount of water and nutrients to plants in a vertical farm is critical for ensuring optimal growth and minimizing waste.
Nutrient solution management	Maintaining an optimal nutrient solution is essential for vertical farming systems, as it ensures that plants receive the necessary nutrients for growth and minimizes the risk of nutrient imbalances or deficiencies.
Irrigation control	Efficient irrigation scheduling is critical in vertical farming to ensure that plants receive the required amount of water and nutrients without causing water stress or wasting resources.

Table 5: water and nutrients sub-system

3.1.2.2.4. Key Environmental Sensing Metrics

In an IoT-integrated vertical farming system, a variety of sensors can be employed to collect data on essential system parameters. The following is a technical overview of the sensors that can be implemented with a description of the problem they solve within the vertical farm. Based on the biology of plants/crops, environmental metrics may affects plants differently. When designing a vertical farm, it's important to know which metrics are relevant and why they are relevant to increase the efficiency of the system (Carbonnel, Stormonth-Darling, Liu, Kuziak, & Jones, 2022).

1. Temperature sensors

1.1. The *water* temperature affects the solubility of nutrients and the rate of plant metabolism, and it can have a significant impact on plant growth and

development. In hydroponic systems, the temperature of the water supply can affect the uptake of nutrients by the plants, as well as the rate of photosynthesis and respiration. If the water temperature is too high or too low, it can cause stress to the plants and inhibit their growth.

1.2. The *air* temperature affects the plant's metabolism, growth, and development. In vertical farms, it's important to control air temperature to ensure consistent conditions for crop growth. Different crops have specific temperature requirements for optimal growth, so maintaining a stable temperature is crucial for maximizing yield and quality.

2. *Humidity sensors*

These devices monitor the relative humidity within the growing environment, crucial for mitigating mold growth and promoting healthy plant development. High humidity levels can cause issues such as mold and mildew, which can damage or kill plants. On the other hand, low humidity levels can cause plants to wilt and dry out, which can also be detrimental to their growth and development. Therefore, it's important to maintain the appropriate humidity levels to ensure optimal plant growth.

3. *Light*

These sensors assess light intensity and quality, including metrics such as photosynthetically active radiation (PAR), which is the range of light wavelengths utilized by plants for photosynthesis. Light sensors enable the optimization of artificial lighting systems in vertical farming setups, ensuring that plants receive the appropriate light spectrum and intensity for efficient photosynthesis and healthy growth.

4. *CO2*

These instruments assess carbon dioxide concentrations in the environment, essential for photosynthesis and overall plant growth. Carbon dioxide (CO2) is an essential component of photosynthesis, the process by which plants convert light energy into chemical energy to produce food. In a closed environment like a vertical farm, the levels of CO2 can drop due to the plants consuming it and can be replenished by external sources.

5. *pH*

These devices measure the pH level of nutrient solutions or growing mediums, ensuring optimal nutrient bioavailability and plant uptake. The pH of a solution will determine the solubility of nutrients, and the form they take. Plants can only absorb nutrients that are dissolved and in certain forms. The solution may contain all the necessary nutrients, however a pH that is too high or low can prevent any uptake. This can be controlled with acid or base injectors through peristaltic pump(s).

6. *Electrical Conductivity (EC)*

These instruments quantify the concentration of dissolved salts in nutrient solutions, assisting in maintaining optimal nutrient levels for plant growth and development. An

electroconductivity (EC) sensor plays an important role in monitoring the nutrient levels in a vertical farm. EC is a measure of the ability of a solution to conduct electricity and is closely related to the concentration of dissolved salts, specifically ions in the solution. In hydroponics, EC is used as a proxy for the concentration of dissolved nutrients in the water supply.

7. *Moisture*

These devices measure the water content in the growing medium, assisting in regulating irrigation and preventing overwatering or underwatering of the plants. Proper moisture levels in the growing medium are essential for healthy root growth and overall plant health. By monitoring moisture levels, vertical farming systems can optimize water use and maintain ideal growing conditions for the plants.

8. *Oxygen*

These sensors measure dissolved oxygen levels in the nutrient solution, which is critical for healthy root development and nutrient uptake. Adequate oxygen levels are necessary for root respiration and maintaining the health of beneficial microorganisms in the growing medium. Low dissolved oxygen levels can lead to root diseases and reduced nutrient uptake by the plants.

9. *Airflow and air pressure*

These sensors monitor air circulation and pressure within the growing environment, ensuring proper ventilation and gas exchange for the plants. Adequate airflow is essential for maintaining optimal temperature and humidity levels, as well as preventing the buildup of harmful gases, such as ethylene, which can negatively affect plant growth.

10. *Pest and disease detection*

Camera-based systems and imaging sensors can help identify signs of pest infestations or plant diseases, enabling early intervention and targeted treatments. Early detection of pests and diseases can minimize crop loss and reduce the need for excessive pesticide use, leading to healthier plants and a more sustainable farming system.

11. *Weight sensing*

These sensors monitor the weight of individual plants or plant parts (such as fruits), providing insights into crop yield and growth patterns. By tracking plant weight, vertical farming systems can optimize cultivation practices, estimate crop yield, and evaluate the effectiveness of different growing conditions and treatments.

3.1.2.3. Integration of Processes, Sensors and Actuators

This sub-chapter will focus on the integration of the identified processes in the context between the processes identified in the chapter prior and the sensors/actuators within the vertical farming system. It addresses aspects related to the requirements to facilitate these processes for the system, using multiple sensors and actuators.

3.1.2.3.1. Integration of Lighting Processes

The integration of lighting processes relies on input from light sensors, and in turn control the output of the lighting system. Typically, these systems use pulse width modulation (PWM) to control the intensity and spectrum of light emitted by LEDs. The controller adjusts the duty cycle of the PWM signal to regulate the intensity of each color channel, enabling the creation of specific light spectra tailored to the plants' requirements.

Figure 5: Light intensity sub-system

Figure 4: light spectrum sub-system

3.1.2.3.2. Integration of Climate Control Processes

The integration of climate control processes necessitate relies on input from temperature, humidity, and gas concentration sensors. These inputs are processed, and control algorithms, such as proportional-integral-derivative (PID) controllers, are employed to adjust the actuators. For instance, the PID controller calculates the error between the desired setpoint and the current temperature, and it modulates the output to heaters or coolers accordingly. The same approach is applied for humidity control and gas concentration control using humidifiers, dehumidifiers, and ventilation systems.

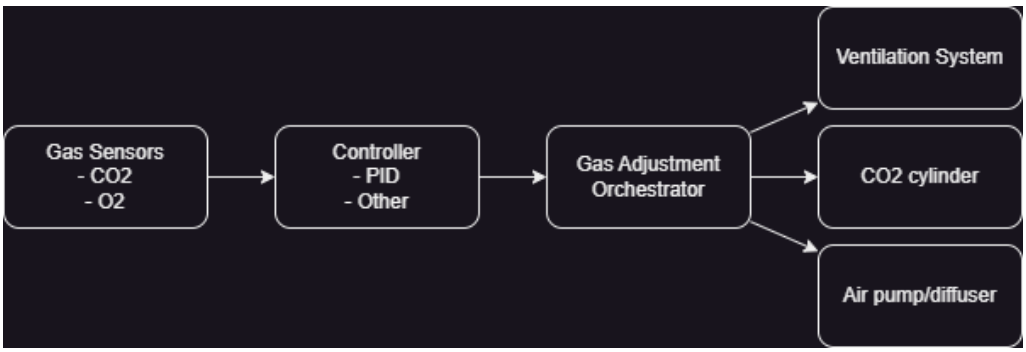

Figure 6: gas sub-system

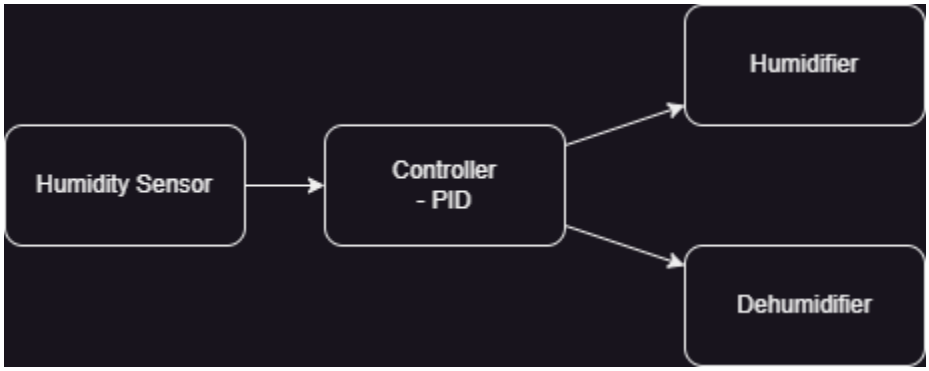

Figure 7: Humidity sub-system

3.1.2.3.3. Integration of Water and Nutrient Management Processes

The control system in a vertical farm collects data from pH, EC, and moisture. Based on this data, the control system adjusts peristaltic pumps, solenoid valves, and water pumps to maintain the desired nutrient solution composition and irrigation schedule. PID controllers or other control algorithms can be utilized to maintain the desired pH and EC levels by activating acid or base injectors and nutrient dosing pumps. In addition, moisture sensors can be used to regulate irrigation cycles, ensuring proper water and nutrient delivery without overwatering, or underwatering the plants.

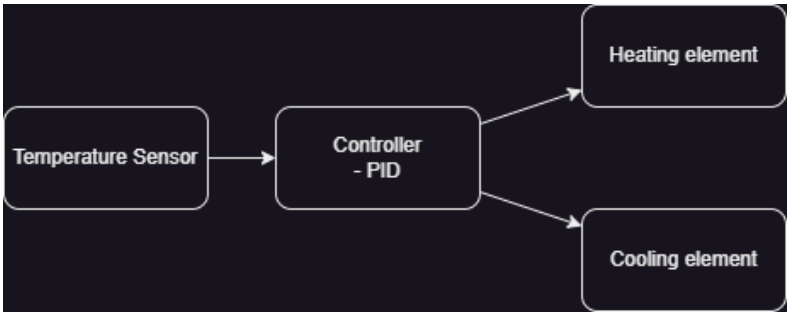

Figure 9: temperature sub-system

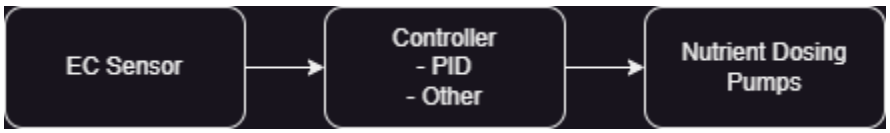

Figure 8: EC sub-system

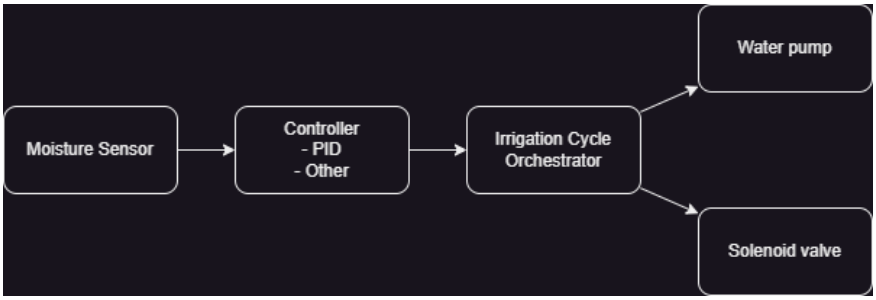

Figure 10: moisture sub-system

3.1.2.4. Control strategies

Expanding upon the previously discussed integrations, control algorithms can be used to implement the strategy with which the IoT system can be controlled. Refer to Appendix C Control strategies' for a full analysis on the most common control strategies.

3.1.3. 3 - Integrating User-Defined Automations with Varying Levels of Complexity into the Vertical Farm Management System

3.1.3.1. Approach

Without automation, IoT solutions offer little more than visualization dashboards and offline data analysis. Automation plays a critical role in ensuring the success of vertical farming systems. Similarly, the data itself that is coming from the connected products has little inherent value. What matters and is ultimately the goal of all IoT solutions is that specific actions are taken to solve very specific, real-world challenges, that are unique for each IoT use case.

However, implementing automation in vertical farming systems presents unique challenges due to the complexity and variability of the system. To address these challenges and provide an overview on how these challenges can be solved, this sub-question will focus on the design of a flexible, adaptable, and user-friendly automations system that facilitates the execution of these business rules.

3.1.3.2. Automation Rules Engine

In the case of an automation rules engine for IoT, business rules represent the core logic that governs the behavior of the system. These rules can include trigger events and corresponding actions that control the behavior of system components based on sensor data and other inputs. The automation rules engine serves as the tool that enables the creation and management of these rules.

Just as different IoT use cases require different rules engine capabilities, different automation use cases may require different levels of automation, with varying degrees of complexity and flexibility. The design and implementation of an automation rules engine must consider the unique requirements of the specific use case and incorporate the necessary capabilities to meet those requirements. In the context of vertical farming systems, an automation rules engine can be used to automate tasks such as adjusting lighting, irrigation, and temperature control based on sensor data and other system inputs. Rules can be created and modified to respond to changing environmental conditions, crop growth stages, and other factors that affect the performance of the system.

Some popular examples include:

1. *Node-RED*

   Node-RED is an open-source automation tool that uses a flow-based programming model to create automation rules. It allows users to drag and drop nodes onto a canvas to define the flow of data and actions. Node-RED is most used as a supplementary tool within a larger system.

2. *Home Assistant*

   Home Assistant is a popular home automation platform that can be used to control various devices and services in a vertical farming system. It includes a rules engine that allows users to define automation rules based on trigger events and actions.

3. *Redis*

   Redis is an in-memory data structure store that can be implemented as an automation engine by providing fast and efficient data storage, retrieval, and communication between different components of the system. Since Redis is only a tool, the rest of the automation engine will have to be implemented and integrated on top of it. Redis can be utilized in an automation engine for:

- State management:
  Storing system state like sensor data, actuator states, and thresholds for quick access and real-time decision-making.

- Caching:
  Improving performance by caching results of complex computations or data queries, useful for large data sets or resource-intensive tasks.

- Message queueing/pub-sub
  Facilitating communication between automation engine components, allowing real-time reaction to system changes.

- Scheduling/timers

Managing scheduled tasks and timers for efficient handling of time-based automations without overloading the backend server.

- Data persistence:
  Providing data persistence options to maintain system state, preventing data loss, and ensuring automation engine reliability.

4. OpenHAB

OpenHAB is another open-source home automation platform that can be used to automate vertical farming systems. It includes a rule engine that supports a wide range of trigger events and actions.

5. AppDaemon

AppDaemon is a Python daemon that provides a structured environment for writing automation apps/scripts for home automation projects. It works well with Home Assistant(documentation is focused on HA) and other home automation platforms that support MQTT messaging and can also be used as a standalone service. AppDaemon allows users to write automations using a sandboxed Python runtime. The daemon will then listen for specified events and trigger the specified scripts accordingly.

6. Mycodo

Mycodo is an open-source environmental monitoring and automation system designed for various applications, including vertical farming. It offers a robust rules engine that enables users to create and manage automation rules based on sensor data and other inputs. This allows for the implementation of customized control strategies for different system components, such as lighting, irrigation, and temperature control. With its user-friendly interface and extensive customization options, Mycodo provides a versatile solution for managing automations in vertical farming systems.

During the analysis of the chosen vertical farm software (Mycodo), it was observed that Mycodo offers many functionalities through its own dashboard UI which may be used in the implementation of a vertical farm, such as PID controls, triggers and actions, but these functionalities were not (yet) accessible through the Mycodo API. This limitation posed a challenge in the development of an automation engine since the ideal solution is for the automation engine to be hosted on the Raspberry Pi with server-side support to manage these automation rules without depending on the server. The server could then be used to manage individual automation engine instances per vertical farm/Raspberry Pi.

However, development on the Raspberry Pi, and more specifically Mycodo, has been avoided because of time and scope constraints and also since the development of the dashboard and its surrounding server components are the focus of the project. This led to designing the user-defined automations within the context of the server, where

the automations engine should be able to at least consume the Mycodo API to autonomously control all the vertical farms. Once a base for this system is functional, it should be possible to modify the Mycodo API to facilitate more features through the automation engine.

3.1.3.3. Integration of the automation rules engine and the server

In conclusion, various approaches can be employed to integrate an automation rules engine for vertical farming systems. Considering the findings of research sub-question 1 - Architectural Patterns for Remote Monitoring, Control, and Automations' and tools analyzed in this report, the following architectures may be considered:

1. Hasura and Node-RED based approach

In this approach, the system utilizes Raspberry Pi with Mycodo and Node-RED for complex automation logic. A server-side custom automation service manages automations through Hasura scheduler API, invoking domain services for various vertical farming processes. Domain services can trigger Node-RED flows, leveraging the full Mycodo ecosystem for advanced features.

Side of system	Main components	Description
Raspberry Pi	- Mycodo - Node-RED	Mycodo and a Node-RED instance will be deployed per Raspberry Pi. The Node-RED instance will be responsible for any complex automation logic that isn't available through the Mycodo API directly.
Server	- Automation service(hasura) - Domain services	The server will contain a general automation service which facilitates management of automations through the Hasura scheduler API. This service will then, based on the user-defined rules, invoke the domain services. A domain service is responsible for one or multiple vertical-farming processes as described in chapter 3.1.2.2: 'Functional Processes within a Vertical Farm (VF)' The domain services may then invoke the Node-RED API to start a flow from within the Raspberry Pi. This is helpful, since the Mycodo API lacks support for some (advanced) features and this approach lets us define fine-grained flows that may completely utilize the Mycodo ecosystem

Table 6: Using Hasura and Node-RED

2. Redis based approach

In this approach, the system is divided into two main components:

- On the Raspberry Pi side, Mycodo is used to control the vertical farm independently, while the server interacts with the Mycodo API to build additional logic based on available features.
- On the server side, it consists of an automation service using Redis as a job queue and domain services. Redis serves as a job scheduler, enabling event-driven services for vertical farm operations. The execution of complex business rules can be achieved by chaining events generated from the scheduler service.

Side of system	Main components	Description
Raspberry Pi	- Mycodo	Mycodo is used to drive the vertical farm independently from the server. The server may invoke the Mycodo API and build additional logic on top.
Server	- Automation service (Redis as a job queue) - Domain services	In this scenario, Redis may be implemented as a job scheduler and may be employed to facilitate scheduler-like features (for example, controlling actuators on a vertical farm when specific events occur). If the backend supports event-driven services, this approach opens the possibility to functionally narrow down each service and invoke each service based on events generated from the scheduler service. These events could then be chained according to the business rules required.

Table 7: redis approach

3. Mycodo-focused approach

In this approach, the system has two main components:

- On the Raspberry Pi side, Mycodo and Node-RED are used. Mycodo provides many features through its UI, and by modifying the existing Mycodo API, more complex functionalities can be supported. Node-RED can facilitate communication between components if needed. This approach has the advantage of keeping the automation engine as part of the vertical farm itself, managed by an API per vertical farm. However,

the downside is that the existing logic in Mycodo's automation engine needs to be analyzed and implemented per use case without breaking any existing functionalities.
- On the server side, an automation management service is used to consume the newly added Mycodo API endpoints. This approach simplifies implementation since all complex logic is contained within the Mycodo instance.

Side of system	Main components	Description
Raspberry Pi	- Mycodo - Node-RED	Since Mycodo offers many features through its own UI, it's possible to modify the existing Mycodo API to support more complex functionalities. These functionalities can then be invoked by a client to manage automations. Where required, Node-RED can be used to facilitate the communication between multiple components. The benefit of this approach would be that the automation engine will be part of the vertical farm itself, with a management API per vertical farm. The downside of this approach would be that the existing logic in Mycodo's automation engine would need to be analyzed and implemented per use case without breaking any existing functionalities.
Server	- Automation management service	The server will need to contain only a management service which consumes the newly added Mycodo API endpoints. Since all complex logic is contained in the Mycodo instance, this approach would be easier to implement compared to the other approaches.

Table 8: Mycodo-centric approach

3.1.3.4. Automations: level 2

Mycodo, with its built-in InfluxDB instance, offers the capability to create feedback loops ranging from basic to complex logic. InfluxDB Checks API is a part of the InfluxDB monitoring and alerting framework. It uses the data stored in InfluxDB and the Flux query language to define checks. The API continuously evaluates the data against specified conditions or thresholds, which are defined using Flux scripts.

The Checks API works in conjunction with other components of the InfluxDB monitoring and alerting framework, such as notification rules and endpoints, to provide a comprehensive solution for monitoring and alerting.

- Checks: These are user-defined scripts written in Flux that query the data and evaluate it against specific conditions. The check generates a status (OK, Info, Warn, or Crit) based on the outcome of the evaluation.
- Notification Rules: These are associated with checks and determine how the alerts should be handled when a specific status is generated. Users can define different notification rules for different status levels.
- Notification Endpoints: These are the destinations for the alerts generated by the checks and notification rules. Examples of notification endpoints include email, Slack, PagerDuty, and (custom) HTTP endpoints.

An example use case in the context of IoT systems, the Checks API can be used to monitor the health and status of individual devices. Users can create checks to alert them when devices go offline, when sensor readings exceed defined limits, or when other anomalies are detected.

Another example use case is anomaly detection in time-series data, such as sudden spikes or drops in a metric. Users can set up checks to trigger alerts when such anomalies are detected, helping in identifying and addressing potential issues.

3.1.3.4.1. Analysis

The InfluxDB API can be used in the dashboard for multiple vertical farms in several ways to enhance the monitoring, analysis, and decision-making process for these systems. Some of the use cases to utilize the InfluxDB API are:

1. Alerts and Notifications: Using InfluxDB Checks API, the dashboard can be configured to generate alerts and notifications based on user-defined thresholds and conditions. This helps in identifying and addressing issues promptly, ensuring optimal performance and preventing potential problems.

2. Custom Feedback Loops: By utilizing InfluxDB's capabilities, the server can create custom feedback loops that automate actions based on specific conditions or thresholds. For instance, if a particular sensor value exceeds a pre-defined limit, the dashboard can trigger a webhook or another action to adjust an actuator, maintaining the desired environmental conditions.

Overall, the InfluxDB Checks API provides a flexible system facilitating level 2 automations easily. Feedback loops can be created and custom webhooks may be exposed for the InfluxDB instance to consume. These webhooks may contain pretty much any kind of business logic, providing a lot of flexibility to implement simple or complex checks. This also let's us conclude that, with the right backend components in place, the Checks API may also be used as part of level 3 automations.

3.1.3.5. Automations: level 3

With a basic feedback loop in place, for example by using the InfluxDB checks API, it's now possible to expand upon the automation's engine with more complex and customizable

automation rules in place, the next step is to make this automation engine a bit smarter. This can be achieved by implementing advanced control mechanisms and utilizing real-time sensor data as input for automation rules.

3.1.3.5.1. Real-time sensor data ingestion (MQTT)

Utilizing sensor data as input for automation rules can lead to more responsive and adaptable automation systems. This can be achieved by employing the MQTT (Message Queuing Telemetry Transport) protocol for communication between the server and the vertical farm system.

By integrating MQTT into the automation engine, the server can:

- Subscribe to sensor data from the vertical farm system and receive real-time updates
- Use the received sensor data as input for evaluating automation rules and making decisions
- Publish commands to the vertical farm system to adjust system parameters (e.g., lighting, irrigation, temperature) in real-time based on the evaluated automation rules
- Easily integrate with MQTT-enabled data processing engines such as Apache Kafka

3.1.3.5.2. Architecture Analysis

With a stream of different sensor data available, it is time to analyze what server components are required to facilitate a (generic) control engine and how this might be designed. While reading about more complex systems for the IoT domain, an architecture proposal was found which uses open-source tools to implement an automation rules engine (Veneri & Capasso, 2018).

This design, on a high-level, contains all the components required to implement an extensive automation component for our IoT-based system, such as the vertical farm. As requirements become more complex, similar components may be added to facilitate the infrastructure for our specific requirements. Refer to Appendix D Industrial Internet-of-Things OSS architecture' for an overview of this architecture.

A summary of the most important components can be described as follows:

- The controllers collect and send sensor data to an MQTT broker and persist the data into a data store (locally on either the controller in the physical environment, or through a gateway-like component).
- The data is also persisted into a time-series based database, except this time it's on a remote location(server).
- The asset registry module stores information about the assets and the relationships between them. These could be physical assets, devices, and systems within an industrial setting. This registry serves as the single source of truth for asset data
- The data from the remote storage is queried by the *advanced analytics* component, which may contain (long running) background-processes based on specific requirements and

strategies. The advanced analytics module queries asset data from the *asset registry* module.
- Apache Kafka is used as a both an event-dispatcher and as a rules-engine using *Kafka streams*. Kafka then processes the data received from the MQTT broker and publishes the processed data to an analysis module.
- The analysis module analyzes the data for real-time operations. The difference between this analytics module and the advanced analytics module is that this one is for fast data-processing analytics, while the advanced module is meant for long-running data-processing analytics.
- By using these components, the platform is now able to produce operations to execute, based on data processed from the controllers.

Of course, our requirements will differ from this system and the constraints will also be different. However, as mentioned before, the high-level functionalities can be implemented with similar approaches to fulfill our specific requirements.

3.1.3.5.3. Alternative Implementation

An alternative to the previously mentioned server architecture to facilitate for level 3 automations is to leverage InfluxDB's Checks API. This approach allows users to manage checks on individual data points and assign notification rules(triggers) based on the data point's status (OK, Info, Warn, or Crit).

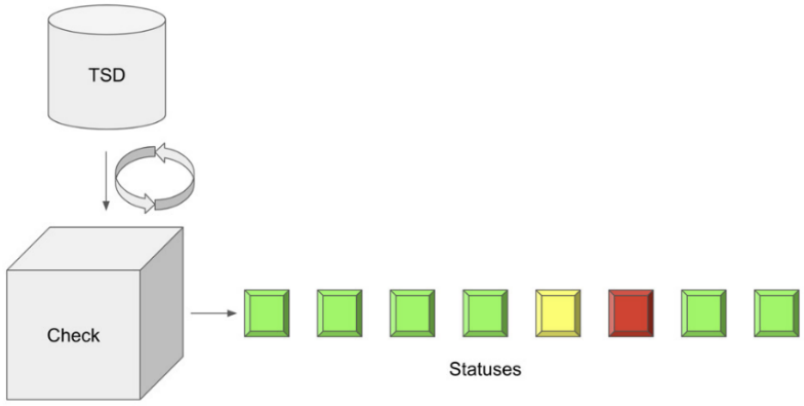

Figure 11: diagram showcasing the check mechanism from InfluxDB (Dotis-Georgiou, 2021)

These webhooks may then in turn invoke the Mycodo API directly or the server if complex business logic is required. By utilizing the InfluxDB Checks API, the server and dashboard can provide a more streamlined experience for users managing multiple vertical farms or Raspberry Pis.

The Checks API enables users to define custom checks and conditions for each data point, which helps in identifying any issues or anomalies in real-time. These checks can be based on specific thresholds or patterns, offering flexibility in monitoring different aspects of the vertical farms. Furthermore, the Checks API allows for seamless integration with other InfluxDB components, such as notifications and alerting systems. This ensures that users are promptly informed about any critical events or deviations, enabling them to take appropriate action as needed.

4. System design

As mentioned before, the primary objective of this project is to research, analyze, design and implement a vertical farm setup with IoT integrations, working in conjunction with a central management dashboard that allows for management, such as monitoring, control and automations of many such vertical farms.

The development of such systems requires a multidisciplinary approach, combining knowledge from agriculture, engineering, and computer science. This is why an extensive design was made to narrow down the scope of the system and potential approaches to developing such a system.

4.1. Use Cases

This section presents a series of use case scenarios to illustrate the functionality and interactions between the user, server and vertical farm. Refer to Appendix A Use case descriptions for more information about each use case.

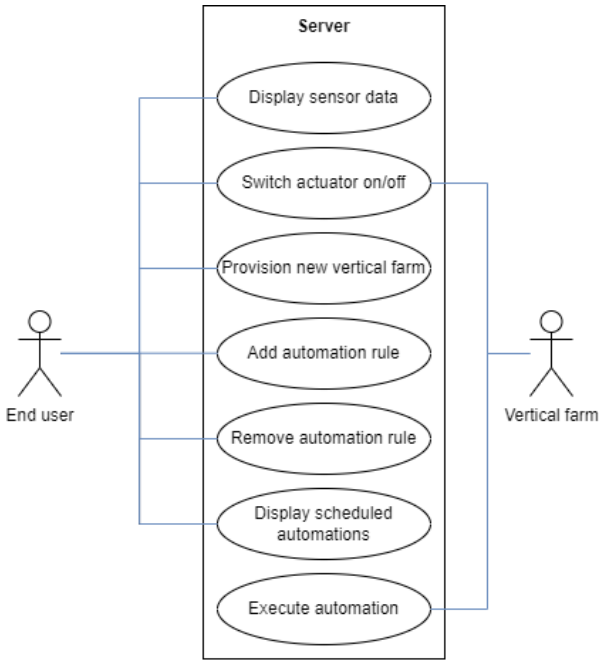

Figure 12: use case diagram, displaying the core functionalities of the system.

4.2. Requirements Analysis

Based on the system description provided in the prior chapters, the rest of the system's requirements are defined by following the Holistic Requirements Model (HRM). The HRM works by categorizing requirements into 3 categories.

4.2.1. Operational requirements

The operational requirements are a set of statements that define how a system should be operated, maintained, and supported to ensure its proper functioning and availability.

#	Requirement	Description
1	Performance	The system must perform its intended function in a reliable, efficient, and effective manner. It must meet the performance standards that a user might expect from an admin dashboard.
2	Scalability	The system needs to be scalable to accommodate evolving user demands, while maintaining optimal performance and functionality. In our context, scalability refers to the ability to support multiple vertical farms/Raspberry Pi's that are deployed and operated simultaneously.
3	Maintainability	The system must be designed to be easily maintained, updated, and repaired. In case of an error, the user should know what has happened and how to fix the issue without having to debug the codebase.
4	Usability	The system must be designed with a user-friendly interface and clear documentation to ensure ease of use
5	Reliability	The system must be reliable, meaning it should perform its functions without errors, failures or unexpected behavior

Table 9: operational requirements

4.2.2. Functional requirements

The functional requirements are a set of statements that define what the system should do, and how it should behave under various conditions. These requirements describe the features, capabilities, and behaviors that the system must have to meet the needs of its users or stakeholders. These requirements have been prioritized by applying the *MoSCoW* framework.

#	Requirement	Description
1	User authentication and authorization	The dashboard should provide a secure login system for users.
2	Provisioning a vertical farm into the system	The dashboard must allow users to add and configure Raspberry Pi devices for each vertical farm, including specifying farm location, device name, and other relevant information.
3	Displaying sensor data	The dashboard must display real-time and historical sensor data from each Raspberry Pi device, including temperature, humidity, light, and nutrient levels.
4	Controlling actuators	The dashboard must enable users to manually control actuators connected to the Raspberry Pi devices, such as lights, pumps, and fans.
5	Adding automation rules	The dashboard must allow users to create and configure automation rules for individual vertical farms based on sensor data, time schedules, or other triggers.

6	Managing automation rules	The dashboard should enable users to edit, delete, and enable/disable automation rules for each vertical farm.
7	Monitoring and notifications	The dashboard should provide real-time monitoring of the vertical farms and allow users to configure alerts for specific events, such as critical sensor readings or system failures.
8	Multi-farm management	The dashboard must support the management of multiple vertical farms, allowing users to switch between farms.
9	System backup and recovery	The dashboard should allow users to create and restore backups of their vertical farm data and configurations, ensuring data safety and business continuity in case of system failures, expansion or data loss.

Table 10: functional requirements

4.2.3. Non-functional requirements

Non-functional requirements refer to the characteristics of the system that describe how it should perform, behave, or appear, but are not directly related to the system's specific features or functions.

#	Requirement	Description
1	Performance	The system should be able to handle large data volumes without experiencing significant delays or performance degradation. For example, the system should be able to process large numbers of data points in the charts without performance issues arising
2	Reliability	The system should be highly reliable and available at all times, with a minimum uptime of 99.9%. For example, the system should have a failover mechanism in case of server failure
3	Scalability	The system should be able to scale up or down as needed to accommodate changes in user demand or data volume. For example, the system should be able to handle an increase in data traffic when connecting more vertical farms without performance degradation
4	Usability	The admin dashboard should be intuitive and user-friendly, allowing users to easily navigate and access the system's features and functions. The rest of the system should also allow for ease of use, for example for further development by other developers
5	Maintainability	The system should be easy to maintain and update, with clear documentation and well-organized code

Table 11: Non-functional requirements

4.3. User stories

Refer to Appendix E User stories' for an overview of formulated user stories.

4.4. Decision-Making Process and Criteria

This section aims to provide a comprehensive overview of the chosen system architecture for the system and design of its components. By specifying the criteria for each component required in our system, we will compare possible options for implementing these components. Following is a list of considerations for each component that need to be made before specifying details about the implementation of the component.

4.4.1. Vertical Farm Software Criteria

Below are the criteria as specified for the software driving the vertical farm:

#	Criteria	Description
1	Data storage	The choice of data storage solution should be made. This includes deciding on the type of databases to use, such as SQL or NoSQL, and where to store the data, such as on the Raspberry Pi, on-premises or in the cloud. There are two types of storage components for our system: ➢ One for storing sensor data, ideally a time-series based engine. ➢ One for storing application(s) data
2	Sensor integration	The vertical farm software should facilitate a straightforward process for integrating a diverse range of sensors into the system. ➢ Should be able to connect to a wide range of sensors. ➢ Should be able to configure the connected sensors (for example, setting the pin number) ➢ Should be able to read out the connected sensors. *: Ideally, the chosen platform should have an API of the chosen Raspberry Pi software and should enable a client to configure a (physically) connected device when required as well as other CRUD-like operations for sensors.
3	Actuator integration	The vertical farm software should facilitate a straightforward process for integrating a diverse range of actuators into the system. This includes accommodating various communication protocols, enabling seamless addition of actuators and providing the means to interact with the actuators. Ideally, CRUD-like operations should be exposed through an API that allows other applications to manage the actuators.
4	Automation capabilities	Decisions must be made regarding the choice of automation engine. This includes selecting the appropriate tools and determining how to trigger automation rules based on sensor readings or other events. Automations can be divided into three separate levels.

		➢ Level 1 Users can manage automations on a fixed schedule, such as turning on Actuator X at 08:00 AM and turning it off at 09:00 AM. ➢ Level 2 Users can manage automations based on sensor data, which is considered a "dumb" feedback loop. An example would be, "if Sensor X exceeds a certain threshold, activate Actuator Y until Sensor X returns to an acceptable range." ➢ Level 3 Users can manage automations that consider multiple sensor values, actuator states, and user-defined thresholds. This level represents a "smart" feedback loop, ideally providing self-correcting behaviors within the vertical farm. This level expands on Level 2 by providing a more extensive and more autonomous automation system. *: This criterion is shared with the server, as automations might be solved on either side of the system.
5	User-friendly configuration	Overall, the vertical farm software should be easily configurable on an individual basis. This may include a (custom) web-based interface, through an API or other tooling.
6	API availability	Should have one or more API's for interacting with the connected sensors, actuators, (runtime) data, sensor data, automations etc.

Table 12: criterions used to compare existing vertical farming platforms

4.4.2. Server Components Criteria

The server will be responsible for any API requests that need to be made by the dashboard. This could be for querying some data, or for anything to do with automations. Aside from the implementation details, the server should be describable as a REST API. Where these API's live are not important for the time being.

For the server components, the following considerations need to be evaluated.

#	Criteria	Description
1	API	The server should have well-structured, consistent and intuitive APIs for the dashboard to consume.
2	Authentication	The server should have proper authentication mechanisms for the APIs to ensure that only authorized users and services can access the API

3	Error Handling	The server should have a consistent and informative error handling strategy for the APIs, providing meaningful information and dealing with the request-response cycle accordingly
4	Asynchronous Processing	The server should support asynchronous processing for time-consuming/background tasks.
5	Logging and Monitoring	Each server component should support comprehensive logging and monitoring mechanisms in place to track performance, identify bottlenecks and quickly debug issues.
6	Automation capabilities	Decisions must be made regarding the choice of automation engine. This includes selecting the appropriate tools and determining how to facilitate the infrastructure required for automation rules based on sensor readings or other events. Automations could be divided into three separate levels of complexity. ➢ *Level 1* Users can manage automations on a fixed schedule, such as turning on Actuator X at 08:00 AM and turning it off at 09:00 AM. ➢ *Level 2* Users can manage automations based on sensor data, which is considered a *primitive feedback loop*. An example would be, "if Sensor X exceeds a certain threshold, activate Actuator Y until Sensor X returns to an acceptable range." ➢ *Level 3* Users can manage automations that consider multiple sensor values, actuator states, and user-defined thresholds. This level represents a *smart feedback loop*, ideally providing self-correcting behaviors within the vertical farm. This level expands on Level 2 by providing a more extensive and more autonomous automation system. *: This criterion is shared with the vertical farm(software), as automations might be solved on either side of the system.

Table 13: server criterions

4.4.3. Considered Options

4.4.3.1. Vertical Farm Software

To gain more information about Raspberry Pi-based vertical farm systems and their technical workings, I looked for existing projects which solved similar problems by providing an environmental monitoring platform and carried out a multi-criteria analysis.

The two contenders were Mycodo and Home Assistant, because compared to other tools they're just a lot more complete. Mycodo is a platform meant for environmental monitoring, while Home Assistant is meant as a generic tool in the bigger context of home automation (including environmental monitoring/control).

To carry out a multi-criteria analysis for Mycodo and Home Assistant, we will evaluate each platform based on the criteria specified in Table 12. The evaluation will be based on a scale of 1 to 5, with 1 being the lowest (poor) and 5 being the highest (excellent).

4.4.3.1.1. Mycodo

Mycodo is an open-source platform for environmental monitoring and automation. It is designed to allow users to monitor and control a range of environmental parameters. Mycodo is highly customizable, with a user-friendly web interface that allows users to configure their monitoring and control systems according to their specific needs. It supports a wide range of sensors and controllers, including popular devices such as the Raspberry Pi. Mycodo also includes a range of data logging and visualization tools, allowing users to analyze historical data and trends over time. On top of all these features, Mycodo provides a rest API for integration with other applications.

#	Criteria	Score
1	Sensor support	5
2	Actuator support	5
3	API availability	4
4	Data storage	5
5	Configuration storage	3
6	Automation capabilities	5
	Total	**27**

Table 14: Mycodo MCA scores

- Sensor support: Mycodo supports I2C, 1-Wire, SPI, and GPIO-based sensors, offering extensive compatibility with various environmental sensors, including custom sensor integrations. Mycodo is designed in a modular fashion; there are pre-made modules for specific sensor models (including the Atlas Scientific ecosystem) as well as the ability to programmatically create a custom implementation if required.
- Actuator control: Mycodo provides a powerful and flexible platform for actuator control in vertical farming, with support for a wide range of actuators and interfaces such as I2C, 1-Wire, SPI and GPIO-based actuators. The REST API makes it easy to integrate the functionalities of these actuators with external applications and tools such as an admin dashboard.
- API availability: Mycodo features a RESTful API that allows for common functionalities to be integrated into external applications

- Data storage: Mycodo uses InfluxDB for time-series data storage, enabling efficient retrieval and visualization of sensor data. This data can be queried through the Mycodo API as well as directly through the InfluxDB API.
- Configuration storage: Mycodo stores configuration settings in an SQLite database, which can be backed up and restored easily. However, these configurations can't be changed through the REST API.
- Automation capabilities: Mycodo offers a Python-based conditional rule-engine system for creating complex automation rules based on sensor data, time-based events, or other conditions. The API provides the ability to drive actuators in a semi-automated way by means of providing a duration to the actuator state.

4.4.3.1.2. Home Assistant

Home Assistant is an open-source platform for home automation that allows users to control and monitor a wide range of smart home devices and systems. It is designed to be highly flexible and customizable, with support for a wide range of devices and protocols.

When it comes to vertical farming with a Raspberry Pi, Home Assistant can be a useful tool for managing and automating various aspects of the setup. For example, Home Assistant can be used to control and monitor environmental factors such as temperature, humidity, and lighting, as well as to automate tasks such as watering and nutrient delivery.

#	Criteria	Score
1	Sensor support	5
2	Actuator support	5
3	API availability	3
4	Data storage	3
5	Configuration storage	4
6	Automation capabilities	3
	Total	**23**

Table 15: Home Assistant MCA score

- Sensor support: Home Assistant supports I2C, 1-Wire, SPI, and GPIO-based sensors, as well as integration with various environmental monitoring platforms through custom integrations or native support.
- Actuator control: Home Assistant supports various actuators and allows users to control them through the user interface or automation rules, using GPIO, I2C, SPI, or other communication protocols.

- API availability: Home Assistant allegedly features a RESTful API built on Python and the Flask web framework, supporting CRUD operations for devices, data, configurations, and automation rules. The API is however quite poorly documented, which makes me question whether all of the listed features are accessible through the API.
- Data storage: Home Assistant uses SQLite or other compatible databases (e.g., MySQL, PostgreSQL) for storing sensor data, managed by the SQLAlchemy ORM.
- Configuration storage: Home Assistant stores configuration settings in YAML files or its integrated storage system, which uses an SQLite database.
- Automation capabilities: Home Assistant has a Python-based automation engine, allowing users to create complex rules and triggers for controlling connected devices. The automation system is built on the Home Assistant Core, which is based on Python and the AsyncIO library. Users can define automation rules in YAML files or create scripts using Python, enabling advanced customization and flexibility in automating various tasks in the vertical farming system.

The multi-criteria analysis conducted on Mycodo and Home Assistant reveals that Mycodo is the better option for a Raspberry Pi-based vertical farming system due to its specific domain of environmental monitoring and many of the features being well documented. Mycodo scored higher than Home Assistant in almost all criteria except configuration storage, making it well-suited for managing and optimizing the environmental conditions in a vertical farming system. Mycodo's support for a wide range of sensors and actuators, flexible automation capabilities, and REST API, as well as its use of InfluxDB for data storage and visualization, offer powerful capabilities for monitoring and analyzing sensor data.

4.4.3.2. Server

4.4.3.2.1. Hasura with Serverless Functions

Since it's part of the graduation company's technology stack; Hasura and Serverless Functions have been tested against the requirements of our system. Hasura is an open-source engine that provides real-time GraphQL APIs on top of existing databases, enabling developers to quickly build and deploy scalable, high-performance applications with a simple and flexible API. It supports a wide range of databases, including PostgreSQL and MySQL, and offers features like authentication, authorization, and event-driven programming.

The approach of the graduation company is meant for cloud-driven apps where hasura is used to act as the primary backend service by providing interfaces that allow developers to implement common backend features in an easier and more declarative way. For example, serverless functions can be mapped to hasura actions and then these actions can be consumed by a front-end client directly, cron jobs are available, authentication/authorization is provided, and table-based event CRUD triggers can be management out-of-the-box. These features provide a solid foundation to easily build web

apps that fit the cloud-environments of the current development landscape.

However, when considering the company stack for an IoT-based system such as the vertical farm, using serverless functions adds a configuration overhead to facilitate communication between serverless functions and the vertical farms. If serverless functions would be used, it would also limit the amount of flexibility of the server compared to developing a custom backend server.

#	Criteria	Score
1	API	4
2	Authentication	3
3	Error Handling	2
4	Asynchronous Processing	3
5	Logging and Monitoring	3
6	Automation capabilities	2
	Total	**17**

Table 16: hasura + serverless MCA scores

The multi-criteria analysis provides an evaluation of using Hasura with serverless functions for the core of the backend implementation of the system.
Here is a summary explaining why each criterion received its respective score:

1. API: Hasura offers a robust and flexible API through real-time GraphQL, allowing for efficient development and deployment of high-performance applications. The API's versatility and simplicity make it a strong choice, resulting in a high score.
2. Authentication: Hasura provides basic authentication and authorization features, which can offer an adequate level of security for the system. However, it may not be as comprehensive or customizable as a custom-built authentication solution, thus receiving a moderate score.
3. Error Handling: While Hasura has some error handling capabilities, it may not be as consistent or informative as a custom backend solution. This may lead to difficulties in managing and addressing errors in the system, which is why it receives a lower score.
4. Asynchronous Processing: Hasura supports asynchronous processing through one-off scheduled and cron-based events. These events can be configured with retry options for extra reliability. This

5. Logging and Monitoring: Hasura provides some logging and monitoring features, but they may not be as comprehensive or customizable as a custom backend solution. Engine related logs are also written to a file, which makes integration with other logging/monitoring systems difficult.
6. Automation capabilities: Hasura's automation capabilities may not be as flexible or extensive as a custom backend solution, especially when considering the specific requirements of the system. This limitation in automation support leads to a lower score. The cron scheduler could be implemented to function as a basic rule engine. However, this approach introduces a couple of new issues into the system. For example, the scheduler uses Postgres for storing metadata. This metadata is wrapped around a primitive implementation which does not scale with the number of vertical farms.

4.4.3.2.2. Custom Backend Server

Developing a custom backend provides several advantages over using Hasura (with serverless functions). For example:

1. Fine-grained control

A custom backend allows for complete control over the architecture, implementation, and functionality. This enables the development of features and optimizations tailored specifically to the vertical farming use case, resulting in better performance and efficiency.

2. Integration flexibility

A custom backend provides the flexibility to integrate with a wider range of third-party services, patterns, tools, and libraries.

3. Scalability and performance

By developing a custom backend, the system can be optimized for specific requirements, ensuring better performance and scalability. This can lead to lower latency and more efficient resource utilization.

4. Security

With a custom backend, there is full control over the security implementation, allowing the system to meet specific security requirements and standards.

5. Maintenance and support

With a custom backend, there is complete control over the system's development, maintenance, and support, allowing for the addressing of issues and implementation of improvements in a timely manner. This can be especially beneficial in an evolving system like a vertical farm.

#	Criteria	Score
1	API	5

2	Authentication	5
3	Error Handling	5
4	Asynchronous Processing	4
5	Logging and Monitoring	4
6	Automation capabilities	4
	Total	**27**

Table 17: Implementing a custom backend scores

4.5. System Architecture

4.5.1. Vertical Farm Components

To get started, a Raspberry Pi-based vertical farm would need to be developed to be able to monitor and control the metrics and test the rest of our system against. The Raspberry Pi serves as the foundation of our vertical farm system; therefore, it is crucial that we select an implementation that aligns with our initial requirements and leaves room for potential future enhancements. To design the functional aspects of a vertical farm, the following building blocks will need to be considered:

Component	Description
Rack	The rack is the physical setup that is going to contain the implemented system. This component will not be a part of this project as there are too many variables involved to fine-tune specific rack-related configurations for a vertical farm. However, new sensors might be added, water supplication and distribution lines might change, etc., so the system's design must take into consideration that the configuration of the rack might change by making these changes reflect back on the system's components.
Raspberry Pi Software	The Raspberry Pi must run a software platform which essentially makes it possible to drive all the functions that the Raspberry Pi would need to fulfill its role as the vertical farm controller.
Raspberry Pi Hardware	The hardware consisting of the Raspberry Pi itself, as well as any sensor- and actuator device. One option would be that the sensors & actuators are hardcoded into the config of the Raspberry Pi, but a better solution would be to have a dynamic system in place that keeps track of connected sensors & actuators so that any hardware setup will work with the system.

Table 18: Components of a vertical farm

For a proof-of-concept, it would be possible to use a simulated/virtual setup for the Raspberry Pi, which would essentially publish sensor data somewhere and keep track of

some actuator states. The virtual device could then expose functionalities that mimic a physical Raspberry Pi. However, since the implementation of the dashboard also depends on the implementation of the vertical farm and it's API(s), a minimum physical setup will be set up and integrated into the proof-of-concept. This approach also increases the testability of the system.

4.5.1.1. Mycodo

Mycodo has been chosen as the Raspberry Pi software to drive the vertical farm. Mycodo provides us with the entire infrastructure required to start prototyping rapidly on other parts of the system instead of building such an advanced tool which would take well over the period of the graduation project. Mycodo provides us with abstractions over common components and features of a greenhouse setup and keeps track of all this state in a python application packaged as a convenient Linux system service.

Mycodo will provide the base for our vertical farm, so that the development of the project can focus on other parts of the system's development. Through initial testing of the API, it was concluded that it should be possible to extend a Mycodo runtime into a scalable management dashboard by consuming the API it provides. These tests were done by setting up a Mycodo instance on a Raspberry Pi, retrieving an API token, and doing HTTP requests through the OpenAPI/Swagger endpoints (UI) that Mycodo provides.

4.5.1.2. Hardware

A Raspberry Pi will be responsible for controlling and monitoring various aspects of the vertical farm using sensors and actuators. These sensors will be used to monitor environmental factors such as temperature, humidity, and nutrient levels. The actuators will then be used to adjust these factors in real-time based on the sensor data.

During the first phase of the project, a proposal was

The *Atlas-scientific* ecosystem is a collection of sensing modules designed for use in hydroponic and aquaponic systems. The ecosystem includes a wide range of sensors for monitoring environmental parameters such as pH, dissolved oxygen, conductivity, and temperature, as well as actuators for controlling pH and dissolved oxygen levels.

The Atlas-scientific ecosystem is perfect for this project, as it allows for a high-quality system, easy interfacing, and a no-soldering solution. The only downside of choosing the Atlas-Scientific sensors is the financial aspect. Because they are very expensive compared to alternatives since the Atlas-Scientific devices have been designed for industrial purposes where precision is crucial and environmental factors are much harsher. However, after discussing the benefits with the customer, we agreed upon this hardware implementation, and they were provided in week 6 of the project.

The following hardware has been provided for the proof-of-concept of the system:

- Atlas-scientific Whitebox T3 carrier board

- Atlas-scientific EZO pH sensor
- Atlas scientific EZO CO2 sensor
- Atlas-scientific EZO Humidity sensor
- Atlas-scientific EZO EC sensor

Based on the results of research sub-question 3.1.2.2: '*Functional Processes within a Vertical Farm (VF)*', the following sensors and actuators have also been added to the proof-of-concept phase:

- Water pump
- Solenoid Valve
- 2x 12VDC Fan
- 2x LED matrix
- Liquid temperature probe
- Camera

This has resulted in the following hardware diagram for the vertical farm:

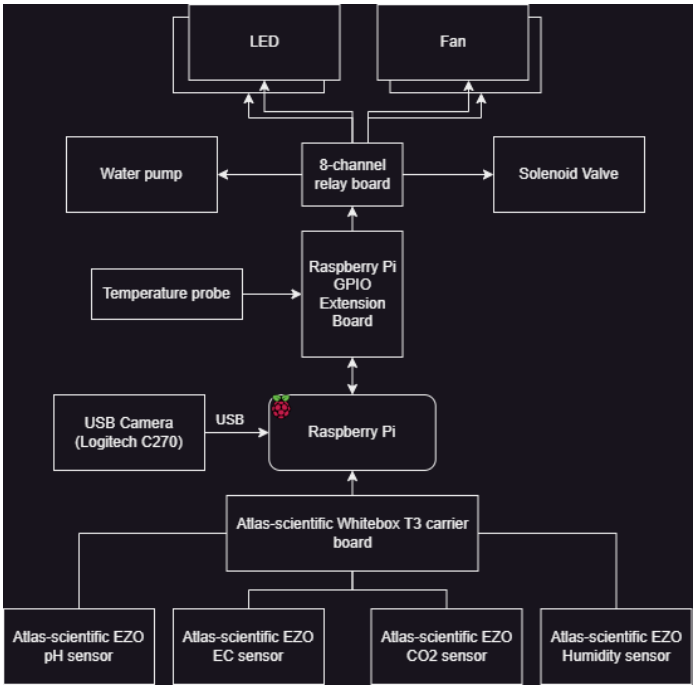

Figure 13: Hardware diagram

4.5.2. Server Components

Based on the findings of research sub-question in chapter 3 - Integrating User-Defined Automations with Varying Levels of Complexity into the Vertical Farm Management System', the following application architecture diagram has been created to facilitate the server with the infrastructure required to implement a similar(but minified) platform of a web-based industrial IoT-platform:

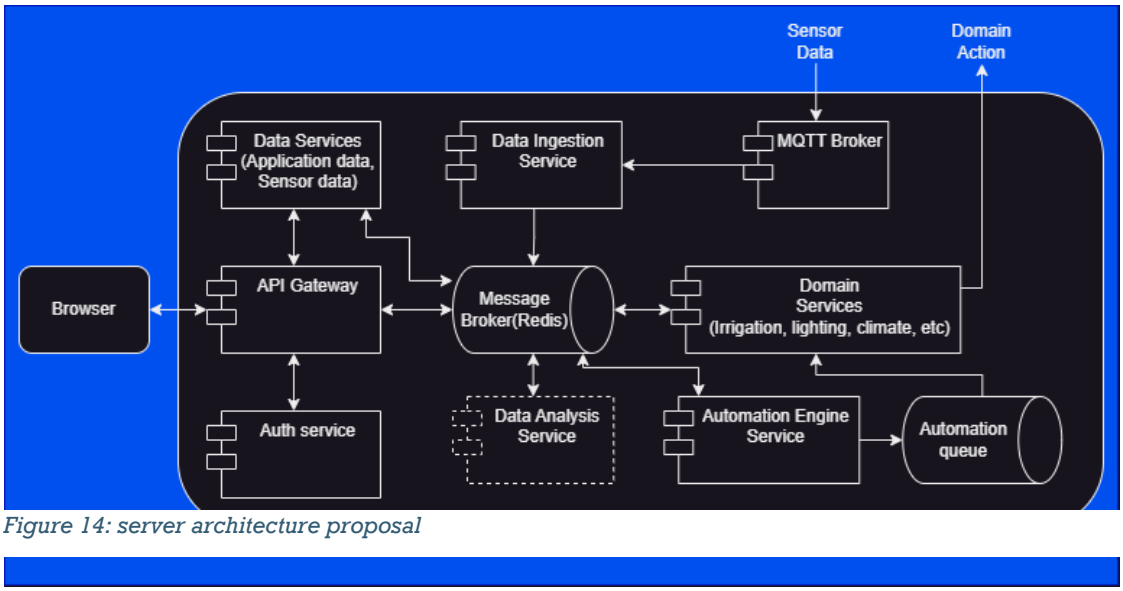

Figure 14: server architecture proposal

A combination of event-driven microservices can be utilized. This should ensure that the server is designed in a modular, scalable, and maintainable manner while separating concerns within the system. Following is a description of each component:

1. *API Gateway*

The API Gateway is a common practice when using microservices. It acts as a single entry-point for all client requests and routes them to the appropriate microservices. The benefits of using an API Gateway include improved security, centralized authentication, and potentially load balancing.

2. *Auth Service*

The Auth Service is responsible for authenticating clients using JWT (JSON Web Tokens). The API Gateway consults the Auth Service before allowing a client to access any microservices, ensuring that only authorized clients can access the system. By separating the authorization process into its own service, we can also easily integrate a cloud-based authentication provider.

3. *Data Services*

Data services include a collection of microservices that integrate with various data solutions, such as Hasura, cloud services, and the Mycodo API. These services facilitate the retrieval and storage of sensor data from the InfluxDB instance (either on the Raspberry Pi or the server).

4. *Message Broker (Redis)*

The Redis message broker enables communication between microservices, facilitating event-driven flows. It allows for the efficient handling of events and messages, ensuring smooth and reliable data flow between services.

5. *MQTT Broker*

The MQTT Broker subscribes to sensor data from the Raspberry Pi's, enabling real-time updates and communication between the server and the vertical farm system.

6. *Data Ingestion Service*

The Data Ingestion Service processes data received from the MQTT broker and makes sure that the data is available throughout the server where required, for example persisting the data or propagating the data to another service. A possible implementation of this service might use Apache Kafka.

7. *Data Analysis Service*

The Data Analysis Service processes real-time sensor data from the Data Ingestion Service and produces events for the Automation Rules Engine, as well as other services if required. This enables dynamic adjustments to the automation rules based on data analysis. As seen in Figure 14, this service is optional. This is because level 2 automations can technically also be achieved by performing simple threshold checks and comparing the current value to it. Based on the result, an automation rule may or may not be triggered.

8. *Automation Service*

The Automation Service manages scheduler-like functionalities to control automations through its own queue. It coordinates the timing and execution of automation tasks. These tasks could be based on a cron expression, an interval or when specific events are produced/consumed.

9. *Domain Services (lighting, irrigation, climate):*

Domain services communicate with the vertical farms to perform certain (vertical farm related) tasks, such as switching lights or water pumps on or off. Since it should also be possible to invoke these services without the automation service, using an event-driven microservice architecture allows us to write the logic once and reuse in any other service.

4.6. Technology Stack

4.6.1. Front-end framework: React

React is a widely used JavaScript framework that facilitates the development of dynamic web UI's. It allows developers to create reusable components, which can be easily integrated with other libraries and frameworks. React also provides a virtual DOM for efficient updates and rendering of UI components.

The options regarding this choice were really between any component-based framework. React is already part of the company stack(and the project boilerplate) and is also the most mature front-end framework with a big developer community for support and documentation. It should also be easier to ask for help regarding react from coworkers.

4.6.2. Back-end framework: NestJS

NestJS provides a scalable and efficient framework for developing server-side components in Typescript within a NodeJS runtime. It is a versatile platform with a declarative yet customizable approach to backend development and offers plugins for many tools required for the project, including MQTT protocol, Kafka, Redis, and even Hasura/GraphQL, if desired. It's a very mature framework with extensive documentation and tutorials available.

NestJS supports microservices as well as event-driven architectural patterns, allowing one project to contain multiple microservices, where the NestJS framework provides the necessary tools for implementing these services. A key advantage of NestJS is its modular approach to backend development, which allows for separate development and independent deployment of each module, making it easier to manage and maintain the backend.

However, the microservice-based approach, combined with a message broker for service-to-service communication, enables the use of different frameworks or runtimes if required. For instance, if a specific library is not supported in a NodeJS environment, an alternative framework or runtime may be employed, if it has support for communication with the message broker's protocol. This flexibility ensures that the project can adapt to various requirements without being limited by a single technology stack.

4.7. Mycodo API integration

The Mycodo install script installs 4 packages which compose the Mycodo runtime:

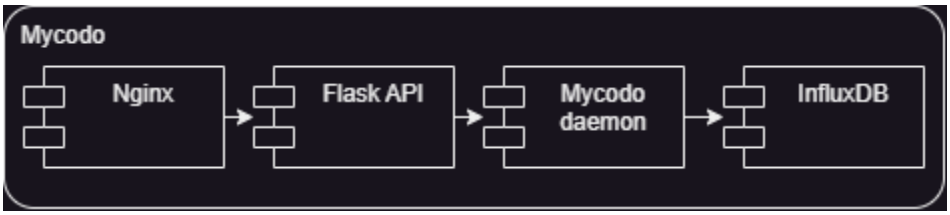

Figure 15: Mycodo software architecture

1. *Nginx*

Nginx is a popular, lightweight, and high-performance web server and reverse proxy server. Nginx hosts the Mycodo web interface, which provides users with access to the dashboard, settings, and various control options for their vertical farm or environmental monitoring system. Nginx also acts as a reverse proxy for the Mycodo runtime and routes HTTP requests to the right internal port.

2. *Flask API*

A lightweight web framework used for creating the Mycodo RESTful API to manage and control the system

3. *Mycodo daemon*

A background service that continuously runs, responsible for managing sensors, actuators, and automation rules

4. *InfluxDB*

A time-series database used for storing sensor data and other time-based metrics collected by Mycodo

While testing the Mycodo Flask API during the initial phase of the project, some conclusions could be drawn around how from a technical context, the API could be integrated with the system. This section describes the motivation for the design around the Mycodo API. The following conclusions were drawn:

#	API endpoint	Takeaways
1	/api/camera	This endpoint allows clients to fetch the last camera image, but this only works with the Raspberry Pi cam-module and not with a USB camera as a USB camera's Mycodo implementation persists the images to the file system.
2	/api/controllers	This endpoint allows clients to query and update the state of any Mycodo controller by providing a ***unique_id*** path parameter. A Mycodo controller is an abstraction over sensors(inputs) and actuators(outputs).
3	/api/daemon	This endpoint allows clients to query the status of the Mycodo daemon and returns the RAM usage and the Mycodo daemon state.
4	/api/functions	This endpoint allows clients to query configuration settings related to all or just one Mycodo function. A Mycodo function is a python script wrapped as an action. This action can be triggered by other Mycodo components. This endpoint does not allow clients to manage functions through the API.
5	/api/inputs	This endpoint allows clients to query configuration settings related to all or just one Mycodo input. A Mycodo input is a sub-abstraction of a Mycodo controller, implemented for sensors. One input in this case equals the configuration settings for one physical sensor.
6	/api/measurements	This endpoint allows clients to create, query sensor data from the InfluxDB instance as three implementations: 1. Measurements found within a time range from start date to end date 2. Last measurement found within a duration from past to present 3. Measurements found within a duration from the past to present

7	/api/outputs	This endpoint allows clients to query configuration settings related to all or just one Mycodo output. A Mycodo output is a sub-abstraction of a Mycodo controller, implemented for actuators. One output in this case equals the configuration settings for one physical actuator.
8	/api/pids	This endpoint allows clients to query configuration settings related to all or just one Mycodo PID controller. A Mycodo PID controller is a self-contained and easy to configure PID implementation. PID controllers can be managed through the Mycodo UI, but not through the API.
9	/api/settings	This endpoint allows clients to query many different configuration settings. Some of these include a 'duplicate' endpoint with a different response body(*). The complete list consists of: - Device measurements - *Inputs(*)* - *Measurements(*)* - *Outputs(*)* - *PIDs(*)* - Triggers - Units (of measurement) - Users

Table 19: Mycodo API analysis

A key takeaway from the Mycodo API design is the assignment of a unique_id property to assets within the runtime, such as sensors and actuators. This unique_id can be queried through the API and used in subsequent requests.

4.8. Data Storage and Retrieval

4.8.1. Application data storage

Until now, a system architecture has been designed where little to no consideration was taken for the choices regarding databases and general storage strategies. Application data is the data that is required for consumption by either the dashboard or server-side services.

4.8.2. Asset Registry

Since the configuration for assets per vertical farm can change at any time, the implementation should facilitate a flexible enough implementation for this asset registry component so that the most up-to-date asset information is highly available.

Based on the findings of chapter 4.7: 'Mycodo API integration'. We discussed importance of the ***unique_id*** within Mycodo so to simplify the integration of Mycodo into the system, a logical design choice would be to implement an asset registry centered around the assets (inputs/outputs) tracked in Mycodo. This can be achieved by periodically fetching the assets and updating the asset registry accordingly. Consequently, any service that requires information about an asset(like it's unique_id) can query the asset registry instead of performing network requests to the Mycodo API every time 'metadata' about an asset is needed. Given the extensive use of the ***unique_id*** metadata property by Mycodo for various API operations, this approach streamlines the process and enhances system efficiency.

By implementing an asset registry and synchronizing it with Mycodo's assets, the system can better manage and utilize the ***unique_id*** metadata property associated with the assets, reducing the need for frequent API calls and improving overall performance. This design choice also ensures that the system remains organized and maintainable, as all the asset information is centralized and easily accessible.

Implementing a (self-updating) asset registry also comes with a major benefit for the operational requirements of the system; since part of the asset registry functionalities is managed by Mycodo, synchronizing these assets also means that the server will support vertical farms with differing sensor- and actuator setups.

4.8.3. Sensor Data Storage

Another important consideration that had to be made while designing the system, is the sensor data storage. It was concluded that Mycodo provides an InfluxDB instance which fits the needs of our system. The benefit of using this method, is that any data related to a specific vertical farm is contained within that environment, locally, while being available through the API.
The drawback of using storage locally on the Raspberry Pi is that any time sensor data has to be queried, it will have to be processed over the network. Depending on the number of data points returned, this approach may not be very performant, but it provides more flexibility. If advanced data analytics becomes a consideration, an alternative method for storing sensor data might have to be employed.

4.8.4. Querying InfluxDB data

With the choices made regarding the system architecture up until now, there are two different methods to query sensor data from a vertical farm:

The Mycodo API uses the InfluxDB API under the hood, so this would be the recommended method of querying sensor data. However, the integration process looks slightly different, and both offer different pros and cons.

1. Using the Mycodo API

To query sensor data using the Mycodo API, an HTTP request would have to be made to the */api/measurement* endpoints, providing the required authentication and parameters. The API will return the requested sensor data in a structured format. Refer to chapter 4.7: Mycodo API integration for the three endpoints provided by the */api/*measurements route. These endpoints have the following parameters:

	Endpoint	Parameters	Response interface
1	/api/measurements/historical/	- Unique_id: string - Unit: string - Channel: integer	List of measurements

		- Epoch_start: integer - Epoch_end: integer	
2	/api/measurement/last	- Unique_id: string - Unit: string	Single measurement
3	/api/measurement/past	- Channel: integer - Past_seconds: integer	List of measurements

Table 20: querying sensor data through the Mycodo API

2. *Directly through InfluxDB*

To query sensor data directly from InfluxDB, a client library is required that supports InfluxDB's API. InfluxData provides an official client library for Node.js environments, which allows seamless interaction with InfluxDB. The primary advantage of using this approach, as opposed to the Mycodo API, is the ability to leverage Flux queries. Flux is InfluxData's functional data scripting language designed for querying, analyzing, and acting on time series data. Features of the flux query language include transforming the queried data, calculating common mathematical functions, aggregation, conditional logic, custom functions and more.

Utilizing Flux queries provides greater flexibility when it comes to parametrizing and customizing data retrieval. This approach should be used for complex and tailored queries to suit specific use cases.

A con about this approach is that, to communicate with the InfluxDB instance directly, the consumer needs to have the proper InfluxDB credentials, bucket name and org id. These can be set when provisioning a device into the system, but ideally a user shouldn't have to worry about these magic strings as it's an additional configuration overhead as well as extra room for errors in the integration of query data for the front-end.

5. Development of the system

This chapter will describe the technical implementation of the system and the system's implementation process will be described.

Since server management was also a non-functional requirement, *cockpit* has been added to the install script of Mycodo to include it with the installation by default. Cockpit allows users to perform server administration tasks through a web UI hosted on the Raspberry Pi. Cockpit allows for the monitoring of system resources, managing network connections, updating software packages, managing user accounts, and performing basic server administration tasks, all through an intuitive and user-friendly web interface. By including Cockpit in the default installation of Mycodo, users can easily manage and maintain their Raspberry Pi without the need for additional tools or extensive command-line knowledge.

For a summary of what the physical system looks like, refer to Appendix B Development of the physical Farm'. This was not included in this chapter to keep the focus of the project on the software of the system.

5.1. Back-end Development

The backend has been developed using a microservices and event-driven approach. This has mostly been accomplished by using the NestJS framework. After considering a couple of final designs, this is the architecture design that has been settled on for the proof-of-concept. This architectural approach with microservices and a message broker helps to decouple the services, improving scalability and reliability of the overall system. This approach also makes it easy to expand on the system with new modules without being limited in frameworks/programming languages.

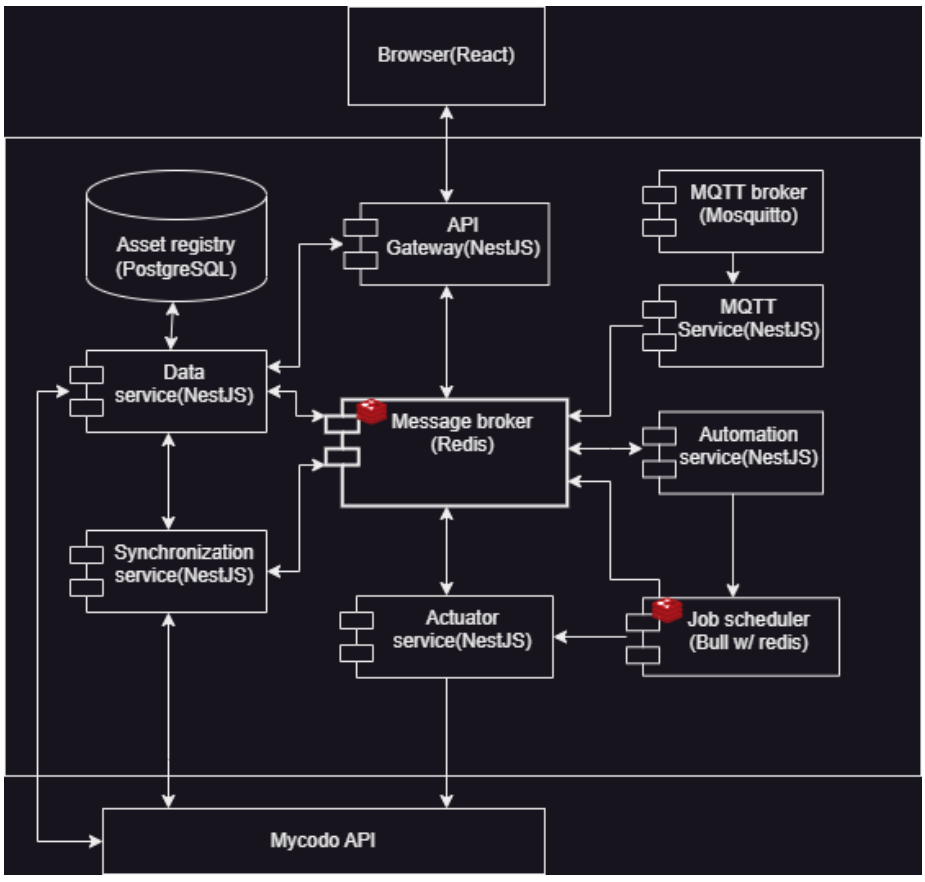

Figure 16: final backend architecture, based on the Industrial IoT architecture

5.1.1. Assets Registry

As described in chapter 4.8.2: Asset Registry, devices, sensors and actuators (assets) are tracked in the asset registry. Since the asset registry will be used as the source of truth for the dashboard, the flow for querying sensor data and visualization in our system follows a specific process that ensures accurate and up-to-date information as well as a generic interface to assets from the Mycodo runtime. This allows services which require information

about assets(like a unique_id), to query the information from the assets registry for usage in subsequent requests. For the proof-of-concept, a PostgreSQL database has been used in conjunction with hasura to facilitate a datastore that acts as an asset registry by keeping track of raspberry pi's, their sensors and actuators.

5.1.2. Data service

In the backend, two types of data can be distinguished: sensor data and application data. The data service implements methods to manage both types of data, providing a unified interface for data storage, retrieval, and manipulation. As the implementation of this "service", hasura has been used during the proof-of-concept phase, because this was initially the chosen backend engine. This decision was driven by the company technology stack preference and for its instant GraphQL API. At the start of the project's development phase, GraphQL was so ingrained into the app that switching, to a HTTP-based approach with an ORM for example, would take a good amount of time and refactoring.

5.1.3. Synchronization service

The synchronization service is responsible for keeping the asset registry updated by periodically fetching the latest data from the Mycodo API. This service ensures that any changes made to the assets, such as adding new sensors or actuators or changing Mycodo configurations, are reflected in the asset registry and available to the other services.

One method to implement these synchronizations is through cron jobs. Cron jobs can be executed periodically to fetch asset metadata from the vertical farms and update the asset registry with the newly fetched data. This would look as follows:

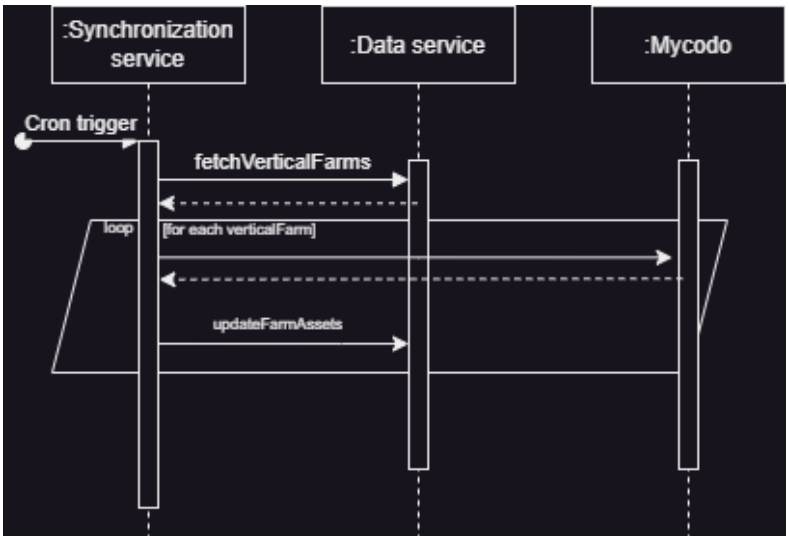

Figure 17: synchronization sequence diagram

This service has only one responsibility: synchronize vertical farms assets with the data service.

5.1.4. Automation service

The automation service is designed to handle automated tasks and processes within the system. To implement this service, the decision was made to use a redis-based implementation, because it's fast, lightweight, and easy to deploy. Another factor is that because NestJS was the chosen (main) backend framework, using their integrated modules became an option; one of these options was the *Bull* npm package.

The automation service has its own instance of redis which acts as a queue system. Bull provides an interface on top of this queue to provide powerful features. The bull API can be used to manage scheduled jobs as well as one-off jobs. The specified queue then processes the jobs in a queue based on the implemented *processor classes*. These classes listen for specific events and further process the job based on methods annotated with the *@Process* annotation. To implement the automations on levels one, two and three, this processing mechanism can be utilized to process small tasks and propagate other events into the message broker. Since the data passed to a job is arbitrary, it's possible to pass all the required information into the job and unpack the data in the processor class.

5.1.5. Actuator service

Actuator control is implemented by providing methods for managing and controlling actuators based on their respective unique_id, allowing for a consistent and efficient approach to actuator control within the system.

The actuator service implements the */api/outputs* API endpoint of Mycodo to provide actuator-focused functionalities like changing the state of an actuator.

Just as sensors are assets, actuators are also treated as assets in the context of Mycodo. This is why the metadata related to actuators is also synchronized by the synchronization service through cron jobs. When this metadata is required, it is queried from the asset registry.

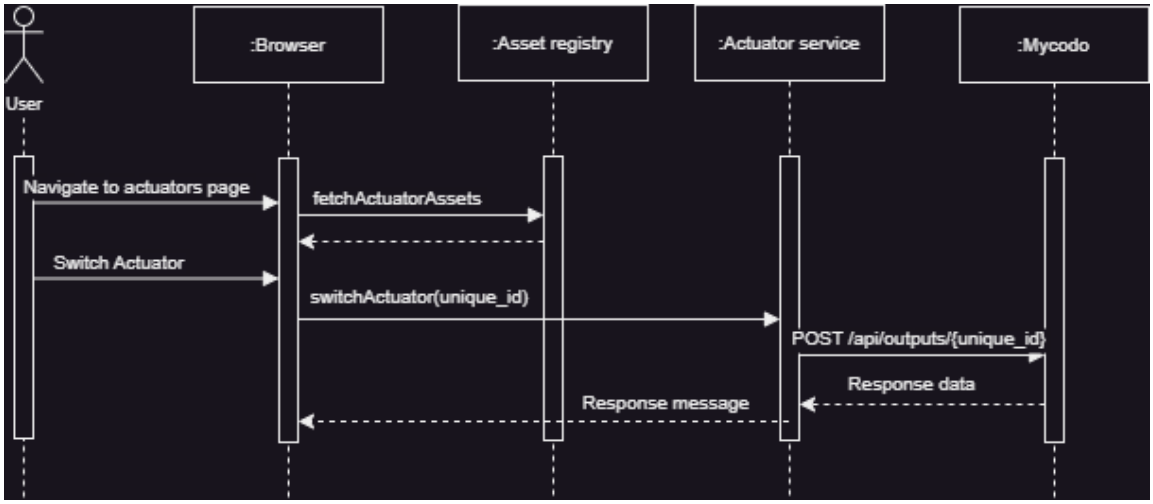

Figure 18: actuator control sequence diagram with relation to server components and their role

5.2. Front-end Development

This section will explain the front-end development process and how it relates to the features defined during the analysis and design phases of the project. The most important UI components and their interactions will be described within the context of use cases and user stories.

The boilerplate project from the graduation company was used to allow for quick prototyping. This boilerplate contains built-in features such as light/dark mode, a routing system, utility hooks, styled UI components and more.

5.2.1. Vertical Farm Provisioning

Provisioning a new vertical farm in the admin dashboard is a simple process. First, Mycodo must be installed and correctly configured, after which the local IP address of the Raspberry Pi will be displayed on the terminal. To add a new Raspberry Pi to the system, to the designated screen and provides details such as the Raspberry Pi name, IP address and API tokens. Once the new Raspberry Pi is added, the server has all the information about a vertical farm to allow the dashboard to consume the available APIs to communicate with the farm.

Then, the dashboard will display the newly added raspberry pi in the drop-down menu on the top bar if the Pi is reachable on the network. Cron jobs execute a round of ping-pong/health to check if the Raspberry Pi's known to the system are reachable on the network and updates their 'online' status accordingly in the asset registry.

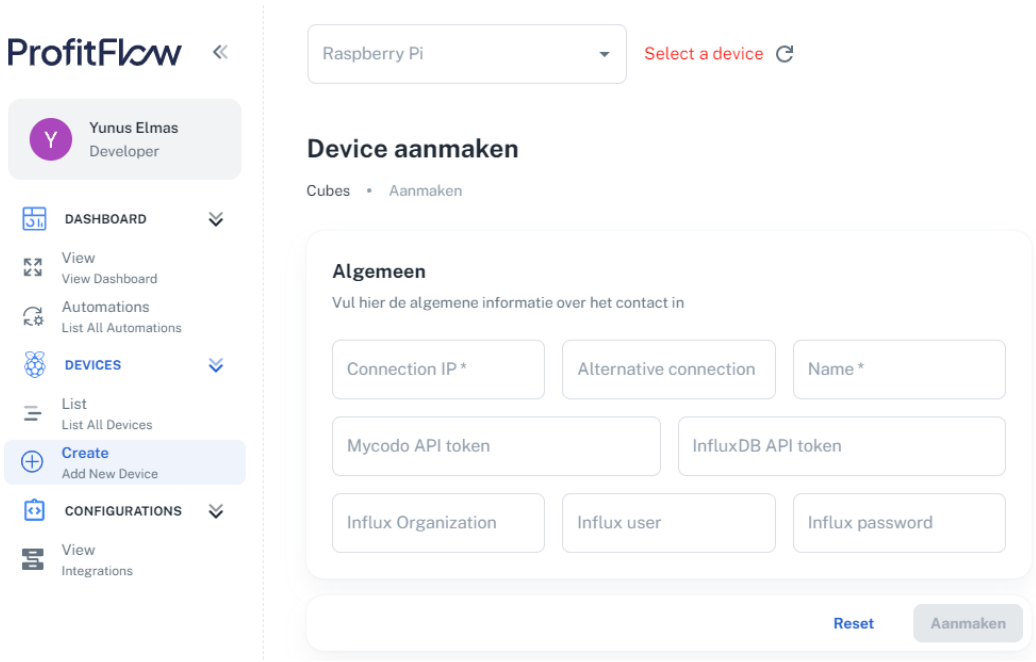

Figure 19: device provisioning screen

5.2.2. Vertical Farm Selection

A drop-down menu in the header of the page periodically fetches a list of "online" Raspberry Pi's that have been provisioned into the server. It then displays these devices in the dropdown list.

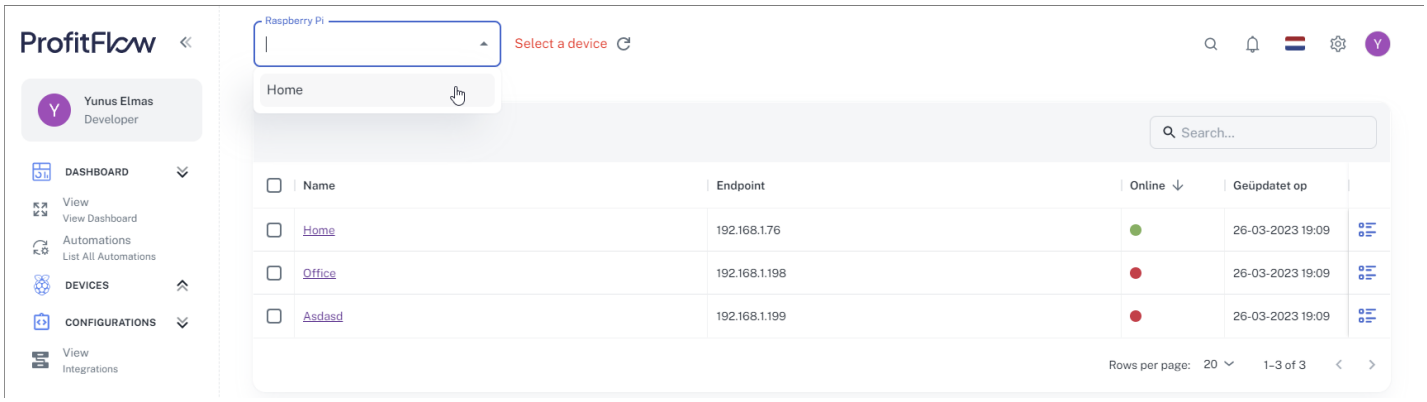

Figure 20: overview of registered devices, with real-time 'online' status

Figure 20 displays the page containing a table to view registered devices. Here we can see that, since this screenshot was taken while I was working from home, the Raspberry Pi(called Home) shows a green(online) status in the table, and it's also the only device showing up in the drop-down menu.

When a user selects a device from the list in the header, the application sets a global state(in-memory, by using the Jotai npm package) containing information about the currently selected Raspberry Pi, including the IP address and Mycodo token. This IP address is then used to dynamically target the correct Mycodo API server by constructing HTTP requests based on the currently selected Raspberry Pi.

5.2.3. Data Monitoring and -Visualization

The charts page displays line charts of data collected from various sensors connected to a Raspberry Pi in the selected vertical farm. Users can filter the sensor data on a time range to be displayed on the charts, with intervals ranging from 15 minutes to 1 week. Refer to Appendix F Data monitoring page' for a full screenshot of the data monitoring page. Here you can see that the charts are based on the hardware connected. Note that the page does not fit on one A4 sized paper so the full window had to be cut, but there are more charts available.

5.2.4. Actuator control

The UI displays a list of card-like components, each representing a single actuator. Metadata is displayed on the actuator's card, such as IDs, interfaces, actuator type and more in one quick overview.

If the actuator has a state associated with it, a toggle button is displayed, allowing the user to turn the actuator on or off. The current state of the actuator is represented by the color of the toggle button, with blue indicating "on" and gray indicating "off."

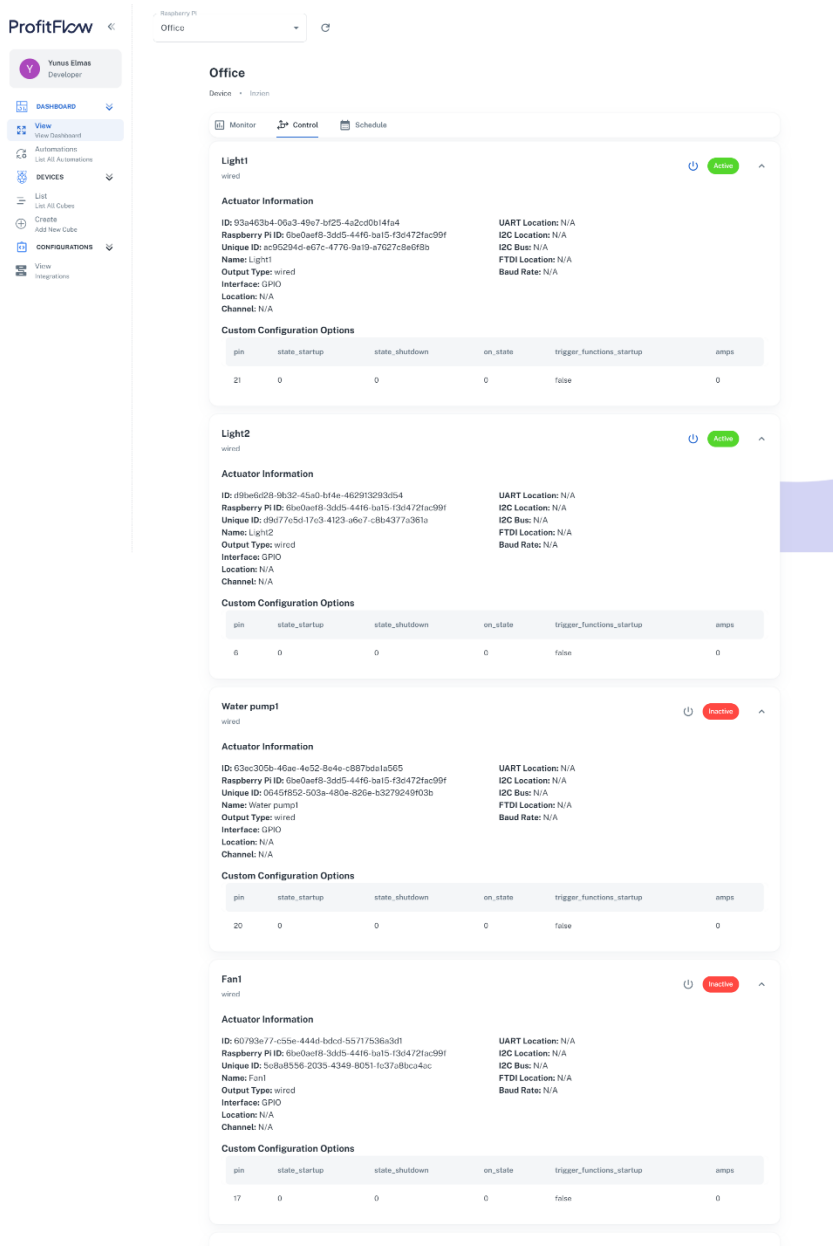

Figure 21: screen containing actuator controls. Each card represents an actuator

6. Conclusion

6.1. Summary of the project

The project aimed to design and implement a proof-of-concept for a multi-farm management dashboard for vertical farms, utilizing Raspberry Pi and Mycodo for sensor data and actuator control. The main goal was to provide a practical system for monitoring and controlling multiple vertical farms from a single dashboard through both hard- and software.

Initially the project would be implemented using the company-provided technology stack(based on serverless functions). This would've been an appropriate strategy, but the customer kept adding constrains such as local hosting. This led to the work done to become obsolete as serverless functions are pretty much just method calls in a local environment with less flexibility.

After this, a lot of research was done on various methodologies and architectural patterns to establish a robust foundation for the project. This phase included the evaluation of different vertical farm-related sub-systems/processes and the integration of user-defined automations with varying levels of complexity. The reason these research questions were chosen is to serve as an addition to the (proof-of-concept) result of this project. The research shows the thought process behind the system's design and concludes potential strategies to incorporate the findings into the system through different methods.

The Mycodo API was integrated into the system, and various data storage and retrieval methods were implemented, including the creation of an asset registry for device, sensor and actuator management.

The resulting system provides a robust and user-friendly platform for managing multiple vertical farms, with features such as monitoring of sensor data, actuator control, automation. The project also improved the maintainability of the system by incorporating the server administration tool Cockpit, which simplifies Raspberry Pi management and maintenance.

In conclusion, the project almost successfully delivered a multi-farm management dashboard for vertical farms(automations have not been integrated with the front-end yet), demonstrating the potential of Raspberry Pi and Mycodo in creating scalable and efficient agricultural systems.

6.2. Reflection on the main research question

The main research question entails: "How can a system be realized that allows for remote monitoring, control and automations for raspberry pi-based vertical farms?".

After doing this project, I can confidently say that there is no ‘‘best-approach’’ to develop a monitoring/control & automations system for vertical farms. There are just a lot of factors involved from the domain itself, as well as the different options to implement such systems being practically endless – so the answer will always end up being ‘‘it depends on your specific requirements’’. There have been some implementation choices during this project

as well which can still use improvement, such as incorporating more cloud technologies, optimizing the communication between physical components and in general applying more common IoT patterns. One takeaway from this project is also that many IoT appliances have the same internal workings, and it take a little bit of creativity to get to an end2end product.

7. Reflection

7.1. Assignment

The project provided me with a variety of learning opportunities and challenges, mainly due to the frequent changes in scope during its development. This wasn't entirely negative, as I had a lot of freedom as a student, but receiving more focused feedback would have helped in narrowing down the scope and making clearer decisions based on the requirements.

My research framework also had to be reworked several times because of the shifting project scope, which led to some lost time and a few obsolete tasks. In the early stages, I spent a significant amount of time researching how to create a custom module for controlling the Raspberry Pi, sensors, and actuators in the vertical farm. However, since the customer's primary expectation was a multi-farm dashboard and the project had a clear timeframe, I decided to use an existing platform for driving the vertical farm and focus on developing the server and dashboard to manage the scope.

Despite these obstacles, I found the technical aspects of the project enjoyable and gained valuable insights into various interesting topics. It required me to explore the technical aspects of vertical farming, IoT, and building custom apps on top of an IoT system. As a developer, I am typically involved in web-based projects, so this assignment took me out of my comfort zone and taught me a lot.

I believe that if the project had clearer boundaries from the start, the results would have been significantly better. For instance, while the automations work on the server-side, I haven't had the chance to integrate them with the front-end yet. Looking back, I think the project, as it was proposed to Saxion before starting the graduation, should not have been accepted. But this is, of course, in hindsight. A lot of energy was spent fine-tuning the project boundaries, requirements, and scope. While this experience has taught me valuable lessons, a more focused starting point would have likely led to a more polished result.

7.2. Personal development

During the graduation, the biggest issue was that I wasn’t getting the domain-related help I needed. On top of this, I had to research, get acquainted with, and implement a system surrounding a vertical farm which I knew nothing about.
It also didn’t help that the customer kept changing the boundaries of the system (hardware connections, testing out different vertical farm setups/solutions that I had to take into consideration during development. So, I had to adapt and create a system generic enough where these factors didn’t matter as much, hence the implementation had to take this into

consideration and be built around being a generic enough solution to facilitate the features independently from the hardware configuration on the Raspberry Pi.

On a personal level, I learned a lot about project management and managing client expectations. The customer told me that he appreciated my communication style, always keeping him in the loop, whether the progress was good or bad. For example, I faced some health issues in the past couple of months, and I learned that these situations can be managed effectively if communicated promptly with relevant parties (company supervisor, customer).

Moreover, starting a project from scratch, analyzing high-level requirements, and continuously managing the project's boundaries, scope, and re-assessing previously gathered requirements provided me with valuable insights into potential pitfalls and strategies to maintain an overview of a project. These experiences have been both challenging and rewarding, contributing to my growth as a professional.

7.3. The company

My experience at ProfitFlow was quite fun but chaotic, largely due to previously mentioned reasons, but also because of the startup environment. Limited domain-related resources within the company made it challenging to receive the necessary support for my graduation project as nobody really had knowledge of IoT. This may have led to me spending more time researching, but not making choices in a timely manner as I didn’t have technical feedback regarding some choices.

Despite these challenges, the company itself was enjoyable to be a part of, as the average age was around 25, and it was clear that everyone was working towards building the company up and becoming better engineers. The dynamic and youthful atmosphere made for an engaging work environment.

Bibliography

Allgower, F. (n.d.). *Model Predictive Control*. Retrieved from uni-stuttgart: https://www.ist.uni-stuttgart.de/research/group-of-frank-allgoewer/model-predictive-control/

Astrom, K. J. (2002). *PID Control.* California Institute of Technology.

Carbonnel, M. d., Stormonth-Darling, J., Liu, W., Kuziak, D., & Jones, M. A. (2022, June 16). *Biology | Realising the Environmental Potential of Vertical Farming Systems through Advances in Plant Photobiology.* Retrieved from https://www.mdpi.com/: https://www.mdpi.com/2079-7737/11/6/922

Dotis-Georgiou, A. (2021, April 19). *InfluxDB's Checks and Notifications System*. Retrieved from influxdata: https://www.influxdata.com/blog/influxdbs-checks-and-notifications-system/

Edinburgh Sensors. (2018, June 11). *Creating the Perfect Environmental and Atmospheric Conditions for Vertical Farming*. Retrieved from Edinburghsensors: https://edinburghsensors.com/news-and-events/perfect-environment-vertical-farming/

Msimbira, L. A., & Smith, D. (2020, 7 10). *The Roles of Plant Growth Promoting Microbes in Enhancing Plant Tolerance to Acidity and Alkalinity Stresses*. Retrieved from frontiersin: https://www.frontiersin.org/articles/10.3389/fsufs.2020.00106/full#:~:text=The%20pH%20range%205.5%E2%80%936.5,availability%20of%20nutrients%20is%20optimal.

National Instruments. (2023, March 30). *The PID Controller & Theory Explained*. Retrieved from ni: https://www.ni.com/nl-nl/shop/labview/pid-theory-explained.html

Nian, R. (2018, June 17). *Controls: Comparison between 4 popular control strategies*. Retrieved from Medium: https://medium.com/@ruinian/controls-comparison-between-4-popular-control-strategies-d1f3a3b61eb1

Veneri, G., & Capasso, A. (2018). *Hands-On Industrial Internet of Things.* Packt.

Table of figures

Table of tables

Appendix

Appendix A Use case descriptions

Name	**Display Sensor Data**
Actors	End user
Pre-conditions	- There is at least one sensor connected to the selected vertical farm. - The user is logged into the dashboard. - The user has the appropriate API tokens set up
Post-conditions	- The user can view sensor data of the connected sensors with means of filtering on date and time
Main Success Path	- The user navigates to the page containing sensor data. - The system retrieves and displays the sensor data on the page
Alternative path	None

Table 21: Use case description, display sensor data

Name	**Actuator Switching**
Actors	End user, system
Pre-conditions	- The actuators are installed and properly connected to the system. - The user is logged into the dashboard. - The user has the appropriate API tokens set up
Post-conditions	- The actuator is switched on or off according to the user's choice
Main Success Path	- The user navigates to the page containing the actuators. - The system displays an overview of available actuators. - The user selects the desired actuator and switches it on or off with a single click
Alternative path	None

Table 22: Use case description, actuator switching

Name	**Provision New Vertical Farm**
Actors	End user
Pre-conditions	- The new Raspberry Pi is configured correctly (Mycodo installed and configured, permissions/tokens generated, database configured) - The user is logged into the dashboard
Post-conditions	- The dashboard is able to interact with the vertical farm - The network status of the vertical farm is continuously updated and displayed accordingly on the dashboard
Main Success Path	- The user navigates to the page containing the "provision new vertical farm" flow. - The system displays a form for the user to configure the vertical farm.

	- The user fills out the form and submits it to the system. - The system displays the network connectivity for the new vertical farm
Alternative path	None

Table 23: Use case description, provision new vertical farm

Name	**Add automation rule**
Actors	End user
Pre-conditions	- The user is logged into the dashboard. - The vertical farm is setup and correctly configured
Post-conditions	- The vertical farm has switched the state of an actuator for the specified amount of time, at the specified datetime
Main Success Path	- The user navigates to the page containing automations. - The user navigates to the "create automation" flow page. - The user specifies the actuator to add an automation to - The user specifies a datetime(start-end) with a state for the actuator. - The user submits the new automation. - The system registers the 'fixed schedule' automation
Alternative path	None

Table 24: Use case description, add automation rule

Name	**Remove automation rule**
Actors	End user
Pre-conditions	- The user is logged into the dashboard. - The vertical farm is setup and correctly configured. - There is an automation rule to delete
Post-conditions	- The automation rule is removed from the schedule of the vertical farm
Main Success Path	- The user navigates to the page containing automations. - The user specifies the automation to remove. - The system displays a confirmation dialog. - The user confirms the prompt. - The system removes the specified automation from the schedule
Alternative path	- The user can also cancel the prompt

Table 25: Use case description, remove automation rule

Name	**Display automation rules**
Actors	End user
Pre-conditions	- The user is logged into the dashboard. - There is at least one automation to display. - The vertical farm is setup and correctly configured.
Post-conditions	- The user can view all the scheduled automations that are currently in effect in their system.
Main Success Path	- The user navigates to the page containing automations. - The system displays an overview of automations
Alternative path	None

Table 26: Use case description, display automation rules

Name	**Execute automation**
Actors	System
Pre-conditions	- There is at least one automation to execute.
Post-conditions	- The automation has been executed at the specified datetime
Main Success Path	- The scheduler polls the automations periodically - The scheduler executes the automation with specified configuration and parameters
Alternative path	None

Table 27: Use case description, executing automations

Appendix B Development of the physical Farm

Since the primary focus of the graduation project isn't on the hardware, this section refers to photos from Appendix X. Based on these photos, a summary of the development process will be described. This is also why considerations like power distribution to the system won't be discussed.

The development of the hardware began with the assembly of the necessary components: the Raspberry Pi, Atlas-scientific electronics, GPIO expansion board, and the 8-channel relay board, as shown in Figure 22. Once these components were prepared, the next step was to connect the actuators to the relay board.

The choice to use a relay board was made for two primary reasons. First, a relay board is essential when controlling high-voltage devices, such as LED matrices or water pumps, through a controller like the Raspberry Pi. Second, relays are driven by the GPIO pins of the Raspberry Pi and are compatible with Mycodo.

Figure 22: physical prototype

The system is neatly contained in an old closet that wasn't being used at the office, and a separate wooden container has been modified to fit the Raspberry Pi with most of its cables packed away.

Appendix C Control strategies

This section will delve into potential control strategies that can be employed for the controller components within the vertical farming system. These control algorithms can be adapted and integrated into various configurations, ranging from basic scripts to complex multi-component applications and even in low level electronic circuits.

It is important to note that specific implementation details are not covered in this section, as the focus is on understanding the core theory behind the control strategies.

I. Open-Loop Control

An open-loop control strategy is a type of control system in which the output is determined solely by the input, without considering any feedback from the system. In this approach, the controller calculates the control action based on a predefined relationship between the input and the desired output, but it does not take into account the actual output or any external disturbances.

Pros	**Cons**
1. Simplicity 2. Easy to implement 3. Low computational cost	1. No compensation for external disturbances or system changes 2. No feedback 3. Terrible control performance
Example	
Consider driving a car with a speed limit of 50 km/h. While driving, your speedometer malfunctions, leaving you without any feedback on your actual speed. As you approach a steep hill, you instinctively press the gas pedal 20% harder to maintain the speed limit, based on your previous driving experience. In this scenario, the control strategy is pressing the gas pedal 20% harder. However, due to the absence of feedback from the broken speedometer, you have no way to monitor and adjust your performance to ensure you are maintaining the desired speed of 50 km/h. This situation illustrates an open-loop control strategy, where the output (speed) is determined solely by the input (pressing the gas pedal) without considering any feedback from the system. **(Nian, 2018)**	

Table 28: open loop control pros and cons

II. Proportional, Integral, Derivative (PID) Control

Proportional, integral, derivative (PID) control is by far the most popular controller in industry today. It is extremely robust, easily implemented and intuitive. PID controllers are used for error rejection.

The basic idea behind a PID controller is to read a sensor, then compute the desired actuator output by calculating proportional, integral, and derivative responses and summing those three components to compute the output (National Instruments, 2023)

The PID controller achieves this through three components: proportional, integral, and derivative actions. These actions work in tandem to reduce the error, improve the system's response, and minimize overshoot or oscillations.

There are three components to PID controllers:

1. *Proportional Control*

Proportional control aims to minimize the current error. The controller gain, Kp, is the hyperparameter in this case. Increasing Kp results in larger control actions being taken, while reducing it leads to smaller control actions.

2. *Integral Control*

Integral control is designed to minimize past errors. The integral gain, Ki, is the hyperparameter for this component. Integral gain effectively eliminates offset in the proportional gain. For instance, if the current error is 10, the proportional gain would make a large change in input to resolve this error. However, this input might be excessive. Integral gain is employed to mitigate this effect.

3. *Derivative Control*

Derivative control focuses on minimizing future errors. The derivative gain, Kd, is the hyperparameter in this situation. This control examines the rate of change in the error and adjusts accordingly. In industrial settings, proportional-integral controllers are often very effective, making the derivative control component unnecessary in some cases.

Pros	Cons
1. Fast performance 2. Minimal overshoot of setpoint 3. Easy to implement 4. Corrects disturbances through feedback	4. Does not optimize the process 5. Fine-tuning might be required (often) 6. Fine-tuning might be cumbersome 7. One controller applies to one control action
Example	
Consider driving a car at a speed of 50 km/h. You have a reliable speedometer and come across the same steep hill. Initially, you press the gas pedal 20% harder to maintain your speed. However, the speedometer reads 65 km/h, so you ease off the gas pedal to pressing only 10% harder. Now you are traveling at 45 km/h. Then, you decide to press a little harder and finally reach the desired speed of 50 km/h. Your action of pressing the gas pedal is the control action, and this time, you have feedback from the speedometer. This feedback allows you to adjust your speed quickly until the desired speed is achieved. In terms of PID control, the initial action of pressing the gas pedal 20% harder is the proportional control, as it attempts to offset the difference between your current speed and the desired speed limit. The integral control comes into play when you adjust your pedal pressure after realizing that pressing 20% harder again would result in going 65 km/h. By remembering the previous outcome, the integral control helps stabilize your control action. Finally, the small adjustments made to reach the desired speed of 50 km/h can be attributed to the derivative control, which considers the rate of change in the error (speed difference) and adjusts accordingly. **(Nian, 2018)**	

Table 29: PID pros and cons

The diagram below illustrates the basic flow of a PID controller:

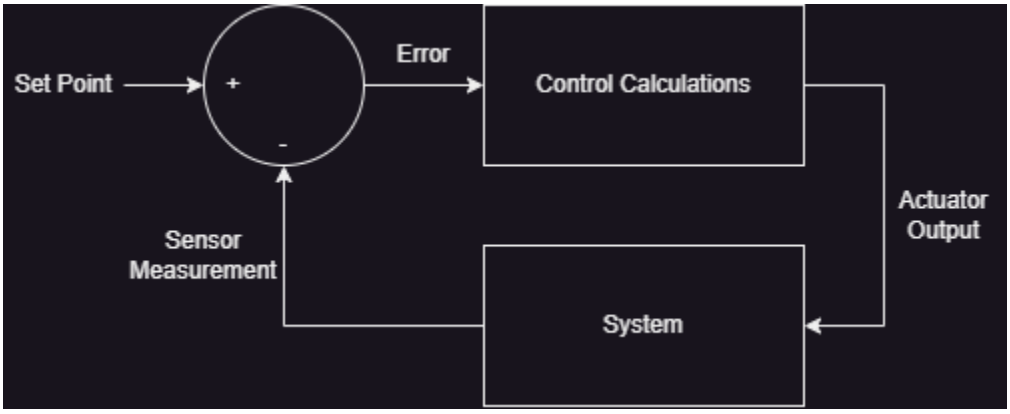

Figure 23: Basic illustration of a PID feedback loop

III. Model Predictive Control (MPC)

Model predictive control (MPC), also referred to as moving horizon control or receding horizon control, is one of the most successful and most popular advanced control methods. The basic idea of MPC is to predict the future behavior of the controlled system over a finite time horizon and compute an optimal control input that, while ensuring satisfaction of given system constraints, minimizes an a priori defined cost functional. (Allgower, n.d.)

The main idea behind MPC is to make proactive adjustments to the system based on predictions, instead of merely reacting to errors or disturbances.

The MPC strategy consists of the following steps:

1. Formulate a model: A mathematical model of the system is developed, capturing its dynamics and interactions between different variables. This model can be based on first principles, empirical data, or a combination of both.
2. Define the prediction horizon: The prediction horizon is the time window over which future behavior of the system is predicted. It plays a crucial role in the performance of the controller, as it determines the number of future time steps that will be considered when making control decisions.
3. Define the control horizon: The control horizon is the time window over which control actions are optimized. It is usually shorter than the prediction horizon, as it is more computationally efficient to optimize control actions over a smaller time window.
4. Optimize control actions: At each time step, the controller uses the system model to predict future behavior over the prediction horizon. It then optimizes control actions over the control horizon to minimize a predefined cost function, subject to constraints on inputs and outputs. This cost function typically represents a trade-off between system performance and control effort.
5. Implement the first control action: The first optimized control action is applied to the system, and the process is repeated at the next time step, using updated information about the system state.

Model predictive control (MPC) is a widely used advanced control strategy in the industry, and its popularity can be attributed to three main factors:

- MPC provides the best control performance when the model used for implementation accurately represents the system dynamics.
- Users can readily comprehend why the controller recommends specific actions by examining the underlying models. This transparency offers a significant advantage over AI-powered control methods, which can sometimes be perceived as "black boxes."
- MPC does not need to directly change the control actions (even though it can), rather, it can recommend the best operating strategies to the operators of the plant to maximize the profit. This allows MPC to be used in conjunction with PID controllers. This concept is called *supervisory* control.

Pros	Cons
1. Fine-grained control 2. Understandability/transparency 3. Allows for high levels of optimization 4. Provides the ability to handle constraints	1. Complexity in implementation/maintenance 2. Necessity for highly accurate process models 3. Performance depends on the quality of the process models 4. Might require periodic updating
Example (direct control)	
While driving down the street at 50 km/h, this time you have a map of the road with incline information and the manufacturer's documentation, which provides models for adjusting the gas pedal pressure based on different inclines to maintain specific speeds. For instance, if the hill's incline is 25 degrees, pressing the gas pedal by 30% will keep the speed at 50 km/h. In this scenario, the speed limit serves as the constraint. By utilizing the road incline data from the map and the speed model from the manufacturer, the MPC controller can efficiently plan the gas pedal press sequence, ensuring that the car consistently travels at exactly 50 km/h.	
Example (supervisory control)	
Imagine the same scenario as before, but this time, the MPC controller acts like a driving instructor, while the driver represents a PID controller. The MPC has no direct contact with the car but optimizes its performance by recommending ideal driving techniques every minute or so. Over time, the car's condition will deteriorate, causing the manufacturer's speed equations to become inaccurate, which will negatively affect the controller's performance. Consequently, updating the models will be necessary to regain optimal control of the car. **(Nian, 2018)**	

Table 30: MPC pros and cons

Figure 24 below illustrates the basic flow of an MPC controller:

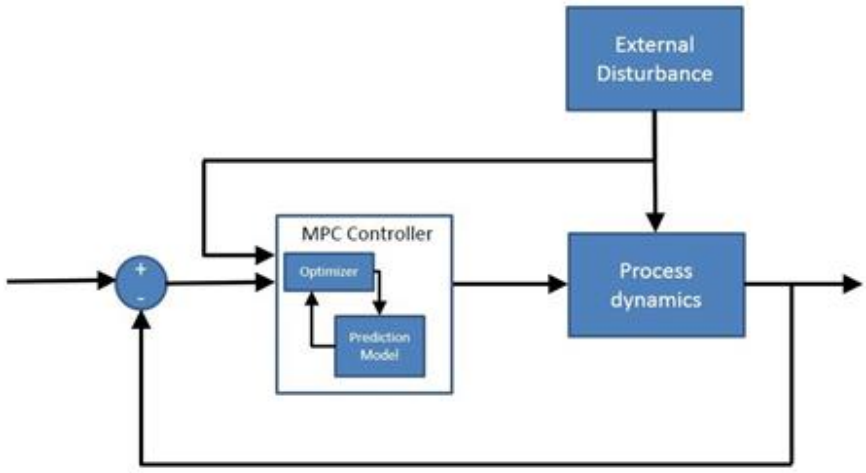

Figure 24: MPC feedback loop

Appendix D Industrial Internet-of-Things OSS architecture

This architecture is based on principles from the domain of Industrial Internet-of-Things (IoT). Industrial IoT refers to the application of IoT technologies in industrial settings, such as manufacturing plants, supply chain operations, and energy management systems. It involves the use of interconnected sensors, actuators, and devices that collect, process, and transmit data to optimize industrial processes, improve efficiency, and reduce costs.

While researching existing systems for "large-scale" operations, the term was introduced into the context of the project through a book called 'Hands-on Industrial Internet-of-Things'. The most important takeaway from the book was the proposal for an industrial IoT platform using open-source technologies:

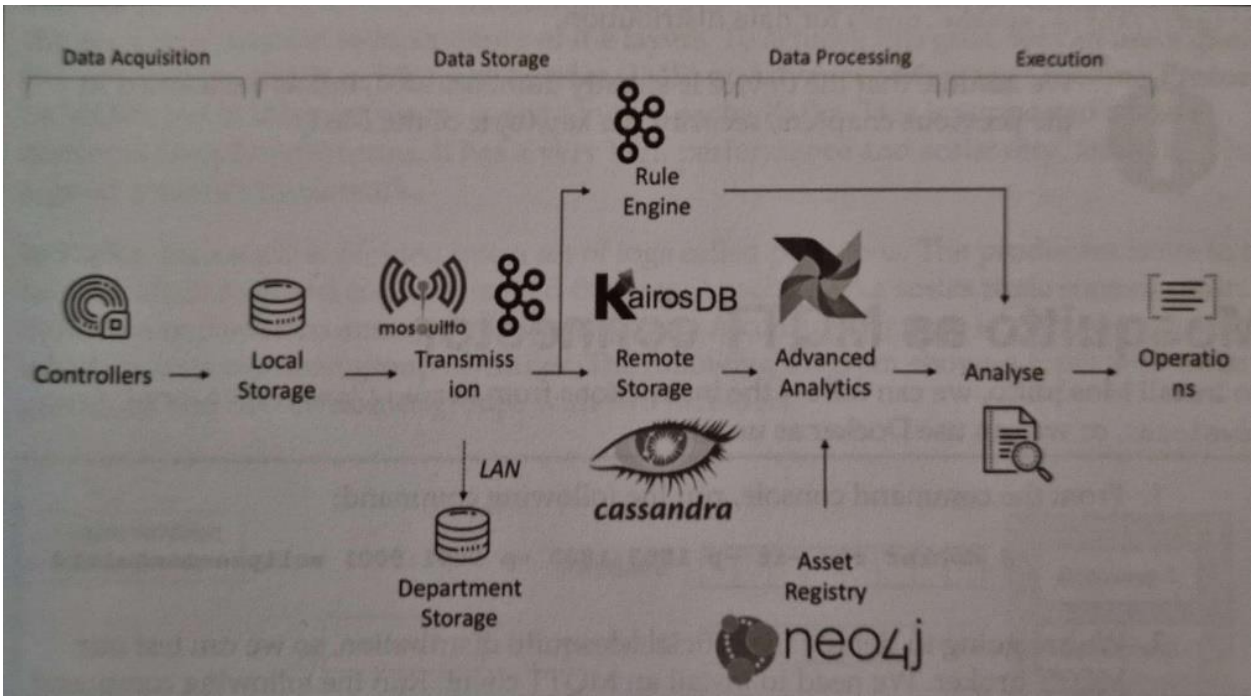

Figure 25: Industrial IoT based architecture for implementing an automation rules engine

Appendix E User stories

Below are the user stories concluded from the functional requirements

Title	Priority	Points
Provision Vertical Farm	Must	5
User story: **As a user, I want to easily provision and connect a new Raspberry Pi powered vertical farm to the system, so I can start managing the vertical farm through the admin dashboard.**		

Title	Priority	Points
Sensor data visualization	Must	3
User story: **As a user, I want to view sensor data, such as temperature, humidity, light, pH, Electroconductivity and CO2 of my connected vertical farms, so I can monitor their environmental conditions and make informed decisions.**		

Title	Priority	Points
Actuator control	Must	5
User story: **As a user, I want to easily control actuators of my vertical farms, such as turning on/off lights or water pumps, through the admin dashboard, so I can optimize the growing environment for my crops.**		

Title	Priority	Points
Automation Rules	Must	8
User story: **As a user, I want to create and manage automation rules with a fixed schedule for various tasks, such as watering or lighting, so I can maintain optimal growing conditions for my crops without manual intervention.**		

Title	Priority	Points

Receive Alerts	Should	4
User story: **As a user, I want to receive alerts when certain conditions or events occur, such as dangerous temperature levels or sensor measurements in extreme highs/lows, so I can take appropriate actions to address the issue and maintain the proper growing environment for the crops.**		

Title	**Priority**	**Points**
Automatic Self-Correction	Could	8
User story: **As a user, I want the system to automatically self-correct certain environmental metrics within specified boundaries, so I can ensure optimal growing conditions without constant manual adjustments.**		

Title	**Priority**	**Points**
Server management	Must	2
User story: **As a user, I want the vertical farm to integrate with a server management tool, so I can monitor and manage the Raspberry Pi on the OS-level so that the vertical farm is (semi) manageable by a non-developer.**		

Title	**Priority**	**Points**
Server management API	Could	5
User story: **As a user, I want the server management tool of the vertical farm to be integrated with the admin dashboard, so I can monitor and manage the Raspberry Pi on the OS-level through the admin dashboard so that the vertical farm is (semi) manageable by a non-developer.**		

Appendix F Data monitoring page

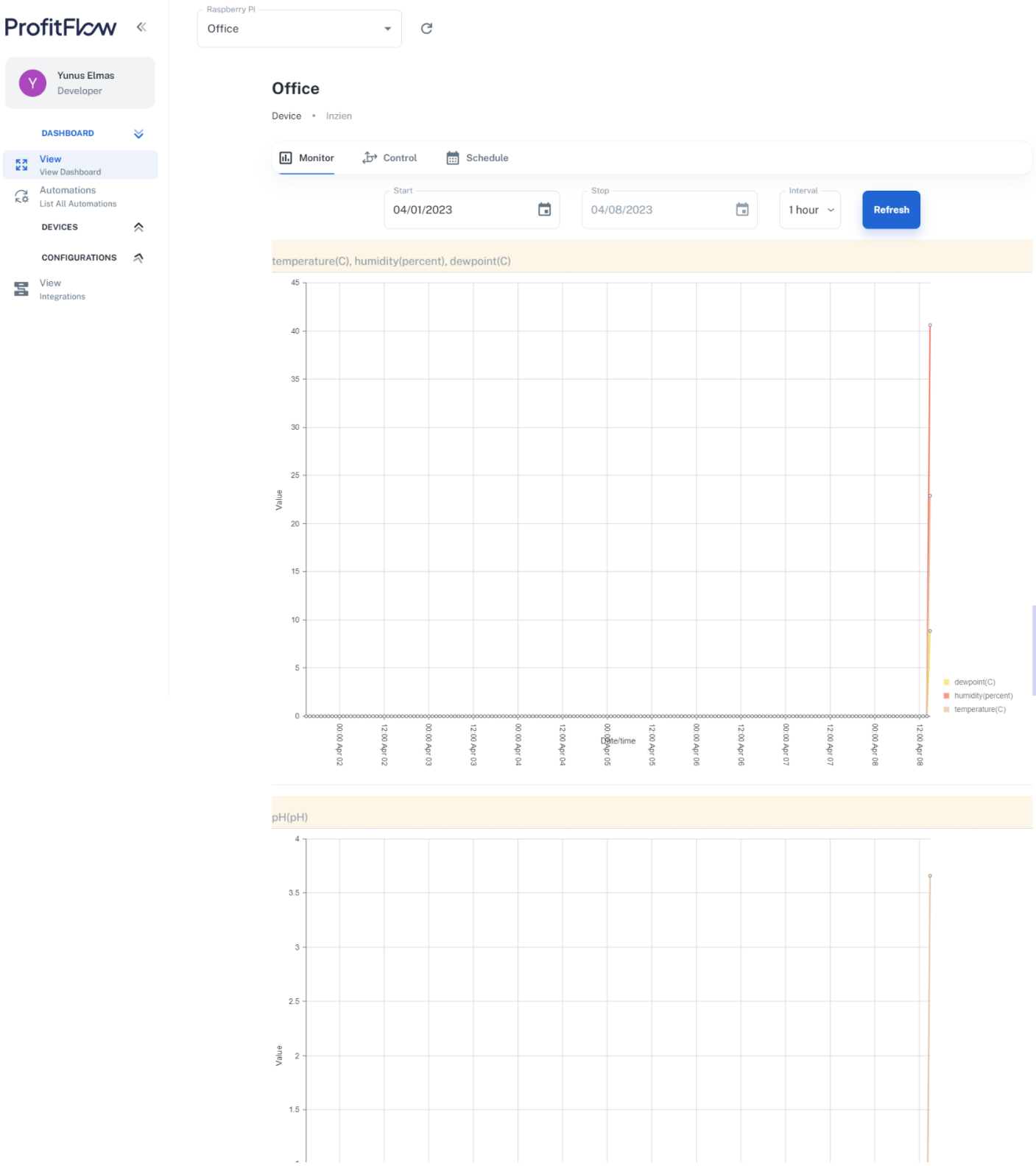

Figure 26: datamonitoring page