

Software Engineering Department

Braude College

Capstone Project Phase B – 61999

24-2-D-6

Artful Accessibility-

Individuals with Mobility Disabilities in Art Exhibition

Mustafa Shama

[mostfashmma@gmail.com](mailto:mostfashmma@gmail.com)

Marysoul Karawany

[marysoul.karawany@e.braude.ac.il](mailto:marysoul.karawany@e.braude.ac.il)

Supervisors:  
Dr. Julia Sheidin



# Table of Contents

Abstract .....	3
1.Introduction.....	3
2. Engineering Process .....	4
2.1 Process Overview .....	4
2.2 Phases of Development .....	5
2.3 System Architecture .....	5
2.4 Solution .....	6
2.4.1 Tools Utilized for the Solution.....	7
2.4.2 System's Technologies .....	8
2.4.3 Formulas .....	9
2.4.3.1 Height adjustment.....	9
2.4.3.2 Optimal viewing distance.....	10
2.4.3.3 Optimal placement of the equipment on wall.....	10
2.4.4 Physical Components and Programming Logic .....	11
2.4.5 Computer Vision for Object Detection.....	13
2.4.6 Backend, Frontend, and Database.....	15
2.4.7 Networking for Communication between IOT devices and Server - MQTT.....	16
2.5 Challenges .....	20
2.6 Testing Plan.....	21
2.7 Conclusion.....	22
3. User Documentation .....	23
3.1 User's guide - Operating instruction.....	23
3.1.1 General Description.....	23
3.1.2 Operating instructions .....	23
3.2 Maintenance guide.....	30
3.2.1 Raspberry Pi 4 Installation Guide.....	30
3.2.2 Steps to Connect Raspberry Pi to HDMI Display and Install VNC Tools.....	31
3.2.3 Setting a Static IP Address on Raspberry Pi (Linux) .....	32
3.2.4 Setting Up Raspberry Pi 4 with Picamera.....	33
3.2.5 Instructions to Make our microcontroller Program Run on Bootup .....	35
3.2.6 Backend and Frontend Installation Guide .....	38
3.2.7 Database Installation Guide.....	39
3.2.8 Roboflow's Machine Learning Model Integration Guide.....	40
3.2.9 MQTT Broker Integration Guide.....	42
3.3 Overview of the System's Process Workflow - Sequence diagram.....	44
3.4 Database description .....	45
4. References.....	46

## Abstract

With a growing focus on accessibility in cultural sites following the UN Convention on the Rights of Persons with Disabilities, museum experiences need to be more inclusive. Innovative technologies are emerging to ensure that all visitors, including those with mobility disabilities, can fully engage with exhibitions. This project introduces a system designed to enhance the experience for individuals with mobility challenges in art exhibitions, using advanced technology to provide an inclusive and seamless experience for all. Our project introduces a system for art exhibitions that adjusts the height of paintings and wall-mounted exhibits to the eye level of visitors with mobility disabilities. Utilizing computer vision algorithms, proximity sensors, and a height-measuring camera, the system detects a visitor's presence and adjusts artwork height in real-time using motors. This approach ensures an inclusive and accessible viewing experience, aligning with ADA standards and enhancing visitor engagement by dynamically adapting to different heights and mobility needs.

## Keywords

Cultural heritage experience, Adaptive design, Museum experience, Human-centered computing, Accessibility, Accessibility systems, Dynamic Adjustment, Machine learning (ML), IOT (Internet of Things).

## 1. Introduction

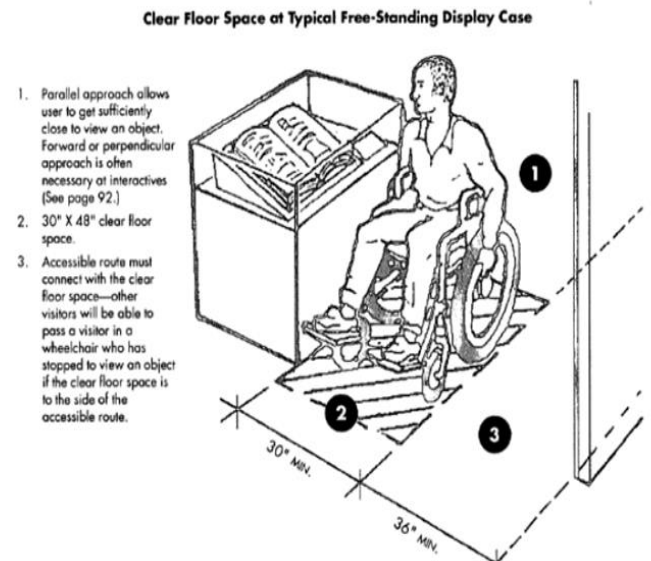
In recent years, cultural and heritage sites, particularly museums, have increasingly acknowledged and accommodated the diverse needs of their audiences. This shift is underscored by a growing recognition of the importance of inclusivity, especially for individuals who may require support to access and fully enjoy these sites as museums transition from mere presenters of objects to providing immersive experiences. Emphasizes that no individual should be excluded from experiencing the benefits of the museum, highlighting the importance of accessibility for all citizens [2]. The significance of inclusivity is further underscored by international agreements, such as the UN Convention on the Rights of Persons with Disabilities, which emphasizes the pivotal role of assistive and digital technologies in the lives of people with disabilities [1]. Consequently, museums strive to be more responsive to the varied needs of their visitors, including those with mobility, hearing, visual, and cognitive disabilities.

Technology has emerged as a vital tool in creating authentic and accessible museum experiences, bridging the gap for individuals who were previously unable to access certain artifacts and sites independently [4]. However, developing technological solutions for individuals with disabilities necessitates a nuanced approach, considering the diverse range of disabilities and specific conditions that may impact how individuals engage with museum activities. Studies have highlighted the challenges faced by individuals with disabilities, such as those who use wheelchairs or have visual impairments, when visiting art museums. These challenges span from physical obstacles encountered outside the exhibition to difficulties experienced within it, including issues with exhibit heights and positioning [2].

Considering these challenges, our current work addresses the specific needs of individuals with mobility disabilities within art exhibitions. While certain accommodations, such as ramps and elevators, provide greater accessibility, positioning exhibit objects remains a crucial aspect that often goes unnoticed. Ensuring that exhibits are displayed at appropriate

heights, allowing wheelchair users and standing individuals to view them comfortably, fosters inclusivity in museum spaces [3] (see Figure 1).

This book explores our efforts to develop innovative solutions to enhance the accessibility of art exhibitions, with the main goal of providing a seamless and enriching experience for all museum visitors, regardless of their mobility limitations. We propose a novel system designed to adjust the height of artwork displays to accommodate visitors' needs, whether seated in a wheelchair or standing. Our system includes a proximity sensor, a camera to detect the visitor's height, and a controller that activates motors to dynamically adjust the display's height. This approach ensures that all visitors have an enriching and equitable museum experience regardless of mobility limitations. By incorporating these adaptive technologies, we provided a seamless and enriching experience for all museum visitors, regardless of their mobility limitations.

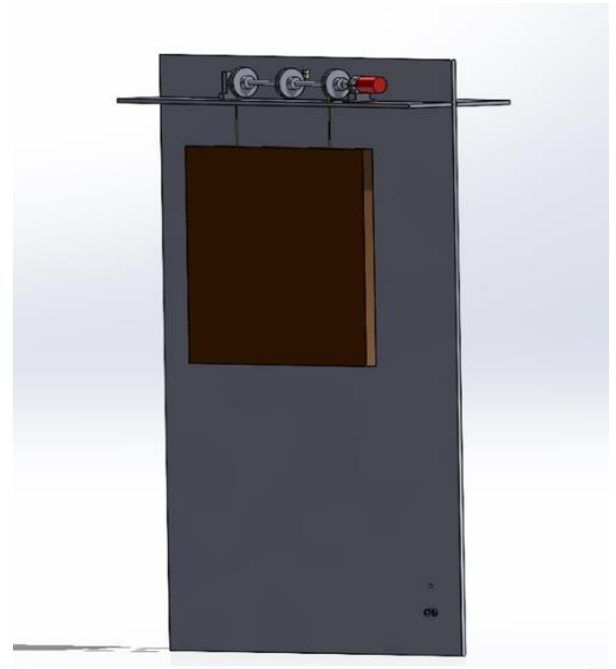


**Figure 1:** The ADA (Americans with Disabilities Act) Standards for positioning Freestanding display cases [5].

## 2. Engineering Process

### 2.1 Process Overview

The engineering process for Phase B builds upon the conceptual framework and design principles established in Phase A. This phase's primary focus was refining the software and hardware system, enabling it to manage multiple paintings within a museum environment while maintaining accessibility for visitors. This project involves collaboration between software engineering and mechanical engineering teams. A group of students from the mechanical engineering department created a prototype to demonstrate the feasibility of the physical adjustment mechanism (see Figure 2). Our software engineering team has developed a system that detects wheelchair users and adjusts a mechanical component to their height. It indicates which image to lower and by how many centimeters, while also including essential maintenance and monitoring features. This phase adhered to the objectives outlined in Phase A, emphasizing scalability, accessibility, and operational efficiency.



**Figure 2:** The prototype.

## 2.2 Phases of Development

The project began with the initiation and requirement refinement phase, which involved a detailed review of the Phase A findings. The focus was on expanding the system to support multiple paintings, and the requirements were refined to ensure compatibility with IoT frameworks and centralized management systems. During this phase, the mechanical and software engineering teams worked independently. The mechanical team concentrated on prototyping the height adjustment mechanism, addressing factors such as artwork weight and stability. Meanwhile, the software team refined the software architecture to handle multi-painting functionality and centralized monitoring through a user interface, while also ensuring seamless integration with hardware components.

The software development and design phase followed, during which the software team implemented essential functionalities to ensure scalability and future integration. This included the development of visitor detection capabilities and the calculation of required height adjustments for paintings based on predefined formulas. These formulas considered details such as the base height, painting height, and optimal viewing range. The team also enabled the dynamic adjustment of multiple paintings based on real-time visitor data and designed a centralized system to monitor and control overall performance in real time.

Finally, the mechanical engineering team focused on prototype testing, where they designed and tested a standalone prototype of the height adjustment mechanism. The prototype, shown in Figure 2, was a critical component for validating the mechanism's functionality and robustness.

## 2.3 System Architecture

The proposed system architecture dynamically adjusts the height of museum paintings to enhance accessibility for wheelchair users. It integrates multiple components, including the frontend, backend server, IoT devices, a database, and a machine learning cloud service, ensuring real-time interaction and scalability (see Figure 3 below). This section explains the key components of the architecture, their roles, and how they interact to achieve the system's goals (see section 3.3 Overview of the System's Process Workflow – Sequence diagram).

The frontend, backend server, IoT devices, database, ML cloud, and MQTT broker are discussed below to provide a comprehensive understanding of the system.

**1. Frontend:** The frontend includes a user login for museum staff authentication, a painting manager for configuring painting dimensions and initial height, and a statistics viewer to monitor visitor engagement. It interacts with the backend via an HTTP REST API for data management and uses a WebSocket connection for real-time updates.

**2. Backend Server:** The backend features an HTTP REST API for communication between the frontend and IoT devices, WebSocket for real-time updates, MQTT logic for data exchange, and multi-device logic for synchronized operations. It processes commands, interacts with the ML cloud for wheelchair detection, and calculates height adjustments.

**3. IoT Devices:** Raspberry Pi devices manage local sensors, detect visitor presence, and log painting details, while M5Stack devices control the mechanical height adjustment. The IoT devices execute real-time commands from the backend and report system status during unexpected events.

**4. Database:** The database stores essential data, including painting configurations, interaction logs, and user credentials. It supports efficient CRUD operations through the HTTP REST API.

**5. ML Cloud:** The ML cloud processes images captured by IoT devices to detect wheelchairs, sending predictions to the backend for height adjustment calculations. Offloading computational tasks to the cloud ensures scalability and quick response times.

**6. MQTT Broker:** The MQTT broker facilitates lightweight, reliable messaging between the backend and IoT devices, enabling efficient bidirectional communication for status updates and command execution.

### System Architecture

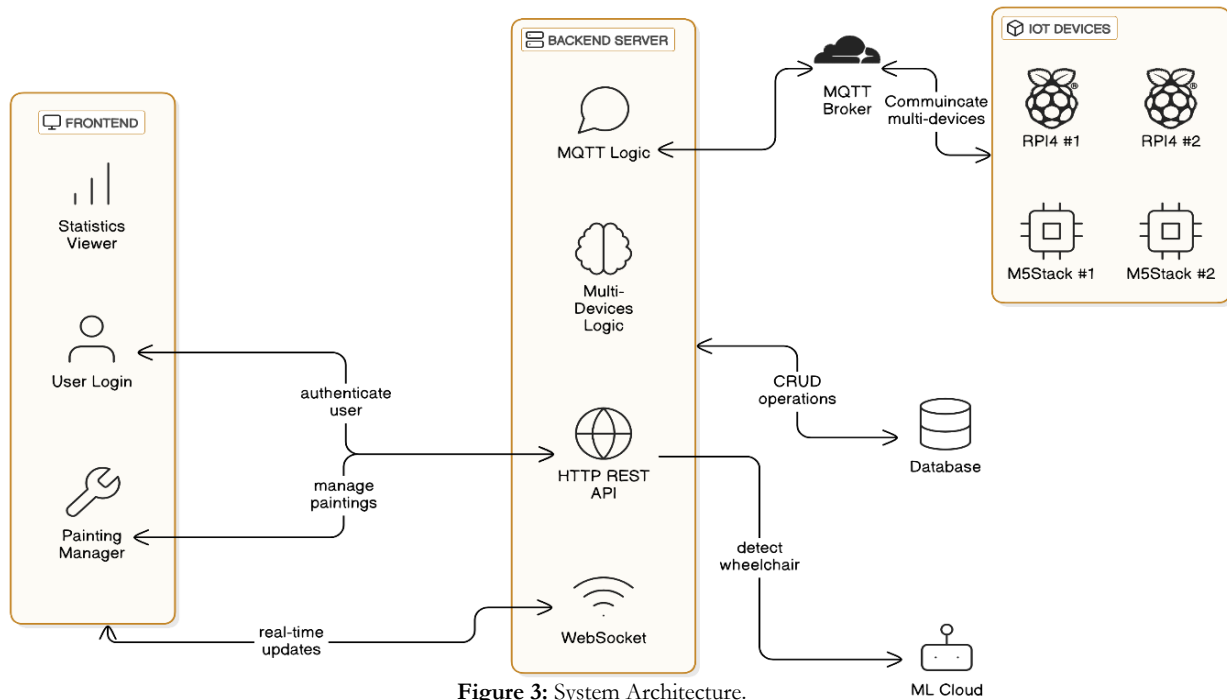


Figure 3: System Architecture.

### Key Features of Architecture

- 1. Scalability:** Modular design supports adding more IoT devices and paintings with minimal adjustments.
- 2. Real-Time Responsiveness:** WebSocket and MQTT protocols enable low-latency communication and dynamic adjustments.
- 3. Fault Tolerance:** Raspberry Pi devices handle power or network interruptions with local logging and Linux OS interruptions.
- 4. User-Friendly Design:** Intuitive frontend interfaces ensure accessibility for non-technical users.
- 5. Cloud Integration:** Computationally intensive tasks, like wheelchair detection, are handled in the cloud to maintain high performance.
- 6. Data-Driven Insights:** Interaction logs provide valuable analytics for optimizing the visitor experience.

## 2.4 Solution

The solution focuses on improving museum accessibility by integrating a dynamic system for adjusting painting heights based on visitor needs, particularly for those in wheelchairs. This section highlights the functionality and real-world application of the system while referencing key calculations that underpin its operations.



The system leverages hardware components, software algorithms, and a centralized control mechanism. A proximity sensor detects visitors, and a camera determines whether a wheelchair user is present. Upon detection, the system calculates the necessary height adjustment for the painting (Formula 1 in section [2.4.3.1](#)). The system also calculates the optimal viewing distance (Formula 2 in section [2.4.3.2](#)), which defines the range for the proximity sensor to trigger detection and camera activation. For accurate distance measurement, these calculated values are then communicated to the mechanical adjustment device for precise implementation. Key innovations include a Maintenance and Monitoring System that tracks real-time interaction data, system health, and connectivity status. The system also incorporates visitor engagement analytics to optimize painting placement and enhance the overall museum experience.

This integrated approach ensures seamless operation across multiple paintings, combining advanced hardware with scalable software to deliver an accessible and engaging environment for all visitors.

### 2.4.1 Tools Utilized for the Solution

In designing our system for dynamically adjusting the height of museum artworks, selecting the appropriate components was essential to ensure functionality, reliability, and efficiency (in section 2.4.4 we explain how we utilized and programmed each component). The system comprises several key components, including the microcontroller, camera, sensors, and supporting peripherals. Each component was carefully evaluated and chosen based on its suitability for the application.

Figure 4 illustrates the system's key physical components:

1. **Microcontroller (A):** We selected the Raspberry Pi 4 Model B (4GB RAM) as the core control unit of the system. With its robust processing power, it manages data processing, sensor inputs, and motor operations seamlessly. The Raspberry Pi 4 also offers extensive connectivity options and compatibility with a wide range of accessories, ensuring reliable system integration. Its ability to support complex applications with efficient energy consumption makes it ideal for our use case. [[Raspberry Pi 4 Model B \(4GB RAM\)](#), [Raspberry Pi 4 Product Brief](#)].
2. **Camera (B):** The Raspberry Pi Camera Module v2 was chosen for its 8MP resolution and seamless integration with the Raspberry Pi microcontroller. It provides high-quality video streams essential for detecting visitors and adjusting the artwork's height accurately, particularly for those using wheelchairs. Its compact design and compatibility with Raspberry Pi systems make it highly efficient for embedded applications requiring precise image capture [[Raspberry Pi Camera Module v2](#) , [Raspberry Pi Camera Module v2 Datasheet](#)].
3. **Sensor (C):** For proximity detection, we used the Ultrasonic Distance Sensor - 5V (I2C). This sensor provides reliable and accurate distance measurement by emitting ultrasonic waves and calculating the time it takes for the echo to return after bouncing off an object. It operates effectively within a range of 2 cm to 4 meters, making it well-

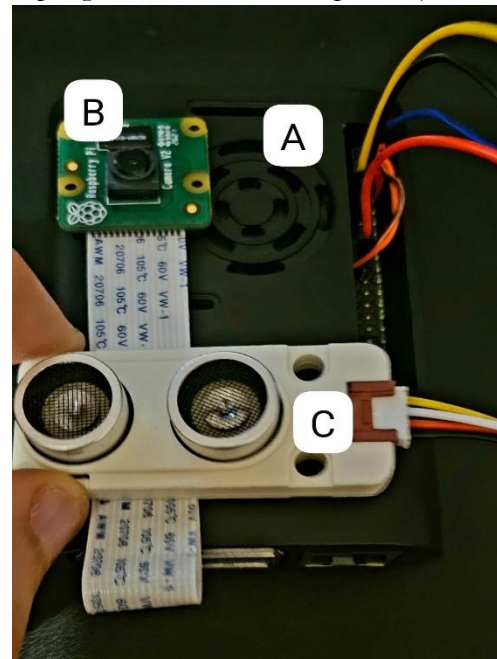


Figure 4: The system's device physical components.

suited for detecting visitors approaching the artwork. The I2C interface simplifies communication with the microcontroller, and the sensor's low power consumption, compact design, and consistent performance under varying environmental conditions ensure reliable operation. These features make it an excellent choice for proximity detection in our system.

4. **Power Supply:** The Official Power Supply for Raspberry Pi 4 (5.1V, 3A, USB-C) was used to provide stable and reliable power to the Raspberry Pi and its connected components. Its certified design ensures the microcontroller operates efficiently without voltage drops or interruptions [[Raspberry Pi 4 Power Supply](#)].
5. **Cover:** To ensure durability and effective thermal management, the Raspberry Pi 4 was housed in a plastic round enclosure with a built-in fan. This design protects the microcontroller while maintaining optimal operating temperatures, ensuring consistent performance even during prolonged operation [ [Raspberry Pi 4 Plastic Round Enclosure with Fan](#) ] .
6. **HDMI Cable:** An HDMI to Micro-HDMI cable (90cm) was included to connect the Raspberry Pi to a display during the setup process. Its length and compatibility ensure easy use and flexibility [[HDMI to Micro-HDMI Cable \(90cm\)](#)].

### 2.4.2 System Technologies

The proposed system, as seen in Figure 3, employs various technologies to integrate software and hardware components seamlessly, ensuring robust and efficient operation. Each technology has been selected to fulfill specific system requirements:

- **Python for Microcontroller Control**

Python serves as the primary language for controlling the Raspberry Pi microcontroller, enabling seamless interfacing with critical system components such as proximity sensors, cameras for real-time image processing, and motors responsible for adjusting painting heights. Python's extensive library ecosystem and strong support for IoT applications make it an ideal choice for managing hardware operations and computational tasks. This ensures precise control and coordination of components, facilitating real-time adjustments to enhance the viewing experience for museum visitors.

- **Roboflow 3.0 for Visitor Detection**

The system utilizes Roboflow 3.0 to train and deploy object detection models, enabling the identification of visitors and wheelchair usage. Roboflow supports the creation of custom datasets and computer vision models, allowing the system to calculate dynamic height adjustments for paintings based on visitor positioning. Its advanced tools for data annotation, model training, and deployment streamline the development process, ensuring robust performance in real-time environments.

- **Networking for Communication**

Networking forms the backbone of the system, facilitating communication between its components. The MQTT protocol (Message Queuing Telemetry Transport) is employed for lightweight, real-time data exchange between the microcontroller, sensors, motors, and backend. MQTT's efficiency and low latency make it particularly well-suited for IoT environments, enabling secure and reliable synchronization of system operations across multiple installations.

- **Node.js for Backend Development**

The backend is developed using Node.js, a JavaScript runtime environment. It handles data processing, communication, and logic management, acting as the central hub for sensor and camera inputs, height adjustment calculations, and data storage. Node.js's asynchronous capabilities ensure responsiveness and efficiency, particularly when managing simultaneous operations for multiple paintings.



- **Frontend Interface for Monitoring and Control**

The frontend is implemented using React.js, providing an intuitive and interactive interface for museum staff. This interface enables real-time monitoring of system performance, including visitor interactions and adjustment history. It communicates with the backend via RESTful and WebSocket APIs, allowing seamless data retrieval and updates.

- **Database for Data Storage**

MongoDB, a NoSQL database, is used to store visitor data, system performance logs, and painting adjustment records. Its scalability and flexibility make it ideal for managing dynamic data and supporting system expansion. The database structure is optimized for efficient storage and querying, facilitating analysis and improving system performance.

- **Git for Version Control**

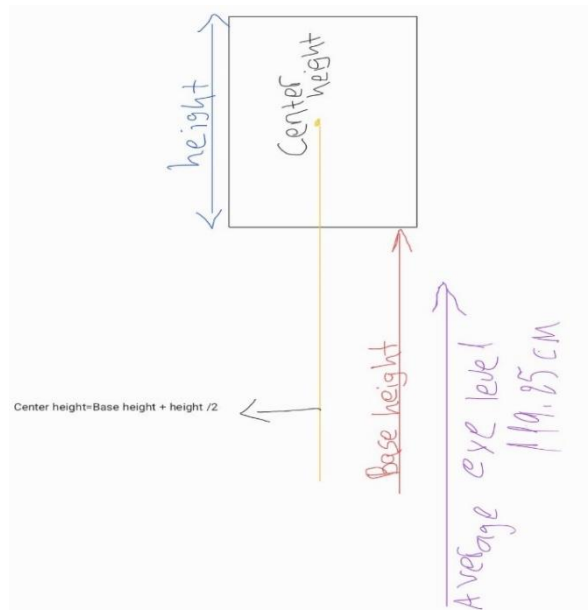
Git is employed to manage the project's version control, enabling collaborative development and efficient tracking of changes across the codebase. Separate repositories are maintained for the backend, frontend, and Raspberry Pi scripts, ensuring specialized workflows for each component. Platforms like GitHub provide centralized repositories for storing, reviewing, and merging code. Git's branching and merging capabilities streamline collaboration, allowing team members to work on features or fixes simultaneously without conflict. Additionally, Git provides a historical record of changes, aiding in troubleshooting and ensuring transparency.

### 2.4.3 Formulas

The system uses precise formulas to calculate the necessary height adjustments for paintings and the optimal viewing distance to determine when a visitor is within range to trigger the system. These calculations ensure accessibility and inclusiveness for all visitors, particularly those in wheelchairs.

#### 2.4.3.1 Height Adjustment

The height adjustment ensures that the center of the painting aligns with the visitor's average eye level, providing optimal visibility and accessibility. These formulas were developed based on accessibility design standards and measurements for wheelchair users to ensure a consistent and inclusive viewing experience. The accompanying diagram (see Figure 4) helps visualize the relationships between Base Height, Painting Height, and the calculated Center of the Painting [[Cornell University Ergonomics Website](#)].



**Figure 5:** Diagram illustrating the components of the formula.

$$\text{Adjustment Needed} = \text{Average Eye Level} - \text{Center Height}$$

$$\text{Center height} = \text{Base height} + \frac{\text{Painting height}}{2}$$

**Formula 1:** Adjustment Needed.

Following Formula 1:

If Adjustment Needed > 0, the painting is too high and must be lowered.

If Adjustment Needed < 0, the painting is too low and must be raised.

Key Components of the Formula (as shown in Figure 4):

- Base Height: The height from the floor to the bottom edge of the painting.
- Painting Height: The vertical size of the painting, measured from the bottom to the top edge.
- Average Eye Level: For wheelchair users, the average eye level ranges from 109 cm to 129.5 cm, with an average value of 119.25 cm [9].
- Center of the Painting: The calculated center height of the painting.

#### 2.4.3.2 Optimal Viewing Distance

The Optimal Viewing Distance defines the range within which the proximity sensor detects a visitor and triggers the system. If the visitor is within this calculated distance, the system activates the camera and begins detecting. This formula, commonly used in TV viewing distance calculations, ensures the visitor is positioned comfortably for an optimal visual experience by maintaining an appropriate distance for viewing the painting without strain or distortion.

$$\text{Optimal Viewing Distance} = \text{Diagonal} * 1.5$$

$$\text{Diagonal} = \sqrt{\text{width}^2 + \text{height}^2}$$

Formula 2: Optimal Viewing Distance.

Key Components of the Formula:

1. Width: The horizontal dimension of the painting.
2. Height: The vertical dimension of the painting.
3. Diagonal Size: Represents the hypotenuse of the painting's dimensions, calculated using its width and height.
4. 1.5 Multiplier: This multiplier, commonly used in TV viewing standards, ensures the visitor is positioned at a distance that allows for optimal visibility and minimal strain.

#### 2.4.3.3 Optimal Placement of the Equipment on Wall

The optimal placement of the Raspberry Pi camera on the wall is critical to ensure accurate detection of wheelchair users. Based on the camera's specifications and field of view, the camera should be mounted at a height of **50-70 cm** from the floor. This placement ensures that the camera can cover the necessary vertical range, from the floor level (0 cm) to approximately **90 cm**, which corresponds to the height of a wheelchair's arms.

The Raspberry Pi camera has a vertical field of view (FOV) of **48.8 degrees**. As shown in the diagram, the camera is positioned at point **B**, with its field of view extending to cover points **A** and **C**, which represent the lower and upper boundaries of the wheelchair's dimensions. This configuration ensures:

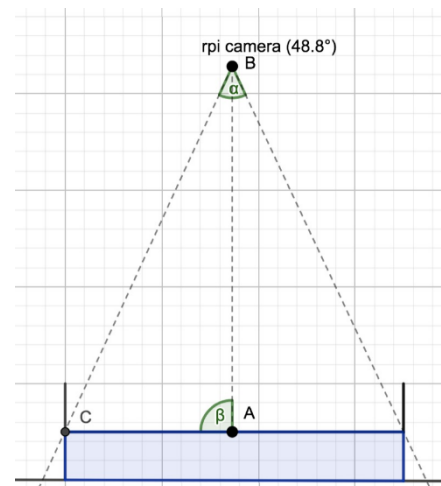


Figure 6: Camera illustration of FOV.

1. **Comprehensive Detection:** The camera can capture the entire vertical profile of the wheelchair, including the bottom wheels and the arms.
2. **Minimized Occlusion:** The placement minimizes blind spots and ensures clear visibility of critical points for accurate detection.

#### **Placement Guidelines**

- **Height:** Mount the camera at 50-70 cm to align its field of view with the target vertical range (0-90 cm).
- **Angle:** Position the camera at a downward tilt to center its FOV on the detection zone.
- **Distance:** Ensure the camera is installed at an optimal distance from the painting or monitored area to maximize coverage while maintaining focus.

By adhering to these guidelines, the camera placement maximizes detection accuracy and enhances the system's reliability in real-world scenarios.

### **2.4.4 Physical Components and Programming Logic**

Figure 4 (in section 2.4.1) illustrates the system's key physical components, including the Raspberry Pi 4B (**A**), the Raspberry Pi Camera Module v2 (**B**), and the Ultrasonic Distance Sensor (**C**). These components work in unison to enable the system's functionality. The Raspberry Pi serves as the central control unit for each painting, and the camera captures visual data, and the ultrasonic sensor measures distances to detect the presence of a person. This section details the functionality of each component and the programming logic behind their operations.

#### • **(A) Raspberry Pi 4B: The Central IoT Device**

The Raspberry Pi 4B is the primary control unit for each separate painting in the system. We've decided to choose the Raspberry Pi because of the level of its flexibility to adapt to our system. In addition to the convenience of integrating hardware components without deep knowledge of electronics.

Beyond managing sensor and camera operations, it handles essential communication and operational tasks to ensure seamless integration and functionality:

#### **1. MQTT Communication with the Backend Server**

The Raspberry Pi manages all MQTT communication with the backend server, facilitating real-time data exchange. It subscribes to relevant topics to receive commands and publishes sensor data, camera frames, or operational updates to keep the backend informed about system status.

#### **2. Program Startup on Boot**

To minimize manual intervention, the Raspberry Pi is configured to automatically start the program upon power-up. Leveraging the Linux OS kernel, this ensures consistent operation in the museum environment without requiring manual setup after every restart.

#### **3. Linux OS Interrupts for Unexpected Shutdowns**

In cases of unexpected shutdowns, such as power loss or Wi-Fi disconnection, the Raspberry Pi uses Linux OS interrupts to send a final MQTT message to a designated topic. This proactive message informs the backend server of the incident, enabling immediate troubleshooting and ensuring system reliability.

#### **4. Frontend-Controlled Shutdown and Restart**

The Raspberry Pi supports remote shutdown and restart commands triggered via frontend controls. Using Linux OS commands, this feature allows museum staff to operate the system without requiring programming knowledge, enhancing accessibility and simplifying overall system management.

#### **5. Painting Log for Identification**

Each Raspberry Pi maintains a local log of painting details, including the system ID (sys\_id), painting name, height, and width. This log allows the backend server to associate the device with its respective painting, enabling seamless identification and operation.

By incorporating these features, the Raspberry Pi ensures autonomous, reliable, and user-friendly operation within the system. Its robust design contributes significantly to the system's overall efficiency and effectiveness, making it an integral part of the architecture.

- **(C) Ultrasonic Sensor:** We've decided to choose the ultrasonic sensor due to wide field view (up to 60 degree), compared to TOF sensor (Time of flight), which has very narrow wide field view, and in addition to the fact that it can't detect black objects due that it absorbs light, and the TOF sensor is based on laser light.

We used a rolling buffer to ensure accurate detection by smoothing sensor readings.

This algorithm keeps the last five distance

measurements and calculates their median to determine whether a person is within range. See Figure 7 illustrates an example where the average of the first five readings is calculated, demonstrating the smoothing process and how the algorithm confirms a person's presence within the range. The algorithm counts at least four consecutive in-range readings to confirm detection before recognizing a person's presence. If the person stays within range during this time, their presence is logged and communicated for further actions. If they move out of range, the system resets after detecting consecutive out-of-range readings. This approach minimizes false triggers and ensures consistent performance.

To calculate distances accurately, the system uses a secondary algorithm. This algorithm calculates the distance based on the sensor's pulse time and adjusts for environmental factors like air temperature and calibration offsets. Formula (3) is used here to adjust the speed of sound according to the current air temperature, ensuring precise calculations. The speed of sound is adjusted according to the temperature to ensure precise calculations, with the Formula (3):

$$\text{Speed of Sound (m/s)} = 331.3 + (0.606 * \text{Temperature}(C^{\circ}))$$

Formula 3: Speed of Sound.

The pulse time from the sensor is used to calculate the raw distance by factoring in the time it takes for sound waves to travel to an object and back. The raw distance is then calibrated using a predefined offset to account for sensor-specific variations. This ensures the sensor consistently provides reliable and accurate measurements for proximity detection. This dual-algorithm approach enables the system to achieve both reliable detection and precise distance calculation, ensuring smooth and efficient operation.

- **(B) Camera:** We used parallel processing to handle image capture and analysis across multiple cameras simultaneously. This approach ensures efficient use of system resources and reduces latency, making it possible to process frames from different cameras concurrently. Each camera operates independently, capturing frames in real-time and analyzing them for specific patterns, such as wheelchair detection.

```
>>> %Run test.py
31.529194999999998
31.082749
30.172686000000002
37.041086
133.387567
133.421909
133.387567
133.387567
133.421909
Person entered range. Distance ~133.387567 cm.
133.696645
133.421909
133.748158000000002
Person confirmed in range (~133.421909 cm). Published to MQTT.
133.387567
Person still in range (~133.421909 cm). No additional wait.
```

Figure 7: Example of calculation.

The backend system uses asynchronous operations to capture frames and send them to a detection service for analysis hosted on cloud. Using MQTT, cameras request frames, and the system waits for responses with timeouts to ensure reliability. Captured frames are processed using machine learning APIs, which analyze the images to detect features like wheelchairs. Detected objects are logged, and actions are triggered accordingly, such as adjusting the artwork's height.

This design leverages the scalability of parallel processing to handle multiple input streams without compromising performance. By running detection tasks concurrently, the system ensures real-time responsiveness, even in environments with high visitor activity, enhancing accessibility and user experience.

### 2.4.5 Computer Vision for Object Detection

The system leverages Roboflow 3.0 Object Detection[[Roboflow](#)], a powerful cloud-based platform, as the backbone for its detection capabilities. Roboflow provides scalable and reliable real-time analysis, enabling the system to handle multiple paintings simultaneously while maintaining high detection accuracy. The platform's active learning mechanism ensures the model remains continuously updated, improving its performance under varying environmental conditions such as lighting changes or visitor behavior and making it highly adaptable to real-world scenarios. The computer vision system is critical for ensuring accessibility within the museum environment. By detecting visitors in real-time and dynamically adjusting the artwork height, the system significantly enhances the viewing experience for wheelchair users and fosters inclusivity for all visitors

- **Detection Workflow and Integration**

When a visitor approaches one of the paintings, the system activates the camera for a specific painting, capturing real-time scene images. These images are immediately sent to Roboflow's cloud-based system for processing. Roboflow analyzes the image using a pre-trained model optimized for object detection, identifying items within the frame, such as individuals and wheelchairs. The detection system relies on confidence thresholds, and if a wheelchair is detected with a confidence level exceeding 80%, the backend triggers the height adjustment mechanism. The detection process is designed for low latency, ensuring real-time responsiveness. The workflow begins when the camera captures a frame encoded and transmitted to Roboflow's API for object detection. The predictions returned by the API are processed locally on the system to determine if height adjustments are needed. If a wheelchair is confirmed, the system calculates the required adjustment using predefined formulas and sends commands to the mechanical components. This end-to-end integration enables immediate and smooth responses, ensuring that visitors experience minimal delays as they engage with the artwork. The system also includes a feedback loop to improve the detection model. Captured images and edge cases are periodically reviewed and added to the training dataset. This allows Roboflow's active learning mechanism to refine the model, adapting to new scenarios and improving its accuracy over time. Such iterative improvement is essential for maintaining reliable detection in diverse museum settings.

- **Accessibility and Scalability**

Roboflow-based detection system enables the museum to scale the solution across multiple paintings without compromising performance. By leveraging cloud-based capabilities, the system ensures that resource-intensive image analysis tasks are handled remotely, reducing the computational burden on local hardware while maintaining high-speed processing. Active learning further enhances scalability by continuously adapting the detection model to new data, allowing the system to function reliably even in complex and high-traffic scenarios. This system's design directly contributes to creating a more inclusive museum environment. Automatically adjusting painting heights based on visitor needs, particularly

for wheelchair users, ensures everyone can comfortably engage with the exhibits. The system's flexibility allows it to be expanded to accommodate various visitor demographics and integrate additional features in the future, such as detecting groups or tracking visitor engagement metrics in real-time. This scalable, accessible approach ensures that the museum can enhance the experience of all visitors while promoting inclusivity and equal access.

- **Integration with Backend and Hardware**

The computer vision component is seamlessly integrated with the system's backend and hardware infrastructure. The backend server facilitates communication between the camera and Roboflow's API, ensuring that images are processed efficiently and the results are acted upon in real-time. The mechanical system responds to height adjustment commands derived from detection outputs, enabling synchronized operations. Integration ensures a unified system where software and hardware work together to provide a seamless visitor experience.

- **Dataset**

The model is built and trained on a dataset comprising over 8,000 images of wheelchairs and people in wheelchairs. This comprehensive dataset ensures robust performance under diverse scenarios, including variations in lighting, angles, and wheelchair designs. The dataset's diversity enables the model to generalize effectively to real-world conditions, improving its detection accuracy and reliability. By leveraging a large, well-curated dataset, the system can handle dynamic environments while maintaining inclusivity and responsiveness. [\[Link to Dataset\]](#)

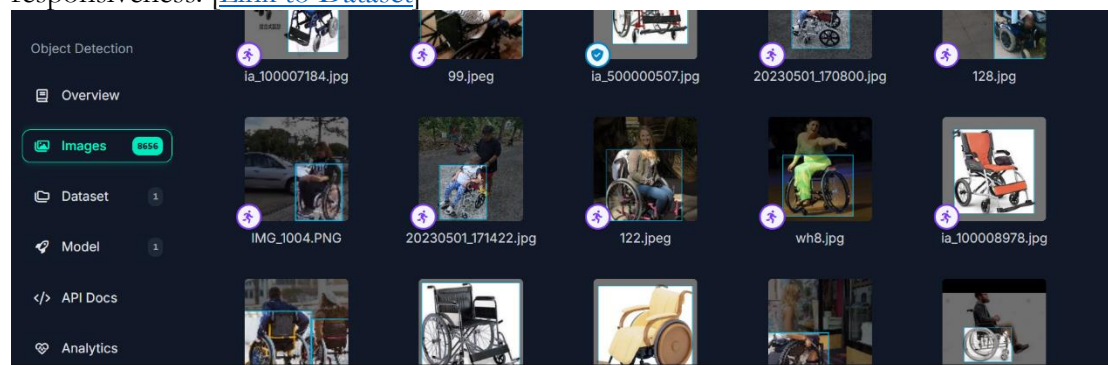


Figure 8: Example of Dataset.

## 2.4.6 Backend, Frontend, and Database

The backend, frontend, and database collectively form the core of the system's functionality. These components enable data processing, user interaction, monitoring, and real-time operations. Below is an organized explanation of each component and its role in the system.

- **The Backend** of the system is a robust and modular architecture, employing various advanced technologies to ensure seamless interaction between hardware, the database, frontend and Roboflow ML Cloud. At its core, it manages dynamic painting adjustments based on visitor interactions while facilitating efficient data handling and real-time responsiveness. With the capability to handle parallel requests at the same time from multiple painting microcontrollers. The backend uses Node.js for server-side logic, enabling high-performance asynchronous operations, and MongoDB with Mongoose for managing a centralized database that stores data about paintings, users, and interaction statistics. Real-time communication with hardware components such as sensors, motors, and cameras is achieved through the MQTT protocol, which allows the backend to send commands and receive updates with minimal latency. WebSocket



communication is integrated into the backend to provide instant updates to the frontend. The backend broadcasts real-time messages about system status, sensor activations, or height adjustments to all connected clients, ensuring that museum staff are informed of ongoing operations as they happen. This enhances user experience by enabling live monitoring of the system. The backend also processes frames captured by the camera. It sends them to a cloud-based object detection API, such as Roboflow, to identify wheelchair users, ensuring accurate and dynamic adjustments. The system's interaction flow includes the frontend sending requests to the backend for managing paintings, retrieving statistics, or authenticating users. The backend processes these requests, which interacts with the database to store or retrieve data and sends the required commands to hardware through MQTT. Real-time updates, such as adjustments triggered by visitor detection, are pushed to the frontend using WebSocket communication. Additionally, the backend seamlessly coordinates hardware actions such as installing new paintings or dynamically adjusting their heights. This sophisticated backend architecture, leveraging technologies like Express.js for API development, Axios for HTTP requests, and WebSocket for instant feedback, ensures a reliable, scalable, and user-friendly system. It forms the backbone of the operation, ensuring that the museum's dynamic painting system is responsive, accessible, and easy to manage.

- **The Frontend** of the system is a dynamic and interactive interface designed to provide seamless access for museum staff to manage and monitor the painting system. It enables critical tasks such as adding, updating, and deleting painting details, viewing real-time system updates, and analyzing visitor engagement statistics. Users interact with a centralized dashboard that aggregates data about painting performance, including metrics like the number of views and viewing durations. The frontend communicates with the backend to retrieve, display, and update data dynamically, ensuring accurate and up-to-date information. Real-time updates are delivered through live notifications and a responsive dashboard, where museum staff can monitor system health, painting adjustments, and sensor activations. The system also enforces role-based access to ensure that only authorized personnel, such as admins and workers, can access specific functionalities. Staff can make edits or adjustments to paintings through an intuitive interface, with changes reflected immediately across the system. Additionally, a responsive design ensures accessibility on various devices, making it easy for staff to interact with the system from different platforms. The frontend seamlessly integrates with the backend and database, ensuring smooth operations and providing museum staff with all the tools they need to manage and monitor the system effectively.
- **The Database** for the system is implemented using **MongoDB**, serving as the central repository for all essential data related to paintings, user management, and visitor statistics. Its flexible, document-based structure is ideal for handling the system's dynamic and interconnected data. The database employs well-defined schemas (see section 3.4 for Database Description) to organize information efficiently, supporting real-time updates and seamless integration with the backend. The **Painting Schema** manages details such as the painting's dimensions, base height, weight, and system status. It includes virtual fields for dynamically calculated attributes like the total height, ensuring accurate and responsive operations without redundant storage. The **User Schema** stores user credentials and roles, enabling secure authentication and role-based access control. Admins can manage system configurations, while workers are restricted to monitoring and operational tasks. The **Painting Statistics Schema** tracks visitor engagement data, including total views, cumulative viewing durations, and daily interaction breakdowns. We utilized JSON fields to store complex and flexible data

structures for attributes such as viewing sessions and daily stats, enabling efficient storage and retrieval of detailed session data. This scheme supports methods for logging new sessions and generating aggregated reports, providing actionable insights into visitor behavior. The database interacts seamlessly with the backend to ensure consistent data flow. Actions performed by museum staff, such as updating painting details or managing user accounts, trigger API requests that update the relevant database records. Similarly, visitor interactions, such as viewing a painting, are logged in real time, keeping statistics accurate and up to date. This database structure, paired with MongoDB's scalability and the flexibility of JSON, ensures efficient performance and supports the system's future growth, whether by adding more paintings or increasing user activity.

## 2.4.7 Networking for Communication between IOT devices and Server - MQTT

This sequence diagram illustrates the flow of communication between multiple microprocessors with the central hub backend server and forwarded to the mechanical team's microprocessor. Utilizing the MQTT protocol to send the value of height adjustment to a specific painting's system dynamically based on user presence and wheelchair detection.

There are 3 types of use cases for MQTT to integrate communication with the mechanical team and the software team microcontroller with backend server.

### 1. Installation of a New Painting System Device

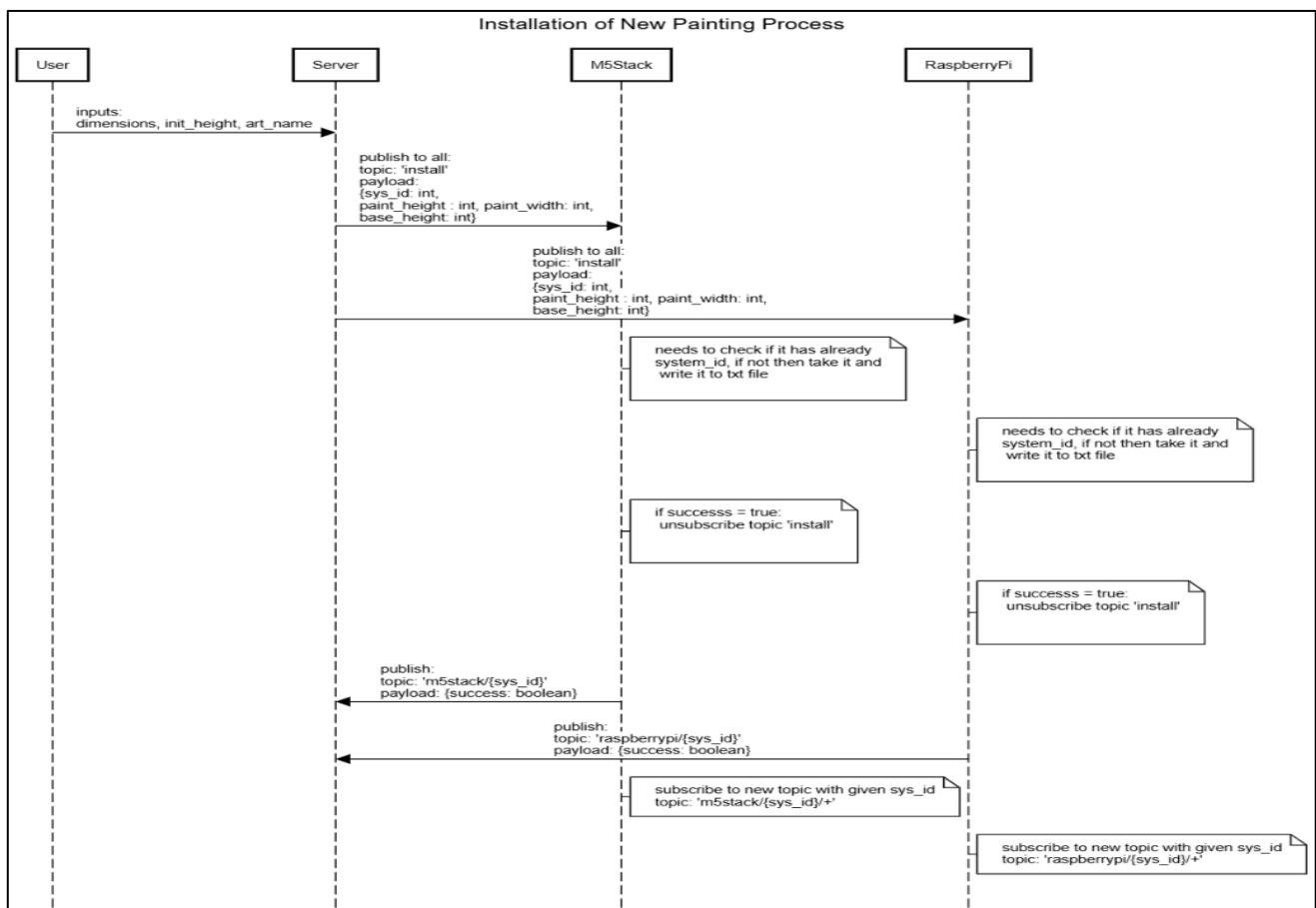


Figure 9: Installation of new a painting system Diagram.

This sequence diagram (Figure 9) showcases the process of installing new painting equipment into the software system while integrating with the mechanical components using MQTT protocols for seamless communication. The process begins with the user providing input through the system interface, which includes the painting's dimensions, initial height, and the name of the artwork. The backend server processes this input and publishes a message to the MQTT "install" topic. This message contains a payload with essential details, such as the system ID (sys\_id), painting height, painting width, and base height. Upon receiving the "install" message, the mechanical microcontroller checks whether the sys\_id is already stored in its local system. If the sys\_id is new, it records the information in a local text file for future reference. Similarly, the Raspberry Pi microcontroller, managed by the software team, performs the same checks and stores the sys\_id if it is new. Once the installation process is completed successfully, both microcontrollers send success acknowledgments back to the server by publishing to their respective topics (m5stack/{sys\_id} and raspberrypi/{sys\_id}). They then unsubscribe from the "install" topic to prevent redundant installations. Finally, the microcontrollers subscribe to new, painting-specific topics (m5stack/{sys\_id}/+ and raspberrypi/{sys\_id}/+), enabling real-time communication and the ability to receive future height adjustment commands specific to the installed painting. This comprehensive process ensures efficient and scalable installation of new painting systems.

## 2. Height Adjustment Process – via MQTT

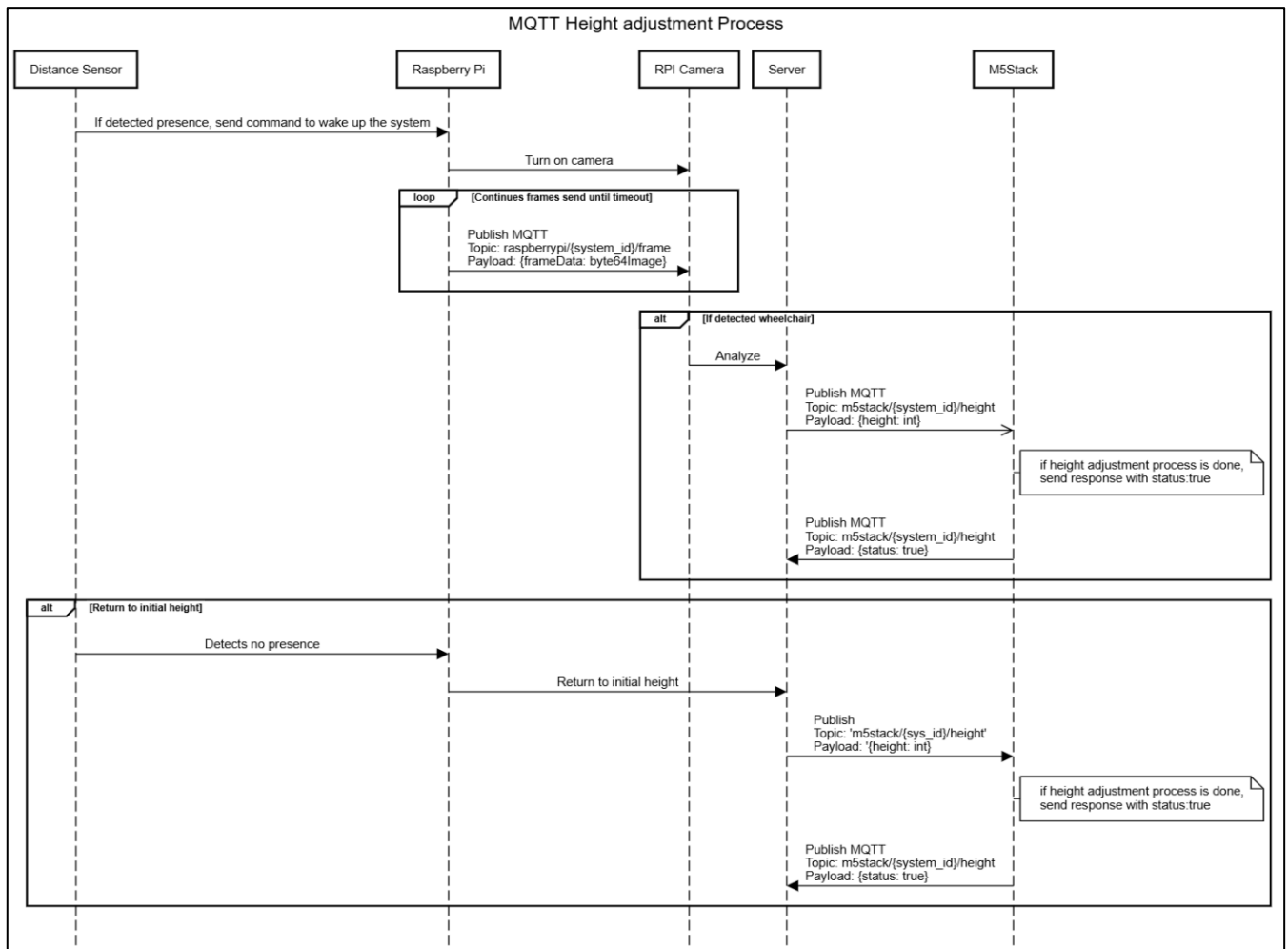
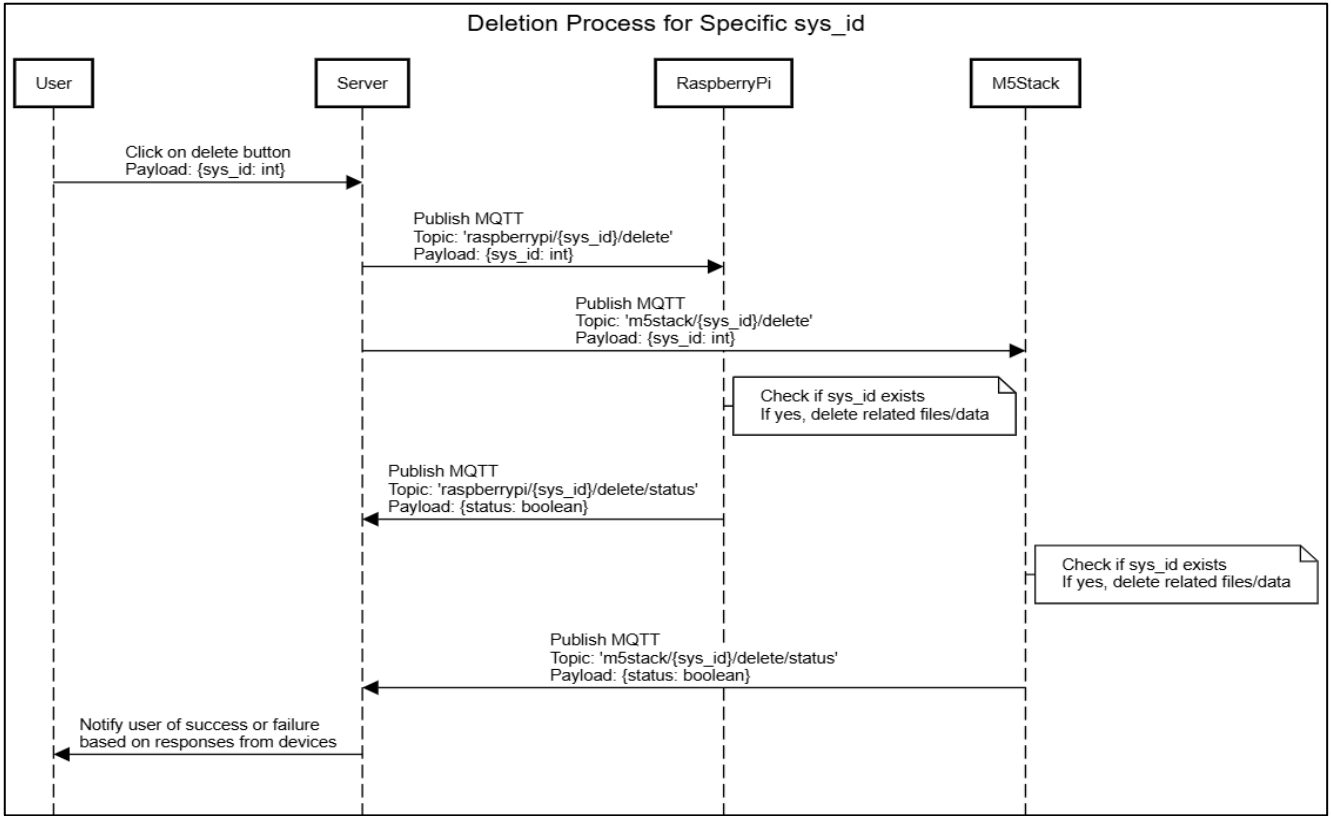


Figure 10: Height adjustment process.

This diagram illustrates the use of MQTT in Height Adjustment Process for a dynamic museum system that adjusts the height of paintings based on visitor detection and interaction. The components involved are the Distance Sensor, Raspberry Pi, RPI Camera, Server, and M5Stack, and the process includes two scenarios: detecting a wheelchair and absence detection. The height adjustment process begins with the distance sensor continuously monitoring the area for visitor presence. When a visitor is detected within range, the sensor sends a command to the ESP32 microcontroller to wake up the system. The ESP32 activates the RPI Camera to capture real-time frames of the visitor. These frames are published to the server over MQTT using the topic `raspberrypi/{system_id}/frame`, with the payload containing the image data in Base64 format. This process continues in a loop until a timeout or specific condition is met. The server analyzes the received frames to determine if the visitor is in a wheelchair. If a wheelchair is detected, the server calculates the required height adjustment and publishes an MQTT message to the M5Stack microcontroller using the topic `m5stack/{system_id}/height`, with the payload containing the calculated height value `{height: int}`. Upon receiving the command, the M5Stack adjusts the painting's height accordingly using its mechanical system and sends a confirmation back to the server via MQTT, indicating success with a payload `{status: true}`. If the distance sensor detects no presence after the visitor leaves, it signals the ESP32 to return the painting to its initial height. The ESP32 publishes an MQTT message to the server using the topic `m5stack/{system_id}/height`, containing the default height value `{height: int}`. The server forwards this command to the M5Stack, which adjusts the painting back to its original position. After completing the adjustment, the M5Stack sends a confirmation `{status: true}` to the server, ensuring the process is logged and verified. This dynamic system adapts to visitor needs in real time, offering accessibility through automatic adjustments and maintaining orderly conditions by resetting the painting to its original height when the area is unoccupied. The system's use of MQTT protocols and inter-component communication ensures responsiveness, efficiency, and a seamless experience for museum visitors.

### 3. Deletion Process for Painting System Device



**Figure 11:** Deletion Process for painting system.

This sequence diagram illustrates the process triggered when a user initiates a request to delete a specific `sys_id`. The steps are as follows: The deletion process begins with the user initiating the action by clicking the delete button on the frontend interface. This triggers a request that sends a payload containing the system ID (`sys_id`) to the server. Upon receiving the request, the server broadcasts an MQTT message to both the Raspberry Pi and the M5Stack microcontroller. The message is published to the topics `raspberrypi/{sys_id}/delete` and `m5stack/{sys_id}/delete`, with a payload containing the `sys_id` to instruct both devices to process the deletion for the specified system. The devices handle the deletion process independently. The Raspberry Pi checks its local storage to determine if the `sys_id` exists, and if found, it deletes any associated files or data. Similarly, the M5Stack performs the same check and deletes relevant files or data if the `sys_id` exists. Once the deletion process is completed, both devices return status responses to the server. The Raspberry Pi publishes its response to the topic `raspberrypi/{sys_id}/delete/status`, and the M5Stack publishes to `m5stack/{sys_id}/delete/status`, with payloads indicating the success or failure of the operation as `{status: boolean}`. The server processes these responses and notifies the user of the outcome, confirming whether the deletion was successfully completed or if any errors occurred during the process. This streamlined deletion workflow ensures efficient management of painting systems while maintaining synchronization between components.

The system leverages parallelization, dynamic communication, and scalability to ensure efficient and adaptive operations. Parallel operations allow tasks such as installation, height adjustment, and deletion to occur simultaneously by broadcasting MQTT messages to multiple devices, enabling independent processing and eliminating bottlenecks. The use of MQTT facilitates real-time, adaptive communication tailored to the current state of the system, allowing devices to act based on their unique states and commands, ensuring flexibility and responsiveness. Additionally, the system is inherently scalable, supporting

the seamless addition of new devices, paintings, or features without requiring changes to existing processes. Using unique `sys_id`-based topic segregation ensures effective communication and management across multiple devices or systems, maintaining the integrity and efficiency of the architecture.

## 2.5 Challenges

The development and implementation of the system presented several challenges that required careful planning, collaboration, and problem-solving. These challenges included both technical and non-technical aspects:

1. **Selecting Optimal Tools and Technologies:** Choosing the most suitable tools and technologies for the system was a significant challenge. With numerous options for hardware components, software frameworks, and communication protocols, the team had to conduct extensive research, testing, and evaluation. The primary focus was ensuring compatibility between the selected components and scalability for future system expansions, making this process both complex and time-intensive.
2. **First-Time Integration of Physical and Software Systems:** The integration of physical components, such as motors, sensors, and cameras, with the software system was entirely new for our team. This involved learning and implementing technologies like MQTT for communication and handling real-time hardware-software interactions. Addressing unexpected compatibility issues and troubleshooting operational challenges required innovative problem-solving and collaboration between different technical domains.
3. **Developing Parallel Multiple Painting System:** One of the significant challenges was developing a system capable of managing and handling multiple painting setups (including microcontrollers, sensors, and cameras) for each painting independently and simultaneously. Ensuring smooth communication between devices and coordinating their actions without conflicts required advanced system design and robust programming logic.
4. **Time Constraints:** Completing the project within a strict timeline was challenging. Balancing the various stages of the project research, development, testing, and documentation demanded efficient time management. The team had to prioritize tasks carefully to ensure that the system met its objectives without compromising the quality of implementation and results.
5. **Developing Accurate Formulas:** Deriving and validating the formulas for height adjustment and optimal viewing distance was a technically demanding process. It required a detailed understanding of accessibility standards, visitor behavior, and system constraints. Ensuring that the formulas were robust enough to handle diverse scenarios while aligning with accessibility guidelines added further complexity to this challenge.
6. **Collaborating with the Mechanical Team:** Working closely with the mechanical engineering team posed its own challenges. Clear communication was essential to ensure the hardware requirements and constraints aligned with the software design. Differences in technical expertise and workflows required effective coordination, regular updates, and collaborative problem-solving to address issues and ensure successful software integration with the mechanical components.

Though difficult, these challenges ultimately contributed to the team's growth and the project's success. They provided valuable learning experiences in interdisciplinary collaboration, technical problem-solving, and project management, laying a strong foundation for future projects and advancements.



## 2.6 Testing Plan

The testing plan evaluates the system's functionality, integration, and performance across all components, focusing on critical areas such as visitor detection, multi-painting operations, communication between software and hardware, and real-time frontend updates. Each test is designed to simulate real-world scenarios and validate the system's reliability, scalability, and usability.

1. **Camera Sensor and Multi-Painting Tests:** The camera sensor's detection capabilities are tested to ensure accuracy in identifying individual visitors, wheelchair users, and their association with multiple paintings. This phase evaluates the system's ability to handle multiple camera sensors simultaneously, simulating scenarios where visitors interact with several paintings simultaneously. Each sensor is monitored to verify that it triggers the correct painting's height adjustment and reflects the corresponding status on the frontend.

To achieve this, **two Raspberry Pi devices with camera and ultrasonic sensors were assembled** (as shown in the Figure 12) to simulate interactions for two separate paintings. This setup allows testing of real-world scenarios where multiple devices operate concurrently, ensuring each sensor independently detects and responds to its respective painting.

- Each device is monitored for the following:
- Accurate association of sensor detection to the correct painting.
- Real-time synchronization of sensor outputs with motor commands.
- Confirmation that height adjustments for each painting are independent and error-free.

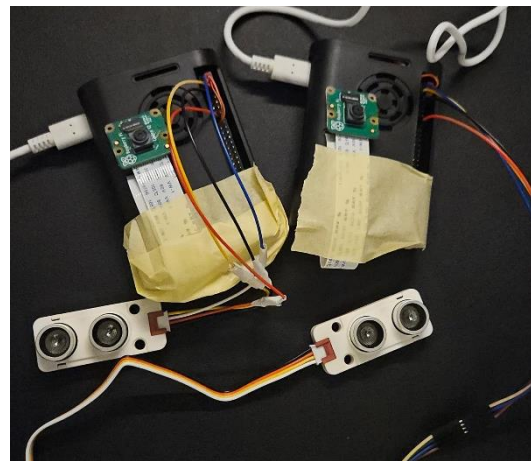


Figure 12: Two Paintings Devices Setup.

Additionally, simulations are conducted to evaluate the synchronization of sensor outputs with motor commands, confirming that each painting adjusts correctly and independently. Results include real-time updates displayed on the frontend for each painting, ensuring seamless multi-painting operation.

2. **Hardware Integration Tests:** The integration between the software system and the mechanical prototype is evaluated to ensure accurate and prompt execution of height adjustments. Commands sent by the software via MQTT are tested for proper communication with the motors. Real-time adjustments are observed, and the response time is measured to ensure the system operates within the expected range of 2-5 seconds after detection. If time permits, the system will be physically connected to the mechanical prototype to validate software and hardware integration. Future testing plans involve deploying the system in real-time scenarios to observe performance under actual operating conditions with multiple paintings.
3. **Frontend Tests:** The frontend interface is tested for its login, painting management, and data visualization functionality. Login functionality is evaluated to ensure users can access the system securely, and role-based access directs them to the correct pages and features. Painting management tests confirm the frontend's ability to add, edit, and delete paintings, with changes reflected accurately in the database and displayed on the interface. The statistics page is tested for its ability to accurately display live visitor

interactions and historical data. Real-time updates for multi-painting operations, such as height adjustments and sensor activations, are verified to ensure the frontend effectively reflects the system's current state.

4. **System Integration Tests:** The entire system is tested for synchronization between all components, including the frontend, backend, hardware, and database. Simulated visitor interactions trigger real-time operations such as detection, data processing, and painting adjustments. Multi-painting scenarios are tested to ensure the system handles concurrent operations effectively, with updates accurately displayed on the frontend. Testing ensures seamless communication between components, validating the system's readiness for deployment. Future plans include scaling tests to evaluate performance in large museum settings with multiple paintings.

## 2.7 Conclusion

The development and implementation of the system have been a remarkable journey of learning, collaboration, and innovation. This project aimed to create a dynamic painting adjustment system that enhances accessibility for all visitors, particularly those using wheelchairs, by integrating advanced computer vision, real-time communication, and mechanical engineering. Through the collective efforts of our team, supervisors, and the mechanical department, we overcame numerous challenges and delivered a system that aligns with our objectives of inclusivity and functionality.

One of the primary challenges we faced was selecting the optimal tools and technologies for the system. After extensive research, testing, and evaluation, we identified the best options that ensured compatibility and scalability. Simulations and rigorous testing of the software components allowed us to refine and validate our approach. With the invaluable guidance of our supervisor and the mechanical lecturer, we were able to make informed decisions and successfully address this challenge.

Integrating physical and software systems, a new experience for our team, posed difficulties. However, through teamwork and collaboration, we gained proficiency in technologies like MQTT and tackled hardware-software compatibility issues.

Regular meetings, both virtual via Zoom and in-person at the college, helped us synchronize our progress with the mechanical team. These updates ensured clear communication and alignment between the software and hardware components. Working with the mechanical team was a rewarding experience, and their dedication to developing a working prototype demonstrated the success of our collaborative efforts. While time constraints limited our ability to integrate the mechanical and software components fully, we developed a comprehensive plan for connecting the two systems. If additional time becomes available, we are prepared to complete the integration and conduct full system tests. If not, we have documented the integration process thoroughly, providing the mechanical team with all the necessary instructions and tools to complete the connection. This project not only achieved its technical goals but also highlighted the importance of interdisciplinary teamwork and problem-solving. By addressing accessibility challenges in museums, it underscores our commitment to creating inclusive environments through innovative technology. The experience of working on this project has been invaluable, fostering growth, collaboration, and a deep sense of accomplishment. Moving forward, this system has the potential to be expanded and refined, paving the way for broader applications in similar environments.

## 3. User Documentation

### 3.1 User's guide - Operating instruction

#### 3.1.1 General Description

The Adaptive Height Adjustment System is designed to enhance engagement in art exhibitions by allowing seamless height adjustments of paintings to suit the needs of those using wheelchairs. The system leverages advanced technology, including sensors and cameras, to detect visitors and adjust artwork positioning automatically. The accompanying Maintenance System dashboard provides users with tools to manage artworks, monitor system performance, and analyze visitor interaction data. This interface is specifically designed to assist museum staff in efficiently tracking and maintaining the system's functionality while ensuring an optimal visitor experience.

User Roles: The system supports two types of users:

1. **Admin:** Responsible for high-level management and analytics, including viewing detailed visitor engagement metrics and generating reports.
2. **Worker:** Focused on operational tasks and system monitoring, ensuring the system runs smoothly and artworks are properly adjusted and maintained.

#### 2.5.1 Operating instructions

General pages for all users:

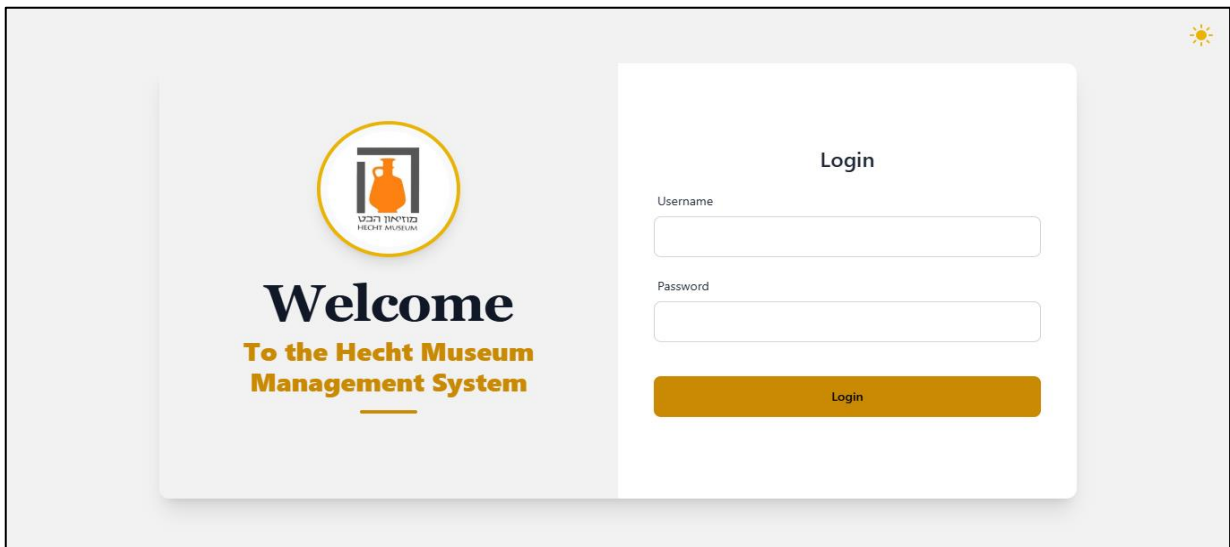
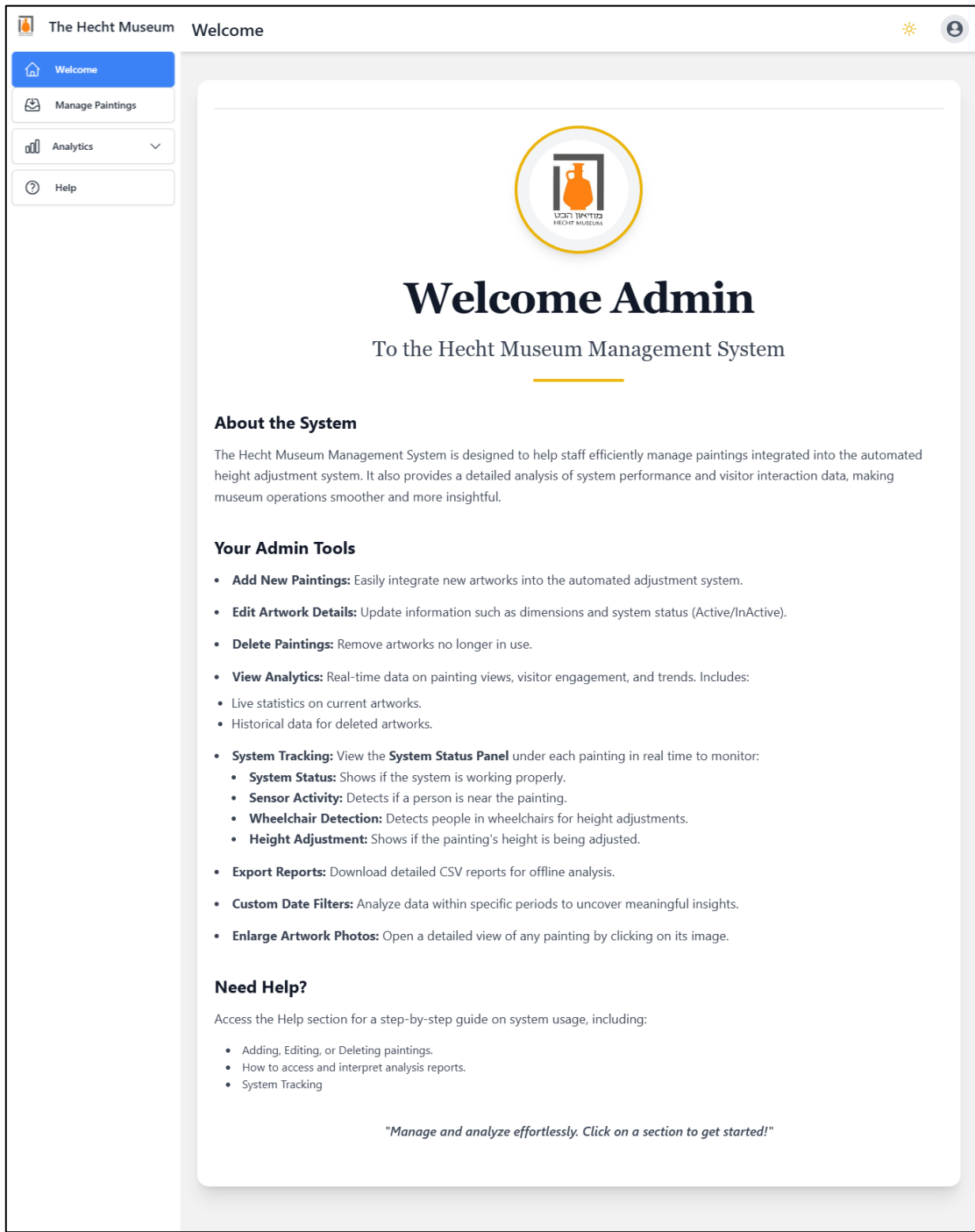
The screenshot shows a web interface for the Hecht Museum Management System. On the left, there is a large light gray box containing a circular logo with an orange silhouette of a museum building and the text 'HECHT MUSEUM' below it. Below the logo, the text 'Welcome' is written in a large, bold, dark blue font, followed by 'To the Hecht Museum Management System' in a smaller, bold, orange font. On the right, there is a white box titled 'Login'. It contains two input fields: 'Username' and 'Password', both with orange borders. Below these fields is a large orange button with the text 'Login' in white. The entire interface is set against a light gray background with a small sun icon in the top right corner.

Figure 13: Log-in Page.

Log-in Page: Enter your username and password to access the system. If you have valid credentials, click the Login button to proceed.

**Welcome Page for Admin:** This page is specifically tailored for administrators, providing them with a personalized greeting and access to all the system's tools and features. It



**Figure 14: Welcome Page for Admin.**

includes a navigation panel to manage paintings, view analytics, and access help. The Admin Tools section highlights functionalities like adding, editing, and deleting paintings, monitoring system status, exporting reports, and analyzing visitor interaction data. This page is designed to support the admin in overseeing system operations and generating insightful reports.

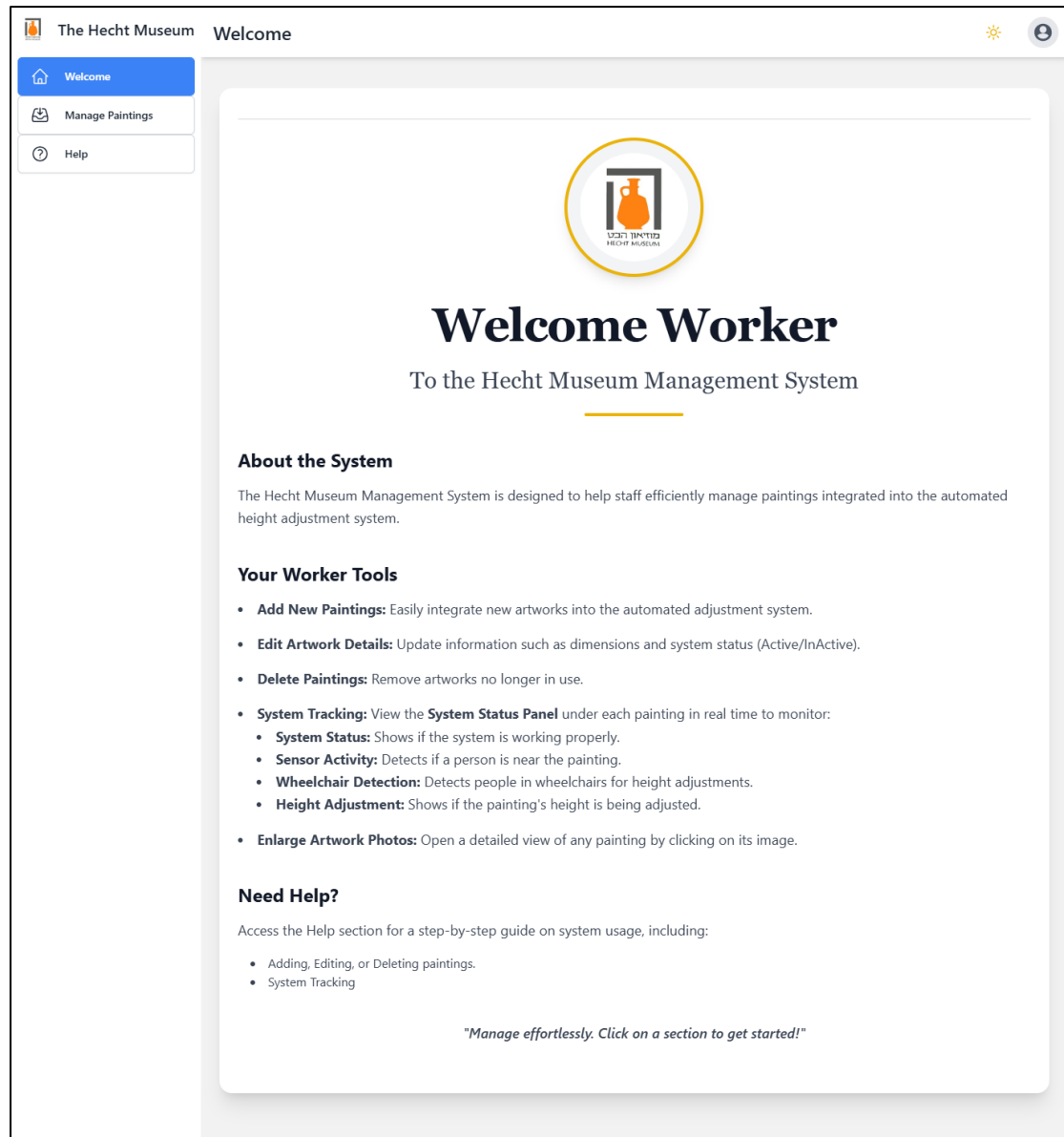


Figure 15: Welcome Page for Worker.

**Welcome Page for Worker:** This page is designed for workers, focusing on their operational responsibilities. It provides a similar layout to the admin page but restricts access to only the tools and features necessary for day-to-day operations. Workers can manage paintings, monitor the system status in real time, and access a Help section for guidance. This ensures that workers can efficiently manage their tasks without access to advanced admin-level analytics or reporting features.

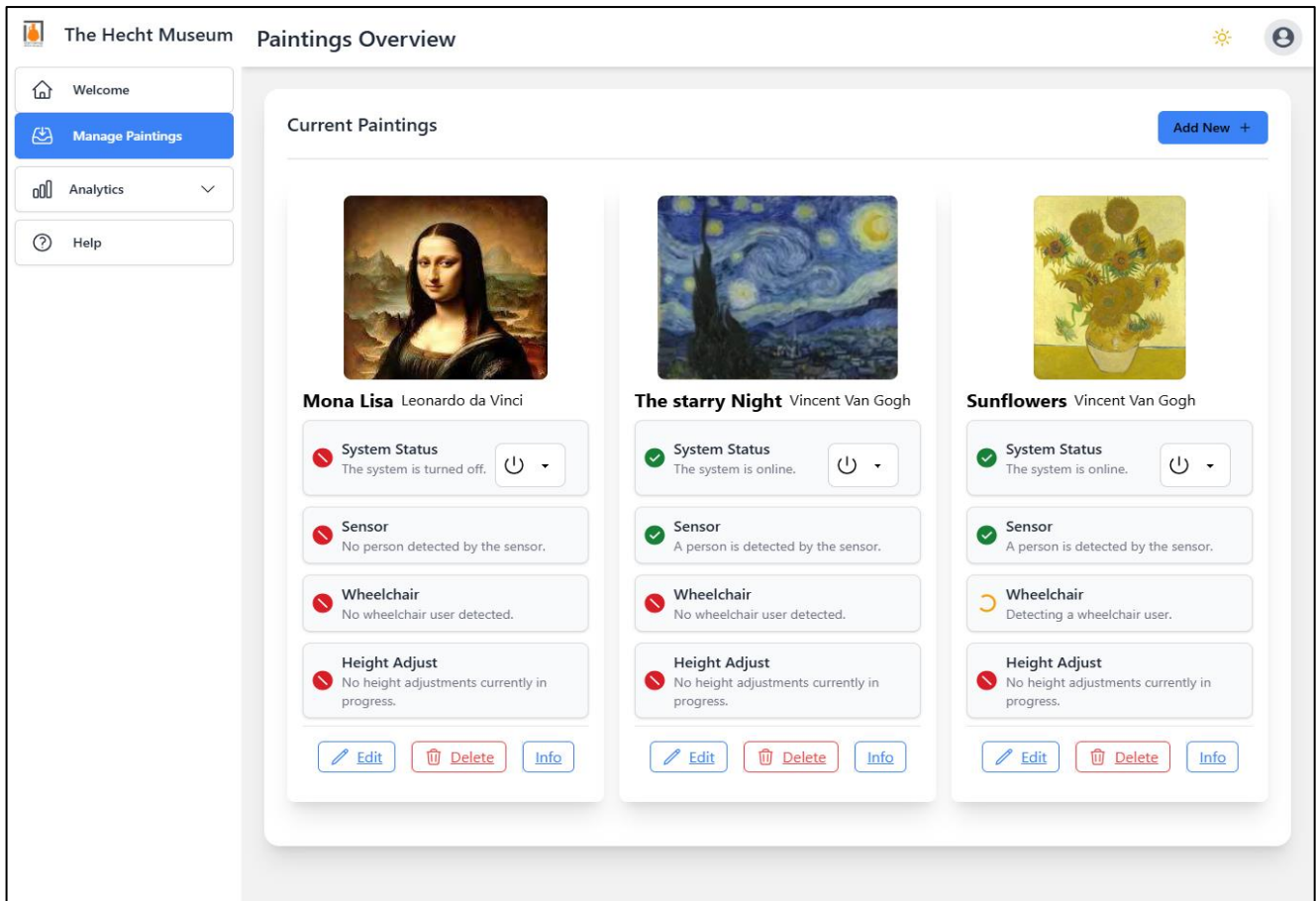


Figure 16: The Paintings Overview Page.

The Paintings Overview Page: allows admins and workers to monitor and manage the paintings in the system. Each painting is displayed as a card with real-time status and management options.

### 1. Painting Cards:

Displays the painting's title, artist, and image, shows the painting's status:

- **System Status:** Whether the system is running.
  - ✔ The system is operational - Indicates that the program is running, the microcontroller is powered on, and it is connected.
  - ✘ The system is turned off - Indicates that the microcontroller is powered down, and no processes are active.
- **Sensor:** If a visitor is detected.
  - ✔ Visitor detected.
  - ✘ No visitor was detected.
- **Wheelchair:** If a wheelchair user is detected.
  - 🔄 The system analyzing if a wheelchair user is present.
  - ✔ Wheelchair user detected.



- ❌ No wheelchair user was detected.
- **Height Adjustment:** If height adjustments are in progress.
- ✅ Adjustment in progress.
- ❌ No adjustment happening.

## 2. Actions:

### - Device start/shutdown:

Staff members can interact with the button next to the system status to perform specific actions to a specific painting's system. When the power button is clicked, a dropdown menu appears with the following options:

**Shutdown MicroController** - Safely powers off the microcontroller.

**Restart MicroController** - Reboots the microcontroller to refresh processes.

**Re/Start Program** - Restarts the running program without rebooting the microcontroller.

- **Edit:** Modify painting details such as dimensions and name.
- **Delete:** Remove the painting from the system.  
Info: View more details about the painting, including height, width, weight, and base height.
- **Add New Painting:** The Add New button at the top-right allows users to add new paintings by entering their details, such as height, weight, painter's name, painting name, width, and base height.
- **Click on Photo:** Clicking on the painting's photo enlarges it for better viewing.

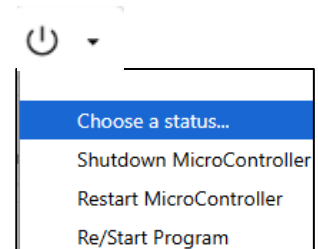


Figure 17: The dropdown menu.

This page provides a clear and quick overview of all paintings and their operational status, including system status, sensor detection, wheelchair detection, and height adjustment progress. It allows users to monitor and manage the paintings efficiently by providing direct access to actions such as editing details, viewing additional information, and removing or adding new paintings. The intuitive interface and real-time updates make it easy for users to ensure all systems are functioning optimally and address any issues promptly.

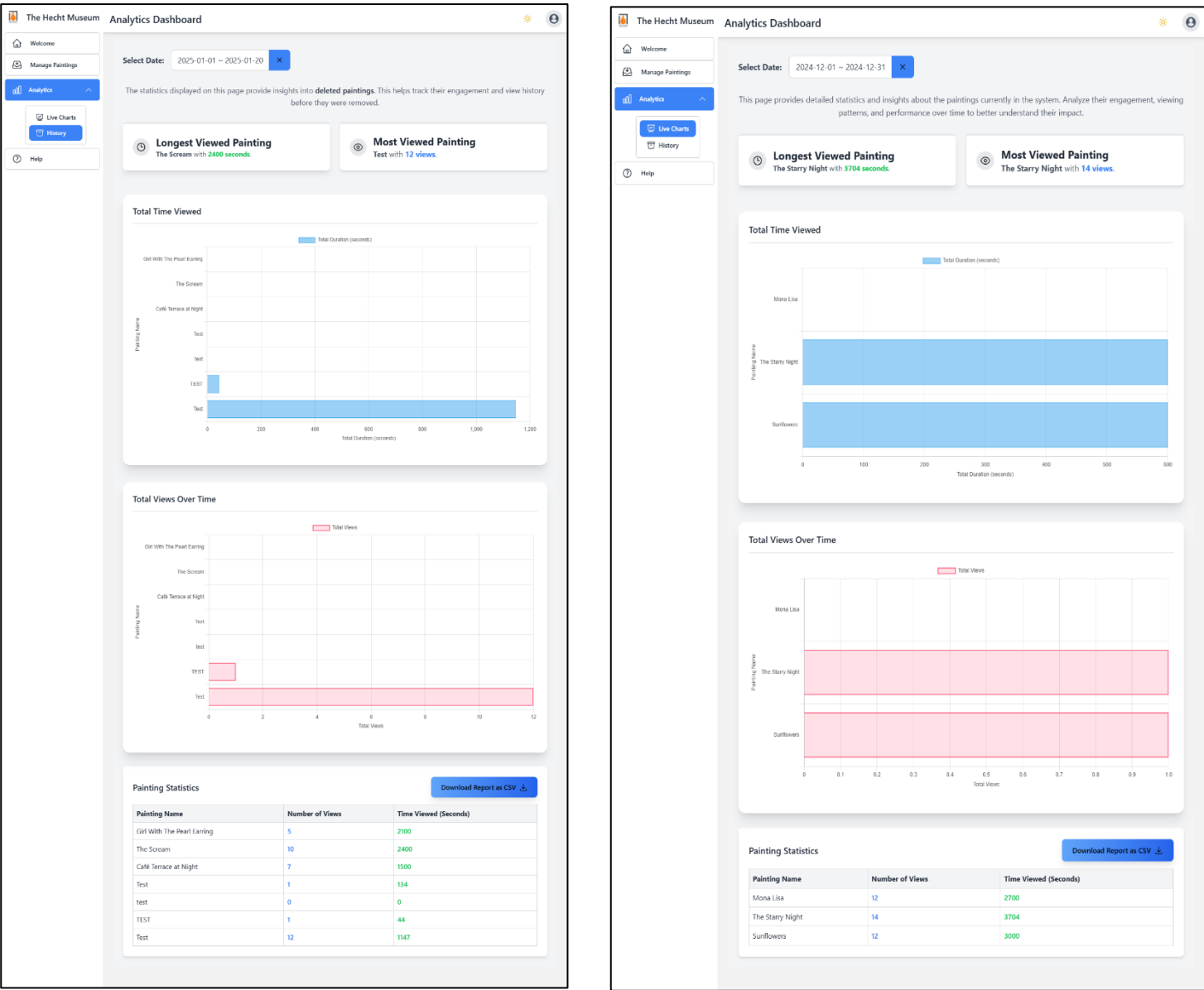
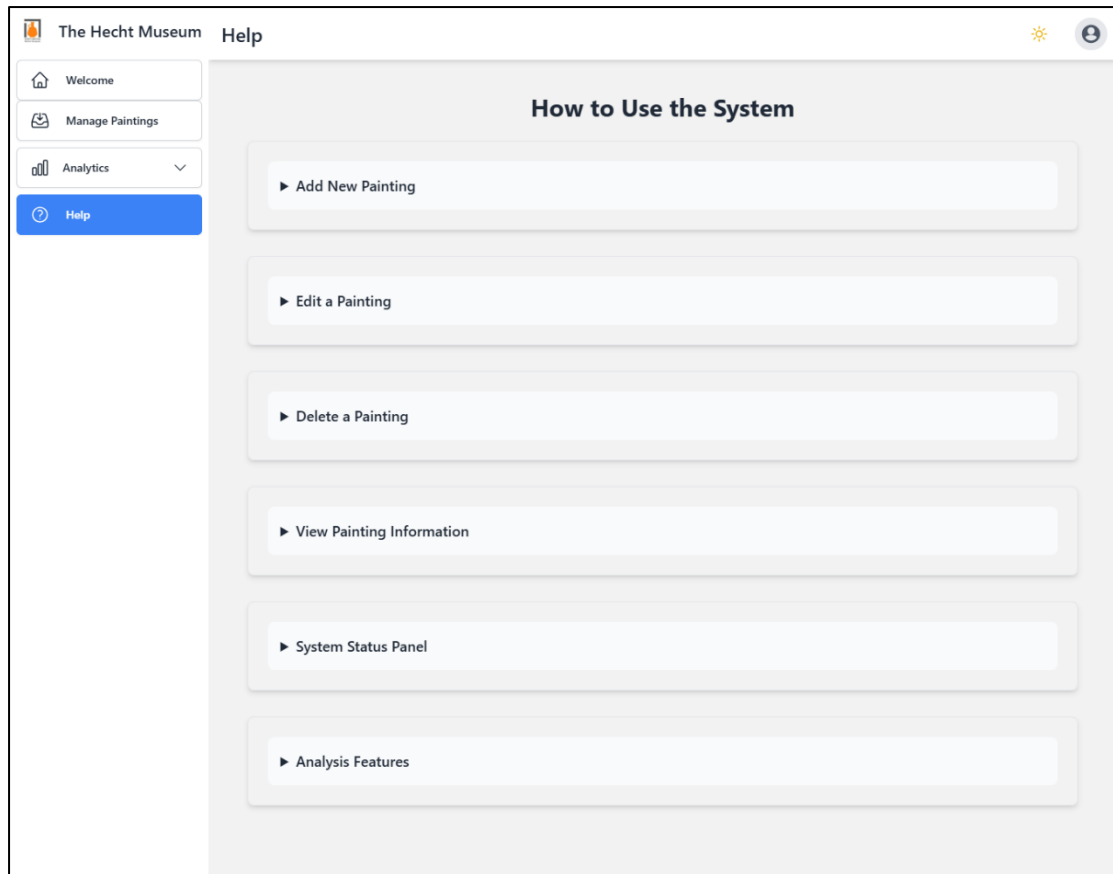


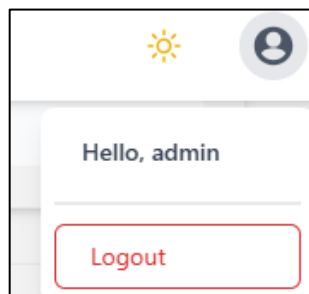
Figure 18: The Analytics Dashboard Live Charts and History.

Analytics Dashboard gives the admin a comprehensive view of painting engagement and system performance. It includes two main sections: Live Charts and History. The Live Charts page offers real-time insights into paintings currently in the system, allowing the admin to track metrics like total viewing time, number of views, and overall engagement trends. The History page focuses on data for paintings that have been removed from the system, providing historical records of their engagement. Both pages allow data filtering according to a selected date range, enabling focused analysis for specific periods. Interactive graphs, detailed statistics tables, and the ability to export data as CSV reports ensure the admin can make informed decisions and maintain thorough records.



**Figure 19:** The Help Page.

The Help Page: provides a detailed guide for both the admin and worker roles, with sections dedicated to each part of the system. For the admin, it includes instructions for adding, editing, deleting, and viewing painting details, as well as understanding the System Status Panel. Admins also have access to an Analysis Features section, which explains how to use the analytics dashboard to track engagement and generate reports. Workers have the same help sections but without the Analysis Features, as they do not have access to this functionality. Each section outlines step-by-step instructions, expected outcomes, and troubleshooting tips, ensuring users can effectively operate the system and resolve any issues.



**Figure 20:** Features.

At the top right of the interface, the user has access to a profile dropdown menu. When clicked, it displays a greeting (e.g., "Hello, admin") and provides the option to logout from the system securely. A dark mode toggle is also available, allowing users to switch between

light and dark themes for a more personalized and comfortable viewing experience. This feature ensures easy account management and user preference customization.

## 3.2 Maintenance guide

Highlighted texts in yellow refer to shell commands.

### 3.2.1 Raspberry Pi 4 Installation Guide

#### 1. Installation

##### 1. Download Raspberry Pi Imager

1. Visit the official Raspberry Pi website: <https://www.raspberrypi.com/software/> .
2. Download the Raspberry Pi Imager for your operating system (Windows, macOS, or Linux).
3. Install the software.

##### 2. Insert the SD Card

1. Insert a compatible microSD card (at least 8GB recommended) into your computer's SD card reader.
2. Backup any important data on the SD card, as it will be erased.

##### 3. Run Raspberry Pi Imager

1. Open the Raspberry Pi Imager.
2. Click "CHOOSE OS" and select the following: Raspberry Pi OS (64-bit)
3. Click "CHOOSE STORAGE" and select your SD card.

##### 4. Write the OS to the SD Card

1. Click "WRITE" to begin the process.
2. Confirm any prompts, and the tool will download and write the OS to the SD card.
3. Wait for the process to complete (it may take a few minutes).
4. Optional, click edit settings

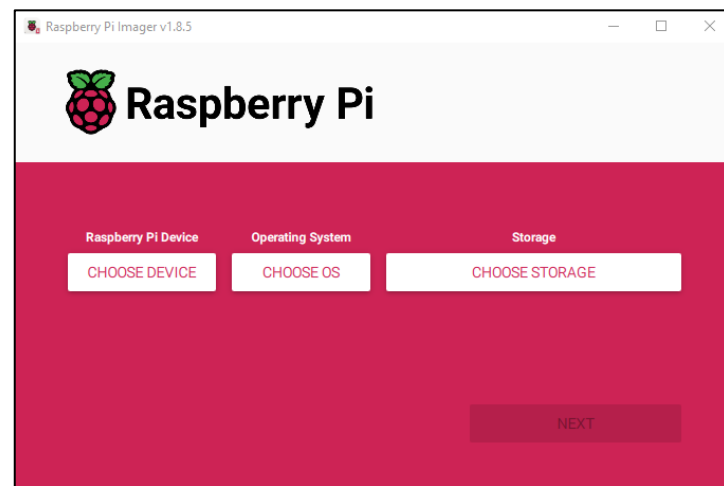


Figure 21: Raspberry Pi Imager.

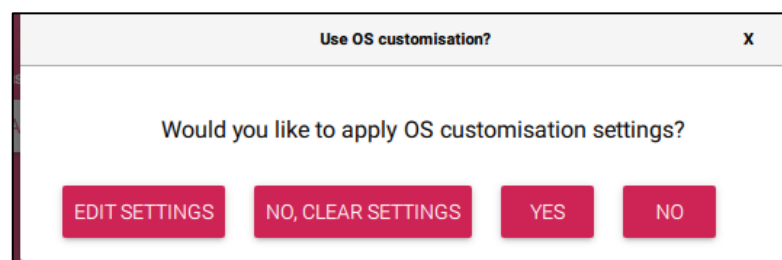


Figure 22: Use OS customisation.

5. Set username and password as you like.  
Also, set your wifi details for automated connection.

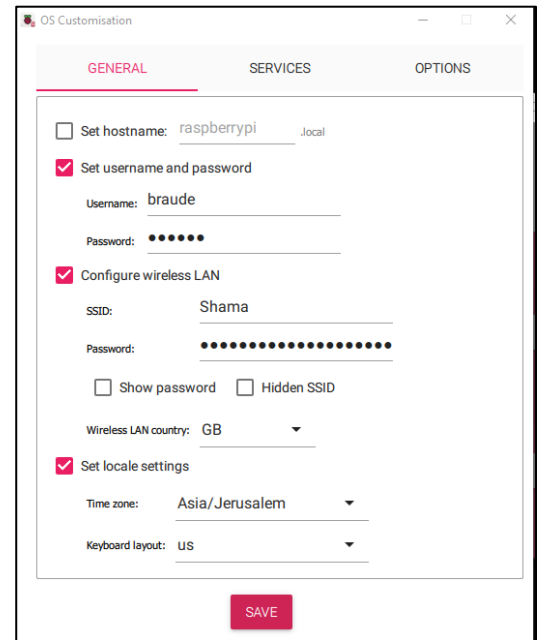
The image shows a window titled "OS Customisation" with three tabs: "GENERAL", "SERVICES", and "OPTIONS". The "GENERAL" tab is selected. It contains several settings: "Set hostname" is unchecked with the value "raspberrypi.local"; "Set username and password" is checked, with "Username" set to "braude" and "Password" masked with dots; "Configure wireless LAN" is checked, with "SSID" set to "Shama" and "Password" masked with dots. Below these are unchecked checkboxes for "Show password" and "Hidden SSID", and a "Wireless LAN country" dropdown set to "GB". At the bottom, "Set locale settings" is checked, with "Time zone" set to "Asia/Jerusalem" and "Keyboard layout" set to "US". A red "SAVE" button is at the bottom right.

Figure 23: Set Details.

### 3.2.2 Steps to Connect Raspberry Pi to HDMI Display and Install VNC Tools

#### 1. Connect Raspberry Pi to an HDMI Display

1. Power Off the Raspberry Pi:
  - Disconnect power from the Raspberry Pi to prevent damage while connecting cables.
2. **Connect the HDMI Cable:**
  - Insert one end of the HDMI cable into the Raspberry Pi's HDMI port (use HDMI0 for dual HDMI models).
  - Connect the other end to your monitor or TV.
3. **Power On the Raspberry Pi:**
  - Plug the power cable back into the Raspberry Pi and wait for it to boot. The Raspberry Pi should display the desktop or console on the screen.

#### 2. Log Into Raspberry Pi

1. If the Raspberry Pi boots into the desktop, log in using your credentials.
2. If it boots to the terminal, log in and start the desktop environment with: **startx**

#### 3. Install RealVNC Viewer

Open the following link to install RealVNC Viewer to windows:

<https://www.realvnc.com/en/connect/download/viewer/>

#### 4. Install RealVNC Server

1. Update the package list:

`sudo apt-get update`

2. Install the RealVNC server:

`sudo apt-get install realvnc-vnc-server`

## 5. Enable VNC Server

1. Open the Raspberry Pi configuration tool:

`sudo raspi-config`

2. Navigate to:

- Interfacing Options → VNC → Enable.

3. Exit the tool and reboot the Raspberry Pi:

`sudo reboot`

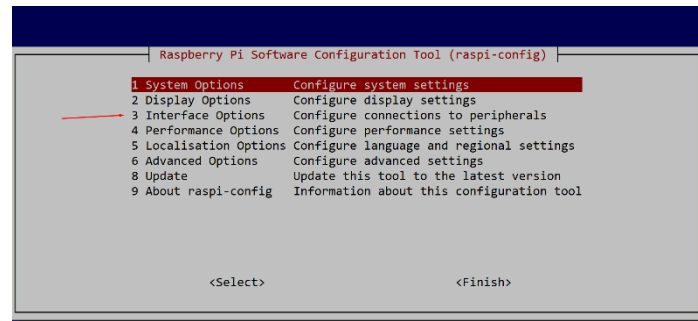


Figure 24: configuration tool.

## 6. Start the VNC Server

If the server doesn't start automatically, manually start it:

`sudo systemctl start vncserver-x11-serviced`

## 7. Connect to VNC

1. Open VNC Viewer on your computer (installed in step 3).
2. Enter Raspberry Pi's IP address (e.g., 192.168.1.x) into the VNC Viewer.
3. Log in using Raspberry Pi's username (pi) and password (default: raspberry, unless changed).

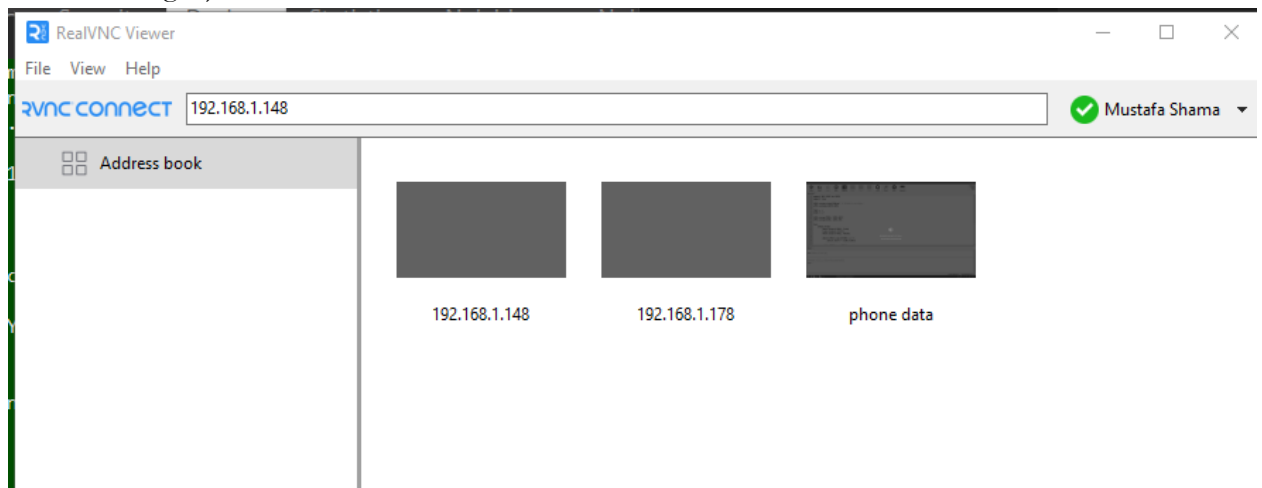
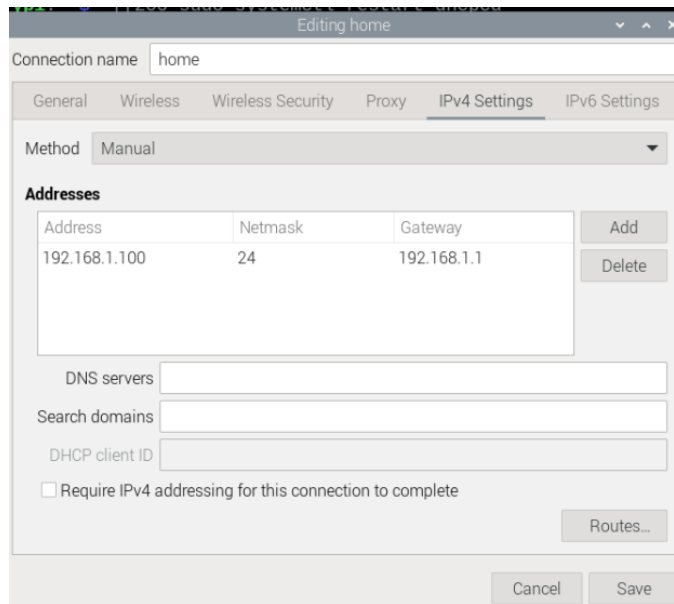


Figure 25: RealVNC dashboard.

### 3.2.3 Setting a Static IP Address on Raspberry Pi (Linux)

1. Open Network Settings:
  - Click on the network icon in the taskbar (usually in the top-right corner of the screen).
  - Select advanced options and then edit connections
  - Select the wireless connection you want to configure
2. Navigate to IPv4 Settings:
  - In the window that appears, go to the **IPv4 Settings** tab.





**Figure 26:** Navigate to IPv4 Settings.

3. **Set the Connection Method to Manual:**
  - From the **Method** dropdown, select **Manual**.
4. **Add a Static IP Address:**
  - Under **Additional static addresses**, click the **Add** button.
  - Enter the following details:
    - **Address:** Your desired static IP (e.g., 192.168.1.100).
    - **Netmask:** Subnet mask (e.g., 255.255.255.0).
5. **Save and Apply Settings:**
  - Click the **Save** button to apply the changes.
6. **Restart Network Services:**
  - Open a terminal and run the following command to restart the network services:
  - `sudo systemctl restart NetworkManager`
7. **Verify the Static IP Address:**
  - Check if the static IP has been applied using:
  - `ip addr show`
  - Ensure your desired static IP address is listed.

#### **Troubleshooting:**

- **Reboot the Raspberry Pi:** If changes don't take effect, reboot the device:
- `sudo reboot`
- **Check for IP Conflicts:** Ensure no other devices on your network are using the same IP address.

This method allows you to configure the static IP using the graphical interface for convenience.

### **3.2.4 Setting Up Raspberry Pi 4 with Picamera**

This guide outlines the steps required to set up a Raspberry Pi 4 with the Picamera2 library to utilize the Raspberry Pi Camera Module.

#### **Prerequisites**

- Raspberry Pi 4 (or another compatible Raspberry Pi model)

- Raspberry Pi Camera Module (v1, v2, or HQ)
- MicroSD card (with Raspberry Pi OS installed)
- Power supply for the Raspberry Pi
- Internet connection
- Display and input devices (optional, for local setup)

### Step 1: Connect the Camera

1. Power off your Raspberry Pi.
2. Locate the Camera Serial Interface (CSI) port on the Raspberry Pi board.
3. Attach the camera ribbon cable to the CSI port:
  - Ensure the metal contacts on the cable face the metal contacts inside the CSI port.
  - Secure the connection by locking the CSI port's clip.
4. Place the camera on a stable surface to avoid damage during setup.

### Step 2: Enable Camera Support

1. Power on your Raspberry Pi.
2. Open a terminal or connect via SSH.
3. Run the Raspberry Pi configuration tool:

```
sudo raspi-config
```

4. Navigate to Interfacing Options > Camera and select Enable.
5. Reboot your Raspberry Pi:

```
sudo reboot
```

### Step 3: Update the System

Update your Raspberry Pi OS to ensure you have the latest software:

```
sudo apt update && sudo apt upgrade -y
```

### Step 4: Install Picamera2

The Picamera2 library provides Python bindings for controlling the Raspberry Pi Camera.

1. Install Picamera2 and its dependencies:

```
sudo apt install -y python3-picamera2 libcamera-dev
```

2. Optionally, install GUI dependencies if you plan to preview camera output:

```
sudo apt install -y python3-pyqt5
```

### Step 5: Test the Camera

1. Verify the camera is detected:

```
libcamera-hello
```

This command should display a preview window if the camera is working properly.

2. Run a basic Python script:

```
python3  
  
from picamera2 import Picamera2  
  
picam2 = Picamera2()  
  
picam2.start_preview()  
  
picam2.capture_file("test.jpg")  
  
picam2.stop_preview()  
  
exit()
```

Check if test.jpg is saved in your working directory.

### Step 6: Troubleshooting

- Camera Not Detected: Double-check the ribbon cable connection and ensure the camera module is enabled in raspi-config.
- Library Errors: Ensure all dependencies are installed. Re-run `sudo apt update` && `sudo apt upgrade` if needed.
- No Preview: Verify your display is working and connected properly.

### Additional Resources

- Official Raspberry Pi Camera Documentation:  
<https://www.raspberrypi.com/documentation/accessories/camera.html>
- Picamera2 GitHub Repository:  
<https://github.com/raspberrypi/picamera2>

## 3.2.5 Instructions to Make our microcontroller Program Run on Bootup

### Prerequisites

1. Python Environment: Ensure Python 3 is installed on your Raspberry Pi.

```
python3 --version
```

2. Required Libraries: Install the necessary Python libraries using pip.

```
pip install paho-mqtt certifi opencv-python numpy
```

3. Program Location: Save your program file (e.g., rpi4\_main.py) in a directory.

### Step 1: Test the Program

1. Run the program manually to ensure it works correctly:

```
python3 /home/<your_username>/Desktop/RPi4-Painting-MC/rpi4_main.py
```

2. Verify that MQTT connects, messages are sent, and the program behaves as expected.

### Step 2: Configure the Program for Bootup

Using systemd (Recommended for Reliability)

#### 1. Create a Service File:

```
sudo nano /etc/systemd/system/mqtt_script.service
```

#### 2. Add the Following Configuration:

[Unit]

Description=Run MQTT Script

After=network-online.target

Wants=network-online.target

[Service]

ExecStart=/usr/bin/python3 /home/<your\_username>/Desktop/RPi4-Painting-MC/rpi4\_main.py

WorkingDirectory=/home/<your\_username>/Desktop/RPi4-Painting-MC

Restart=always

StandardOutput=append:/home/<your\_username>/startup.log

StandardError=append:/home/<your\_username>/startup.log

KillSignal=SIGTERM

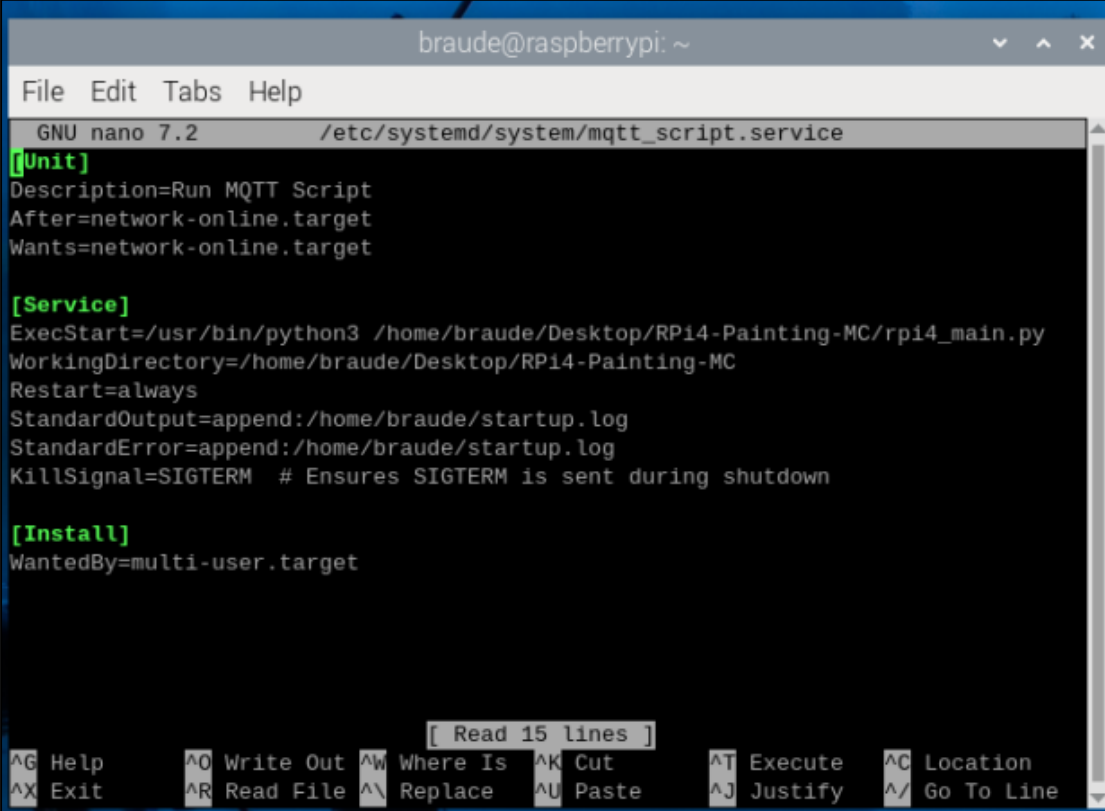
[Install]

WantedBy=multi-user.target

#### 3. Enable and Start the Service:

```
sudo systemctl enable mqtt_script.service
```

```
sudo systemctl start mqtt_script.service
```



```
braude@raspberrypi: ~  
File Edit Tabs Help  
GNU nano 7.2 /etc/systemd/system/mqtt_script.service  
[Unit]  
Description=Run MQTT Script  
After=network-online.target  
Wants=network-online.target  
  
[Service]  
ExecStart=/usr/bin/python3 /home/braude/Desktop/RPi4-Painting-MC/rpi4_main.py  
WorkingDirectory=/home/braude/Desktop/RPi4-Painting-MC  
Restart=always  
StandardOutput=append:/home/braude/startup.log  
StandardError=append:/home/braude/startup.log  
KillSignal=SIGTERM # Ensures SIGTERM is sent during shutdown  
  
[Install]  
WantedBy=multi-user.target  
[ Read 15 lines ]  
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location  
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line
```

#### 4. Test the Service:

- Check **Figure 27: mqtt\_script file.** the status:

```
sudo systemctl status mqtt_script.service
```

- Check the log file for output:

```
cat /home/<your_username>/startup.log
```

- Reboot the Raspberry Pi and ensure the service starts:

```
sudo reboot
```

#### Step 3: Debugging and Verification

1. If the program doesn't run, check the log file for errors:

```
cat /home/<your_username>/startup.log
```

2. Ensure the program path is correct in the crontab or systemd service file.

#### Notes

- To stop the service if needed:

```
sudo systemctl stop mqtt_script.service
```

- After bootup , use command `pgrep -f rpi4_main.py` to check if process is running.

It will print the process id

Your program should now run automatically on boot and handle shutdown signals gracefully.

### 3.2.6 Backend and Frontend Installation Guide:

#### Prerequisites

##### 1. Install Node.js and npm:

- Download and install the latest LTS version of Node.js from the [Node.js official website](#).
- Verify the installation by running the following commands:

```
node -v
```

```
npm -v
```

- Ensure npm (Node Package Manager) is installed along with Node.js.

##### 2. Install Git:

- Download and install Git from the [Git official website](#).
- Verify the installation:

```
git --version
```

##### 3. Install a Code Editor (optional but recommended):

- Install [Visual Studio Code](#) or your preferred code editor.

##### 4. Clone or Obtain the Project:

- Get the project source code either by cloning the repository or downloading the zip file.

#### Step-by-Step Installation

##### 1. Clone the Project Repository:

- Open a terminal or command prompt.

- Clone the repository (replace <repository-url> with the actual URL):

```
git clone <repository-url>
```

- Navigate to the project directory:

```
cd <project-folder>
```

## 2. Install Dependencies:

- Navigate to the backend (Node.js) directory and install dependencies:

```
cd backend
```

```
npm install
```

- Navigate to the frontend (React.js) directory and install dependencies:

```
cd ../frontend
```

```
npm install
```

## 3. Run the Backend:

- Start the backend server from the backend directory:

```
node ./index.js
```

## 4. Run the Frontend:

- Start the frontend development server from the frontend directory:

```
npm start
```

## 5. Access the Application:

- Open your web browser and navigate to the default development server URL:
  - Frontend: <http://localhost:3000>

## 3.2.7 Database Installation Guide

### 1. Download MongoDB:

- Visit the [MongoDB Community Server Download Page](#).
- Select your OS, version (recommended: latest stable release), and package (.msi for Windows).

## 2. Run the Installer:

- Double-click the downloaded .msi file.
- Follow the setup wizard:
  - Choose "Complete" setup type.
  - Opt to install MongoDB as a Windows Service (recommended) during installation.
- Select the option to install **MongoDB Compass**, a GUI for MongoDB (optional but recommended).

## 3. Set Up MongoDB:

- After installation, MongoDB will be configured to run as a Windows Service by default.
- Start MongoDB using the command:

```
net start MongoDB
```

- Stop MongoDB using:

```
net stop MongoDB
```

## 4. Verify Installation:

- Open a terminal or command prompt and run:

```
mongod --version
```

- If MongoDB is running as a service, you can connect to it using:

```
mongo
```

## 3.2.8 Roboflow's Machine Learning Model Integration:

### 1. Make an account at Roboflow

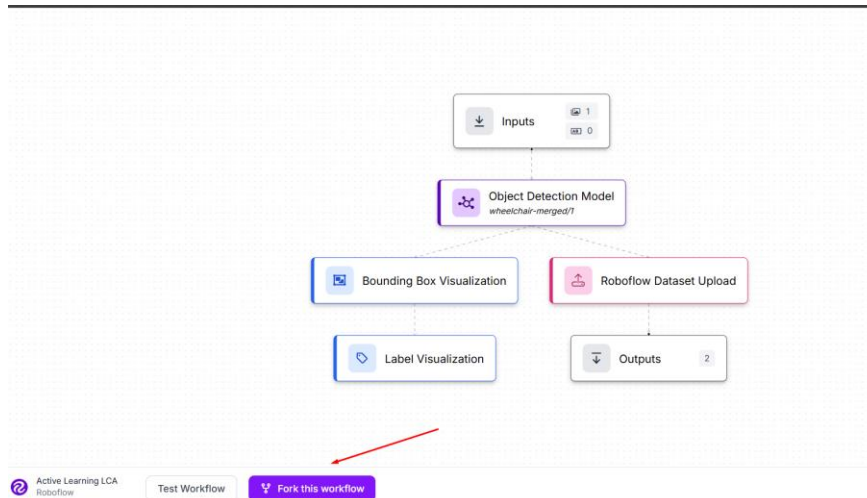
<https://app.roboflow.com/login>

### 2. Fork the Machine Learning Model of this Project

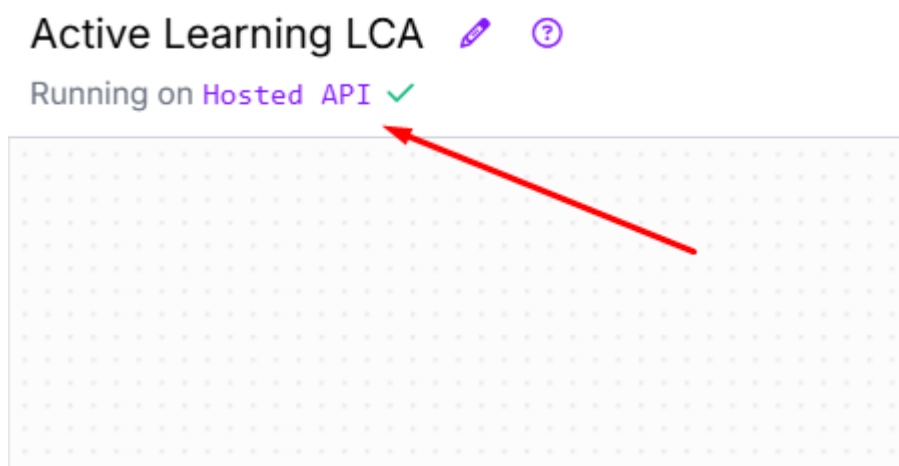
#### 1. Go to Model Link: [click here](#)




## 2. Fork this workflow




## 3. Click on hosted API





## 4. Connect to Server




Choose an Inference Server

 Hosted API

 Dedicated Deployment

 Localhost

 Other

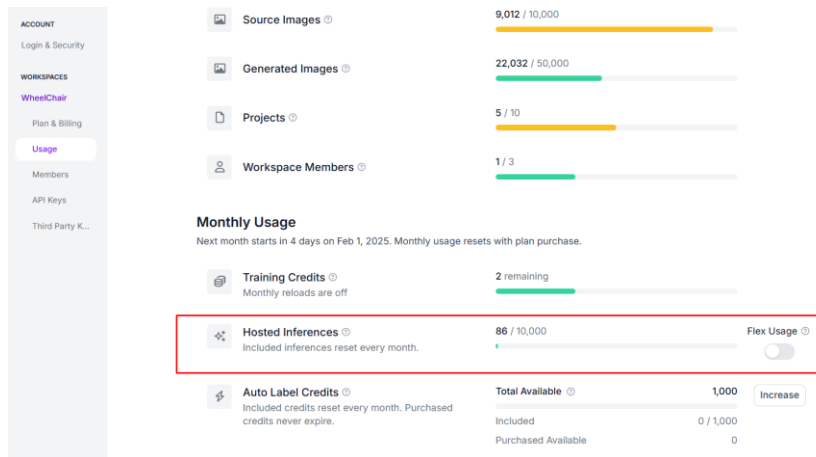
Use Roboflow's Cloud to build and run your workflow.  
Supports basic blocks running on images (no GPU or video). Auto-scales up to infinity and down to zero.  
You only pay for what you use.

Server URL

https://detect.roboflow.com

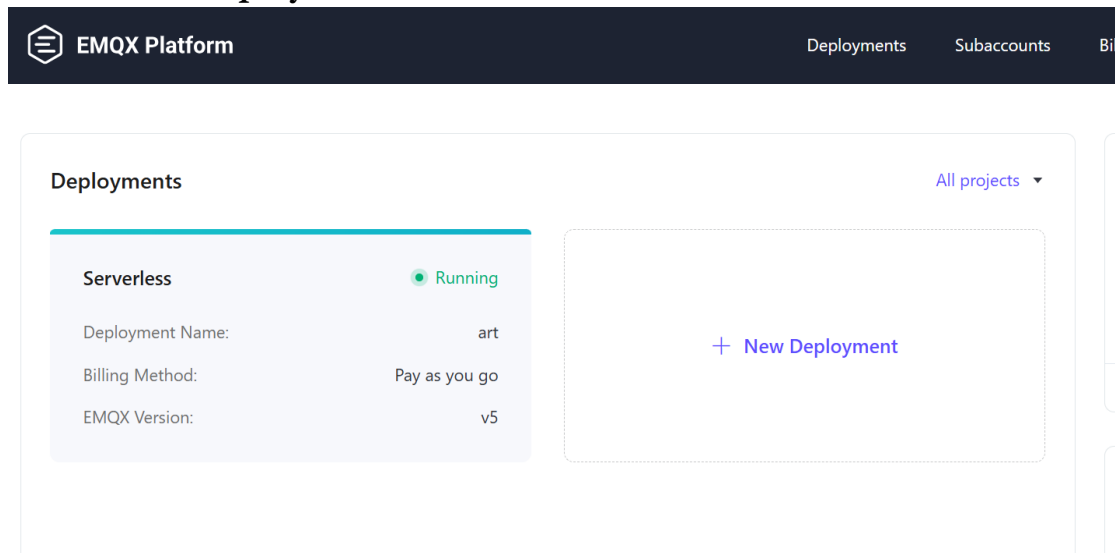
Connect

## 5. Upgrade to Roboflow subscribed account to host the Model on cloud.



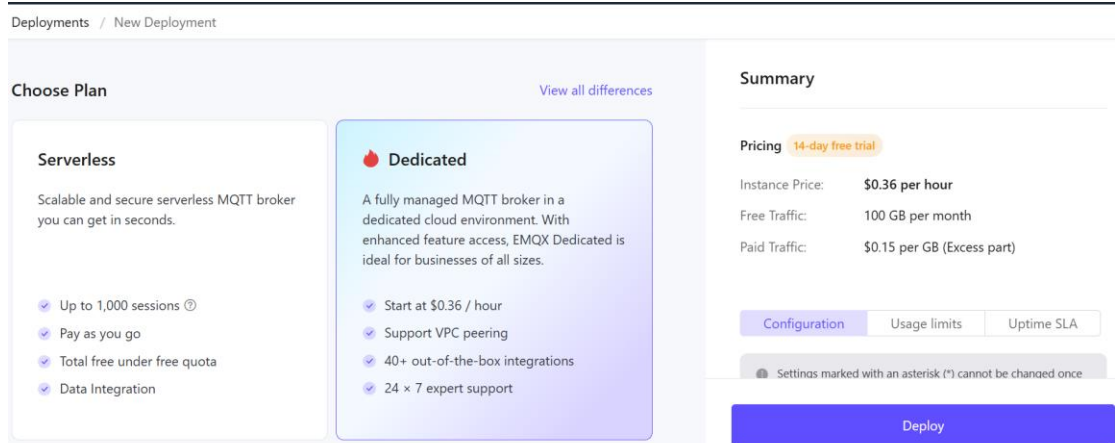
### 3.2.8 MQTT Broker Integration:

1. Register on emqx: [click here](#)
2. Start new deployment



### 3. Choose Plan

Its recommended to pay for subscriptions on to go with Serverless for testing and development



### 4. Get the mqtt info from Project overview dashboard

Overview

Access Control

Monitor

Data Integration

Diagnose

### MQTT Connection Information

Address: j81f31b4.ala.eu-central-1.emqxsl.com

MQTT over TLS/SSL Port: 8883

WebSocket over TLS/SSL Port: 8084

CA Certificate: [↓](#) CA Certificate Expiration: 2031.11.10

## 5. Integrate the MQTT server details in Backend and Rpi project code

### - Backend code:

Open backend project and go to services -> mqttService.js

Around line 30, change the credentials of mqtt service from emqx as shown in step 4

```

30 class MQTTService extends IMQTTService {
31     constructor() {
32         super();
33         this.mqttClient = mqtt.connect('mqtt://j81f31b4.ala.eu-central-1.emqxsl.com', {
34             username: "art",
35             password: "art123",
36             // clientId: "art_backend111",
37             port: 8883,
38             protocol: 'mqtts',
39         });

```

### - Rpi code:

Open Rpi project and go to rpi4\_main.py file go mqtt\_setup function

change the credentials of mqtt service from emqx as shown in step 4

change the credentials of username

```

416 # MQTT Setup
417 def mqtt_setup():
418     global mqtt_client
419     logging.info('MQTT setup!')
420     # Configure the LWT (Last Will and Testament)
421     try:
422         mqtt_client = mqtt.Client(client_id=f"esp32_{hex(int(time.time() * 1000))[2:]}", protocol=mqt
423         mqtt_client.username_pw_set("art", "art123")

```

And change credentials of address from step 4 in mqtt\_client.connect

```

        try:
            logging.info(f"Attempt {attempt + 1}: Connecting to MQTT broker...")
            mqtt_client.connect("j81f31b4.ala.eu-central-1.emqxsl.com", port=8883)
            mqtt_client.loop_start()
            logging.info("MQTT connected successfully.")
            break
        except Exception as e:

```

### 3.3 Overview of the System's Process Workflow - Sequence diagram

The real-time painting height adjustment system dynamically adjusts the height of a painting based on the face level of each visitor. The process involves multiple components, including a proximity sensor, camera, microcontroller, motor, and cloud services, working together to ensure optimal viewing conditions. The sequence of interactions begins when a visitor approaches the exhibit and continues through adjusting the painting's height, maintaining the optimal position, and resetting once the visitor leaves.

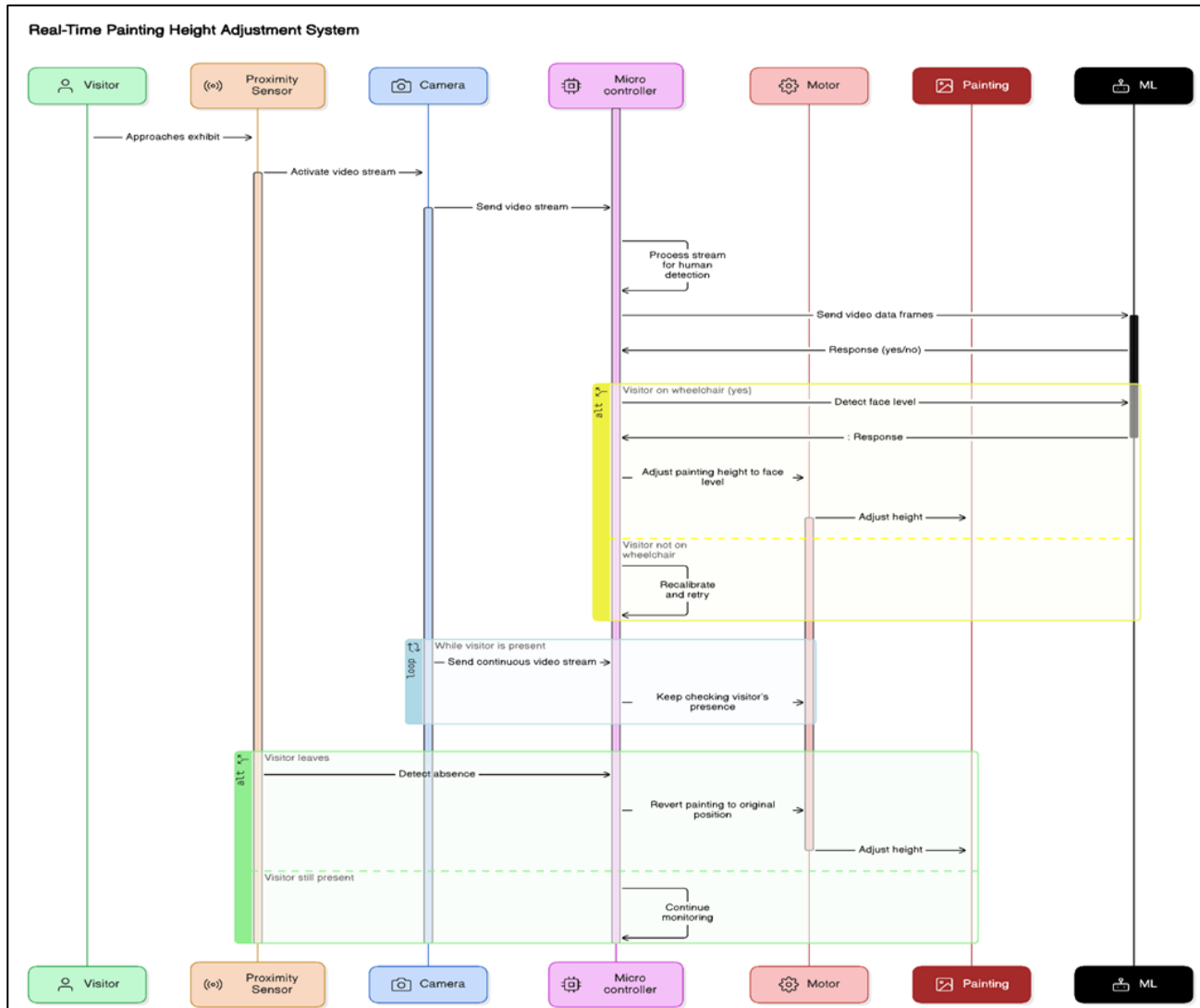


Figure 28: Sequence diagram.

## 3.4 Database Description

- **User Table:** Stores user credentials and roles.

Fields:

username (PK): Unique login name for the user.  
password: User's login password.  
role: User's assigned role (e.g., admin, worker).


User	
username 	varchar
password	varchar
role	varchar

Table 1: User Table.

- **Painting Table:** Stores detailed information about each painting.

Fields:

- sys\_id (PK): Unique identifier for the painting.
- name: Name of the painting.
- painter\_name: Name of the painter.
- base\_height: Base height of the painting.
- height: Current height of the painting.
- width: Width of the painting.
- weight: Weight of the painting.
- microcontroller: Microcontroller linked to the painting.
- status: Painting status (Active or Inactive).
- sensor: Indicates if the painting has a sensor.
- wheelchair: Whether wheelchair accessibility is supported.
- height\_adjust: Whether height adjustment is enabled.
- photo: Binary photo data of the painting.
- lastUpdated: Timestamp of the last update.
- created: Timestamp of creation.

Painting	
sys_id 	int
name	varchar
painter_name	varchar
base_height	int
height	int
width	int
weight	int
microcontroller	varchar
status	varchar
sensor	boolean
wheelchair	int
height_adjust	boolean
photo	blob
lastUpdated	datetime
created	datetime

Table 2: Painting Table.

- **PaintingStats Table:** Tracks viewing statistics and engagement metrics for paintings.

Fields:

- sys\_id (PK): Unique identifier for the stats entry.
- painting\_id: Identifier referencing the associated painting.
- totalViews: Total number of views recorded.
- totalViewDuration: Total duration (in seconds) of all views.
- averageViewDuration: Average duration (in seconds) per view.
- viewingSessions: JSON array data of individual viewing session details.
- dailyStats: JSON array data summarizing daily view statistics.


PaintingStats	
sys_id 	int
painting_id	int
totalViews	int
totalViewDuration	int
averageViewDuration	int
viewingSessions	json
dailyStats	json

Table 3: PaintingStats Table.

## 4. References

1. United Nations (2006), "Convention on the rights of persons with disabilities", available at: [www.un.org/disabilities/convention/conventionfull.shtml](http://www.un.org/disabilities/convention/conventionfull.shtml)
2. Packer, J.: Beyond learning: Exploring visitors' perceptions of the value and benefits of museum experiences. *Curator: The Museum Journal*, 51(1), pp. 33-54, (2008).
3. Wintzerith, S.: "Inclusive without Knowing It." In *The New Museum Community: Audiences, Challenges, Benefits*, edited by Nicola Abery, pp. 458–76. Edinburgh: MuseumsEtc, (2013).
4. Borg, J., Lindstrom, A., Larsen, S.: "Assistive technology in developing countries: a review from the perspective of the convention on the rights of persons with disabilities", *Prosthetics and Orthotics International*, Vol. 35No. 1, pp. 20-29, (2011).
5. Salmen, J. P.: *Everyone's Welcome: The Americans with Disabilities Act and Museums*. American Association of Museums, 1575 Face St., NW., Suite 400, Washington, DC 20005, (1998).