HARD & SOFT 2025

BUCHAREST 3 TEAM
POLITEHNICA UNIVERSITY OF BUCHAREST

FINAL DOCUMENTATION

STUDENTS

MARIUS-SEBASTIAN ANTACHE
REMUS-IOAN COMAEGA
VICTOR-STEFAN FLORESCU
MIHAELA POPESCU-STEFAN

COACHES

AS. DRD. ING. DIANA BAICU
DR. ING. MIHAI CRACIUNESCU

Table of Contents

1. Introduction 3
2. Principles and Theory 3
   2.1. Mecanum Wheels Movement 3
   2.2. Robot Operating System 5
   2.3. Graph Traversal Algorithms 5
3. Preliminary work 7
4. The Maze 7
   4.1. Navigation 10
   4.2. Exploration 10
   4.3. Glass Detection 10
   4.4. Bumper Detection 12
   4.5. Alcohol Detection 13
   4.6. Exit Detection 13
   4.7. Energy Management 14
   4.8. Testing Methodology 14
5. Architecture 15
   5.1. Hardware Development 15
   5.2. Software Architecture 18
      5.2.1. ROS Architecture 18
      5.2.2. Maze Solver 19
      5.2.3. Backend & Cloud Architecture 20
      5.2.4. Frontend Interaction 21
      5.2.5. Interconnection 21
   5.3. GUI 22
6. Security Measures & Ethical Hacking 22
   6.1. Threat Analysis and Security Vulnerabilities Considered 24
7. Task Allocation 25
8. Conclusions 26
9. Appendix – Bill of Materials 27

1. Introduction

This year's challenge required us to develop a robot car capable of navigating a maze using autonomous-driving technology. Across the labyrinth, various obstacles and speed bumps can be encountered. Additionally, the robot must be capable of detecting the presence of alcohol and magnetic fields. All data collected from the maze, along with the robot's position, must be transmitted to an application running on a server.

For this task, the MentorPi robot kit has been provided, which is equipped with a Raspberry Pi 5, a motor driver and sensor data acquisition board, an RGB camera, and a LiDAR. Furthermore, additional sensors such as an ultrasonic distance sensor, vibration sensors, alcohol sensor and a Hall sensor have been provided. Integration of these components is required, and no additional essential parts, such as processing units or replacement sensors, should be used to complete the tasks.

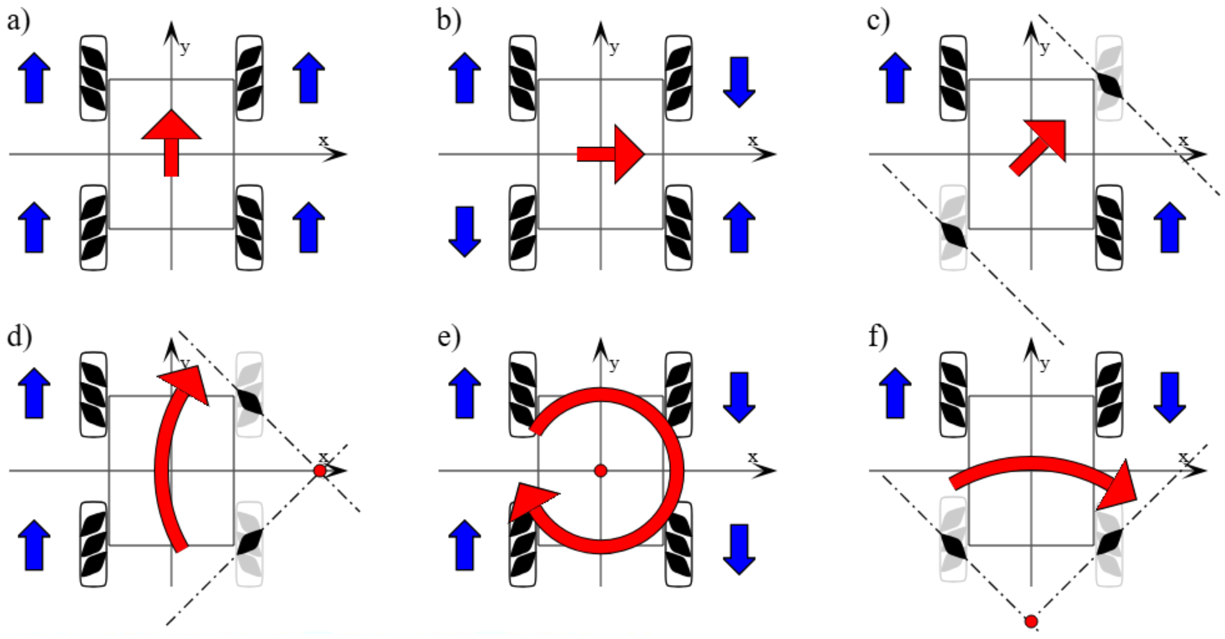
The maze is a structure with wooden walls, of 20 cm in height, and contains both true and false exits. A true exit is identified by the presence of two magnets on the side of the walls, while false exits lack such markers. As the robot navigates through the maze, it may encounter multiple glasses filled with liquid. Some of these glasses contain water, while others contain alcohol. The robot must accurately detect and differentiate between them, transmitting their positions to the server for proper storage and analysis.

2. Principles and theory

2.1. Mecanum Wheels movement

The MentorPi's chassis is based on omni-wheels (also known as Mecanum wheels), which allow the robot to move in any direction – forward, backward, sideways, and diagonally. The omni-directional movement capability is enabled first of all by their structure, being composed of angled rollers which rotate freely along the circumference, and by the fact that each wheel is controlled independently by a motor.

These characteristics provide enhanced maneuverability in restricted spaces and allow for complex movements like spinning and lateral shifts, without requiring a minimum turning radius (as it happens with Ackerman steering, for example). The image below depicts the wheel control and the resulting directions of the robot, as well as the movement equations:



$$V_A = V_X - V_Y - V_\omega * \left(\frac{1}{2}L + \frac{1}{2}H\right) \quad , \quad V_B = V_X + V_Y + V_\omega * \left(\frac{1}{2}L + \frac{1}{2}H\right)$$

$$V_C = V_X - V_Y - V_\omega * \left(\frac{1}{2}L + \frac{1}{2}H\right) \quad , \quad V_D = V_X + V_Y + V_\omega * \left(\frac{1}{2}L + \frac{1}{2}H\right)$$

Figure 1. Omni-wheels movement characteristics

The formulas above describe the individual wheel velocities for a mecanum-wheeled robot. The terms Vx and Vy represent the robot's desired linear velocities in the forward/backward and lateral directions respectively, while Vω represents its desired angular velocity (rotation around its center). The expression (1/2L+1/2H) accounts for the distance from the robot's center to each wheel, combining its length L and width H. Each wheel's velocity is calculated by combining the translational components (Vx and Vy) with a rotational component derived from Vω, where the sign and direction of each term reflect the specific placement and orientation of the wheel. This set of equations is used in inverse kinematics to determine how fast each wheel should spin to achieve a given motion vector, making omnidirectional movement possible.

The movement of the robot is controlled through the ROS holonomic controller: the high-level position and rotation displacement in the world's coordinates is translated into the necessary speeds for each individual wheel. The movement equations mentioned above were implemented within this controller.

2.2. Robot Operating System

ROS 2 (Robot Operating System 2) is a modular middleware framework designed to support the development of scalable, real-time, and distributed robotic systems. At its core, ROS 2 provides a decentralized communication infrastructure using a Data Distribution Service (DDS)-based publish-subscribe model, allowing nodes to exchange data asynchronously through topics, or synchronously via services and actions. Nodes are grouped into packages that contain specific functionality such as sensing, control, or planning. Having a widespread community and support, the packages for most sensors or actuators are available as open-source software (created either by individuals or by manufacturing companies), or they can be developed from scratch when required.

While nodes can publish or subscribe to topics, the underlying DDS handles discovery, serialization, and transport — enabling inter-node communication even across different machines or networks. Additionally, ROS 2 can integrate with powerful tools like rviz2 for visualization, and ros2bag for data logging, which have been used in the current project to display, check and verify the data from the integrated sensors, as well as the output of the processing algorithms.

2.3. Graph traversal algorithms

To navigate a maze, a robot first needs to represent the physical space as an abstract graph, where key locations become nodes and connecting paths become edges. Using its sensors, especially the LiDAR, camera and encoders, the robot divides the space into a grid of cells, which can be connected or not to one-another. The task of navigating the maze, meaning to decide which path to explore from the starting location with the goal of finding the end location, can then be translated into a problem of defining a strategy for moving around in the available space. Two algorithms are described below, namely Breadth First Search (BFS) and Depth First Search (DFS).

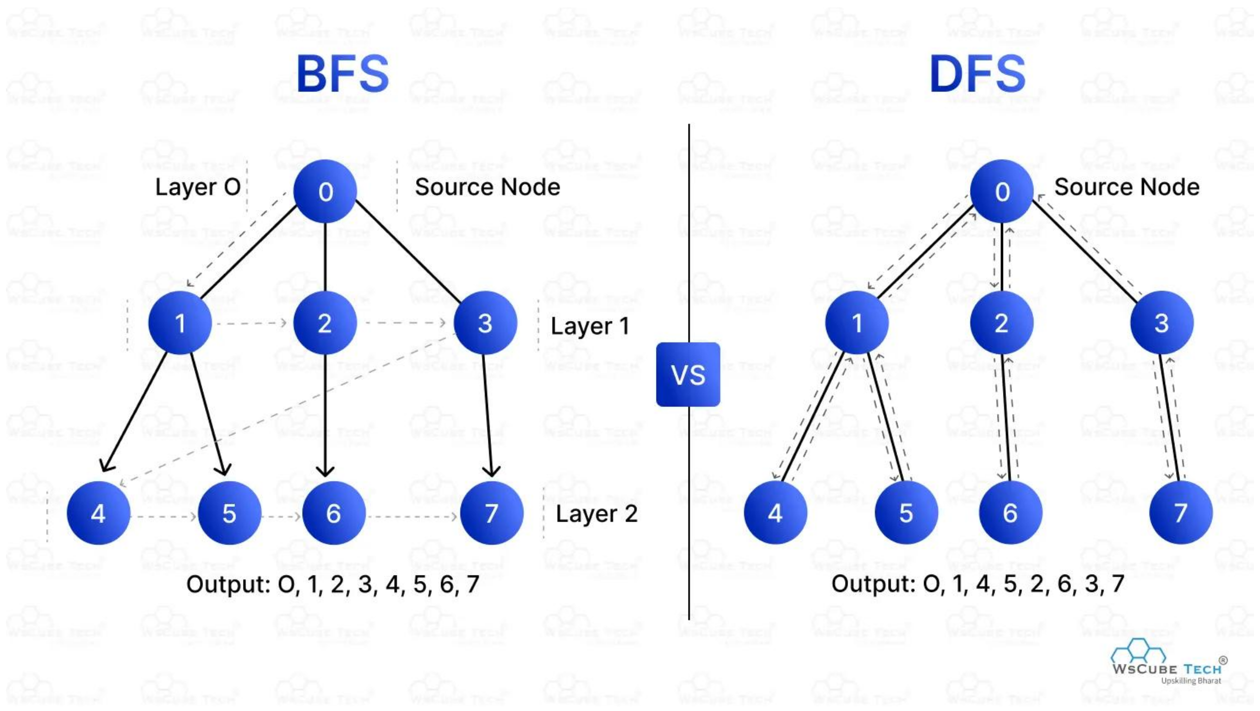


Figure 2. The major types of graph traversal algorithms: BFS and DFS

Breadth First Search (BFS)

BFS is a graph traversal algorithm that explores the graph in layers. Starting from a source node, it visits all its immediate neighbors before moving on to nodes at the next depth level. It uses a queue to manage the traversal order, ensuring that nodes are explored in the order they were discovered. BFS is especially useful for finding the shortest path in an unweighted graph, performing level-order traversal in trees, or exploring all reachable states in state-space problems. Due to its nature, BFS guarantees the minimum number of steps to reach a node, but can be memory-intensive on large or wide graphs.

Depth First Search (DFS)

DFS explores as far as possible along a branch before backtracking. Starting from a node, it recursively visits an unvisited adjacent node, continuing this path until it reaches a dead-end, then backtracks to explore other unexplored branches. DFS typically uses a stack, either implicitly via recursion or explicitly with a stack data structure. DFS is well-suited for problems like topological sorting, cycle detection, maze solving, and exploring connected components in a graph. It is generally more memory-efficient than BFS but does not guarantee the shortest path unless all edge weights are equal and controlled traversal is used.

Given the nature of the maze completion, as well as the energy efficiency requirement, while both methods have been tested, the DFS algorithm has been considered as a navigation strategy in order to find the right exit as fast as possible.

3. Preliminary work

The team started with an in-depth evaluation of the provided car kit and sensors. The documentation of the MentorPi car kit revealed several ROS2 launch configurations, enabling SLAM procedures, velocity control of the car using the microROS expansion board, a sensor fusion node merging information from wheel encoders, LiDAR and IMU for increased pose estimation and autonomous navigation capabilities using the nav2 stack (the navigation package included in ROS2).

Initial testing using the ROS navigation stack revealed unsatisfactory results, the nav2 nodes needing precise parameter tuning for costmap creation (the function that defines obstacles and influences the way the robot avoids them) and path generation inside the closed environment of the maze.

Given the structured nature of the maze, a custom set of navigation functions, which are going to be described in the sections below, were implemented, using the odometry derived information and the LiDAR data.

Initially, the graphical user interface (GUI) was served directly from the main processing server to ensure maximum performance and minimal latency, which was particularly useful during development and testing phases, where speed was critical. However, as the system evolved, scalability and flexibility became a priority. To address this, the GUI was decoupled from the processing backend and deployed on a self-hosted web server with a self-signed certificate, allowing for modular deployment and isolated maintenance. For secure and publicly accessible deployment, a reverse SSH tunnel was later introduced to route traffic through a trusted proxy server. This proxy serves the interface over a CA-signed HTTPS connection, ensuring encrypted communication and certificate validation on the client side, while preserving the secure internal structure of the original architecture.

4. The Maze

Given that the width of the corridor is known, for processing, the maze was divided into cells. The starting position of the robot is designated as cell number 0, located at coordinates (0,0).

The first approach for reproducing a maze was to combine a series of boxes to create corridors, as in the image below. While this method allowed for fast reconfiguration, it occupied significant extra space and limited the dimension of the maze, while also not replicating the real-world conditions, like materials and structure.

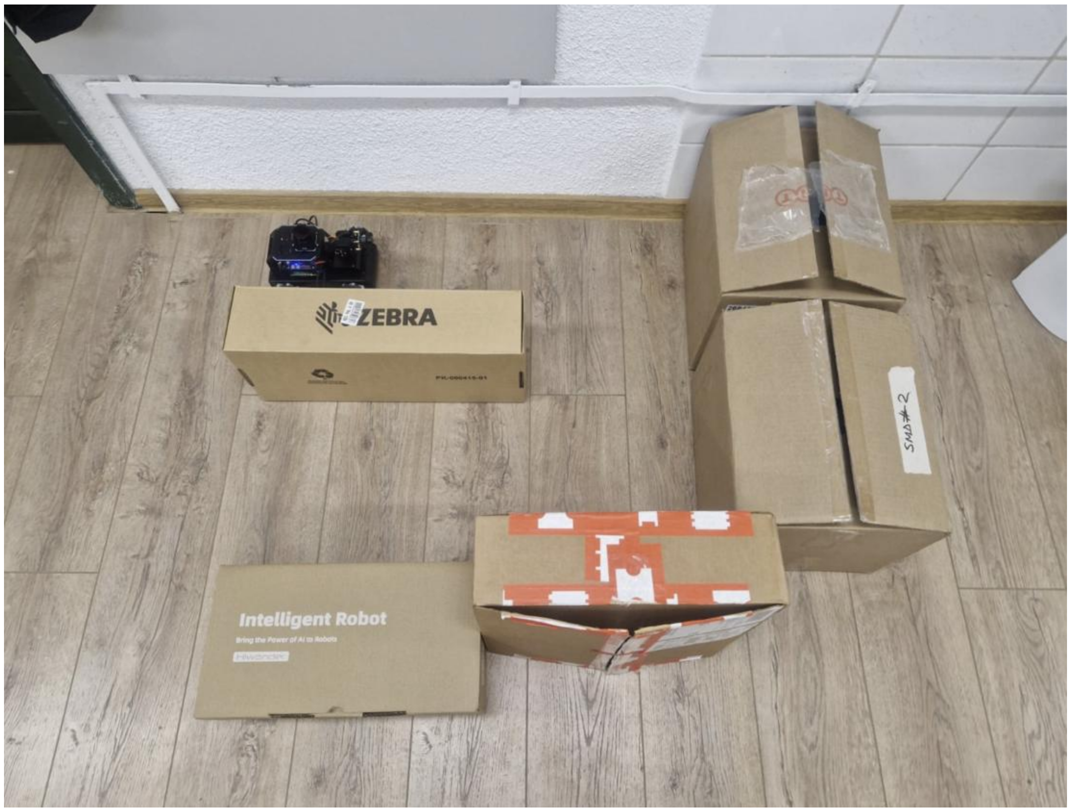


Figure 3. Initial maze made out of boxes

In order to test the system in real world conditions, a maze was then designed in a 3D design software (Solidworks) and imported in the Gazebo simulator within ROS. The maze and its dimensions can be observed in the image below. The layout was configured in order to integrate some of the commonly found structural characteristics of mazes: dead ends, intersections, straightaways, etc. This strategy allows for improved system robustness and edge case detection during development.

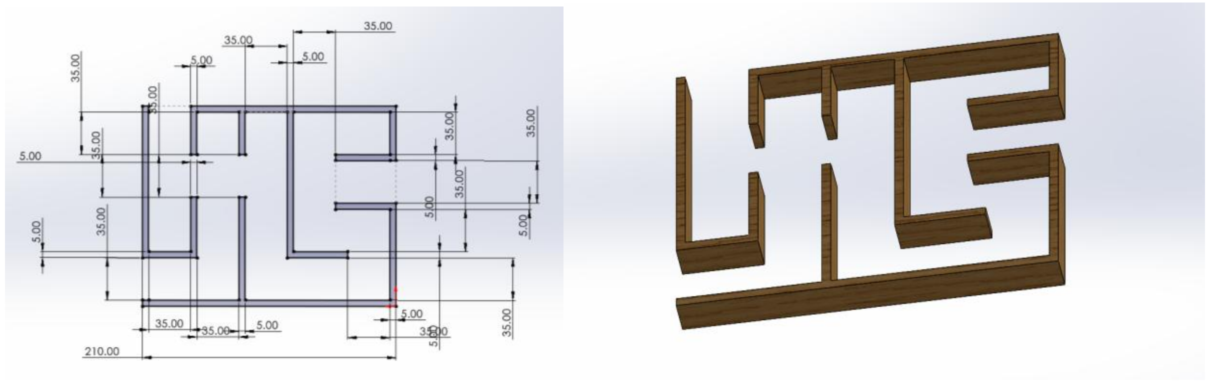


Figure 4. Solidworks design of maze

Based on the design, materials were procured to replicate the design in the lab. The maze was manufactured using wood planks, cut into 35 cm pieces. This was done at a wood shop, as it can be seen in the image below.

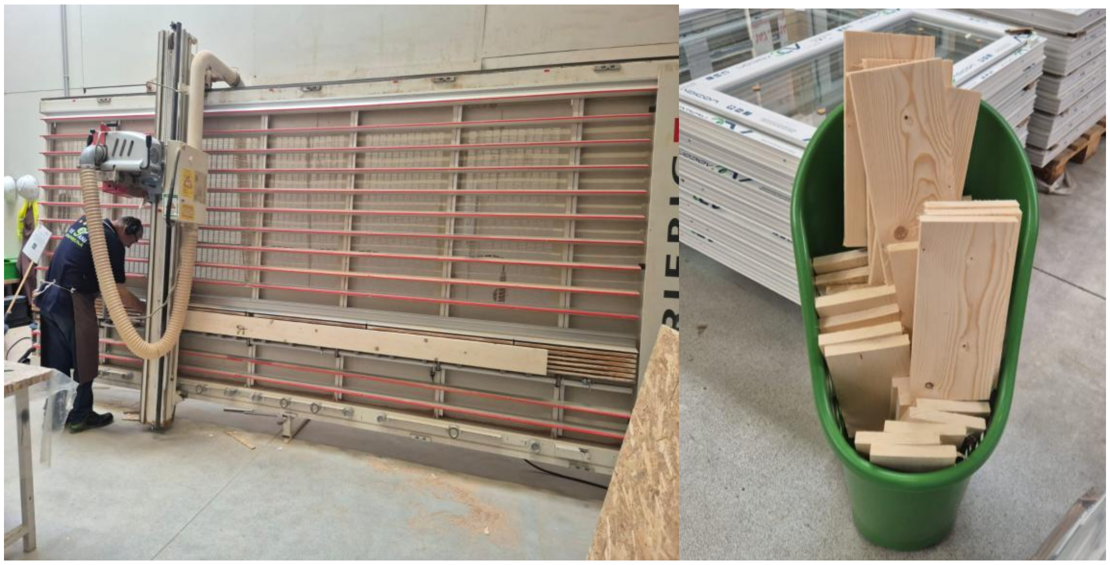


Figure 5. Manufacturing of building blocks for maze design

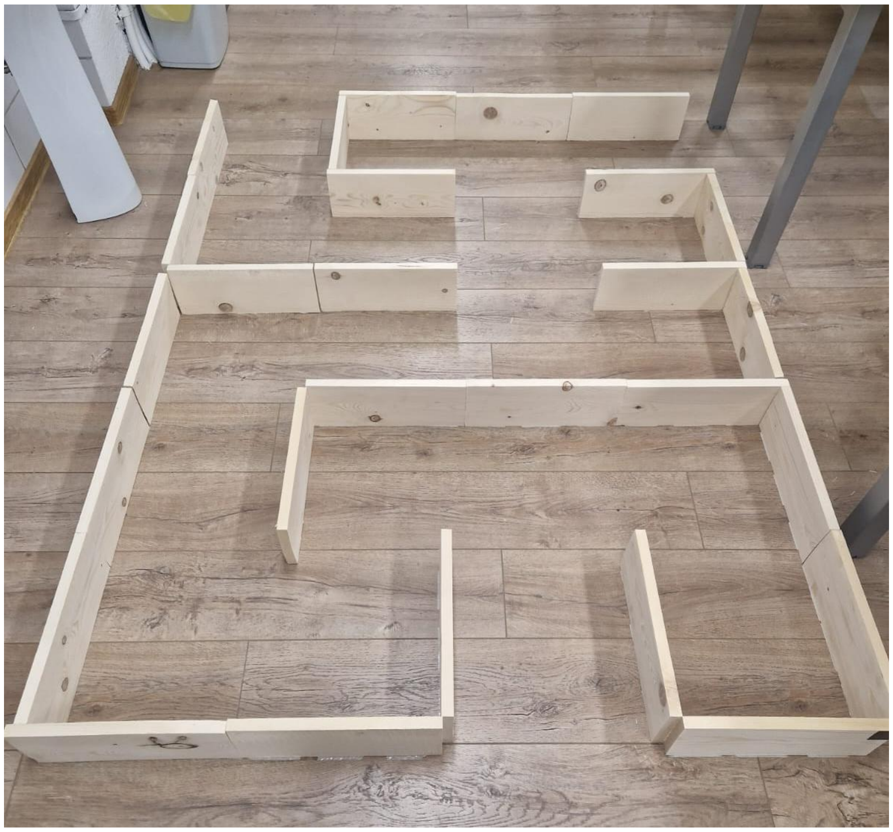


Figure 6. Maze assembly inside the lab

4.1. Navigation

The navigation logic uses the cell structure to choose one between 4 possible neighbors of the current cell as the next navigation refrence. The current cell in which the robot is located is calculated by dividing the current pose by the known cell size of 35cm.

The walls are used during navigation to reposition the car in the middle of each visited cell, correcting accumulated error in the odometry. More information about this topic can be found in the section "***Software architecture***".

4.2. Exploration

While BFS is preferred for exploring the maze completely, the imposed time limit for the challenge dictates that the DFS approach is preferable.

4.3. Glass Detection

The glass detection system is designed to operate in real time using the camera mounted on the robot. Images captured by the camera are sent to a dedicated server, where a custom-trained YOLO model (You Only Look Once) is run. More specifically, the *yolov11n* architecture was used as it is the smallest and fastest base yolo model available. This has been specifically trained on the Stakan1 dataset from universe.roboflow.com, which contains 2851 unique photos of different types of glasses filled with water at varying levels. On top of that, the images have varying backgrounds and lighting conditions. All of these variations present in our dataset can help the model better generalize for unknown shapes of glasses.

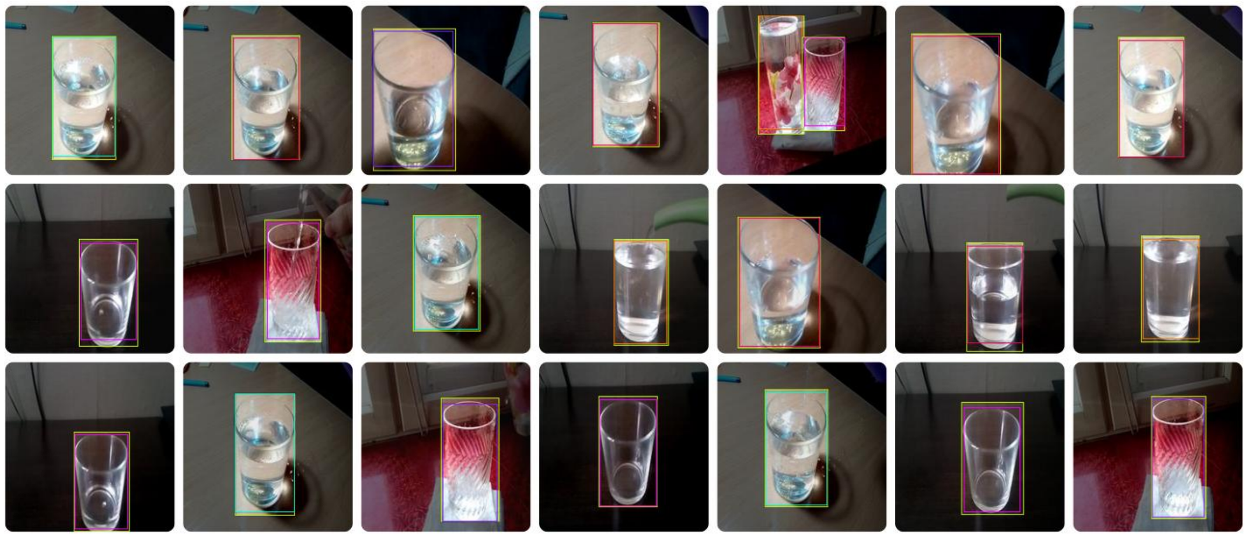
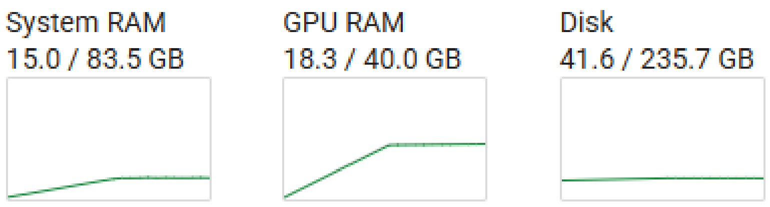


Figure 7. Glass dataset

The final glass detection model was trained using a Google Colab virtual machine, with an Nvidia A100 GPU with 40 GB of VRAM. The best performing model for glass recognition and localization was achieved after a training run of 200 epochs and a batch size of 300 images, taking up 18 GB of VRAM.



System RAM
15.0 / 83.5 GB

GPU RAM
18.3 / 40.0 GB

Disk
41.6 / 235.7 GB

Figure 8. Resource utilization on the Google Colab infrastructure

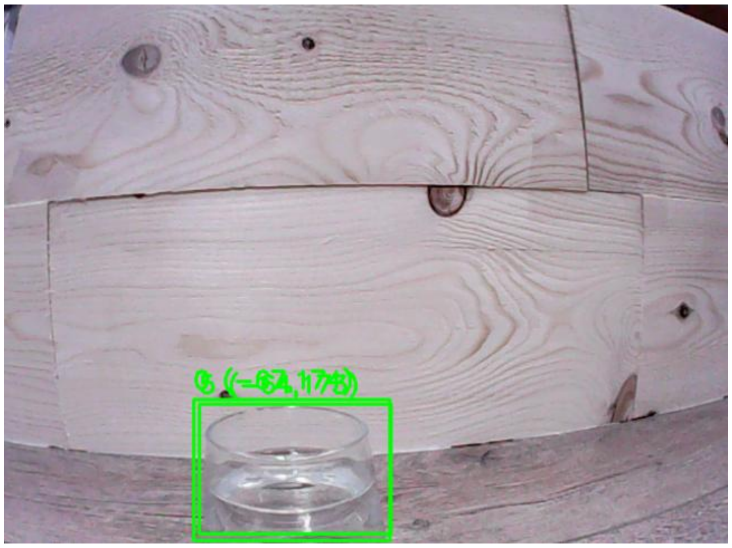


Figure 9. Model inference on the live feed of the robot camera

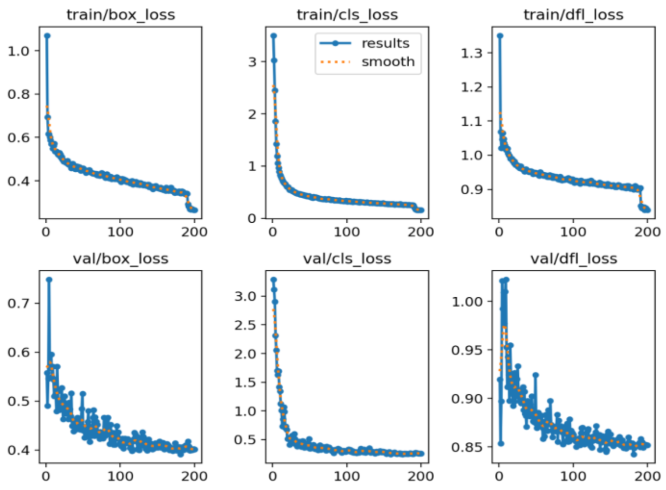


Figure 10. Loss value minimization over 200 epochs

Once the server processes the image, it returns a detection signal to the client, indicating whether a glass object has been identified. This result is then handled by a bridge node in the robot's ROS architecture. The bridge node publishes the detection result as a message on a specific topic, making it accessible to any other node in the system.

For instance, the node responsible for navigation and obstacle avoidance subscribes to this topic. Upon receiving a message that confirms the presence of a glass, it can adapt its path planning algorithm to navigate around the obstacle safely. This modular design allows multiple functional nodes to react appropriately to the detection, ensuring coordinated and autonomous behavior of the robot in dynamic environments. The coordinates of the detected glass are placed in a list inside in order to avoid detection loops and assure every glass is checked only once.

4.4. Bumper Detection

The figure below demonstrates a simulated bumper detection system using data from the on-board IMU (Inertial Measurement Unit). The top graph shows the simulated pitch angle of the vehicle as it moves over speed bumps. The pitch angle rises sharply as the front wheels climb the bumper, remains elevated while the vehicle is on the bump, and then drops quickly as the rear wheels descend. The bottom graph represents the vertical acceleration, calculated as the derivative of the pitch angle. This results in two distinct spikes: a positive spike during the sharp upward motion and a negative spike during the descent, clearly identifying the transition over the bumper.

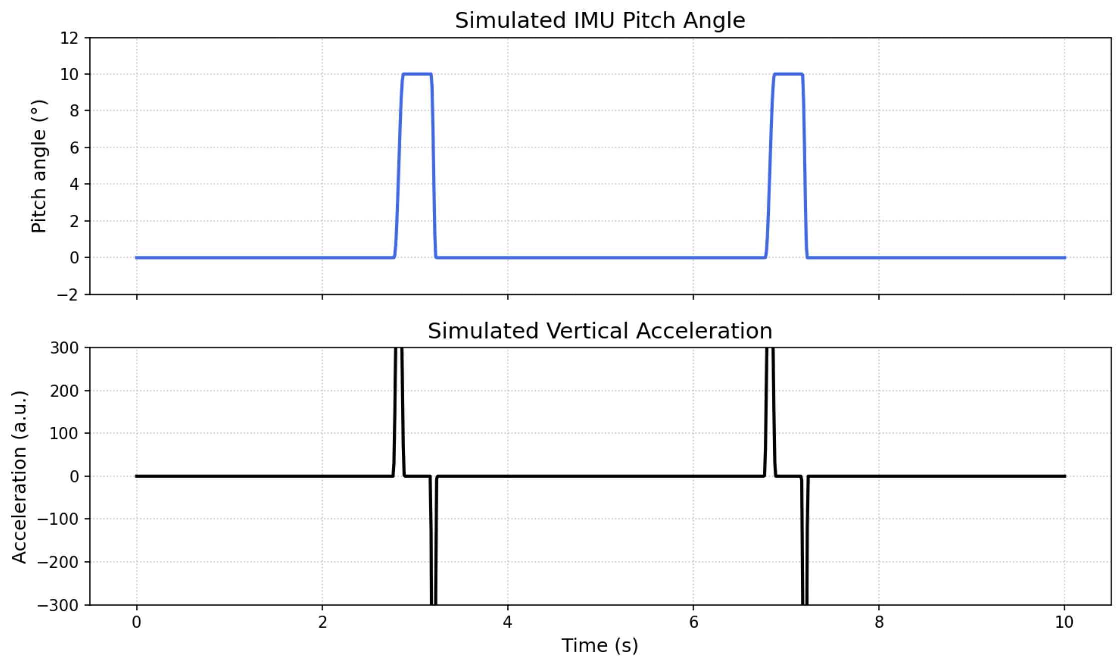


Figure 11. IMU data when simulating a speed bump passing

This method of detecting events through sudden angular changes and their derivatives can be generalized to detect any type of vibration or abrupt motion, not just speed bumps. In practice, the IMU data is cross-checked with a dedicated vibration sensor (such as the provided SW-420 sensor), which is highly sensitive to even small oscillations. When both sensors detect synchronized spikes, the system confirms that an actual vibration event has occurred, enhancing robustness and reducing false positives in motion or impact detection scenarios.

4.5. Alcohol Detection

Once the glass is detected, rotation is controlled in order for the robot to face the glass. The distance to the glass is then tuned using the ultrasonic distance sensor.

With the alcohol sensor positioned above the liquid, multiple measurements are taken to determine the presence of alcohol. A mounted fan ensures air flow towards the sensor, providing an increased detection range - up to 20% more.

4.6. Exit Detection

The detection of the exit from the maze is implemented by verifying a lidar scan between -90 deg and 90 deg relative to the car's forward position. Once all measurements are above a certain threshold, the cell is marked as an exit cell.

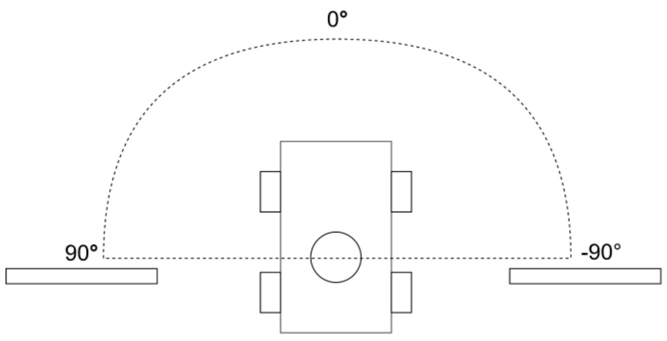


Figure 12. Lidar area scanned for exit check

Differentiating between the correct and false exits is then implemented with a separate procedure in which the car reverses and aligns with the closest parts of the walls by scanning 2 narrow lidar cones on the left and right of the car until the surfaces are detected. Finally, the robot will move until a certain distance close to the right wall , followed by continuous readings of the hall sensors for a predetermined amount of time. If the Hall sensor reads a magnetic field, then the exit will be marked as the true one.

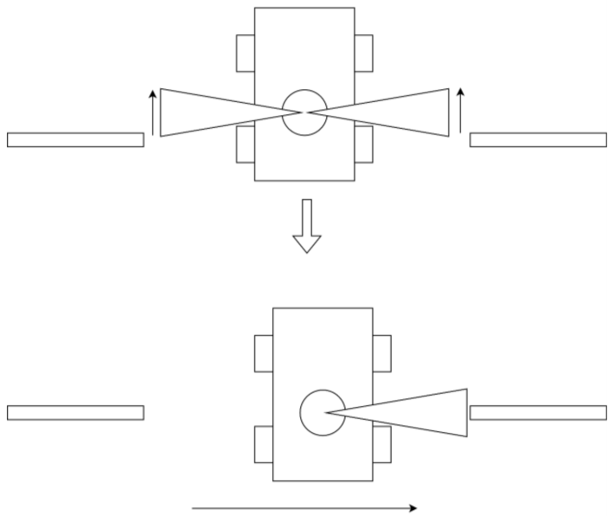


Figure 13. Lining up procedure with the exit magnet

4.7. Energy Management

The power consumption of our system is estimated to be between 22W and 35W, providing an autonomy of 24 to 42 minutes. This estimation is based on both the typical usage and maximum load scenarios for each of the components used in the system. In the worst-case scenario, the robot can operate for over 20 minutes on a full battery. This simulation involves continuous movement, sensor data processing, and communication with the server. The values have been obtained by doing measurements using the lab equipment, providing a better understanding of robot performance. In real life conditions, the autonomy of the robot is estimated to be around 30 to 35 minutes, more than satisfactory to complete the task at hand.

4.8. Testing methodology

Given that only one run is permitted in the demonstration round, emphasis was placed on robustness during testing. For integration of a new component that modifies physical behavior, the robot was required to finish 5 runs without errors. The consecutive runs also allow for parameter tuning (such as proportional terms or distance thresholds) based on observations.

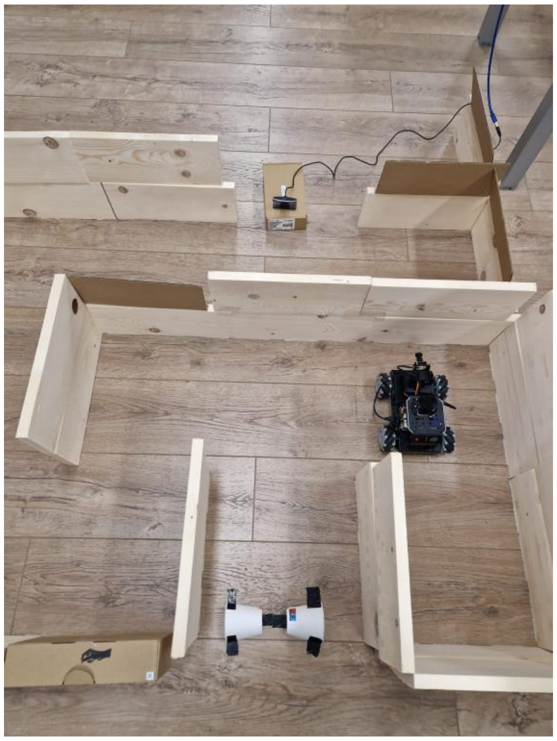


Figure 21. Robot during a demo run in the lab created maze

5. Architecture

5.1. Hardware Development

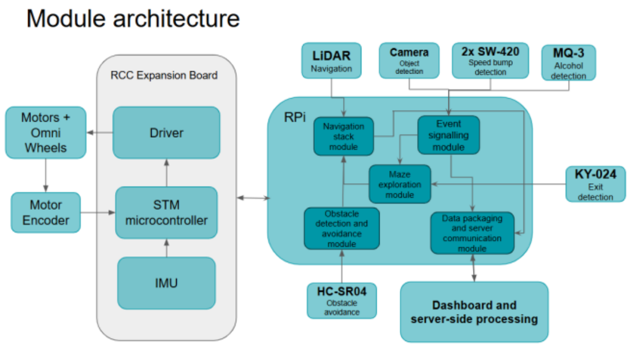
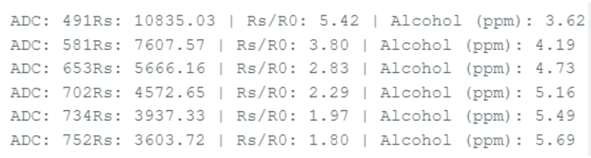


Figure 14. Module Architecture

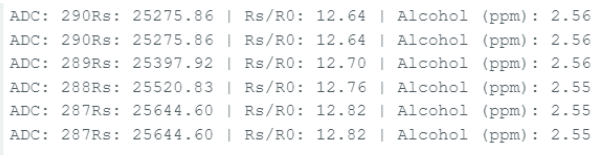
The main hardware challenges stemmed from the architectural limitations of the Raspberry Pi 5 and the calibration of the analogue sensors. Given that the provided development board lacks a built-in analog-to-digital converter (ADC), the analog outputs of the MQ-3 gas sensor and KY-024 linear magnetic Hall sensor could not be read directly. To calibrate the sensors, they were interfaced with an Arduino board, and for their final integration, the output read from the built-in comparator was used.

Another constraint was the limited number of supply lines on the Raspberry board, which required a careful allocation of power resources. Fortunately, both the vibration and Hall sensors were compatible with 3.3V and 5V supply levels, providing flexibility in terms of pin mapping. For interfacing the HC-SR04 ultrasonic sensor, which outputs a 5V logic level on the echo pin, a voltage divider circuit was implemented on a breadboard with two resistors, which was also useful in the power distribution.

Among the provided sensors, the MQ-3 gas sensor imposed the most demanding calibration. Due to its internal heating element, it required a burn-in period of over 24 hours during the first days to achieve stable and reproducible readings - around 2.5ppm for clean air and 7.6 for the exposure to isopropyl alcohol. Using the sensor's datasheet, the R0 resistance was computed under known conditions by taking average readings.



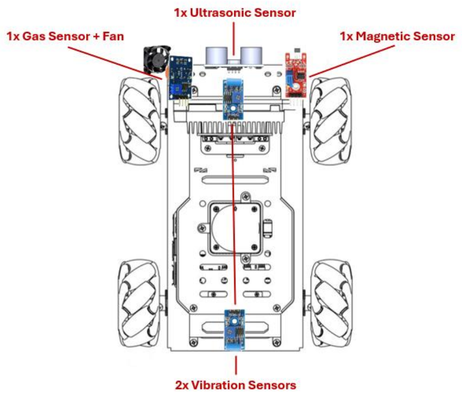
```
ADC: 491Rs: 10835.03 | Rs/R0: 5.42 | Alcohol (ppm): 3.62
ADC: 581Rs: 7607.57 | Rs/R0: 3.80 | Alcohol (ppm): 4.19
ADC: 653Rs: 5666.16 | Rs/R0: 2.83 | Alcohol (ppm): 4.73
ADC: 702Rs: 4572.65 | Rs/R0: 2.29 | Alcohol (ppm): 5.16
ADC: 734Rs: 3937.33 | Rs/R0: 1.97 | Alcohol (ppm): 5.49
ADC: 752Rs: 3603.72 | Rs/R0: 1.80 | Alcohol (ppm): 5.69
```



```
ADC: 290Rs: 25275.86 | Rs/R0: 12.64 | Alcohol (ppm): 2.56
ADC: 290Rs: 25275.86 | Rs/R0: 12.64 | Alcohol (ppm): 2.56
ADC: 289Rs: 25397.92 | Rs/R0: 12.70 | Alcohol (ppm): 2.56
ADC: 288Rs: 25520.83 | Rs/R0: 12.76 | Alcohol (ppm): 2.55
ADC: 287Rs: 25644.60 | Rs/R0: 12.82 | Alcohol (ppm): 2.55
ADC: 287Rs: 25644.60 | Rs/R0: 12.82 | Alcohol (ppm): 2.55
```

Figure 15. MQ-3 sensor output during calibration time and after stabilization

To ensure an efficient prototyping and lay out sensors placement in case of possible conflicts, a model of the robot chassis was created, along with relevant environmental references. The initial arrangement proved to hinder the execution of certain routines such as taking turns, avoiding obstacles or positioning in the proximity of the glasses. As a result, a revised arrangement involved a nested fixture of the ultrasonic sensor and its alignment on the median axis with the gas sensor and the LIDAR, as can be observed in the images below.



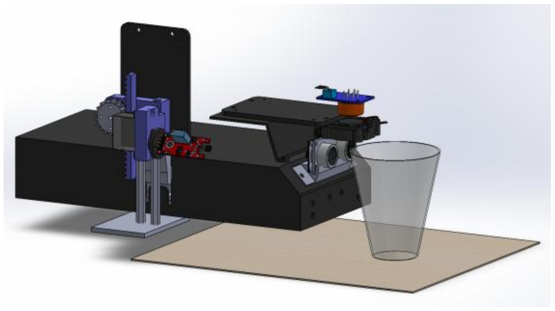


Figure 16 a. Preliminary positioning of the sensors b. CAD Hardware Model

Since the placement and the polarity of the magnet that defines the final exits were unknown, a rack-pinion mechanism was designed in order to ensure the height of the Hall sensor is adjustable. In order to bring minimal changes to the hardware provided, this system was designed around the two servomotors that were used for the pan and tilt movement of the camera. Moreover, to achieve a successful measurement, an adapter piece between the servo shaft ending and the Hall sensor was modelled, which can rotate the sensor during the exit detection routine at 180 degrees. For the linear movement to be transmitted, the geared mechanism was paired with two guide rails, represented by two metal distancers, to ensure stability and proper coupling. The assembly also required a base fixture for the distancers and a vertical mount to adhere to the robot's chassis.

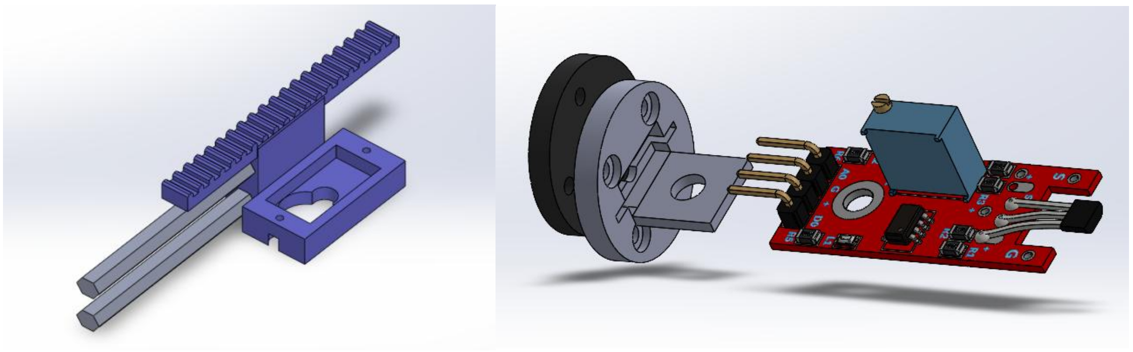


Figure 17. Hall Sensor Positioning Mechanism Modeled Parts

Several other components were designed using an educational license of Solidworks, such as a vibration sensor mount, an ultrasonic sensor holder and distancers for the gas sensor. The above-mentioned parts were 3D printed using PLA on a desktop FDM printer, ensuring rapid prototyping and iterative design improvements throughout the integration process.

Figure 18. The process of realizing the 3D printed parts

5.2. Software Architecture

The diagram below illustrates the interaction between the backend, ROS2 components, and the frontend — effectively summarizing the full architecture:

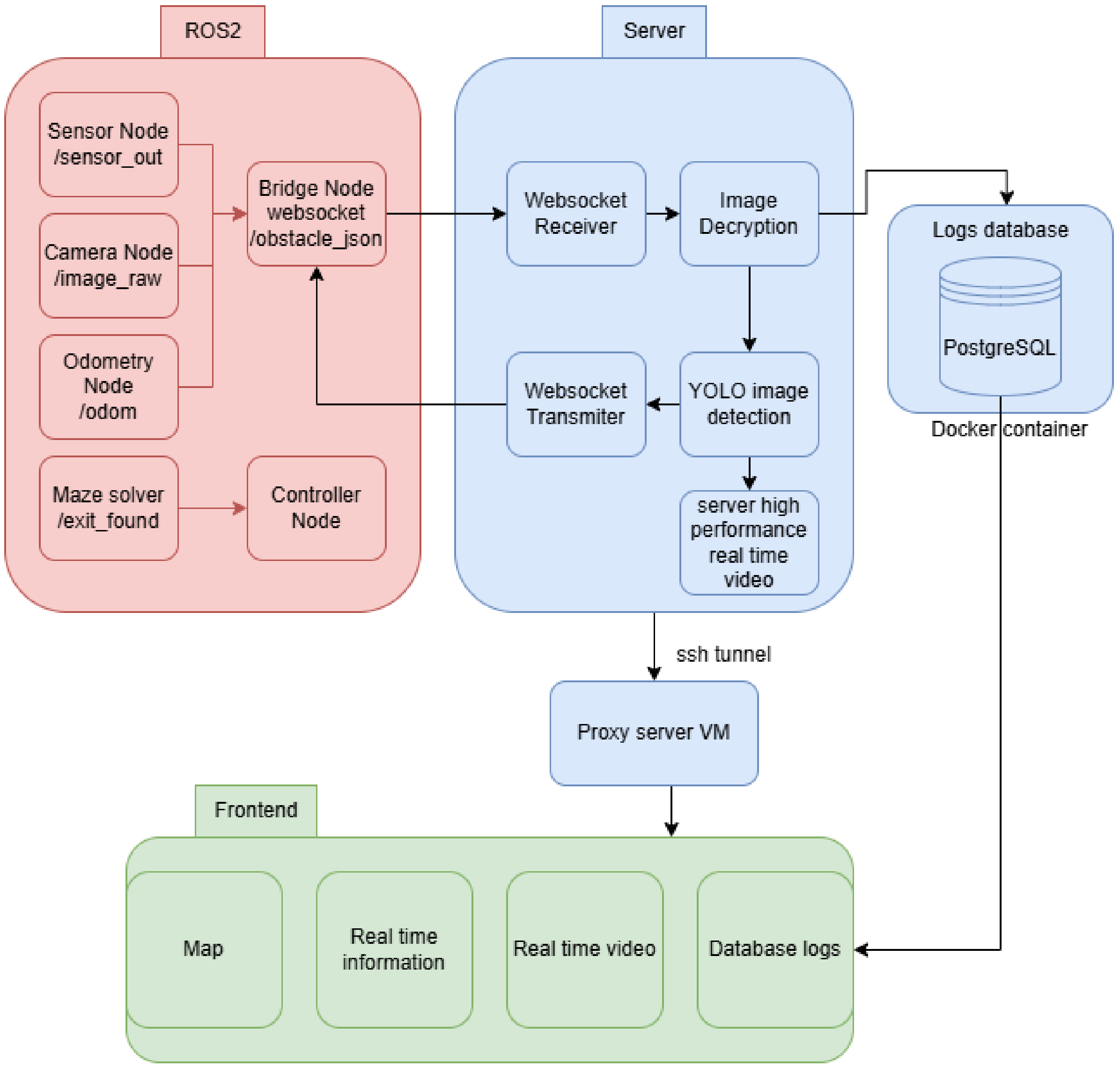


Figure 19. Software components and their interaction

5.2.1. ROS Architecture

Each node in ROS2 is a process that performs a specific task. Nodes communicate by publishing or subscribing to topics, which are named channels over which messages are sent. A publisher node sends data to a topic, while a subscriber node listens for data on that topic. This enables the possibility to build decentralized and modular architectures with relative ease, especially regarding inter-process communication.

In our setup:

- Sensor Node (/sensor_out): Publishes environmental and system-level sensor data, such as gas detection, vibration, or magnetic field readings.
- Camera Node (/image_raw): Publishes raw camera frames that can later be used for image-based inference or steganographic encoding.
- Odometry Node (/odom): Publishes the robot's position and movement information, which is critical for trajectory tracking and map reconstruction.
- Maze Solver Node (/exit_found): Indicates whether the robot has found a predefined target (e.g., an exit in a maze), contributing to the decision-making process

All of these nodes send their data to a central **Bridge Node**, which consolidates the information and encodes it into an image for transmission. The Bridge Node then acts as a WebSocket client, sending a JSON-encoded message (/obstacle_json) through a secure connection to the backend server.

The Controller Node consumes data from multiple topics to guide the robot's decisions and send actuation commands, closing the loop between perception and control.

5.2.2. Maze solver

The maze solver process contains the logic for exploration and navigation and obstacle callbacks. Each of the procedures mentioned run iteratively on separate timers. with only one of the timers being active at once.

Exploration is achieved with a custom implemented DFS algorithm. The main control loop searches for clear cells using the lidar and marks visited cells. Paths are being stored in a stack, so that if a dead end is found, backtracking on these stored paths may occur until an unvisited cell is found. Once the decision regarding the next cell has been made, control is passed to the navigation procedure.

The navigation procedure contains several control algorithms for reaching the center of the desired cell, while correcting accumulated odometry error. These algorithms include:

- A **proportional regulator** for facing towards the next desired cell, with the error being represented by the difference between current yaw angle and the reference yaw angle.
- A **second proportional regulator** for placing the robot equally distant in regards to the walls. If only one wall is present, the car follows the detected wall 17.5 cm.

- A **wall locking procedure** that is enabled if the next cell has a forward facing wall relative to the, enabling the car to stop at 17.5 cm away from the wall.
- A **gap clearing regulator** that, if a gap is detected in the left or right wall during forward motion, enables the robot to place itself so that the gap is fully visible by the lidar.

5.2.3. Backend & cloud architecture

The backend is modular, efficient, and secure. It was built from the ground up to ensure full control over its architecture and to optimize overall system performance. At a high level, the server manages real-time communication between the frontend and the robot, and also handles image processing using a custom-trained YOLO model designed specifically for this project.

The system architecture is organized as follows: the robot, which runs on ROS 2, includes a Bridge Node that collects data from various onboard sensors, encodes this data into images using steganographic techniques (see the Steganography section for details), and transmits the resulting payloads over a secure, self-signed WebSocket connection to the backend server. Upon reception, the server decrypts the embedded information, decodes the data, and stores it into a PostgreSQL database hosted within a Docker container on the same machine. This containerized setup enhances both security and portability.

Once the data is stored, it is updated in real time and made available via the /get endpoint, allowing the frontend to retrieve the most recent information efficiently. For maximum performance and frame rate consistency, a real-time video feed of the robot is displayed directly on the server side, bypassing the need for streaming over the network.

The frontend, communicating with the backend over HTTPS, receives and displays an integrated view of the robot's state, including the latest image, live telemetry such as sensor readings and odometry data, detected events, and historical logs fetched from the database. This data is presented to the user in an accessible and structured format.

In addition to communication and visualization, the backend is also responsible for dynamic map generation. Using odometry data, the server reconstructs the robot’s trajectory and estimates the presence and location of surrounding walls, rendering them into a visual map that evolves in real time as the robot explores its environment.

The backend is implemented in Python and uses multithreading to ensure efficient parallel handling of tasks, such as WebSocket communication, data decryption, image processing, and database operations. User authentication is handled via a third-party OAuth2 provider (Google Sign-In), ensuring secure access without the need to manage password storage locally, thereby reducing the attack surface of the system.

5.2.4. Frontend Interaction

The frontend is a web-based interface that interacts with the backend over HTTPS. It serves several roles:

- **Map**: Displays a reconstructed spatial map of the robot's environment, generated from odometry data and obstacle estimation performed by the server.
- **Real-time Information**: Presents live telemetry such as sensor readings, event detections, and position updates in a clean, human-readable format.
- **Real-time Video**: Streams a high-performance video feed sourced directly from the robot, rendered locally on the server and accessible through the frontend.
- **Database Logs**: Enables users to browse historical logs stored in the PostgreSQL database, facilitating debugging, analytics, or offline review.

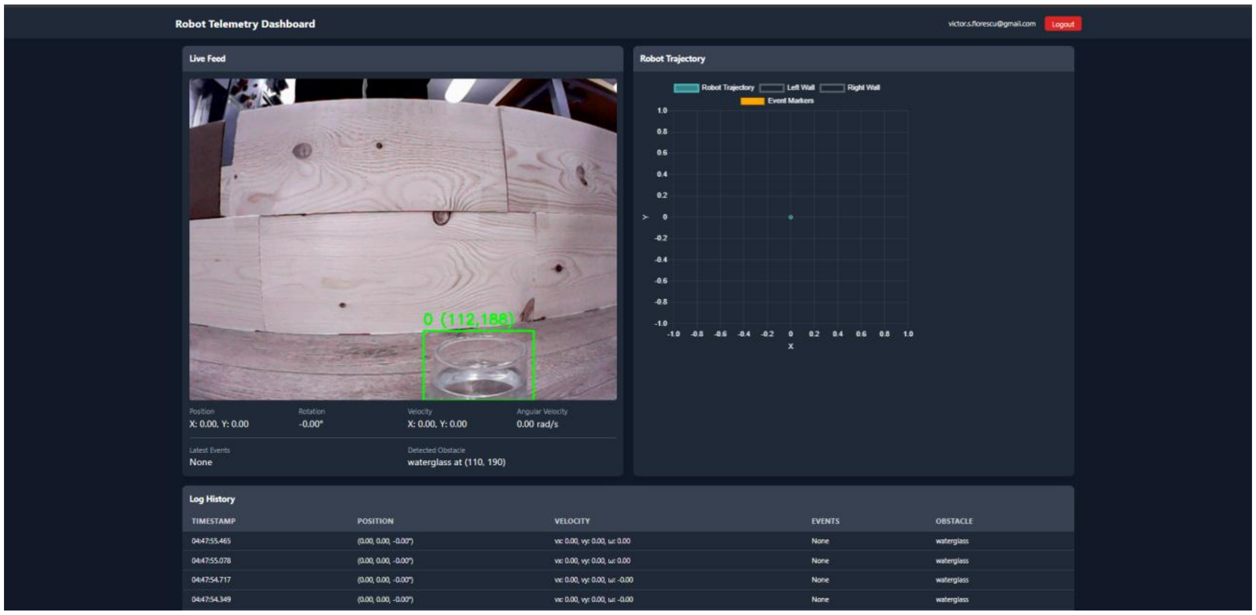
5.2.5 Interconnection

This architecture demonstrates a clear separation of tasks and activities:

- ROS 2 handles the data generation and low-level robotic behavior.
- The server performs secure communication, data decryption, real-time processing, inference, and storage.
- The frontend presents all relevant data in an intuitive interface for human interaction.

By encoding messages from ROS 2 into images and transmitting them via WebSocket, the system ensures efficient data bundling and avoids the complexity of managing multiple concurrent communication channels.

5.3. GUI



Position
X: 0.00, Y: 0.00

Rotation
-0.00°

Velocity
X: 0.00, Y: 0.00

Angular Velocity
0.00 rad/s

Latest Events
None

Detected Obstacle
waterglass at (110, 190)

Log History

TIMESTAMP	POSITION	VELOCITY	EVENTS	OBSTACLE
04:47:55.465	(0.00, 0.00, -0.00°)	vx: 0.00, vy: 0.00, ω: 0.00	None	waterglass
04:47:55.078	(0.00, 0.00, -0.00°)	vx: 0.00, vy: 0.00, ω: 0.00	None	waterglass
04:47:54.717	(0.00, 0.00, -0.00°)	vx: 0.00, vy: 0.00, ω: -0.00	None	waterglass
04:47:54.349	(0.00, 0.00, -0.00°)	vx: 0.00, vy: 0.00, ω: -0.00	None	waterglass

Figure 20. The Graphical User Interface

The Graphical User Interface is a robot telemetry dashboard designed for **real-time monitoring** of robotic perception and motion. On the left, it displays a live video feed from the robot's camera, with visual object detection overlays—here, a water glass is detected and highlighted using bounding boxes and coordinates. Below the video feed, numerical telemetry such as position, velocity, rotation, and detected events are shown dynamically.

On the right, the robot's trajectory graph visualizes its path in a 2D space, including walls and event markers for spatial context. The log history table at the bottom records time-stamped data entries, providing a structured view of past positions, velocities, and events, enabling traceability and debugging. The interface integrates responsive design, secure Google login (shown at the top), and clearly separates telemetry from visualization, making it effective for both live operation and post-analysis.

6. Security Measures and Ethical Hacking

The security of our system has been a top priority from the ground up, and multiple layers of protection have been implemented to ensure confidentiality, integrity, and controlled access across all components.

Firstly, all sensitive credentials, including API keys, OAuth2 secrets, WebSocket certificates, and encryption keys, are **stored exclusively in environment variables**, never hardcoded in the codebase or exposed in version-controlled repositories. This separation of secrets from code ensures that, even in the event of *source code leakage*, no critical credentials or tokens are compromised. The backend is configured to access these variables at runtime, following industry best practices for secure deployment workflows.

For communication security, **WebSocket transmissions between the robot and the server are encrypted using TLS certificates** issued by a trusted Certificate Authority (CA). This ensures end-to-end encryption with formally validated identities, eliminating the risks typically associated with self-signed certificates. The use of CA-signed certificates guarantees that the encrypted tunnel not only protects the confidentiality and integrity of the transmitted data, but also provides verifiable authenticity of the endpoints involved. This configuration adheres to industry standards for secure communications and significantly strengthens the system's resilience against *interception, spoofing, or man-in-the-middle attacks*.

All frontend–backend interactions are **secured using HTTPS**, which uses industry-standard **TLS encryption** to safeguard data in transit. This guarantees that all communications initiated by the frontend—whether related to user authentication or data retrieval—are encrypted and protected from *passive interception or active tampering*.

Authentication is managed through OAuth2, with Google Sign-In as the identity provider. This delegation removes the need for our system to store, manage, or validate user passwords, substantially reducing the attack surface and mitigating common security risks such as *password leakage, brute-force attacks, credential stuffing, or insecure password storage*. Tokens provided by Google are validated on the backend using Google's public key infrastructure, ensuring both their authenticity and integrity.

Additionally, all backend endpoints enforce strict access control policies. Only authenticated users presenting valid tokens are permitted access to protected resources. For future scalability and granular permission management, a role-based access control (RBAC) layer can be added on top of the OAuth2 identity layer.

Beyond software-level protections, we also employed a **secure networking configuration**. The robot's local server—self-hosted on our network—is connected to a DigitalOcean **proxy server via an SSH tunnel**. This mechanism allows encrypted, authenticated traffic to flow from the robot to the cloud server without exposing any local ports or requiring insecure router-level port forwarding. The SSH tunnel ensures confidentiality and guarantees that only *authorized systems can access the transmission channel*, significantly improving our deployment's security posture.

An additional layer of security is achieved through the use of **steganographic encoding**, which is employed to conceal critical telemetry and event data within image frames captured by the robot. Prior to embedding, each message is encrypted using a hybrid cryptographic scheme: the actual data is encrypted with a symmetric key using the Fernet protocol, while the symmetric key itself is encrypted with RSA using the server's public key. This ensures that only the intended recipient, in possession of the corresponding private key, can decrypt the payload. The fully encrypted message is then uniformly distributed across the image using a Least Significant Bit (LSB) embedding algorithm. By modifying only the least significant bits of pixel values, the algorithm introduces minimal perceptual distortion, making the hidden data virtually undetectable to the human eye. This technique not only secures the message from unauthorized access, but also disguises its presence, thereby adding a layer of camouflage on top of standard encryption. As a result, even *if the transmission is intercepted*, the **combination of encryption and steganography renders the payload unintelligible and non-obvious**, significantly enhancing the overall confidentiality of the data pipeline.

6.2. Threat Analysis and Security Vulnerabilities Considered

Throughout the project, we evaluated and mitigated various potential security breaches, ranging from basic oversights to advanced system-level exploits. Below is a non-exhaustive list of vulnerabilities and how they were addressed:

1. Weak or Default Router Passwords
   Problem: Unauthorized access to the local network.
   Solution: Router credentials were changed, guest networks were disabled, and firewall settings were hardened.

2. Unsecured Local Services (e.g., open ports)
   Problem: Direct access to the robot's internal services from external networks.
   Solution: All local ports were closed from the public internet, with access only available via the SSH tunnel.

3. ROS 2 DDS Discovery Broadcast Leakage
   Problem: ROS 2 may expose topic and node metadata over the network, especially during discovery.
   Solution: We restricted ROS 2 communication to a closed internal subnet and used domain ID isolation to limit cross-device discovery.

4. Default Raspberry Pi Access Point (AP) Exposure
   Problem: Out-of-the-box Raspberry Pi OS may automatically host an access point or leave services like VNC/SSH enabled.
   Solution: We initially disabled and later entirely removed the default access point. Network connectivity was re-established using a secure, hidden SSID configured via wpa_supplicant, with automatic AP recovery explicitly disabled.

5. Lack of Firewall Rules on the Robot
   Problem: Unfiltered incoming/outgoing connections on the robot's OS could lead to exploits or exfiltration.
   Solution: We employed UFW (Uncomplicated Firewall) to restrict all unnecessary inbound/outbound traffic, allowing only essential protocols and ports.

6. Brute-force or Replay Attacks on WebSocket Connections
   Problem: Attacker may replay or forge messages if communication is not properly secured.
   Solution: TLS encryption combined with session-level authentication ensures that only verified agents can communicate. Payloads are also cryptographically signed where necessary.

7. Credential Leakage through Logs or Debug Output
   Problem: Secrets being accidentally printed in logs or debug terminals.
   Solution: Logging mechanisms were filtered, and sensitive values were masked or completely excluded from any form of output.

8. Improper Environment Variable Permissions
   Problem: Sensitive data might be read by unauthorized users on the host system.
   Solution: Environment variable files were protected with strict user-level file permissions, and deployment scripts ensured no leakages into shell history or process listings.

9. Outdated System Packages or Dependencies
   Problem: Known exploits in libraries or system software.
   Solution: We used Docker containers with pinned, audited dependencies for backend services, and regularly applied patches to the robot's OS image.

7. Task Allocation

Our team approached the project with a strong sense of organization, commitment, and mutual accountability. From the beginning, we divided the workload in a balanced manner, assigning responsibilities based on each member's strengths and areas of interest. This allowed us to work in parallel on different layers of the system, such as backend infrastructure, frontend integration, ROS2 node development, and image processing, while maintaining coherence and compatibility across components.

- *Victor Florescu* was responsible for implementing the server side, the web interface, and the communication between them. In addition, he designed the overall system architecture, focusing on the communication flow and data pipeline from end to end. This included defining how data generated by various ROS2 modules, interconnected through topics and custom nodes, is encoded, securely transmitted to the backend server, decrypted and stored in a Dockerized PostgreSQL database, and ultimately delivered to the frontend via HTTPS. His role encompassed both the structural design of these components and their practical integration into a robust, real-time system.

- *Marius Antache* and *Remus Comeaga* worked together to implement the decision and control algorithms, handling the robot's behavior and response logic in various situations, such as obstacle detection, trajectory planning, and target tracking. As part of these tasks, they also developed and integrated a PID controller that continuously maintains the robot at equal distances between the walls. This control mechanism takes advantage of the robot's omnidirectional wheels, enabling lateral adjustments and fine movement corrections, which improve both stability and accuracy in the narrow corridors of the maze.

- *Mihaela Popescu* was tasked with the hardware side of the project, taking responsibility for the physical integration and reliability of the robot's sensing and structural systems. This included the reading, calibration, and installation of a variety of sensors, such as ultrasonic, Hall effect, gas, and vibration, ensuring that each of these provided accurate and consistent data. She was also in charge of the maze construction, designing the layout and materials to simulate a realistic navigation environment. Mihaela was also responsible for the CAD modeling of custom components required for mounting sensors and stabilizing hardware elements on the chassis. The designs were improved gradually, based on feedback, to position the sensors optimally. She also played a key role in verifying the electrical connections, ensuring that all components were powered and interfaced with the control board.

We maintained a shared to-do list and task board, updated regularly to reflect progress, priorities, and blockers. Every task was assigned a deadline, and we tried to stay within the allocated time. Communication played an important role in not having dead-times or not focusing excessively on a single feature.

Due to the fact that a single platform was available, parallelization was a key aspect that helped us stay on track. For example, for the GUI's development, a simulated agent that sends generated data has been used, while the detection network has been tested on a laptop. Additionally, another similar development board was used to setup the sensors, and while testing the control of the robot, the alcohol detection procedure was under development.

8. Conclusions

The final implementation of the robot successfully integrates multiple sensing and computational capabilities to autonomously navigate a constrained and partially unknown maze environment. The robot is able to detect and classify environmental events, such as obstacles, vibrations and magnetic fields while accurately maintaining its orientation and trajectory. Throughout the exploration process, it continuously recorded its state, captured real-time imagery, and transmitted a bundled data payload to a centralized server for logging, analysis, and live monitoring.

From a system perspective, this performance is the result of a coordinated multi-layered architecture, combining ROS2-based framework with a secure backend pipeline and an intuitive frontend dashboard. The robot leverages its onboard omnidirectional motion profile, sensor fusion algorithms, and adaptive control strategies to make context-aware decisions and respond to dynamic elements in the environment. Each interaction with the physical world, whether detecting a false exit or passing over a vibration-inducing obstacle, was transformed into structured digital information, processed and displayed within milliseconds.

Throughout the project, the team encountered and overcame several technical and organizational challenges. Among the most significant were achieving low-latency, lossless communication over WebSocket with embedded steganographic data, managing concurrent development across hardware and software subsystems, and iterating on a real-time control loop robust enough to handle both continuous sensor noise and sudden environmental changes. The iterative design–test–refine cycle proved essential in enhancing system reliability under operational constraints.

Beyond the technical results, this project offered valuable lessons in collaborative engineering, system integration, and the practical limitations of working with real hardware in competition-grade scenarios. It emphasized the importance of modular design, robust fault isolation, and the ability to adapt architectural decisions in response to evolving requirements. The competition context provided a high-pressure, real-world validation platform that honed not only our technical competencies but also our capacity to communicate, prioritize, and deliver under time constraints.

9. Appendix – Bill of Materials

The following components, although not originally included in the provided hardware package, were added to the final setup in order to ensure mechanical stability, better detection, and sensor mounting flexibility:

- 5V fan (mounted in front of the Gas Sensor for detection enhancement)
- Compact breadboard (for auxiliary connections)
- Assorted screws, spacers, nuts, washers
- 3D-printed parts:
  – Rack and pinion assembly
  – Custom servo support bracket
  – Modular sensor mounts
- Plexiglass platform (dimensions: 51 × 78 mm)
- Cable ties (for wire management and secure anchoring of components)

Each of these additions contributed to the reliability and performance of the overall system, ensuring that the robot could operate under prolonged runtime conditions without mechanical failure or data acquisition inconsistencies.