Politecnico di Torino

MSc in Mechatronic Engineering

A.a. 2025/2026

Sessione di Laurea Marzo 2026

ChatGPT-powered autonomous patrol mobile robot: a ROS2-based approach for intelligent surveillance

Supervisors:
Prof. Alessandro Rizzo
PhD Pangcheng David Cen Cheng

Candidate:
Enrico Ianni Ficorilli

Abstract

Autonomous monitoring and surveillance systems are increasingly critical for indoor security and facility management. Historically these applications have relied on static, rigid and deterministic algorithms. Traditional robotic perception utilizes standard object detection models that, while capable of identifying bounding boxes, lack true semantic understanding and contextual reasoning. Furthermore, classic systems struggle to correlate high-level visual identity with precise low-level spatial data, limiting their ability to react intelligently to dynamic, unstructured environments. The integration of Large Language Models (LLMs) and Vision-Language Models (VLMs) into robotic frameworks represents a significant milestone in overcoming these limitations, shifting the paradigm from programmatic execution to semantic situational awareness.

This thesis explores a novel architectural approach to robotic surveillance by utilizing ChatGPT as a semantic parser and reasoning engine. Centered on the TurtleBot 4 mobile platform operating within the ROS 2 Jazzy Jalisco middleware, the research implements a "Patrol and Perceive" logic. A custom ROS 2 package was engineered to facilitate asynchronous communication between the robot's local nodes and the OpenAI API. To bridge the gap between AI reasoning and physical space, the system utilizes a novel sensor fusion approach. Real-time visual data captured by the robot's OAK-D camera provides the LLM with the necessary context to identify problems, while a dynamically masked percentile-based filtering algorithm processes RPLIDAR data to calculate the precise distance to the target. Significant systems engineering challenges are documented and resolved, specifically the configuration of FastDDS Discovery Servers within a Windows Subsystem for Linux (WSL 2) environment, and the optimization of the Nav2 autonomy stack to prevent behavioral instabilities during locomotion.

To validate the proposed architecture, physical experiments were conducted across three distinct operational scenarios: a narrow corridor, a high-glare entrance hall, and a combined patrol route. The system's robustness was evaluated under varying surveillance frequencies (0.5s and 0.2s intervals). The results demonstrate that the proposed system successfully bridges the gap between high-level AI reasoning and low-level physical actuation, accurately identifying human presence and localizing threats in real-time. While high-frequency testing revealed a minor distance mismatch during partial visual occlusions, the overall architecture proved highly resilient. This work concludes that LLM-based parsers significantly enhance mobile security applications, providing a robust and highly capable foundation for the next generation of reasoning-driven autonomous service robots.

Acknowledgements

Table of Contents

List of Figures VIII

1 Introduction 1
1.1 Goal 2
1.2 Thesis structure 2

2 State of the art, Hardware and Software Architecture 4
2.1 State of The Art 4
2.2 Comparative Analysis of LLM-Robotic Frameworks 6
2.3 Gaps in current research 6
2.4 The TurtleBot 4 8
2.5 System architecture 8
2.5.1 iRobot Create 3 8
2.5.2 Perception and compute 9
2.6 Large Language Model integration architecture 10
2.6.1 The OpenAI API interface 10
2.6.2 Latency management and asynchronous execution 11
2.7 Sensing and actuation suite 11
2.7.1 Luxonis OAK-D Pro Camera 11
2.7.2 RPLIDAR A1 12
2.8 Connectivity and power distribution 13
2.8.1 Internal communication bus 13
2.8.2 Power management and docking 13
2.9 Software stack and ROS 2 integration 14
2.10 The Robot Operating System (ROS 2) 14
2.10.1 Node-based architecture and modularity 14
2.10.2 Communication Primitives 15
2.10.3 DDS and the discovery service 17
2.10.4 Quality of Service (QoS) Profiles 18
2.10.5 The ROS 2 ecosystem 18
2.11 Navigation Stack (Nav2) 18

2.11.1 The Server-Client architecture 19
2.11.2 Behavior trees and task execution 20
2.11.3 Costmaps and environmental representation 20
2.12 The Transform library (TF2) 21
2.12.1 The Transform tree 21
2.12.2 Localization and the map-odom transform 21
2.13 Simultaneous Localization and Mapping 22
2.13.1 SLAM problem formulation 22
2.13.2 Graph-based SLAM 22
2.14 Visualization and debugging (RViz 2) 22
2.14.1 The Visualization Pipeline 22
2.14.2 Costmap visualization 23
2.15 Gazebo Sim 24
2.15.1 The physics engine and component-based architecture 24
2.15.2 Simulation Description Format (SDF) 25
2.15.3 The network bridge 25
2.16 WSL 2 environment 26
2.16.1 Architectural overview 26
2.16.2 Networking and ROS 2 middleware 26

3 Implementation **27**
3.1 System architecture 27
3.2 Environment setup and workspace creation 27
3.2.1 ROS 2 Jazzy on WSL 27
3.2.2 Workspace Initialization 27
3.3 Dependency management 28
3.4 Network architecture and troubleshooting 28
3.4.1 Cross-platform communication 28
3.5 Time synchronization 28
3.6 Autonomous navigation configuration 29
3.6.1 The Nav2 Stack 29
3.6.2 Angle instability 29
3.6.3 Fix: omnidirectional tolerance 29
3.7 Performance and stability optimizations 29
3.7.1 Controller computational load 30
3.7.2 Costmap and lifecycle tuning 30
3.8 Lidar data processing and sensors 31
3.8.1 Depth perception 31
3.9 Geometric calibration and coordinate frames 31
3.10 Hardware filtering and persistent obstacle 32
3.11 Horizon limitations and statistical adjustments 32

3.12 Smart Patrol Node . . . 32
3.12.1 Imports and dependencies . . . 33
3.12.2 Libraries . . . 33
3.12.3 Class initialization . . . 34
3.12.4 QoS tuning . . . 35
3.13 Sensor logic . . . 35
3.13.1 Semantic analysis . . . 36
3.13.2 Patrol loop . . . 38
3.13.3 Concurrent surveillance . . . 38
3.13.4 Execution entry point . . . 39

4 Experimental setup and methodology 40
4.1 Pre-flight checks and network initialization . . . 40
4.1.1 Environment sourcing and daemon management . . . 40
4.1.2 Teleoperation and safety check . . . 41
4.2 Mapping and localization . . . 41
4.2.1 Map generation and visualization . . . 41
4.3 Navigation stack initialization . . . 42
4.3.1 Discovery server configuration . . . 42
4.4 Smart Patrol node deployment . . . 42
4.4.1 Waypoint selection and compilation . . . 43
4.5 Experimental scenarios . . . 43
4.5.1 Scenario A: Narrow corridor . . . 43
4.5.2 Scenario B: Entrance hall . . . 43
4.5.3 Scenario C: Combined route . . . 44
4.5.4 Scenario D: Combined route 5Hz variant . . . 44
4.5.5 Scenario A: narrow corridor patrol (0.5s frequency) . . . 45
4.5.6 Scenario B: entrance hall (0.5s frequency) . . . 50
4.5.7 Scenario C: combined route (0.5s) . . . 53
4.5.8 Scenario D: high-frequency stress test (0.2s polling) . . . 59

5 Discussion and Future Work 64
5.1 Discussion of findings . . . 64
5.1.1 Architectural and middleware viability . . . 64
5.1.2 Navigational reliability and controller tuning . . . 65
5.1.3 Trade-off between semantic and spatial intelligence . . . 65
5.2 System limitations . . . 66
5.2.1 Cloud latency and open-loop perception . . . 66
5.3 Future work . . . 67
5.3.1 Visual servoing and dynamic threat tracking . . . 67
5.3.2 Migration to local Vision-Language Models (Edge Computing) 69

5.3.3 Transition to 3D sensor fusion 69
5.3.4 Multi-agent orchestration via LLM 69
5.4 Conclusion 70

List of Figures

2.1 TurtleBot 4 [**clearpath_tb4**] 8
2.2 Raspberry Pi 4B [**clearpath_tb4**] 9
2.3 TurtleBot 4 UI board [**clearpath_tb4**] 10
2.4 OAK-D-Pro Camera [**clearpath_tb4**] 12
2.5 RPLIDAR A1M8 [**clearpath_tb4**] 13
2.6 ROS2 Domain ID [**mathworks_ros2**] 15
2.7 ROS2 Topics [**mathworks_ros2**] 15
2.8 ROS2 Services [**mathworks_ros2**] 16
2.9 ROS2 Actions [**mathworks_ros2**] 17
2.10 Navigation2 Architecture Overview [**nav2_docs**] 19
2.11 Map generation in RViz 2 [**clearpath_tb4**] 23
2.12 Map visualization [**clearpath_tb4**] 24
2.13 TurtleBot 4 in Gazebo Sim 25

4.1 Physical view of the narrow corridor used for Scenario A. The robot patrols between the two waypoints along this hallway. 45
4.2 RViz visualization of the Scenario A global costmap and designated waypoints (points in red). 47
4.3 Terminal log indicating a successful "ALARM" state and accurate distance measurement. 48
4.4 Terminal log illustrating the latency miss scenario. The person enters and exits the camera's field of view between two 0.5-second checks, resulting in a wrong distance estimation. 49
4.5 Physical view of the Scenario B. 50
4.6 RViz visualization of Scenario B, showing sparse obstacle boundaries. 51
4.7 Terminal log indicating a successful "ALARM" state and accurate distance measurement. 52
4.8 Terminal log illustrating the latency miss scenario. The person enters and exits the camera's field of view between two 0.5-second checks, resulting in a wrong distance estimation. 52

4.9 Partial physical view of the combined route used for Scenario C. The robot patrols between the 12 waypoints along this hallway. 53
4.10 Partial physical view of the combined route used for Scenario C. The robot patrols between the 12 waypoints along this hallway. 54
4.11 RViz visualization of Scenario C, showing the full patrol route encompassing both the corridor and entrance hall. 56
4.12 Terminal log indicating a successful "ALARM" state and accurate distance measurement. 58
4.13 Terminal log illustrating the latency miss scenario. The person enters and exits the camera's field of view between two 0.5-second checks, resulting in a wrong distance estimation. 58
4.14 Terminal output showing the system catching a path planning failure and skipping to the subsequent waypoint to maintain the patrol route. 62

Chapter 1

Introduction

The human-robot interaction has been a topic of interest by the scientific community for the past decades. Through carefully crafted feedback loops and fine tuning, increasing levels of competence and dexterousness have been achieved, which can be adapted to a very broad range of applications.

This thesis revolves around a specific strategy used to send instructions to the robot: it involves typing a command in natural language (conversational type), parsing it in a way that the Robot Operating System (ROS) understands, and finally executing it. The parser and its job are done through the LLM's Application Programming Interface (API), in this specific case ChatGPT. When an LLM is used in its domain setting, i.e. the publicly available interface on the web, the interaction with it is constrained to that isolated segment. This is where the API comes in. It acts like the waiter in a restaurant: it receives the input, sends it to the LLM's servers, which then compute the most appropriate answer, and send it back to the receiver. By using the API, the LLMs' abilities can be leveraged in custom environments.

The past years have seen the rise of Large Language Models (LLMs), stochastic entities that are focused on interpreting and manipulating symbols, represented as tokens, ultimately producing an alphabet that the model is trained on. Tokens can be letters, numbers, proteins, basically distinguishable items that can be categorized and connected. With the right training and subsequent fine tuning the resulting model's skills can be leveraged in a variety of situations.

The environment used in this thesis is Ubuntu 22.04 Noble Nombat, along with Ros Jazzy. The LLM's job is parsing (reshaping) the natural language commands that have been correctly received in a way that ROS "understands".The high level description of the method used is:

- **Setup:** starting a simulation on Gazebo or connecting the real robot (a TurtleBot 4) to the designated device used to send instructions.

- **Patrol:** starting a patrol loop that waits for commands (published as ROS Topics).
- **Parsing:** the natural language topics are parsed by the LLM and sent back.
- **Movement:** ROS interprets the commands and executes them, making the robot move.

1.1 Goal

To achieve the desired communication pipeline between user and robot, the following specific technical objectives have been identified:

- **Architectural integration in ROS 2 Jazzy:** design and implement a dedicated ROS 2 package structure (built using the `ament_python` build system) capable of managing asynchronous communication between local robotic nodes and remote cloud-based LLM servers.
- **Natural language to robotic action mapping:** develop a Parser Node that utilizes the OpenAI API to translate text input into ROS commands. This involves handling the serialization of data, managing API latency, and ensuring the robot executes the derived behavior via the `/cmd_vel` or navigation topics.
- **Autonomous perception and feedback loop:** integrate visual feedback from the TurtleBot 4's OAK-D camera into the LLM prompt. The objective is to enable closed-loop reasoning where the LLM follows commands and analyzes an image of the robot's surroundings to decide the most appropriate subsequent action.
- **Validation of the interface:** verify the portability of the developed software stack between the Gazebo simulation and the physical TurtleBot 4 hardware, ensuring that the package configurations (including `setup.py`, `package.xml`, and the `libexec` directory structure) are robust and compliant with ROS 2 standards.

1.2 Thesis structure

The thesis is structured as follows: Chapter 2 provides an overview of the state of the art in the field of human-robot interaction, along with comprehensive descriptions of the TurtleBot 4 and ROS2. Chapter 3 describes the methodology used to achieve the goals outlined in Chapter 1, including the design and implementation of the ROS package and the integration of the LLM. Chapter 4 presents the results of the

experiments conducted to validate the interface, done in real life scenarios. Finally, Chapter 5 concludes the thesis with a discussion of the findings and potential future work in this area.

Chapter 2

State of the art, Hardware and Software Architecture

2.1 State of the Art

The integration of Large Language Models (LLMs) into the robotic domain marks a transformative departure from classical, deterministic control loops toward semantic, reasoning-driven architectures, giving a new approach to the Parser role in human-robot interaction. At the heart of this evolution is the endeavor to bridge the gap between high-level human intent and low-level robotic actuation, a challenge that has been approached through various lenses ranging from pure programmatic synthesis to multimodal spatial awareness.

One of the many contributions to this field is the `ROSGPT` framework developed by Anis Koubaa [**rosgpt**]. `ROSGPT` established a foundational "translation broker" architecture, utilizing a proxy node to facilitate a handshake between the ROS 2 ecosystem and OpenAI's GPT models. By leveraging prompt engineering techniques like Chain-of-Thought and instruction tuning, it successfully demonstrated that unstructured human speech could be mapped onto context-specific robotic primitives. However, while `ROSGPT` excelled at intent extraction, the field soon recognized a recurring problem: unintended outputs. This occurs when an LLM generates syntactically correct but contextually dangerous or non-functional code. To address this, projects like `RobotIQ` [**raptis2025robotiq**] introduced high-level function libraries and embedded Domain-Specific Languages (eDSLs). `RobotIQ` prevents the LLM from generating "drifted" code that might compromise the robot's physical integrity by forcing it to select from a pre-validated list of logical sequences.

A parallel trajectory in programmatic synthesis is seen in the `Codebotler` project from UT AMRL [**codebotler**]. `Codebotler` advances the state of the art by providing a rigorous evaluation framework called `RoboEval`. This benchmark

uses temporal logic properties to verify that the generated Python programs satisfy specific task requirements across multiple initial states, allowing for complex, multi-stage sequencing that simple command-parsing cannot achieve. Yet, even with robust code generation, a robot remains blind unless it can reconcile these instructions with its physical environment. This necessity led to the rise of frameworks like `TCC-IRoNL` (Interaction through Natural Language) [**linusnep**], which moved toward a multi-modal approach: it enables robots to engage in natural conversation while simultaneously processing the visual context of the scene, allowing for a more human-like context shared between the operator and the machine, with the help of VLMs (Vision-Language Models).

This visual dimension is further refined in `VisionGPT` by the AIS-Clemson lab [**visiongpt**], which focuses on real-time anomaly detection for assistive navigation. `YOLO-World` for open-vocabulary object detection and GPT-3.5's reasoning are combined in order to provide audio-based situational awareness, proving that LLMs can act as high-level safety monitors in dynamic, crowded urban settings. However, interpreting a 2D image is fundamentally different from understanding 3D space. The `RoboSpatial` project by NVIDIA Research [**robospatial**] addresses this by providing a massive dataset specifically designed to teach models reference-frame comprehension, distinguishing between ego-centric, world-centric, and object-centric perspectives. Such spatial grounding is a prerequisite for any robot that must follow a command like "go behind the chair," which is inherently relative.

Building upon these spatial foundations, the `NavCoT` (Navigational Chain-of-Thought) framework [**navcot**] introduces the concept of the LLM as a world model. In `NavCoT`, the agent imagines the future surroundings based on the instruction and then selects the candidate observation that best aligns with that imagination. This parameter-efficient in-domain training significantly reduces the domain gap between an LLM's text-heavy training data and the pixel-heavy reality of navigation. Similarly, `SORT3D` [**sort3d**] integrates a spatial reasoning toolbox with sequential LLM reasoning to achieve zero-shot generalization in unseen environments. Complex referential statements are decomposed into a series of toolbox function calls, delegating relative spatial math to rule-based logic while keeping the LLM as the high-level conductor.

As these systems become more autonomous, the focus shifts toward planning and coordination. The `LLM-Based-mobile-robot-path-planning` project [**llm_path_plan**] explores the use of prompt engineering, where the LLM must handle task allocation for various types of scenarios. This decentralization of logic is echoed in the `DELTA` framework from Bosch Research [**delta**], which investigates multi-agent task planning. Here, the LLM acts as a global coordinator, decomposing complex mission objectives into sub-tasks for specialized agents. Finally, to ensure these models continue to improve in the field, `Visual-RFT` (Visual Reinforcement Fine-Tuning) [**visual_rft**] introduces a method for models to learn from visual

feedback in data-scarce scenarios, ensuring that the VLM's reasoning grounding remains accurate even as the robot encounters novel obstacles. Together, these projects form a rich array of research that transitions the robot from a machine that follows orders to an agent that understands context, space, and intent.

2.2 Comparative Analysis of LLM-Robotic Frameworks

To synthesize the diverse methodologies presented in the preceding discussion, Table 2.1 provides a structured comparison. This table categorizes each project based on its core methodology (parsing, code generation, or spatial reasoning), its primary modality, and the specific innovation it introduces to the ROS ecosystem.

2.3 Gaps in current research

While the state of the art demonstrates rapid progress in LLM-robotics integration, several critical gaps remain which this thesis aims to address:

1. **Spatial-semantic latency:** frameworks like `ROSGPT` often treat the LLM as an offline planner. There is a lack of research on real-time, closed-loop systems where the LLM continuously re-evaluates the path based on dynamic visual data.

2. **Deployment on constrained hardware:** many multimodal frameworks (like `TCC-IRoNL`) rely on heavy, cloud-based GPUs or onboard NVIDIA Jetsons. There is limited exploration of deploying these architectures on standard educational platforms like the Raspberry Pi 4-based TurtleBot, where computational resources are strictly limited.

3. **Hallucination management in navigation:** while code generation drift is well-documented, spatial hallucination where an LLM confidently directs a robot to a non-existent room based on a misinterpretation of a map remains an under-researched safety hazard.

Table 2.1: Comprehensive comparison of State-of-the-Art LLM-Robotic integration frameworks

Project / Reference	Primary method	Key modality	Middleware	Main contribution & Innovation
`ROSGPT`	Intent parsing	Text	ROS 2	Introduced the proxy node to map human speech to ROS primitives using prompt engineering.
`RobotIQ`	Restricted eDSL	Text	ROS 2	Mitigates code drift by restricting LLM output to pre-validated function libraries and high-level APIs.
`Codebotler`	Code generation	Text	ROS 1/2	Evaluates programmatic policies using `RoboEval` and temporal logic for verifiable task execution.
`TCC-IRoNL`	Multimodal VLM	Vision + Text	ROS 2	Enables conversational shared context through concurrent visual processing and natural language dialogue.
`VisionGPT`	VLM + Detection	Vision + Audio	ROS 2	Integrates `YOLO-World` for open-vocabulary anomaly detection and assistive audio feedback for navigation.
`RoboSpatial`	Spatial grounding	3D Spatial	Custom/GZ	Provides massive datasets for teaching models ego-centric vs. world-centric reference frame comprehension.
`NavCoT`	World modeling	Visual CoT	ROS 2	Employs Chain-of-Thought to imagine future observations, reducing the domain gap in pixel-heavy navigation.
`SORT3D`	Tool-use reasoning	3D Vision	ROS 2	Decomposes referential statements into toolbox calls for zero-shot generalization in unseen 3D environments.
`DELTA`	Multi-agent plan	Decentralized	ROS 2	Coordinates heterogeneous teams by decomposing global missions into sub-tasks for specialized agents.
`Visual-RFT`	RL fine-tuning	Vision	Custom	Introduces reinforcement fine-tuning to maintain accurate reasoning grounding in data-scarce scenarios.

2.4 The TurtleBot 4

The TurtleBot 4 represents the fourth generation of the world's most popular open-source robotics platform for education and research. Developed through a collaboration between Clearpath Robotics and iRobot, the TurtleBot 4 is the first of its kind to be built entirely on the Robot Operating System 2 (ROS 2). This chapter provides an exhaustive technical breakdown of the platform, detailing its tiered architecture, integrated sensing capabilities, and the computational framework that enables high-level autonomous behaviors.

Figure 2.1: TurtleBot 4 [**clearpath_tb4**]

2.5 System architecture

The TurtleBot 4 is characterized by a hierarchical design that separates low-level mobility from high-level perception and compute. This modular stack ensures that the real-time requirements of motor control do not interfere with the resource-intensive tasks of vision processing and AI reasoning.

2.5.1 iRobot Create 3

The foundation of the TurtleBot 4 is the iRobot Create 3 educational robot. This mobile base acts as the physical body of the system, providing the necessary torque, odometry, and power management.

The Create 3 utilizes a differential drive kinematics model, featuring two independently driven wheels and two passive casters for balance. This configuration allows for a zero-turning radius environment used in this study. Internally, the base is equipped with a high-resolution optical floor tracking sensor and wheel encoders that provide a robust odometry stream (/odom). Furthermore, the base integrates a comprehensive safety suite, including infrared cliff sensors to prevent falls and a 360-degree mechanical bumper to detect physical collisions.

2.5.2 Perception and compute

Mounted atop the mobile base is the "perception and compute" assembly, which houses the high-level computational brain and the primary visual sensors. Unlike its predecessor, the TurtleBot 3, which relied on the Raspberry Pi 3, the TurtleBot 4 leverages the significantly more powerful **Raspberry Pi 4 Model B**. This unit provides the necessary ARM-based multi-core processing power to handle the ROS 2 middleware and the concurrent execution of the ChatGPT Smart Patrol Node.

Figure 2.2: Raspberry Pi 4B [**clearpath_tb4**]

The UI Board features LEDs for multiple statuses, a 128x64 user display and user button. USB 3.0 ports, power ports and Raspberry Pi pins are also available for custom hardware integration.

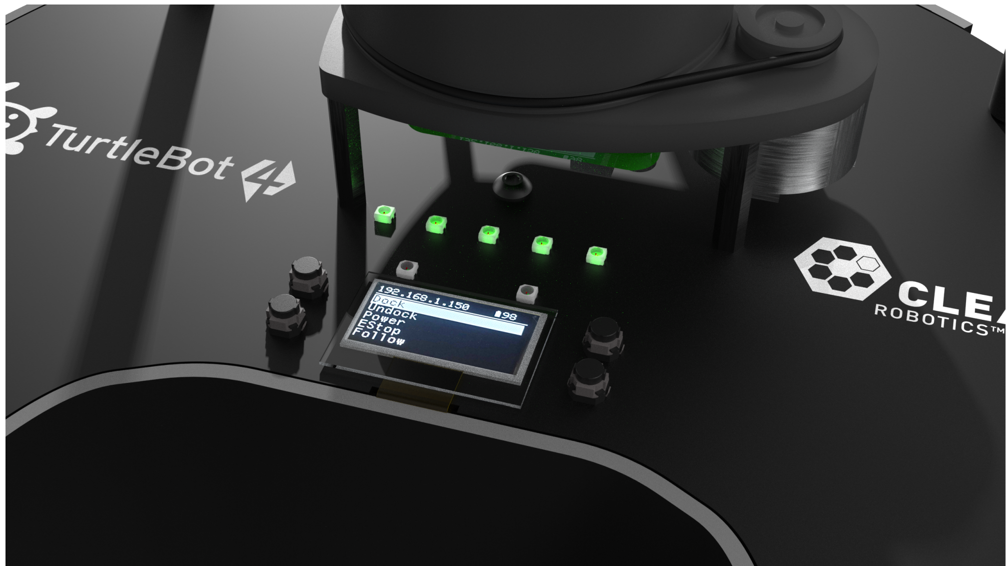


Figure 2.3: TurtleBot 4 UI board [**clearpath_tb4**]

2.6 Large Language Model integration architecture

The integration of OpenAI's GPT models into the robotic control loop introduces a non-deterministic element into a traditionally deterministic system. This section details the software architecture used to bridge the API of the LLM with the asynchronous nature of ROS 2.

2.6.1 The OpenAI API interface

The Smart Patrol Node acts as a client for the OpenAI API. Unlike local inference, this relies on a remote procedure call (RPC) mechanism. The architecture must handle the latency inherent in network requests, which typically ranges from 500ms to 3 seconds depending on the token load.

Tokenization and context window

The interaction with the model is constrained by the context window. For the GPT-4o model used in this study, the input is tokenized, in other words converted from raw text into numerical vectors. The system architecture uses a memory buffer. To prevent context overflow and reduce latency, only the last N interactions and the current system state are injected into the system prompt.

Prompt engineering as software configuration

In this architecture, the system prompt functions similarly to a configuration file. It defines the robot, its available tools (functions), and safety constraints. The *Zero-Shot* prompting strategy is used, where the model is provided with examples of valid JSON outputs (e.g., `{"action": "patrol", "target": "kitchen"}`) to ensure the output can be parsed programmatically by the ROS node.

2.6.2 Latency management and asynchronous execution

To prevent the robot from freezing while waiting for an LLM response, the API call is offloaded to a separate thread. While the LLM operates, the main ROS thread continues to process sensor data (LIDAR obstacle detection) to ensure the robot remains reactive to immediate physical threats.

2.7 Sensing and actuation suite

The TurtleBot 4 is equipped with a diverse array of sensors that allow it to perceive its environment in both two and three dimensions. This multi-modal data is essential for the LLM to make informed decisions.

2.7.1 Luxonis OAK-D Pro Camera

The most distinctive component of the TurtleBot 4 perception suite, and also the central piece for this project, is the Luxonis OAK-D Pro camera. It integrates a 4K IMX214 colour sensor and a pair of OV9282 stereo sensors, along with an infrared laser dot projector and an infrared illumination LED, performing better in low light conditions and providing depth information.

Figure 2.4: OAK-D-Pro Camera [**clearpath_tb4**]

For the purpose of this thesis, the OAK-D Pro serves as the eyes of the ChatGPT interface. The camera utilizes an onboard Myriad X Vision Processing Unit (VPU), which offloads the heavy lifting of stereo depth calculation and neural inference from the Raspberry Pi. By publishing a synchronized stream of color and depth images, the OAK-D Lite allows the robot to understand both the identity of an object and its precise location in 3D space.

2.7.2 RPLIDAR A1

In addition to visual sensing, the TurtleBot 4 features an RPLIDAR A1 sensor mounted at the highest point of the robot. This 2D laser scanner operates by rotating a laser diode 360 degrees, taking up to 8000 samples per second. It provides the robot with a planar slice of its environment, which is vital for traditional SLAM (Simultaneous Localization and Mapping) and obstacle avoidance. Even when the LLM is driving the high-level logic, the LIDAR data acts as a safety guardrail to prevent the robot from colliding with obstacles that might be outside the camera's field of view.

Figure 2.5: RPLIDAR A1M8 [**clearpath_tb4**]

2.8 Connectivity and power distribution

The TurtleBot 4 features a sophisticated power and data distribution system designed for long-duration autonomy.

2.8.1 Internal communication bus

Communication between the Raspberry Pi and the Create 3 base occurs over a dedicated Ethernet bridge (wired) or Wi-Fi. This ensures high-speed data transfer for the IMU (Inertial Measurement Unit) data and motor commands. The Raspberry Pi communicates with the OAK-D camera via a USB 3.0 link to support the high bandwidth required for 4K video streams.

2.8.2 Power management and docking

The system is powered by a 26 Wh Lithium Ion battery. A key feature of the TurtleBot 4 is its ability to autonomously dock with its charging station. This is

particularly important for long-term experiments where the robot may need to operate for extended periods without manual recharging.

2.9 Software stack and ROS 2 integration

The software architecture of the TurtleBot 4 is natively built on ROS 2 Galactic (and adapted for Jazzy in this research). This means that every sensor and actuator is exposed as a ROS 2 Topic, Service, or Action.

- **Discovery service:** the TurtleBot 4 uses the Simple Discovery Protocol to find the control workstation on the network.
- **Namespacing:** the TurtleBot 4 is designed to work in multi-robot environments, meaning its topics are often namespaced, allowing for easy scalability.
- **Diagnostics:** the TurtleBot 4 publishes a constant stream of diagnostic data, monitoring the temperature of the Raspberry Pi and the health of the internal nodes, which was crucial during the debugging of the WSL 2 environment.

In conclusion, the TurtleBot 4 is a well balanced ecosystem of hardware and software. Its ability to provide high-fidelity visual context while maintaining precise low-level motor control makes it the ideal platform for exploring the intersection of Large Language Models and mobile robotics.

2.10 The Robot Operating System (ROS 2)

The Robot Operating System 2 (ROS 2) is a framework that provides a hardware abstraction layer, low-level device control, message exchange between processes, and package management. By utilizing a distributed architecture, ROS 2 allows for the development of modular robotic software that is both scalable and fault-tolerant.

2.10.1 Node-based architecture and modularity

The fundamental unit of execution in ROS 2 is the `Node`. In the context of this thesis, each software component such as the ChatGPT parser, the TurtleBot 4 motor driver, and the OAK-D camera interface operates as an independent one. Nodes communicate with each other in the same network, identified with a unique ID.

This modularity is based on a separation principle. For instance, the node developed for this project doesn't need to know the physics of the robot's wheels, but only how to publish a specific message type. This abstraction allows to swap a simulated robot for a physical one without changing the high-level logic.

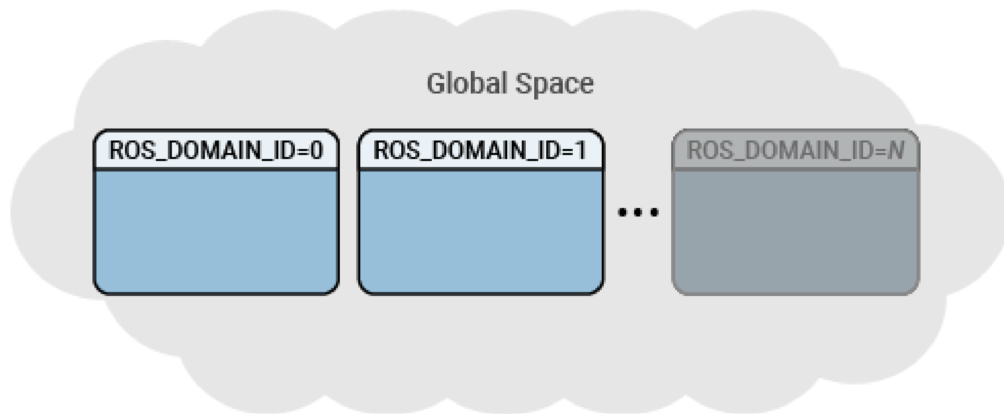


Figure 2.6: ROS2 Domain ID [**mathworks_ros2**]

2.10.2 Communication Primitives

To exchange data between these modular nodes, ROS 2 provides three primary communication models, each suited for different robotic tasks:

- **Topics (Publisher/Subscriber):** topics are used for continuous data streams. In this project, the `/oakd/rgb/preview/image_raw` topic is a constant stream of visual data, while `/cmd_vel` is a stream of velocity commands. Multiple nodes can listen to the robot's sensors simultaneously.

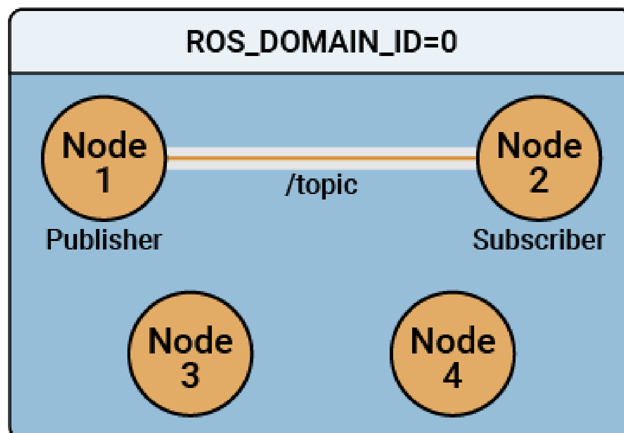


Figure 2.7: ROS2 Topics [**mathworks_ros2**]

Services (Client/Server): services are used for discrete interactions. A common example in this system is a service call to trigger a specific AI analysis or to reset the robot's odometry.

•

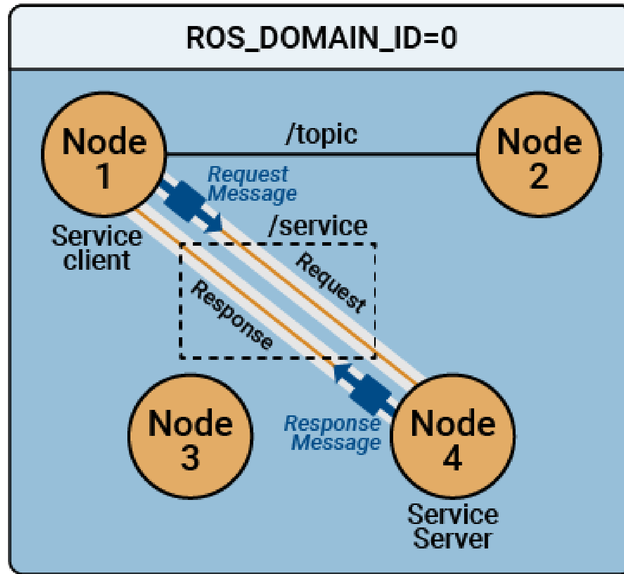


Figure 2.8: ROS2 Services [**mathworks_ros2**]

Actions (Goal/Result/Feedback): actions are intended for long and complex tasks. An action allows the robot to start a Patrol goal, provide periodic feedback on its progress (e.g., "50% of waypoint reached"), and finally return a Result upon completion.

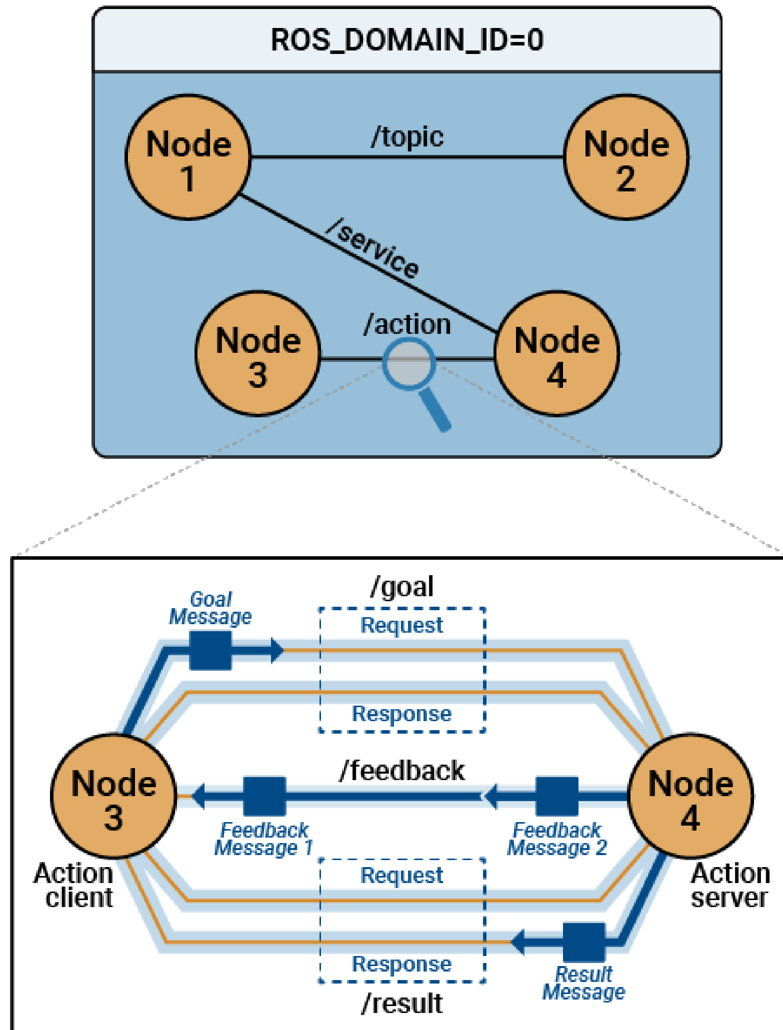
- 

Figure 2.9: ROS2 Actions [**mathworks_ros2**]

2.10.3 DDS and the discovery service

The defining characteristic of ROS 2 is its reliance on **Data Distribution Service (DDS)**. In the previous version, ROS 1, a central ROS Master connected nodes. If the Master crashed, the entire system failed.

ROS 2 removes this single point of failure through the **Discovery Service**. When the TurtleBot 4 is powered on, its internal nodes broadcast Discovery packets

using the Simple Discovery Protocol (SDP). These packets contain the node's IP address and its communication requirements. This allows our control workstation to discover the robot over a Wi-Fi network without any manual configuration, enabling the decentralized peer-to-peer network.

2.10.4 Quality of Service (QoS) Profiles

A critical technical addition in ROS 2 is the ability to define **Quality of Service (QoS)** profiles. This allows us to tune how data is handled based on its importance:

- **Reliability:** it is either `Reliable` (guaranteeing that every command from the LLM is received) or `Best Effort` (prioritizing the latest camera frame even if some frames are lost in transit).
- **Durability:** for static information, like the robot's description, there is `Transient Local` durability, which ensures that even if a node starts late, it receives the last published message.
- **History:** a "keep last" policy can be defined to ensure the robot only acts on the most recent command, preventing a backlog of old instructions from causing erratic behavior.

2.10.5 The ROS 2 ecosystem

Finally, the software lifecycle is managed by two primary tools:

1. `colcon`: the build tool used to build and install ROS 2 packages. For `ament_python` packages, it processes the Python setup and generates the necessary shell scripts to configure the environment.
2. **Launch system:** a Python based automation tool that allows us to start Gazebo, spawn the TurtleBot 4, and initialize the AI Parser node with a single command, ensuring the correct startup sequence and parameter passing.

2.11 Navigation Stack (Nav2)

While ROS 2 provides the communication backbone, the actual autonomy of the Smart Patrol robot is powered by the Navigation 2 (Nav2) stack. Nav2 is a professional-grade navigation system designed to enable mobile robots to move from point A to point B safely and efficiently. It replaces the monolithic `move_base` node from ROS 1 with a modular, task-based architecture built upon Behavior Trees.

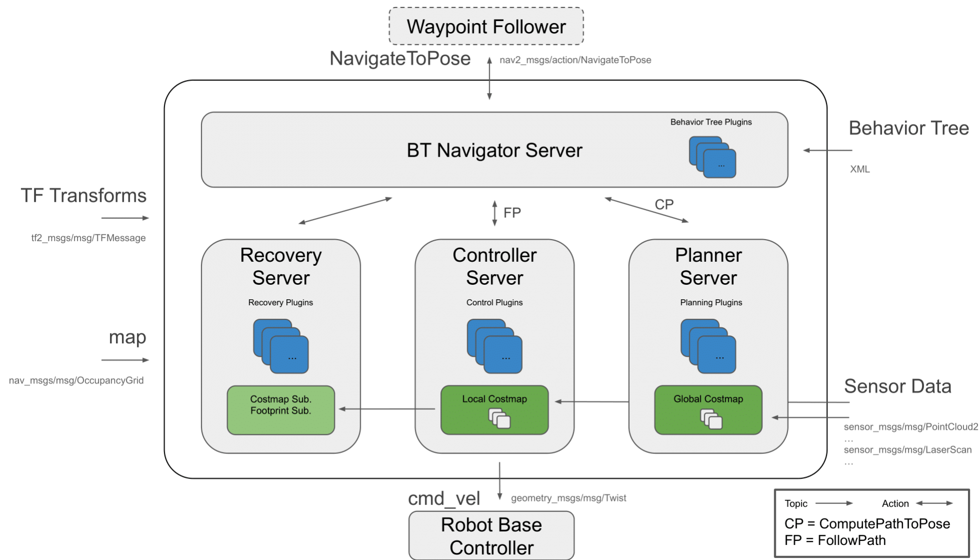


Figure 2.10: Navigation2 Architecture Overview [**nav2_docs**]

2.11.1 The Server-Client architecture

Unlike traditional control loops, Nav2 operates as a collection of independent Action Servers. This separation of concerns allows to swap algorithms (e.g., changing the path planner) without rebuilding the entire stack. The `smart_patrol_node` acts as a client, sending high-level goals to these servers.

Planner Server: responsible for global path planning. It takes the robot's current pose and the goal pose as input and produces a feasible path (a list of waypoints) that avoids static obstacles defined in the Global Costmap.

Controller Server: responsible for local execution. It takes the global path and generates immediate velocity commands (`cmd_vel`) for the robot's wheels. The controller operates at a high frequency (typically 20Hz) to react to dynamic obstacles.

Behavior Server: manages recovery behaviors. If the robot becomes stuck (e.g., surrounded by obstacles), this server executes a sequence of clearing actions, such as spinning in place to update the Lidar map or backing up to find a new trajectory.

Waypoints Follower: a specialized server that accepts a list of poses (the patrol route) and orchestrates the execution of the Planner and Controller servers to visit each point in sequence.

2.11.2 Behavior trees and task execution

Nav2 utilizes Behavior Trees (BTs) to orchestrate these servers. A BT is a hierarchical structure of nodes that control the flow of execution, offering superior flexibility compared to finite state machines.

- **Control Flow Nodes:** dictate the logic (e.g., "Sequence", "Fallback"). For example, a standard navigation tree might define: "Try to follow the path; IF blocked, triggers the Recovery Fallback; IF that fails, abort."
- **Action Nodes:** the leaves of the tree that perform the actual computations, such as calling the `ComputePathToPose` action or sending velocity commands via the Controller.

For the Smart Patrol application, this architecture allows the LLM to trigger high-level behaviors (e.g., `NavigateToPose`) without needing to manage the low-level recovery behaviors required if the robot encounters an unmapped obstacle.

2.11.3 Costmaps and environmental representation

To navigate safely, the robot maintains two distinct 2D occupancy grids, known as costmaps. Each costmap is composed of multiple "layers" that are superimposed to calculate the final traversability cost of a cell.

Global costmap

The global costmap is a static map of the environment, typically generated via SLAM (Simultaneous Localization and Mapping) prior to the patrol mission. It represents permanent obstacles such as walls and heavy furniture.

- **Static layer:** loads the occupancy grid from a map file (PGM/YAML).
- **Inflation layer:** adds a buffer zone around obstacles. Cells close to a wall are marked with high cost, ensuring the global planner prefers paths down the center of hallways rather than hugging the walls.

Local costmap

The local costmap is a rolling window (typically $3m \times 3m$) centered on the robot. It is updated in real-time using raw sensor data.

- **Obstacle layer:** marks cells as occupied based on Lidar scans. This allows the robot to react to dynamic entities, such as humans walking or chairs that have been moved.

- **Rolling window:** as the robot moves, old data is replaced, preventing the memory from filling up with stale data.

2.12 The Transform library (TF2)

A fundamental requirement for any robotic system is the ability to manage coordinate frames. A robot is a collection of moving parts; the camera moves relative to the base, and the base moves relative to the world. The TurtleBot 4 utilizes the `tf2` library to maintain the mathematical relationship (translation and rotation) between these components over time.

2.12.1 The Transform tree

The system maintains a distributed, tree-structure of coordinate frames:

1. **`map` (Global Frame):** the fixed frame of reference for the building. The robot's position relative to the patrol route is calculated here. This frame is discontinuous; it can shift if the SLAM algorithm corrects the robot's position loop-closure.

2. **`odom` (Odometry Frame):** the local frame of reference. This frame is continuous and smooth, derived from the wheel encoders and IMU. It is critical for the local controller, which requires smooth velocity feedback. However, due to wheel slip, `odom` drifts away from `map` over time.

3. **`base_link` (Robot Frame):** the frame attached to the geometric center of the robot chassis.

4. **`laser_frame` / `camera_link` (Sensor Frames):** static transforms defining where sensors are mounted relative to the chassis. For example, if the Lidar detects an obstacle at $x = 1.0m$, `tf2` is used to transform that point into the `map` frame to mark it on the global costmap.

2.12.2 Localization and the map-odom transform

The interaction between the `map` and `odom` frames is the core responsibility of the localization system (AMCL or SLAM Toolbox). The navigation stack publishes a dynamic correction transform (`map` $\rightarrow$ `odom`). This transform represents the error drift accumulated by the wheel encoders. By constantly updating this correction, the robot ensures that the LLM's spatial reasoning ("I am in the kitchen") remains accurate even after long patrol sessions where wheel slip is inevitable.

2.13 Simultaneous Localization and Mapping

For a robot to patrol an environment autonomously, it must first understand the layout of that environment. This process is known as Simultaneous Localization and Mapping (SLAM). In this project, the **SLAM Toolbox** package was utilized to generate a 2D occupancy grid map of the patrol area.

2.13.1 SLAM problem formulation

The SLAM problem asks two questions simultaneously: "Where am I?" and "What does the world look like?"

1. **Mapping:** the robot uses its Lidar sensor to measure the distance to obstacles. As the robot moves, these measurements are integrated into a global map.

2. **Localization:** the robot uses its odometry (wheel encoders) to estimate its movement. However, odometry drifts over time due to wheel slip. SLAM corrects this drift by aligning the current Lidar scan with the previously built map (scan matching).

2.13.2 Graph-based SLAM

The `slam_toolbox` implementation uses a graph-based approach.

- **Nodes:** each node in the graph represents the robot's pose at a specific point in time.

- **Edges:** the connections between nodes represent spatial constraints derived from odometry or scan matching.

2.14 Visualization and debugging (RViz 2)

Robotics development differs from standard software engineering because the system state includes physical properties like position, velocity, and sensor data. To inspect this state, the **ROS Visualization tool (RViz 2)** was employed as the primary graphical interface.

2.14.1 The Visualization Pipeline

RViz 2 subscribes to ROS topics and renders the data in a 3D view.

TF tree visualization: RViz subscribes to the `/tf` and `/tf_static` topics to render the robot model (`URDF`) in its correct configuration.

Sensor data: the Lidar scans (`/scan`) are rendered as red point clouds.

Navigation Goals: RViz serves as the control interface for sending initial pose estimates (`2D Pose Estimate`) and navigation goals (`Nav2 Goal`) to the action servers described in Section 2.11.

2.14.2 Costmap visualization

A crucial role of RViz in this project was visualizing the Navigation Costmaps. By enabling the Inflation Layer, the "danger bubbles" around obstacles can be observed. This visual feedback was instrumental in tuning the inflation radius parameter in the `custom_nav2_params.yaml` file; one could physically see if the robot was marking a narrow corridor as impassable due to excessive safety padding.

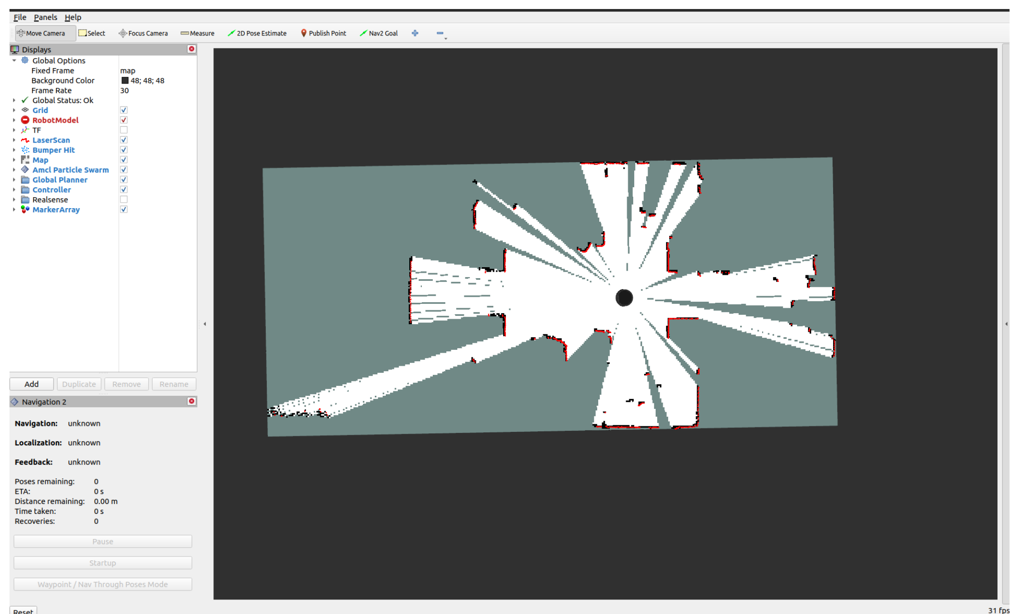


Figure 2.11: Map generation in RViz 2 [**clearpath_tb4**]

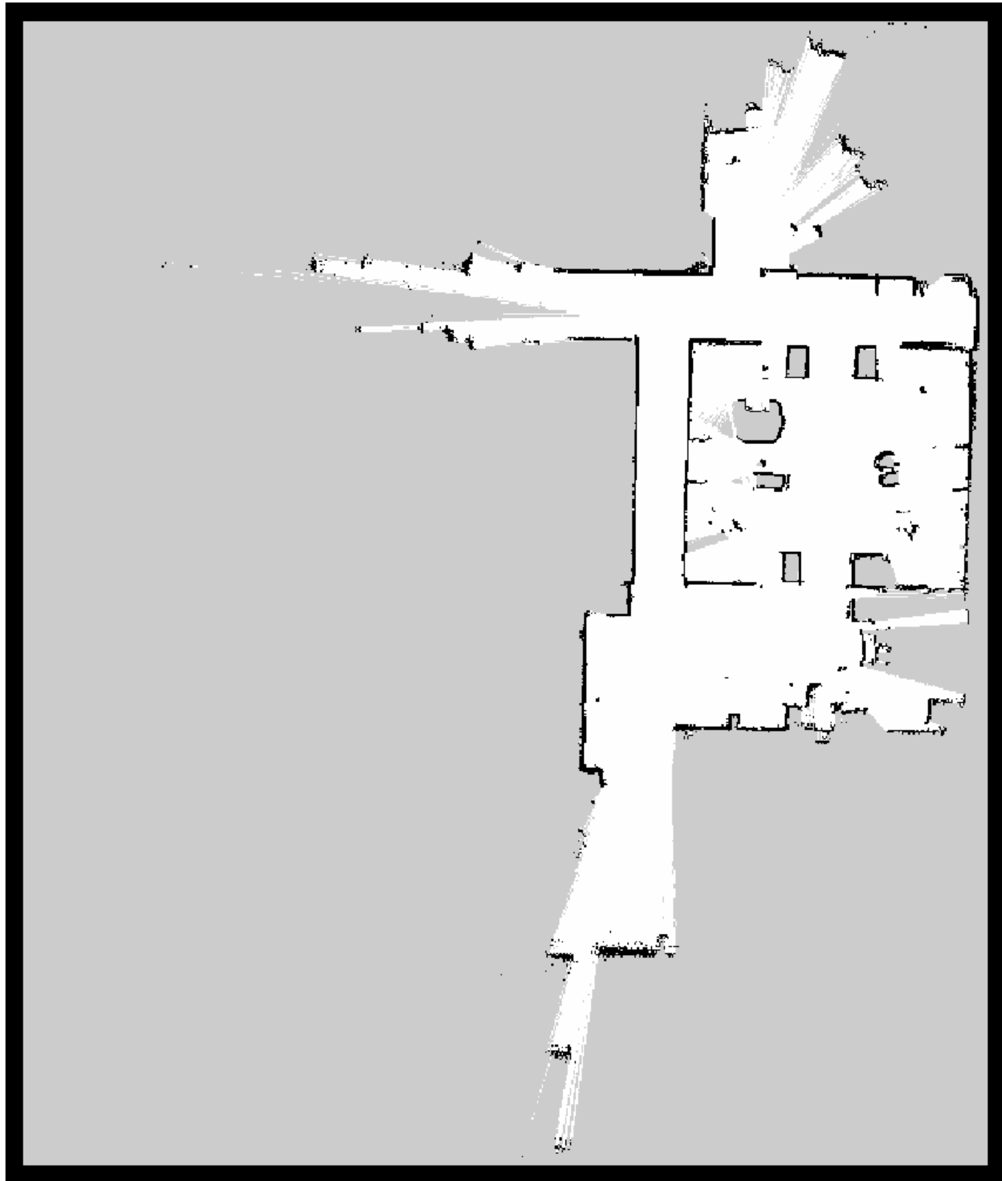


Figure 2.12: Map visualization [**clearpath_tb4**]

2.15 Gazebo Sim

2.15.1 The physics engine and component-based architecture

At the heart of Gazebo Sim is the physics engine, which handles the rigid-body dynamics, collision detection, and friction calculations for the TurtleBot 4.

The simulation operates on an **Entity Component System (ECS)**. Every object in the world is defined as an Entity. Data about that entity (such as its position or mass) is stored in Components, and Systems (like the Physics System) iterate over these components to update the state of the world at every time step.

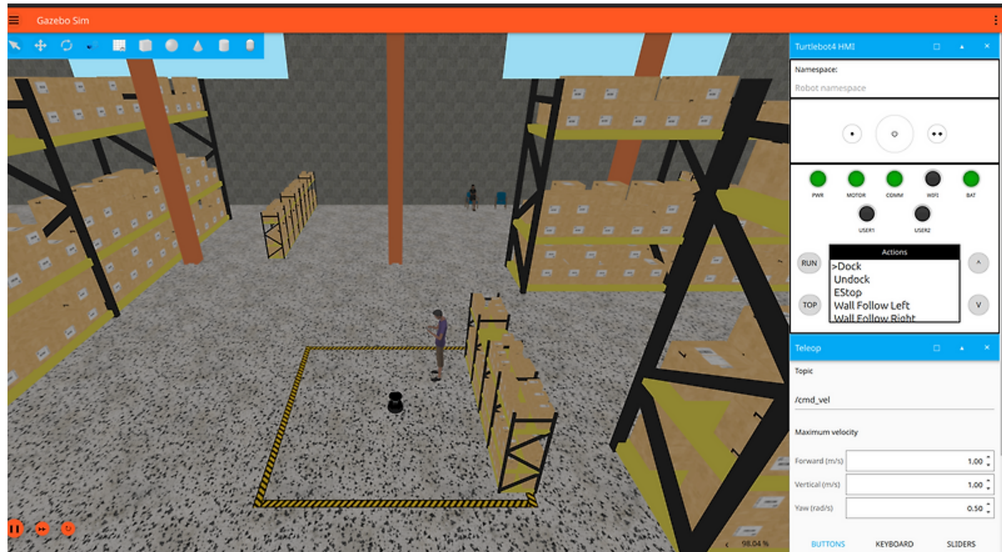


Figure 2.13: TurtleBot 4 in Gazebo Sim

2.15.2 Simulation Description Format (SDF)

The Simulation Description Format (SDF) is an XML-based language, used to describe objects and environments for robot simulation. While ROS 2 often uses URDF (Unified Robot Description Format) for robot kinematics, SDF is superior for world-building because it can describe lighting, global physics properties, and complex models.

2.15.3 The network bridge

A simulation is useless if the robot cannot communicate with the outside world. The `ros_gz_bridge` acts as the translator between Gazebo and ROS 2. It maps Gazebo topics (like the camera feed or LIDAR data) to ROS 2 topics. This bridge is what allows our Patrol Node to see into the virtual world and send velocity commands back into the physics engine.

2.16 WSL 2 environment

The development and testing phase of this project were conducted on a Windows 11 host using the Windows Subsystem for Linux 2 (WSL 2). ROS development normally requires a native Ubuntu installation or a heavy Virtual Machine. WSL 2 provides a more efficient alternative by running a Linux kernel inside a lightweight utility virtual machine, allowing for near-native performance while maintaining access to Windows-based productivity tools.

2.16.1 Architectural overview

Unlike WSL 1, which translated Linux system calls into Windows NT system calls, WSL 2 utilizes a highly optimized Hyper-V subset. This project relies on several key features of the WSL 2 architecture:

- **Full Linux kernel:** WSL 2 includes its own Linux kernel, built from the latest stable branch. This is essential for ROS 2 Jazzy, as it ensures compatibility with complex networking and file system requirements.
- **Ext4 VHDX:** the Linux environment is stored in a virtual hard disk (VHDX), which supports standard Linux file permissions and high-speed I/O operations, critical for `colcon` build processes.
- **Interoperability:** WSL 2 allows for the seamless execution of Windows binaries from the Linux terminal and vice-versa, facilitating the use of Windows-based IDEs like VS Code with Linux-based ROS nodes.

2.16.2 Networking and ROS 2 middleware

Networking in WSL 2 is handled via a virtualized Ethernet adapter that sits behind a Network Address Translation (NAT) service. While this works seamlessly for internet-facing tasks like calling the OpenAI API, it can introduce challenges for ROS 2 discovery.

Chapter 3

Implementation

3.1 System architecture

The system comprises three main computational nodes:

1. **Robot (TurtleBot 4):** runs the low-level drivers, Lidar driver (`rplidar_ros`), and the FastDDS Discovery Server [**fastdds**].

2. **Workstation (WSL 2):** runs the high-level logic (`smart_patrol`), the Navigation Stack (Nav2) [**nav2**], and the visualization tools (RViz).

3. **Cloud (OpenAI API):** processes visual data for semantic analysis [**openai_api**].

3.2 Environment setup and workspace creation

3.2.1 ROS 2 Jazzy on WSL

Setting up ROS 2 Jazzy [**ros2**] on Windows Subsystem for Linux (Ubuntu 22.04) provided a robust development environment but introduced specific constraints regarding USB device access and network bridging.

3.2.2 Workspace Initialization

The project resides in a custom ROS 2 workspace named `rosnode_chatgpt`. The directory structure was established to separate standard packages from the custom `rosnode_patrol` logic.

```
1 # Create the workspace directory
2 mkdir -p ~/rosnode_chatgpt/src
3 cd ~/rosnode_chatgpt/src
4
```

```
5  # Create the custom package
6  ros2 pkg create --build-type ament_python rosnode_patrol --
       dependencies rclpy sensor_msgs geometry_msgs
       nav2_simple_commander
7
8  # Build the workspace
9  cd ..
10 colcon build --symlink-install
11 source install/setup.bash
```

Listing 3.1: Workspace Creation Commands

3.3 Dependency management

The `rosnode_patrol` package relies on several key libraries that had to be manually integrated into the environment:

- **Nav2 Simple Commander:** a Python API for interacting with the ROS 2 Navigation Stack [**nav2**].
- **OpenAI Client:** for communicating with the GPT-4o vision model [**openai_api**].
- **CV Bridge:** for converting ROS image messages into OpenCV formats [**opencv**].

3.4 Network architecture and troubleshooting

3.4.1 Cross-platform communication

A critical hurdle in this project was establishing reliable communication between the TurtleBot 4 (running native Ubuntu) and the workstation (running WSL). Standard ROS 2 uses multicasting for node discovery, which is frequently blocked by Windows firewalls and WSL's Network Address Translation (NAT).

To overcome this, a **FastDDS Discovery Server** architecture [**fastdds**] was implemented.

3.5 Time synchronization

A failure mode was identified where the robot rejected navigation commands immediately upon receipt. This was caused by clock drift. The Raspberry Pi 4 lacks an onboard Real-Time Clock (RTC) battery. Upon reboot, its system time often reverted to the build date or Unix Epoch. If the workstation timestamp was

significantly ahead of the robot's timestamp, the laser scan data would be discarded as "from the future". Synchronization was achieved by changing the TurtleBot 4's time on the Raspberry Pi, ensuring that both devices shared a common time reference.

3.6 Autonomous navigation configuration

3.6.1 The Nav2 Stack

The ROS 2 Navigation Stack (Nav2) [**nav2**] provides the behavioral tree and path planning capabilities. For this project, the `nav2_simple_commander` API was used to issue high-level tasks, such as "Go to Waypoint A". However, the default configuration for the TurtleBot 4 proved too computationally heavy and rigid for our specific environment, leading to navigation failures and timeouts.

3.6.2 Angle instability

Initial deployment revealed a behavioral issue: upon reaching a waypoint, the robot would often enter a state of rotation, attempting to align with the specified orientation. This was particularly problematic in path planning, where the robot would lose seconds trying to correct its heading, and for the project's use case, the final orientation was not critical, time was of a more immediate concern.

3.6.3 Fix: omnidirectional tolerance

To resolve this, the `yaw_goal_tolerance` parameter was increased to 2π radians (6.28).

$$\text{Tolerance} = 2\pi \approx 6.28 \text{ rad} \tag{3.1}$$

Setting the tolerance to a full circle tells the local planner that any final heading is acceptable. The robot now considers the goal reached the moment it arrives at the X/Y coordinates, regardless of which way it is facing. This completely eliminated the unnecessary rotation behavior.

3.7 Performance and stability optimizations

The default Nav2 configuration is tuned for powerful desktop computers. Running the full stack on the TurtleBot 4 (Raspberry Pi 4) and a WSL workstation introduced significant latency, causing the controller to miss loop rates and trigger recovery behaviors unexpectedly.

3.7.1 Controller computational load

To ensure smoother operation and prevent the warnings, the DWB/MPPI controller parameters (based on the Dynamic Window Approach [**dwa**]) were optimized in order to reduce the computational cost of calculating trajectories:

- **Time Steps (Reduced** $56 \rightarrow 32$**):** this parameter defines how far into the future the local planner simulates potential paths. Reducing this horizon lowers the CPU load significantly, allowing the planner to react faster to immediate changes.
- **Batch Size (Reduced** $2000 \rightarrow 600$**):** this controls the number of path candidates generated per iteration. A reduction to 600 provided a sufficient variety of paths for the hallway environment while drastically reducing the computational overhead.

3.7.2 Costmap and lifecycle tuning

Further adjustments were made to the Global and Local Costmaps to improve behavior in narrow spaces and handle network latency:

- **Inflation Radius (**$0.45 \rightarrow 0.25$ **meters):** the default safety padding was too aggressive for the testing environment. Reducing the inflation radius allows the robot to navigate through narrower corridors without falsely detecting a collision and triggering a "stuck" state.
- **Bond Timeout (**15.0**s):** the lifecycle manager's timeout was increased to tolerate network jitter between the WSL host and the robot.
- **Attempt Duration (**20.0**s):** given the complex network routing, the allowable time for establishing connections to the action servers was extended to prevent premature aborts.

```
1  controller_server:
2    ros__parameters:
3      FollowPath:
4        plugin: "dwb_core::DWBLocalPlanner"
5        # Yaw Tolerance set to 2*PI (360 degrees) to prevent
     spinning
6        yaw_goal_tolerance: 6.28
7        xy_goal_tolerance: 0.25
8
9        # Parameters tuned for obstacle avoidance and performance
10       BaseObstacle.scale: 0.02
11       PathAlign.scale: 32.0
```

```
12          GoalAlign.scale: 24.0
13          PathDist.scale: 32.0
14          GoalDist.scale: 24.0
15          RotateToGoal.scale: 32.0
16
17      # Lifecycle and Timeout Adjustments
18      bond_timeout: 15.0
19      action_server_result_timeout: 20.0
```

Listing 3.2: Optimized controller parameters (snippet from `custom_nav2_params.yaml`)

3.8 Lidar data processing and sensors

3.8.1 Depth perception

While the camera provided the *identity* of the target ("A person"), it could not reliably provide the *distance*. The Lidar sensor provides accurate distance but no semantic identity. An algorithm was developed to correlate the two.

3.9 Geometric calibration and coordinate frames

During initial testing, the distance readings were erratic. Objects known to be in front of the robot were not detected, or the distance reported was constant regardless of movement.

An analysis of the TurtleBot 4's URDF (Unified Robot Description Format) revealed a coordinate frame offset. The Lidar data array is populated counter-clockwise starting from the robot's negative Y-axis (Right side).

- **Array Index 0:** Right Side ($-90°$)
- **Array Index 25%:** Front ($0°$)
- **Array Index 50%:** Left Side ($+90°$)

The code was updated to center the detection cone at the 25% index mark:

```
1 front_idx = int(num_readings * 0.25)
```

3.10 Hardware filtering and persistent obstacle

After fixing the orientation, a persistent obstacle was detected at exactly 0.60–0.65m directly in front of the robot, inhibiting its ability to reliably estimate the person's distance.

To resolve this without hardware modification, a **Dynamic Masking Algorithm** was implemented. This algorithm divides the front field of view into two zones:

1. **Center Zone (±5°):** the center of the view is heavily filtered. Any reading below 0.75m is discarded as chassis noise.

2. **Peripheral Zone (±5° to ±15°):** the sides are filtered less aggressively (0.40m), allowing the robot to detect legs or obstacles that are close but off-center.

3.11 Horizon limitations and statistical adjustments

A final limitation observed was a distance ceiling of approximately 1.8 meters. Even when a person stood 3 meters away, the sensor reported 1.8m.

To allow the robot to see past this floor horizon, the statistical calculation was modified. Instead of taking the absolute minimum distance ('np.min'), which would always select the floor, the algorithm uses the **10th percentile**.

```
1
2 valid = front_cone[front_cone != float('inf')]
3
4 if len(valid) == 0:
5     return float('inf')
6
7 return np.percentile(valid, 10)
```

Listing 3.3: Distance algorithm

This change successfully allowed the robot to ignore the sparse floor readings and lock onto the denser cluster of points representing the person's legs, extending the effective range.

3.12 Smart Patrol Node

This section details the architecture and implementation of the `smart_patrol_node`. This ROS 2 node serves as the central intelligence of the security robot, integrating

three distinct subsystems: autonomous navigation (Nav2 [**nav2**]), visual semantic analysis (OpenAI GPT-4o [**openai_api**]), and filtered distance estimation (Lidar sensor fusion).

The code is written in Python using the `rclpy` client library and is designed to run asynchronously. This architecture ensures that potentially blocking operations, such as waiting for an API response from OpenAI or processing heavy navigation tasks, do not freeze the robot's perception loop. This allows the reaction to obstacles in real-time.

3.12.1 Imports and dependencies

The software relies on a specific set of libraries to bridge the gap between ROS 2 middleware, standard computer vision tools, and cloud-based AI services.

```
1  import rclpy
2  from rclpy.node import Node
3  from sensor_msgs.msg import Image, LaserScan
4  from geometry_msgs.msg import PoseStamped
5  from nav2_simple_commander.robot_navigator import BasicNavigator,
       TaskResult
6  from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
7  from rclpy.executors import MultiThreadedExecutor
8  import cv2
9  from cv_bridge import CvBridge
10 import threading
11 import time
12 import base64
13 import numpy as np
14 from openai import OpenAI
```

Listing 3.4: Library imports

3.12.2 Libraries

- **rclpy & Node:** the fundamental building blocks for any ROS 2 Python application [**ros2**]. They handle the node's lifecycle, logging, and communication with the ROS graph.
- **sensor_msgs:** provides the standard message definitions for the camera (`Image`) and the Lidar (`LaserScan`).
- **rclpy.qos:** Quality of Service (QoS) settings are critical in this implementation. Specific policies (`ReliabilityPolicy`, `HistoryPolicy`) ensure that the robot can receive high-bandwidth camera data over a potentially lossy Wi-Fi connection without lagging.

- **MultiThreadedExecutor:** this executor allows the node to process multiple callbacks simultaneously. This is essential because the navigation loop, camera callback, and Lidar callback must all run concurrently without blocking one another.
- **nav2_simple_commander:** this API abstracts the complex Nav2 action servers (planning, controlling, recovering) into simple function calls like `goToPose()` [**nav2**].
- **CvBridge:** converts ROS image messages (raw byte arrays) into OpenCV [**opencv**] numpy arrays required for processing.
- **OpenAI:** the client library used to interact with the GPT-4o model [**openai_api**].

3.12.3 Class initialization

The `__init__` method sets up the robot's hardware interfaces, prepares the navigation stack, and initializes the AI client.

```
1  class SmartPatrolNode(Node):
2      def __init__(self):
3          super().__init__('smart_patrol_node')
4  
5          # 1. Setup OpenAI
6          try:
7              self.client = OpenAI()
8          except Exception as e:
9              self.get_logger().error(f"OpenAI Init Failed: {e}")
10             return
11 
12         # 2. Setup sensors (Camera + Lidar)
13         self.bridge = CvBridge()
14         self.latest_image = None
15         self.latest_scan = None
16         self.sensor_lock = threading.Lock()
17 
18         # Define "Sensor Data" QoS profile (Best Effort)
19         qos_policy = QoSProfile(
20             reliability=ReliabilityPolicy.BEST_EFFORT,
21             history=HistoryPolicy.KEEP_LAST,
22             depth=10
23         )
24 
25         # Subscribe to Camera
26         self.create_subscription(
27             Image,
28             '/oakd/rgb/preview/image_raw',
29             self.image_callback,
```

```
30              qos_policy
31          )
32
33          # Subscribe to Lidar
34          self.create_subscription(LaserScan, '/scan', self.
    scan_callback, 10)
35
36          # 3. Setup Navigation
37          self.navigator = BasicNavigator()
38
39          # Define patrol waypoints
40          self.waypoints = [
41              # chosen points go here, as (x, y, z) coordinates in
    the map frame
42          ]
43
44          # 4. Start the patrol thread
45          self.patrol_thread = threading.Thread(target=self.
    run_patrol_loop)
46          self.patrol_thread.start()
```

Listing 3.5: Node initialization

3.12.4 QoS tuning

A critical challenge in wireless robotics is bandwidth management. Streaming raw HD images can saturate the network. To mitigate this, a custom Quality of Service (QoS) profile using `ReliabilityPolicy.BEST_EFFORT` is defined.

Unlike standard "Reliable" (TCP-like) communication, "Best Effort" (UDP-like) discards dropped packets. This ensures that the `latest_image` variable always contains the most current view of the world, rather than frames buffered during a network lag spike.

3.13 Sensor logic

This function represents the core sensor logic. It transforms raw Lidar data into a reliable distance estimate for the specific object detected by the camera, accounting for hardware obstructions and mounting tilt.

```
1       def get_front_distance(self):
2           with self.sensor_lock:
3               if self.latest_scan is None:
4                   return float('inf')
5
6               ranges = np.array(self.latest_scan.ranges)
```

```
7             num_readings = len(ranges)
8
9             # 1. Geometric calibration
10            # TurtleBot 4 Lidar Index 0 is Right (-90 deg). Front
     is 25%.
11            front_idx = int(num_readings * 0.25)
12            deg_idx = num_readings / 360.0
13
14            # 2. Cone of vision
15            # 30-degree cone (+/- 15 deg)
16            half_cone = int(15 * deg_idx)
17            start = front_idx - half_cone
18            end   = front_idx + half_cone
19
20            indices = np.arange(start, end) % num_readings
21            front_cone = ranges[indices]
22
23            # 3. Dynamic masking
24            cone_angles = np.linspace(-15, 15, len(front_cone))
25            mask_center = np.abs(cone_angles) < 5
26            mask_sides  = np.abs(cone_angles) >= 5
27
28            # Filter logic:
29            # Center (+/- 5 deg): Ignore < 0.75m (Hides chassis/
     cables)
30            # Sides: Ignore < 0.40m (Allows close detection)
31            front_cone[mask_center & (front_cone < 0.75)] = float(
     'inf')
32            front_cone[mask_sides & (front_cone < 0.40)] = float('
     inf')
33            front_cone[front_cone == 0.0] = float('inf')
34
35            # 4. Floor horizon Fix
36            valid = front_cone[front_cone != float('inf')]
37
38            if len(valid) == 0:
39                return float('inf')
40
41            return np.percentile(valid, 10)
```

Listing 3.6: Lidar filtering logic

3.13.1 Semantic analysis

The `check_environment` function captures the visual scene and sends it to the cloud for analysis. It implements a retry mechanism to handle camera buffer underruns.

```python
def check_environment(self):
    img_copy = None

    # Retry logic for robust image capture
    for attempt in range(3):
        with self.sensor_lock:
            if self.latest_image is not None:
                img_copy = self.latest_image.copy()
                break
        time.sleep(0.5)

    if img_copy is None:
        return

    # Prepare image for ChatGPT
    _, buffer = cv2.imencode('.jpg', img_copy)
    base64_image = base64.b64encode(buffer).decode('utf-8')

    # Prompt engineering for classification
    prompt = (
        "You are a security robot. Briefly describe what you see in one sentence. "
        "If you see a PERSON (even incompletely, like a pair of legs with shoes, etc), "
        "start your sentence with 'ALARM: PERSON DETECTED'. "
        "Otherwise, start with 'SAFE:'."
    )

    try:
        # API Call
        response = self.client.chat.completions.create(...)
        ai_text = response.choices[0].message.content.strip()

        # Fuse with Lidar Data
        distance = self.get_front_distance()

        if "ALARM" in ai_text:
            self.get_logger().error(f"!!! {ai_text} !!!")
            self.get_logger().error(f"Estimated Distance: {distance:.2f} m")
        else:
            self.get_logger().info(f"{ai_text}")

    except Exception as e:
        self.get_logger().error(f"AI Error: {e}")
```

Listing 3.7: Semantic analysis function

3.13.2 Patrol loop

The run_patrol_loop function orchestrates the robot's movement. It runs in a separate thread to prevent blocking the sensor callbacks.

```
1      def run_patrol_loop(self):
2          # Wait for Nav2 Stack to fully initialize
3          nav2_nodes = ['controller_server', 'planner_server', '
       bt_navigator', ...]
4          for node_name in nav2_nodes:
5              self.navigator._waitForNodeToActivate(node_name)
6
7          while rclpy.ok():
8              for pt in self.waypoints:
9                  # Construct Pose Goal
10                 goal_pose = PoseStamped()
11                 goal_pose.header.frame_id = 'map'
12                 goal_pose.pose.position.x = float(pt[0])
13                 goal_pose.pose.position.y = float(pt[1])
14                 goal_pose.pose.orientation.w = 1.0
15
16                 self.navigator.goToPose(goal_pose)
17
18                 # Monitor progress while moving
19                 i = 0
20                 while not self.navigator.isTaskComplete():
21                     i += 1
22                     # Perform surveillance every 0.5 seconds
23                     if i % 5 == 0:
24                         self.check_environment()
25                     time.sleep(0.1)
```

Listing 3.8: Navigation Loop

It is of note that the time interval between surveillance checks is a critical parameter. Too frequent checks could overload the system, while too infrequent checks could miss important events. The reported interval of 0.5 seconds was thoroughly tested to balance these concerns, ensuring that the robot remains responsive to its environment while maintaining efficient navigation. Further tuning and testing of this parameter can be found in the next chapter.

3.13.3 Concurrent surveillance

A key feature of this implementation is the ability to perform surveillance during navigation. The inner while loop checks the task status every 0.1 seconds. Every 5th iteration (0.5 seconds), it triggers check_environment(). This ensures continuous monitoring of the area rather than stop-and-go behavior.

3.13.4 Execution entry point

The `main` function initializes the ROS 2 context and the `MultiThreadedExecutor`.

```
def main(args=None):
    rclpy.init(args=args)
    node = SmartPatrolNode()

    executor = MultiThreadedExecutor()
    executor.add_node(node)

    try:
        executor.spin()
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()
```

Listing 3.9: Main execution block

By using `executor.spin()`, the main thread enters an infinite loop handling the high-frequency sensor callbacks, while the `patrol_thread` (created in `__init__`) executes the navigation logic in parallel.

Chapter 4

Experimental setup and methodology

This chapter details the operational procedures established to deploy the Smart Patrol system in a real-world environment. Unlike simulation, physical deployment involves complex initialization sequences to ensure synchronization between the workstation (WSL 2) and the robot (TurtleBot 4). The experiments were conducted in the DET department at PoliTo, specifically targeting three distinct environments: a narrow corridor, a wide entrance hall, and a combined patrol route encompassing both zones.

4.1 Pre-flight checks and network initialization

A strict startup protocol was developed to mitigate the non-deterministic network behaviors observed in the Windows Subsystem for Linux (WSL) environment. The following sequence is executed upon every system boot to ensure a reliable communication bridge.

4.1.1 Environment sourcing and daemon management

The first step in any ROS 2 session is strictly defining the environment variables. While many users automate this via `.bashrc`, manual sourcing provides confirmation that the correct overlay workspace is active.

```
1 source ~/.bashrc
```

Listing 4.1: Workspace Initialization

Following environment setup, the ROS 2 Daemon, a background process responsible for node discovery, must be reset. In a mixed-OS environment (Windows Host

+ Ubuntu Container), the daemon often caches stale IP addresses from previous sessions. If the robot's IP has changed (e.g., from DHCP lease renewal), the daemon will continue routing traffic to the old address, resulting in a state where topics appear listed but carry no data, or are not listed at all.

```
ros2 daemon stop
ros2 daemon start
```

Listing 4.2: Discovery Daemon Reset Sequence

Validation: To verify connectivity before launching heavy applications, the topic list is queried. A successful connection is confirmed only if the robot-side topics (e.g., `/battery_state`, `/scan`) are visible.

```
ros2 topic list
```

Listing 4.3: Connectivity Verification

4.1.2 Teleoperation and safety check

Before enabling autonomous navigation, a manual control check is performed to ensure the motor drivers are responsive and the emergency stop (E-Stop) is disengaged. The `teleop_twist_keyboard` node is used for this purpose. This step is crucial to confirm that the robot can receive and execute velocity commands without problems.

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -p
    stamped:=true
```

Listing 4.4: Manual Control Command

4.2 Mapping and localization

The operational environment was mapped using the `slam_toolbox` in asynchronous mode to create the static 2D occupancy grid required for the global Planner.

4.2.1 Map generation and visualization

The SLAM process was initiated via the standard TurtleBot 4 launch file. Simultaneously, the visualization stack was launched in a separate terminal to inspect the map quality in real-time.

```
# Terminal 1: SLAM Logic
ros2 launch turtlebot4_navigation slam.launch.py

```

```
# Terminal 2: Visualization
ros2 launch turtlebot4_navigation view_navigation.launch.py
```

Listing 4.5: SLAM and Visualization launch

During this phase, the robot was manually driven around the perimeter of the testing zones. Special care was taken to rotate the robot 360 degrees at key junctions to ensure the loop-closure algorithms could capture sufficient feature descriptors.

4.3 Navigation stack initialization

Once the map is saved, the autonomous navigation stack (Nav2) is deployed. This is the most resource-intensive phase of the setup, requiring precise configuration of the Discovery Server to handle high-bandwidth traffic.

4.3.1 Discovery server configuration

The TurtleBot 4 utilizes a FastDDS Discovery Server to manage traffic between the robot and the workstation. The IP address of the robot is dynamic; therefore, it must be visually confirmed from the robot's onboard UI screen and exported as an environment variable before launching Nav2.

```
# Check IP on robot UI screen first
export ROS_DISCOVERY_SERVER=192.168.0.115

ros2 launch turtlebot4_navigation nav2.launch.py \
    slam:=false \
    use_sim_time:=false \
    params_file:=/home/user/custom_nav2_params.yaml
```

Listing 4.6: Nav2 launch sequence

Parameter explanation:

- `slam:=false`: loads a pre-existing static map instead of generating a new one.
- `use_sim_time:=false`: forces the nodes to use the system clock, essential for hardware deployment.
- `params_file`: points to the custom configuration file detailed in Chapter 3, containing the relaxed yaw tolerances and optimized controller frequency.

4.4 Smart Patrol node deployment

The final step is the deployment of the custom logic node: `rosnode_patrol`.

4.4.1 Waypoint selection and compilation

The patrol route is defined as a list of coordinate waypoints $[x, y, yaw]$ within the `smart_patrol.py` script. These coordinates can be obtained by the "Publish Point" feature of Nav2.

To ensure that the ROS path environment is correctly updated with any new dependencies or a set of new points, a targeted build is performed before execution.

```
# 1. Compilation
cd ~/turtlebot4_ws
colcon build --symlink-install --packages-select rosnode_patrol

# 2. Execution from Source
source /opt/ros/jazzy/setup.bash
cd /mnt/c/Users/user/Desktop/tesi/rosnode_chatgpt
source install/setup.bash
ros2 run rosnode_patrol smart_patrol
```

Listing 4.7: Node compilation and execution

4.5 Experimental scenarios

To evaluate the robustness of the system three distinct operational scenarios were designed, varying in environmental complexity and surveillance frequency.

4.5.1 Scenario A: Narrow corridor

Objective: test navigation in a simple environment.

- **Environment:** a uniform hallway with artificial lighting.
- **Configuration:** surveillance checks every **0.5 seconds**.
- **Challenge:** the 0.5 second interval made some distance checks miss targets due to the heavy filtering, where the robot's sensors fail to correctly distinguish a person if they are only partially visible within the narrow field of view.

4.5.2 Scenario B: Entrance hall

Objective: second test for a simple environment.

- **Environment:** wide open space with glass doors and variable natural light.
- **Configuration:** surveillance checks every **0.5 seconds**.

- **Challenge:** natural light can wash out the camera sensor, potentially affecting the CLIP model's ability to detect people. The 0.5 second interval also led to missed detections when a person was only briefly in the frame, similar to the corridor scenario, but the wider field of view provided more opportunities for detection.

4.5.3 Scenario C: Combined route

Objective: determine the impact of surveillance frequency on detection latency and in a more complex scenario.

- **Environment:** a loop connecting the corridor and entrance.
- **Configuration:** surveillance checks every **0.5 seconds**.
- **Observation:** combination of the two previous scenarios, in order to test the system's endurance and stability over an extended patrol. The same detection challenges were observed, with the 0.5 second interval still causing some missed detections when a person walked briskly across the robot's field of view.

4.5.4 Scenario D: Combined route 5Hz variant

Objective: determine the impact of surveillance frequency on detection latency and in a more complex scenario.

- **Environment:** a loop connecting the corridor and entrance.
- **Configuration:** high frequency checks every **0.2 seconds**.
- **Observation:** increasing the frequency to 5Hz (0.2s) provided more opportunities to catch a target. However, the problem didn't go away when people stood next to the robot in a partial way, as the heavy filtering still caused some missed detections. The increased frequency did help in scenarios where a person was only briefly visible.

4.5.5 Scenario A: narrow corridor patrol (0.5s frequency)

Objective: Test the navigation controller's stability and the sensor fusion's accuracy in a highly constrained, simple environment with a 0.5-second polling interval.

Environment topology and route configuration

The first test environment was a uniform, narrow hallway with consistent artificial lighting. This environment tests the limits of the Lidar's vision filter, as the physical walls are in close proximity to the robot's lateral sensors.

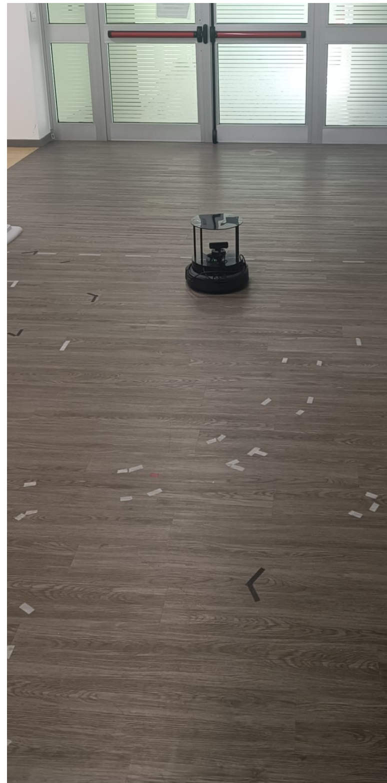


Figure 4.1: Physical view of the narrow corridor used for Scenario A. The robot patrols between the two waypoints along this hallway.

The patrol route was defined by a sequence of two waypoints along the longitudinal axis of the hallway.

Waypoint ID	X Coordinate (m)	Y Coordinate (m)	Orientation (Yaw)
P1	5.9	-0.07	0.0
P2	10.0	-1.35	0.0

Table 4.1: Coordinate configuration for the Scenario A patrol route. The robot navigates between these points in a loop.

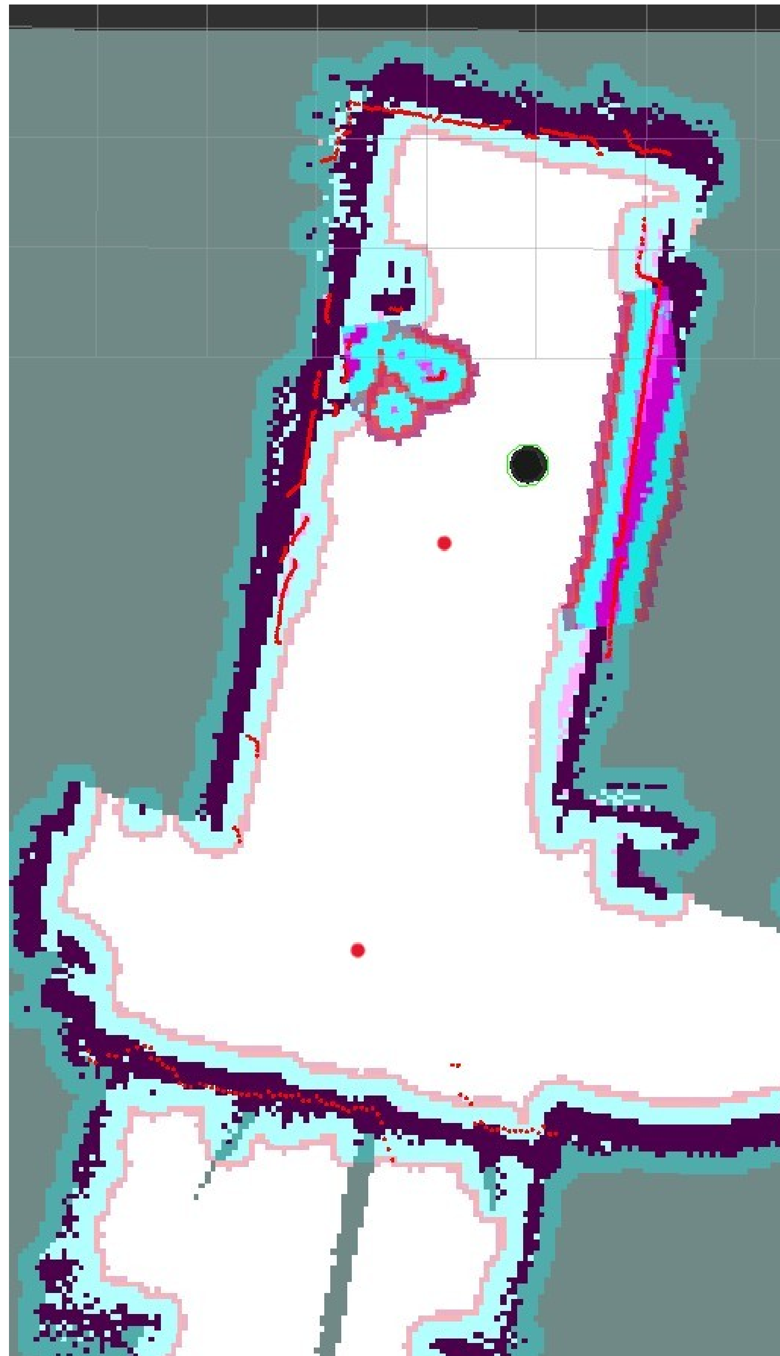


Figure 4.2: RViz visualization of the Scenario A global costmap and designated waypoints (points in red).

Nominal execution: successful semantic detection

Under nominal conditions, the Nav2 local planner successfully routed the robot down the edges of the corridor without triggering recovery behaviors. The artificial lighting provided excellent contrast for the OAK-D camera.

When a human subject stood squarely in the center of the hallway, the system performed perfectly. The LLM correctly identified the subject, and because the subject was directly in front of the robot, the physical geometry fell perfectly within the $\pm 15°$ Lidar cone.

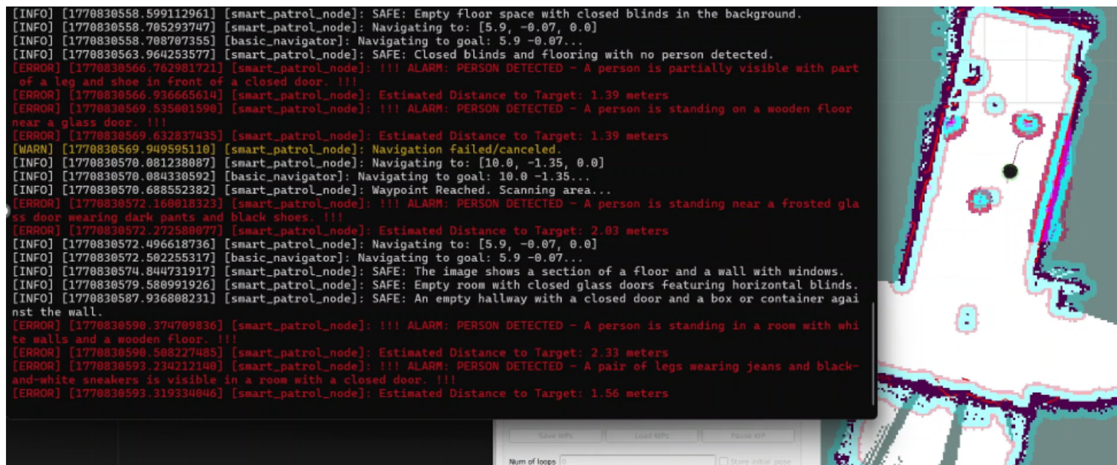


Figure 4.3: Terminal log indicating a successful "ALARM" state and accurate distance measurement.

Edge case analysis: distance estimation failure due to latency

The 0.5-second polling interval introduced a notable vulnerability. Because the robot only checks its environment every half-second, a person walking briskly perpendicular to the robot could pass through the edge of the camera's field of view between API calls. In these instances, the robot's physical movement combined with the API latency caused the distance measurement to be inaccurate. The LLM would correctly identify the presence of a person, but by the time the Lidar data was processed, the person had already moved out of the detection zone, resulting in a distance reading concerning the closest frontal object.

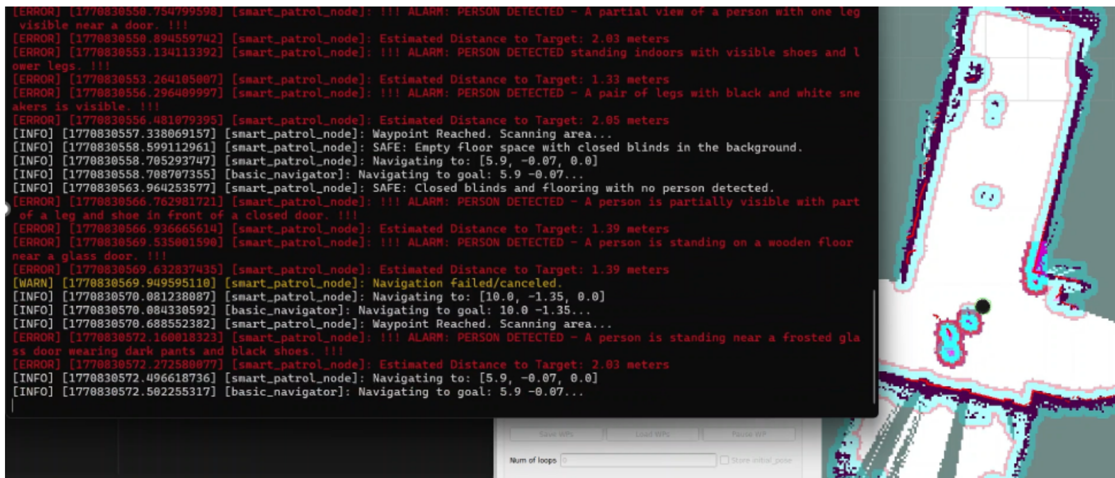


Figure 4.4: Terminal log illustrating the latency miss scenario. The person enters and exits the camera's field of view between two 0.5-second checks, resulting in a wrong distance estimation.

4.5.6 Scenario B: entrance hall (0.5s frequency)

Objective: Test visual reasoning limits and detection accuracy in an environment similar to the first scenario, but adding slight more complex route.

Environment topology and lighting dynamics

The entrance hall presented a different challenge compared to the corridor. It is a wide-open physical space featuring glass doors and a more unstructured layout.

Figure 4.5: Physical view of the Scenario B.

The patrol route was defined by a sequence of two waypoints along the longitudinal axis of the hallway.

Waypoint ID	X Coordinate (m)	Y Coordinate (m)	Orientation (Yaw)
P1	10.0	-3.07	0.0
P2	10.0	-0.58	0.0
P3	10.9	2.1	0.0
P4	9.7	3.3	0.0

Table 4.2: Coordinate configuration for the Scenario B patrol route. The robot navigates between these points in a loop.

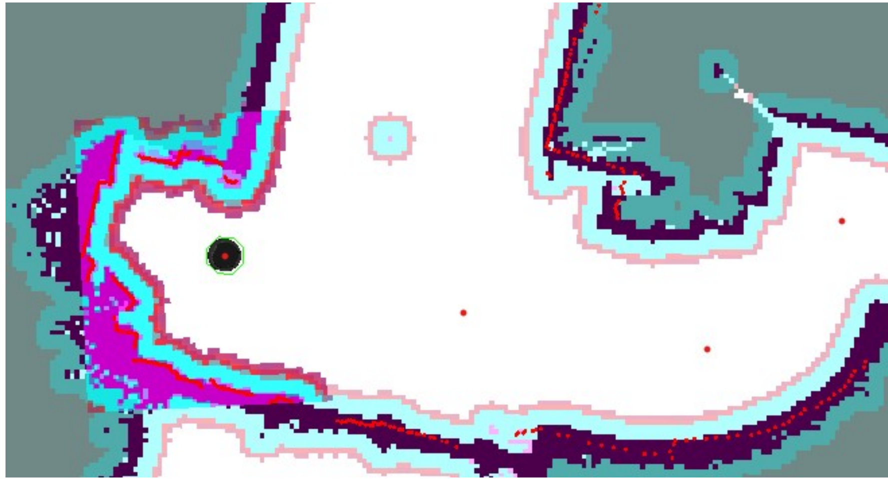


Figure 4.6: RViz visualization of Scenario B, showing sparse obstacle boundaries.

Nav2 challenges in re route execution

Normally, Nav2 is able to navigate without much issue in all of the environments tested. However, the curve proved to be a challenge: if the points were not carefully placed, the robot struggled to find a valid path and would get stuck in a recovery loop. This was likely due to the fact that the robot's local costmap was not able to generate a clear path around the curve, especially if the waypoints were too close to the walls or if there were any dynamic obstacles (e.g., people walking by) that further constrained the available space.

The first group was composed of only three points, with only one bridging the entrance hall and the end of the curve. When the robot had to navigate from one such extreme to another, Nav2 automatically chose the shortest path, which was very close to the obstacles.

Repetition of correct detection and latency miss scenarios

The same detection scenarios observed in the corridor were also present in the entrance hall. When a person stood directly in front of the robot, the system performed perfectly. However, when a person walked briskly across the robot's field of view, the 0.5-second polling interval caused some detections to be missed, resulting in inaccurate distance measurements.

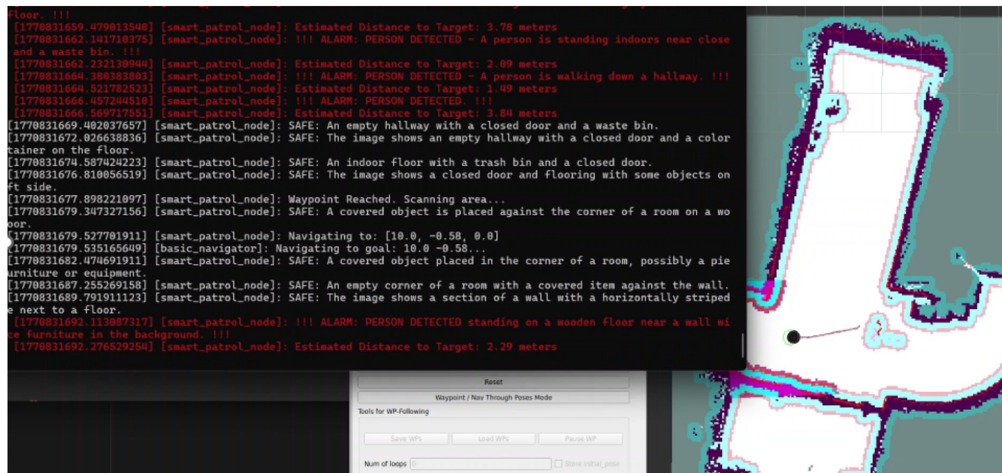


Figure 4.7: Terminal log indicating a successful "ALARM" state and accurate distance measurement.

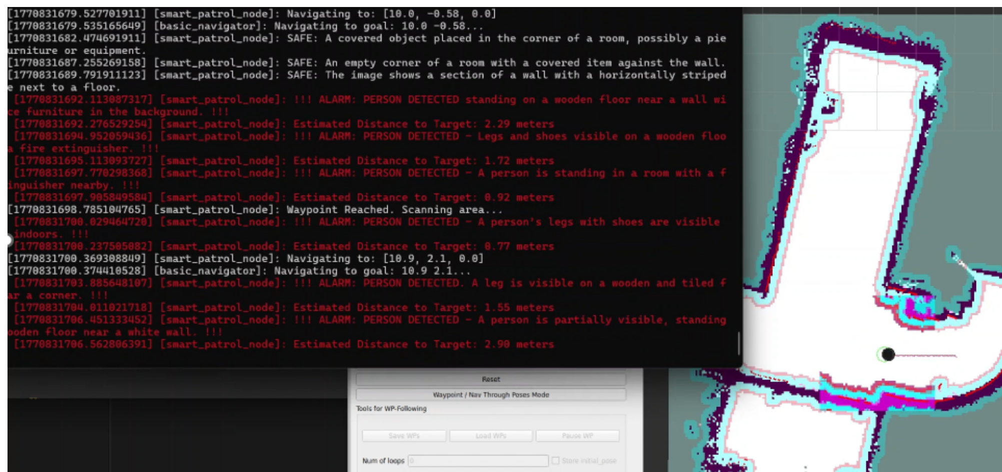


Figure 4.8: Terminal log illustrating the latency miss scenario. The person enters and exits the camera's field of view between two 0.5-second checks, resulting in a wrong distance estimation.

4.5.7 Scenario C: combined route (0.5s)

Objective: Combine the Scenarios A and B into a continuous patrol route to test the system's endurance.

Route integration

This scenario combined the previous waypoints into a continuous, looping patrol.

Figure 4.9: Partial physical view of the combined route used for Scenario C. The robot patrols between the 12 waypoints along this hallway.

Figure 4.10: Partial physical view of the combined route used for Scenario C. The robot patrols between the 12 waypoints along this hallway.

The patrol route was defined by a sequence of 12 waypoints along the two zones.

Waypoint ID	X Coordinate (m)	Y Coordinate (m)	Orientation (Yaw)
P1	4.2	3.8	0.0
P2	3.8	2.6	0.0
P3	4.45	1.75	0.0
P4	4.75	0.15	0.0
P5	4.25	-1.3	0.0
P6	3.85	-2.25	0.0
P7	3.5	-3.2	0.0
P8	3.3	-4.1	0.0
P9	2.5	-2	0.0
P10	1.1	-1.3	0.0
P11	-0.2	0.8	0.0
P12	-2	0.15	0.0

Table 4.3: Coordinate configuration for the Scenario C patrol route. The robot navigates between these points in a loop.

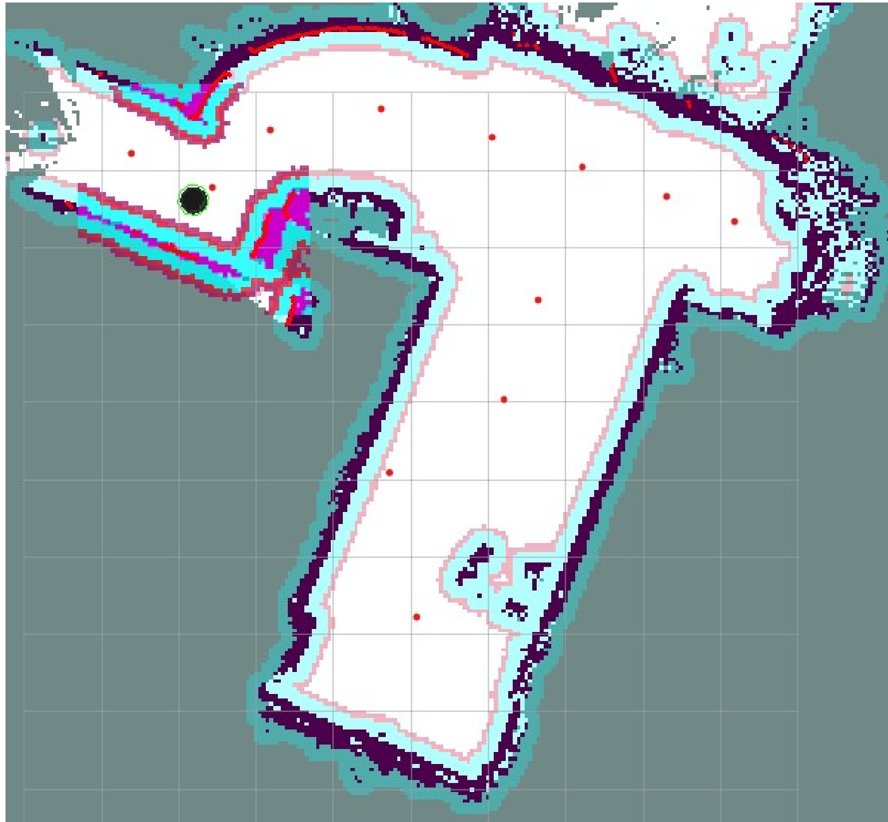


Figure 4.11: RViz visualization of Scenario C, showing the full patrol route encompassing both the corridor and entrance hall.

Route

The poins were reused to make the loop between the two zones, and the same detection scenarios observed in the previous two scenarios were also present here. The robot successfully navigated the entire route without getting stuck, but the 0.5-second polling interval still caused some distances to be miscalculated when a person walked briskly across the robot's field of view, effectively demonstrating the same latency miss scenario observed in the previous two scenarios. The increased complexity of the route did not introduce new detection challenges, but it did provide a more rigorous test of the system's endurance and stability over an extended patrol.

```
1            # entrance curve, left to right
2            [4.20, 3.80, 0.0],
3            [3.80, 2.60, 0.0],
4            [4.45, 1.75, 0.0],
5            [4.75, 0.15, 0.0],
```

```
6                # hallway entrance, left to right
7                [4.25, -1.30, 0.0],
8                [3.85, -2.25, 0.0],
9                [3.50, -3.20, 0.0],
10               [3.30, -4.10, 0.0],
11               # go back to the center
12               [3.50, -3.20, 0.0],
13               [3.85, -2.25, 0.0],
14               # hallway corridor, top to bottom
15               [2.50, -2.0, 0.0],
16               [1.10, -1.30, 0.0],
17               [-0.20, 0.80, 0.0],
18               [-2.0, 0.15, 0.0],
19               # go back to the center
20               [-0.20, 0.80, 0.0],
21               [1.10, -1.30, 0.0],
22               [2.50, -2.0, 0.0],
23               [3.85, -2.25, 0.0],
24               # go back to the beginning of the curve
25               [4.25, -1.30, 0.0],
26               [4.75, 0.15, 0.0],
27               [4.45, 1.75, 0.0],
28               [3.80, 2.60, 0.0],
```

Listing 4.8: Loop waypoints configuration

Repetition of correct detection and latency miss scenarios

This combined route scenario presented the same problems as the previous two scenarios, with the same correct detection and latency miss cases. The increased patrol complexity did not mitigate the latency issue, as the 0.5-second polling interval still allowed for situations where a person could briefly enter and exit the camera's field of view without being detected in time for an accurate distance measurement.

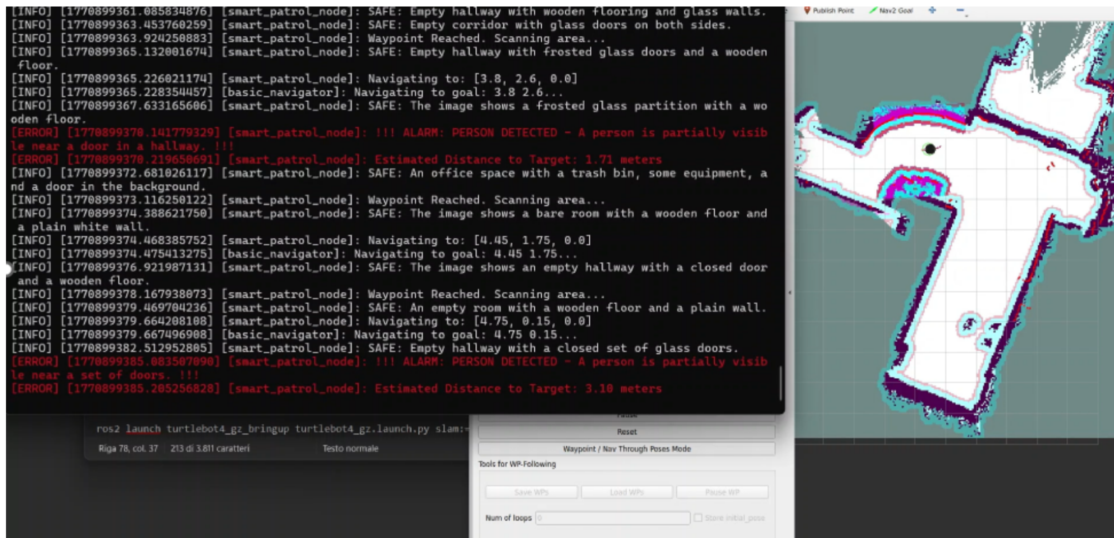


Figure 4.12: Terminal log indicating a successful "ALARM" state and accurate distance measurement.

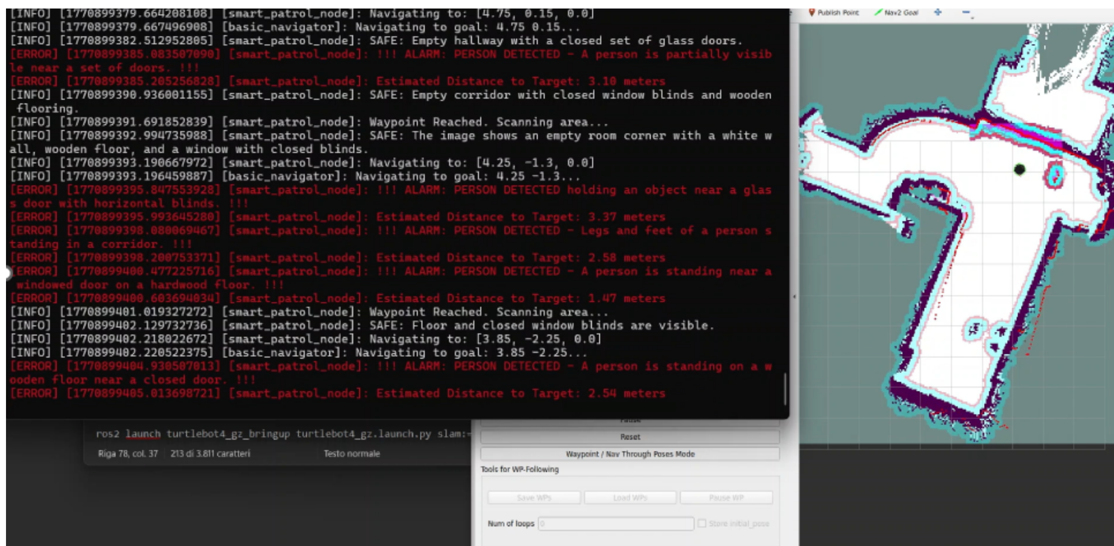


Figure 4.13: Terminal log illustrating the latency miss scenario. The person enters and exits the camera's field of view between two 0.5-second checks, resulting in a wrong distance estimation.

4.5.8 Scenario D: high-frequency stress test (0.2s polling)

Objective: Determine if increasing the surveillance polling frequency mitigates the latency vulnerabilities observed in previous scenarios, and to evaluate the impact of this increased frequency on the overall system stability and network load.

Test parameters and rationale

The previous scenarios (A, B, and C) definitively established that a 0.5-second (2Hz) polling interval is insufficient for a dynamic security environment. Humans walking briskly transverse to the robot's axis of movement were frequently lost between frames, resulting in erroneous Lidar distance calculations because the physical object had departed the central cone before the AI response triggered the Lidar callback.

To counteract this, the 'time.sleep()' parameter within the asynchronous surveillance loop was reduced from 0.5 seconds to 0.2 seconds, increasing the semantic polling rate to 5Hz. The robot was deployed on the same 12-waypoint loop utilized in Scenario C (Section 4.5.3) to provide a 1-to-1 baseline comparison.

System endurance and network load

A primary concern when configuring the 5Hz interval was the potential for network congestion and API rate-limiting. The system must capture an image, encode it to Base64, transmit the payload over the WSL 2 network bridge to the OpenAI servers, await the LLM inference, decode the response, and execute the ROS 2 callbacks.

Observations: Surprisingly, the system architecture proved highly resilient to the increased load. The MultiThreadedExecutor and the explicit separation of the Nav2 action server logic from the Vision-Language loop prevented the robot's movement from stuttering. The TurtleBot 4 maintained a fluid velocity profile while continuously streaming images to the cloud. The API response times were consistently low, meaning the asynchronous queue remained stable without buffering stale images.

Latency miss mitigation and detection improvement

The reduction of the polling interval yielded an immediate and quantifiable improvement in semantic responsiveness. By evaluating 2.5 times more frames per second, the probability of capturing a moving target in the center of the camera's field of view increased drastically.

In previous scenarios, a person walking across the hallway might only be captured in one frame (often near the edge). At 5Hz, that same person was typically captured

in three consecutive frames (entry, center, exit).

This yielded two major improvements:

1. **Lower miss rate:** the robot no longer missed subjects passing quickly in front of it. The visual detection rate for transverse movement approached 100%.

2. **Improved distance accuracy:** since the AI detected the person while they were still physically occupying the center of the frame (rather than catching them on their way out), the synchronized Lidar callback successfully pinged the person's geometry. The frequency of erroneous measurements plummeted for targets crossing the robot's path.

Kinematic and semantic paradox

While the high-frequency polling solved the issue of missing fast-moving targets, it paradoxically isolated and highlighted a separate architectural limitation, a mismatch in the field of view.

As demonstrated above, the camera often captured intruders just as they entered the frame. The Luxonis OAK-D camera possesses a wide horizontal FOV (approximately 70°). Conversely, to prevent the Nav2 stack from hallucinating obstacles from the corridor walls, the Lidar software filter (defined in the `get_front_distance` function) was strictly constrained to a narrow 30° mathematical bound.

When a person stands partially occluded or is just entering the edge of the camera's vision (e.g., at an angle of 25° relative to the robot's center):

1. **Semantic layer (Success):** the LLM receives the image, identifies the partial human geometry (e.g., an arm or leg on the far edge), and correctly triggers the `ALARM` state.

2. **Spatial layer (Failure):** the Lidar function is triggered, but because the person's physical angle (25°) falls outside the rigid bounds of the filtered `front_cone` array ($\pm 15°$), the Lidar logic ignores the person entirely.

3. **Resulting error:** the Lidar beams pass straight past the person and strike the back wall, resulting in the system reporting an erroneously large distance (e.g., measuring the wall at 5.0m instead of the person at 1.0m, or reporting `inf`).

Conclusion of Scenario D: the 5Hz stress test proved that increasing the semantic polling rate is vital for real-world security applications, as it successfully eliminates the temporal latency that allows targets to slip by undetected. However, the resulting FOV mismatch proves that an intelligent autonomous system cannot rely on static sensor fusion logic. The robot possesses the semantic intelligence

to know a threat is present on its periphery, but it currently lacks the dynamic kinematic reflexes, such as visual servoing or automatically rotating its chassis to center the detected threat, that are required to accurately measure it. This limitation highlights a critical area for future development in LLM-driven robotics.

Global planner failures and recovery behaviors

Normally, Nav2 is capable of navigating through all of the environments tested without significant issue. However, the unstructured layout of the entrance hall, combined with the specific placement of waypoints around the curve, introduced severe path planning challenges. If the waypoints were not meticulously placed the robot struggled to find a valid kinematic path, resulting in the system aborting the goal and logging a "navigation failed/canceled" warning.

Unreachable goals: to understand this failure, it's important to analyze how the Nav2 Global Planner calculates a path. The global costmap is represented as a 2D grid where each cell holds a cost value ranging from 0 (free space) to 254 (lethal obstacle), with the Inflation layer propagating a decreasing cost gradient outward from any lethal obstacle up to a defined `inflation_radius`.

If a user-defined waypoint (P_x, P_y) is placed too close to a physical wall, its corresponding costmap cell might possess a value of 253 (inscribed cost) or higher. When the A* or Dijkstra algorithm utilized by the `NavFn` planner attempts to search for a route to this coordinate, the mathematical heuristic determines the node is entirely unreachable, as moving into that cell would result in a guaranteed physical collision.

Dynamic obstruction and controller timeouts: even if a waypoint is mathematically valid on the global costmap, dynamic obstacles can cause local failures. During Scenario B, there were instances where people walked through the entrance hall, inadvertently standing directly on the robot's intended path.

Since the local costmap updates at a high frequency (utilizing the Lidar `/scan` topic), the system immediately recognized the path was blocked. If the surrounding space was too constrained to calculate a detour, the controller halted the robot. Once the `wait_for_server_timeout` threshold was exceeded, the controller server reported a failure to the main Behavior Tree.

Recovery behaviors and system resilience

When the primary `ComputePathToPose` or `FollowPath` actions fail, the Nav2 Behavior Tree does not immediately abort the mission. Instead, it triggers a sequence of automated recovery behaviors designed to clear the error state.

During the curve navigation in the entrance hall, the robot was observed executing the following recovery sequence:

1. **`ClearEntireCostmap`:** the robot assumes its local costmap is polluted with stale data (e.g., a person who has already walked away but left an obstacle in the memory). It wipes the local costmap clean and waits for a fresh Lidar scan.

2. **`Spin`:** if clearing the costmap fails to open a valid path, the robot rotates 360° in place. This allows the Lidar to sweep the entire immediate area, updating the local costmap with highly accurate, 360° spatial data.

3. **`BackUp`:** if spinning fails, the robot attempts to reverse slightly to extract itself from what it perceives as a tight corner, before asking the Global Planner to try generating a path again.

Waypoint follower logic and route continuity

If all recovery behaviors are exhausted and a path still cannot be generated (e.g., the waypoint is permanently inside a wall's inflation radius, or a physical object completely blocks the hallway), the underlying `NavigateToPose` action definitively aborts.

In a poorly designed system, this would cause a fatal crash, freezing the robot in place indefinitely. However, the Smart Patrol architecture leverages the Nav2 `WaypointFollower` server. The custom `smart_patrol_node` is programmed to catch this specific `TaskResult.FAILED` or `TaskResult.CANCELED` status code.

Instead of terminating the script the node accepts the failure, logs the warning, and immediately commands the robot to proceed to the next sequential coordinate in the `waypoints` array.

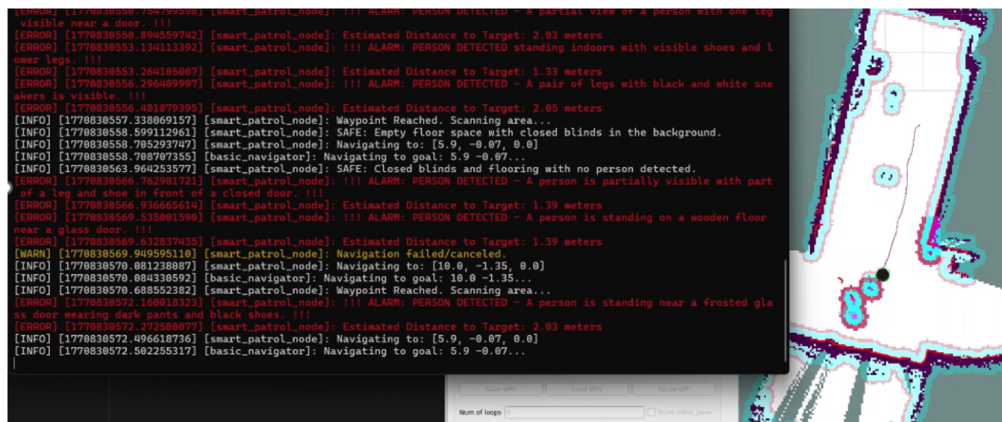


Figure 4.14: Terminal output showing the system catching a path planning failure and skipping to the subsequent waypoint to maintain the patrol route.

This fault-tolerant design is a critical requirement for autonomous security patrols. The system demonstrated that a localized path planning failure at one specific geographic point does not compromise the whole path; instead, the robot skips the unreachable area and resumes surveillance.

Chapter 5

Discussion and Future Work

This final chapter synthesizes the empirical data gathered during the experimental deployments detailed in Chapter 4. It provides a critical evaluation of the Smart Patrol architecture, discussing the successful integration of LLMs with classical robotic navigation, while addressing the systemic limitations discovered during high-frequency physical testing. Finally, it outlines a comprehensive roadmap for future research, proposing architectural and kinematic upgrades to improve the system's robustness, responsiveness, and real-world applicability in enterprise security contexts.

5.1 Discussion of findings

The primary objective of this thesis was to design, implement, and validate an autonomous patrol robot capable of semantic scene understanding using an LLM, specifically operating within a ROS 2 and Windows Subsystem for Linux (WSL 2) ecosystem. The findings indicate that while the system successfully achieved its core objectives, the intersection of cloud-based AI and local kinematic control introduces novel challenges in spatial-temporal synchronization.

5.1.1 Architectural and middleware viability

One of the most significant technical achievements of this research was the stabilization of the WSL 2 to native-Ubuntu communication bridge. Traditional robotics research relies almost exclusively on native Linux environments due to the complexities of the Data Distribution Service (DDS) middleware, which heavily utilizes UDP multicast for node discovery.

The implementation of the FastDDS Discovery Server proved highly effective in bypassing the limitations of WSL 2. Throughout communication between the

TurtleBot 4 (acting as the centralized Discovery Server) and the Windows host (acting as the client), the system maintained a robust connection. It successfully streamed high-bandwidth topics, such as the `/tf` coordinate transforms and raw `Image` arrays, without triggering the problematic network states that stalled early development. This effectively validates WSL 2 as a highly capable, lightweight alternative to dual-booting or hardware-heavy virtual machines for complex ROS 2 development.

5.1.2 Navigational reliability and controller tuning

The Navigation 2 (Nav2) stack demonstrated exceptional reliability once the default parameters were tuned to accommodate the specific physical constraints of the TurtleBot 4 hardware and the environment.

The yaw tolerance optimization

A critical early failure mode was the rotational behavior upon reaching a waypoint. The default `yaw_goal_tolerance` of 2.8° caused the robot to enter a micro-correction state: this resulted in significant stalling at each waypoint, disrupting the fluidity of the patrol route.

In a physical environment with floor friction, wheel slip, and minor odometry drift, achieving a 2.8° alignment is often physically impossible without continuous over-correction. By relaxing the `yaw_goal_tolerance` to 6.28 radians (2π, or 360°), the system essentially disabled the `GoalAlign` critic's authority upon arrival. The robot prioritized positional accuracy (X/Y coordinates) over rotational accuracy. This optimization was crucial for maintaining the fluidity of the continuous patrol route, allowing the robot to immediately transition to the next waypoint without stalling.

5.1.3 Trade-off between semantic and spatial intelligence

The most profound finding of this research lies in the divergence between semantic intelligence (knowing *what* is there) and spatial intelligence (knowing *where* it is).

The OpenAI GPT-4o model demonstrated remarkable zero-shot reasoning capabilities. The model successfully operated as a binary classifier ("SAFE" vs. "ALARM"), proving capable of identifying partially occluded humans under varying lighting conditions. It successfully abstracted the visual data, negating the need to train a custom YOLO or SSD model on a specific dataset.

However, the empirical data from the high-frequency Scenario C highlighted a severe architectural bottleneck when linking this semantic classification to the Lidar's spatial data. The Lidar filtering algorithm, mathematically constrained to a $\pm 15°$ vision cone, acted as a rigid spatial gatekeeper.

The frequency-latency paradox and temporal drift

The experiments varying the surveillance polling frequency (0.5s vs 0.2s) exposed a critical paradox in the system's design. This can be modeled by defining the total temporal latency (T_{total}) of a single perception loop:

$$T_{total} = t_{capture} + t_{network_up} + t_{inference} + t_{network_down} + t_{ros_callback} \tag{5.1}$$

Because T_{total} is heavily influenced by the non-deterministic $t_{inference}$ (the time the LLM takes to generate a response), the physical state of the world changes while the robot "thinks". If an intruder is moving at velocity v_{target}, their physical displacement during the API call is $\Delta d = v_{target} \times T_{total}$.

- **At 0.5s (2Hz):** the API calls were infrequent enough that moving targets often entered and exited the camera's field of view entirely between checks. This resulted in incorrect distances (the person was detected, but the distance was wrong).
- **At 0.2s (5Hz):** the visual detection rate approached 100%, successfully catching targets on the periphery of the frame. However, because Δd was still non-zero due to T_{total}, by the time the `ALARM` triggered the Lidar callback, the intruder had physically moved outside the rigid $\pm 15°$ Lidar cone.

Consequently, the faster the robot "thought," the more frequently it correctly calculated the distance to the target, resulting in less incorrect readings (e.g. the Lidar pinging the background wall instead of the intruder). This proves that high-level AI reasoning cannot compensate for rigid, static sensor fusion algorithms in dynamic environments.

5.2 System limitations

While the Smart Patrol architecture serves as a successful proof-of-concept, several limitations restrict its immediate deployment in enterprise security environments.

5.2.1 Cloud latency and open-loop perception

The reliance on a cloud-based API introduces an uncontrollable external variable. While the asynchronous Python threading prevented the ROS 2 node from crashing while waiting for the API, it did not solve the issue of data staleness.

Furthermore, the current architecture operates in an open-loop manner regarding threats. The vision system detects a threat, and the Lidar attempts to measure it,

but the navigation stack (Nav2) remains completely isolated from this information. The robot acts as a tool to its own navigation stack: it continues driving its pre-programmed route regardless of what the AI detects. A true security robot must possess the capability to halt its patrol and actively track a detected anomaly.

5.3 Future work

To evolve this system from a foundational prototype into a robust, autonomous security agent, several architectural and kinematic upgrades are proposed for future iterations of this research.

5.3.1 Visual servoing and dynamic threat tracking

To solve the distance measurement and the partial occlusion problem detailed in Chapter 4, the robot must develop kinematic reflexes. Future implementations should introduce a "Visual Servoing" feedback loop.

Instead of relying on a static $\pm 15°$ Lidar cone to blindly guess if the target is still there, the system should parse the bounding box coordinates (or a general spatial location like "Left", "Center", "Right") inferred by the LLM. If the LLM detects an intruder on the left edge of the frame, the `smart_patrol_node` should immediately preempt the Nav2 patrol goal and issue a `cmd_vel` rotational velocity command to turn the robot's chassis to the left.

By actively rotating to center the threat in the camera's field of view, the target will naturally be forced into the center of the Lidar's processing cone, guaranteeing an accurate distance measurement and eliminating the spatial mismatch.

Theoretical implementation of kinematic reflexes

To achieve this, the `smart_patrol_node` could be expanded to include a preemptive Proportional (P) controller. The theoretical code implementation below demonstrates how the node would cancel the current Nav2 goal and publish a direct `Twist` message to center the target.

```
1 def execute_visual_servoing(self, target_position):
2     """
3     Interrupts Nav2 and rotates the robot to center the detected
    threat.
4     target_position: String ("left", "right", "center") provided
    by the LLM
5     """
6     # 1. Cancel the current patrol route
7     if not self.navigator.isTaskComplete():
```

```
8            self.get_logger().warn("Threat detected! Canceling Nav2
        Goal.")
9            self.navigator.cancelTask()
10
11       # 2. Setup direct velocity publisher
12       cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)
13       twist_msg = Twist()
14
15       # 3. Proportional Control Logic
16       # Positive angular.z turns Left, Negative turns Right
17       rotation_speed = 0.5
18
19       if target_position == "left":
20           self.get_logger().info("Servoing: Rotating Left to center
        target.")
21           twist_msg.angular.z = rotation_speed
22       elif target_position == "right":
23           self.get_logger().info("Servoing: Rotating Right to center
         target.")
24           twist_msg.angular.z = -rotation_speed
25       else:
26           self.get_logger().info("Target centered. Halting for Lidar
         scan.")
27           twist_msg.angular.z = 0.0
28
29       # 4. Execute rotation briefly
30       cmd_vel_pub.publish(twist_msg)
31       time.sleep(0.5)
32
33       # 5. Stop the robot and trigger Lidar distance check
34       twist_msg.angular.z = 0.0
35       cmd_vel_pub.publish(twist_msg)
36
37       accurate_distance = self.get_front_distance()
38       self.get_logger().info(f"Verified Distance: {accurate_distance
        }m")
```

Listing 5.1: Theoretical ROS 2 implementation for Visual Servoing and target centering.

This code effectively transforms the robot into an active tracker. Upon detecting a threat, it immediately halts its patrol, rotates to center the target, and then performs a Lidar scan to obtain an accurate distance measurement. This would significantly enhance the system's reliability in real-world security scenarios, where targets are often moving and partially occluded.

5.3.2 Migration to local Vision-Language Models (Edge Computing)

To eliminate the vulnerabilities associated with cloud dependency and variable API latency (T_{total}), future architectures should migrate the semantic reasoning to the edge.

With the rapid advancement of quantized, small-parameter models, it is becoming feasible to run complex VLMs directly on local hardware. While the Raspberry Pi 4 on the TurtleBot lacks the necessary compute, the system architecture could be modified so the WSL 2 workstation,which often possesses a dedicated GPU, hosts the model locally.

By hosting a local inference server, the system would reduce the inference network latency to single-digit milliseconds. This would dramatically shrink the Δd temporal drift, allowing the robot to react to intruders almost instantaneously, and ensuring it remains fully operational even during external internet outages.

5.3.3 Transition to 3D sensor fusion

The current sensor fusion architecture relies heavily on a 2D Lidar "slice" of the world (the `LaserScan` message). This is inherently fragile for security applications, as the 2D plane can easily pass between a person's legs or shoot over small, dynamically introduced obstacles.

The Luxonis OAK-D camera utilized in this research is equipped with stereo depth sensors capable of generating rich 3D point clouds. Future work should bypass the 2D RPLIDAR entirely for threat localization. Instead, the architecture should fuse the LLM's semantic output directly with the OAK-D's depth map. By analyzing the depth pixels corresponding to the visual bounding box where the LLM identified the intruder, the robot can achieve highly accurate 3D localization without relying on physical hardware alignment or mathematical cone filters.

5.3.4 Multi-agent orchestration via LLM

Finally, the semantic nature of LLMs makes them ideal for multi-agent systems. Future research could deploy multiple TurtleBots in a single facility. If Robot A detects an anomaly but loses track of it, the LLM could dynamically generate a new Nav2 coordinate goal for Robot B to intercept the target based on the global facility map. The LLM would serve as the central dispatcher for an entire fleet of autonomous security agents.

5.4 Conclusion

The integration of Vision-Language Models into the ROS 2 ecosystem marks a definitive paradigm shift in autonomous robotics. This thesis has successfully demonstrated that unstructured, generative AI can be securely tethered to deterministic robotic middleware to create a functioning Smart Patrol system.

By documenting the successful implementation of WSL 2 networking architectures, optimizing the Nav2 stack for physical constraints, and critically exposing the bottlenecks between semantic reasoning speed and spatial measurement physics, this research lays the empirical groundwork for the next generation of dynamic, self-correcting autonomous agents. The transition from machines that simply navigate space to machines that semantically understand and react to it is no longer a theoretical concept but an achievable reality.