POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineer

Master's Degree Thesis

A Modular LLM-Based Framework for Semantic Navigation and Perception in Mobile Robotics

Supervisors

Prof. Alessandro RIZZO

Ph.D. Pangcheng David CEN CHENG

Candidate

Giuseppe Antonio GENTILE

July 2025

Abstract

As robots become increasingly integrated into human environments, the ability to interact intuitively through natural language has become a crucial goal. Traditional robotic control systems require structured inputs and predefined behaviors, limiting their adaptability in dynamic, real world environments. Recent advances in Large Language Models (LLM) offer a new paradigm: harnessing language as a general interface for reasoning, perception, and decision making. However, integrating LLMs with embodied agents presents fundamental challenges, including grounding instructions in physical space, ensuring safety, and linking symbolic language and low-level robotic actions.

This thesis explores the integration of Large Language Models (LLMs) into robotic systems for natural language-driven control and perception. The proposed architecture connects a GPT-based reasoning agent with a ROS 2 navigation stack and a visual pipeline comprising BLIP for visual question answering and YOLOv8 with depth sensing for object localization. Natural language instructions are parsed into structured commands using prompt injection, verified for safety via an Abstract Syntax Tree (AST) parser, and dispatched to the robot for execution.

The system is tested in a simulated environment with a TurtleBot4 platform. Experimental results demonstrate reliable performance in goal navigation and partial success in end-to-end instruction-to-verification tasks. While predefined commands achieve a 100% success rate, semantically inferred goals reveal the limits of symbolic mapping and perception coverage. Vision modules exhibit strong accuracy on binary visual questions and acceptable spatial detection within proximity thresholds.

This work contributes a modular, extensible framework for embodied LLMs, validating its potential for grounded, multimodal interaction. Future extensions are discussed to address real-time feedback integration, memory, and multi-turn dialogue.

Acknowledgements

First and foremost, I dedicate this thesis to my parents, whose unwavering love, sacrifices, and encouragement have provided me with the invaluable opportunity to pursue my studies and achieve this milestone. Their belief in my potential has been a constant source of motivation and inspiration.

I am deeply grateful to my sisters, Anna and Micaela, whose endless support and reassurance have carried me through challenging times. Their empathy, kindness, and presence have truly made a difference, making difficult moments more manageable and successes sweeter.

My heartfelt appreciation goes to Mariarosa, my other half, who has continually encouraged me to persevere and strive for excellence. Her patience, understanding, and relentless positivity have been pivotal in helping me reach the finish line with confidence and pride.

A special thank you goes to my beloved nephews and nieces—Silvano, Sofia, Francesco, Paolo, Anita, and little Andrea, who is yet to arrive. Their joy, innocence, and affection have been a delightful source of inspiration, reminding me of the beauty and simplicity of life's most precious moments.

Finally, I wish to express my sincere gratitude to my lifelong friends, Donato, Giorgio, Danilo, Mario and Michele. Having been by my side from the very first year of university, they have provided constant friendship, humor, support, and countless memorable experiences that have enriched my academic journey and my personal growth.

Table of Contents

List of Tables

List of Figures

Acronyms

LLM
Large Language Model

ROS
Robot Operating System

SLAM
Simultaneous Localization and Mapping

IMU
Inertial Measurement Unit

RGB-D
Red Green Blue + Depth

CoT
Chain of Thought

ToT
Tree of Thought

PLM
Pre-trained Language Model

RNN
Recurrent Neural Network

LSTM
Long Short-Term Memory

TF

Transform Library (for coordinate frames in ROS)

LiDAR

Light Detection and Ranging

SLAM Toolbox

Simultaneous Localization and Mapping Toolbox

vSLAM

Visual Simultaneous Localization and Mapping

GPS

Global Positioning System

AMCL

Adaptive Monte Carlo Localization

Nav2

Navigation 2 (ROS 2 Navigation Stack)

BT

Behaviour Tree

FSM

Finite State Machine

AI

Artificial Intelligence

CaP

Code as Policies

VLM

Vision-Language Model

VQA

Visual Question Answering

ViT

Vision Transformer

BLIP

Bootstrapped Language-Image Pretraining

YOLOv8

You Only Look Once version 8

Chapter 1

Introduction

The integration of natural language understanding with robotic control represents one of the most promising frontiers in the field of embodied Artificial Intelligence (AI). As robots increasingly enter human-centered environments such as homes, hospitals, and warehouses, the ability to interpret and act upon natural language commands is essential for intuitive and seamless Human-Robot Interaction (HRI).

Traditional robotic systems rely on pre-programmed behaviors and constrained user interfaces, often requiring technical expertise or structured inputs. In contrast, Large Language Models, such as GPT-3.5 and GPT-4 [1], offer a new paradigm where high-level tasks can be described using natural language and automatically translated into executable code or symbolic actions.

However, integrating LLMs into robotics presents non-trivial challenges. These include ambiguity in user instructions, real-time constraints, safety verification, and the need to ground abstract reasoning in concrete sensory data and robot capabilities [2, 3].

Recent advancements in LLMs have demonstrated their capabilities not only in natural language processing (NLP) tasks, but also in semantic reasoning, action planning, and code generation [4]. These characteristics suggest the potential for LLMs to serve as high-level controllers in robotic architectures, where they interpret natural language and generate structured plans executable within frameworks like Robot Operating System 2 (ROS 2) [4].

This thesis investigates how such models can be used to enhance autonomous robot behavior by bridging language and control. The goal is to create a system where a robot can navigate and perceive its environment based solely on human instructions, without requiring pre-defined scripts or hard-coded logic.

Despite the recent progress, several key questions remain open:

- Can an LLM reliably interpret human language and produce executable robotic plans?

- How can symbolic goals be mapped from natural language and executed in a ROS 2-based robotic pipeline?
- To what extent can a robot autonomously verify the success of an instruction using visual perception?

This work proposes a hybrid framework that couples a GPT-based reasoning module with a ROS 2 navigation stack and visual perception agents including BLIP [5](Bootstrapping Language-Image Pre-training) and YOLOv8 [6].

The primary objectives of this thesis are:

- To design a modular architecture for robotic systems capable of understanding and executing natural language commands;
- To integrate a LLM-based reasoning agent into a ROS 2 simulation environment using TurtleBot4 [7];
- To implement a vision pipeline based on Visual Question Answering (VQA) with BLIP and 3D object localization using YOLOv8 and depth data;
- To ensure robustness and safety through static code validation (via Abstract Syntax Tree parsing);
- To evaluate the system performance through a series of end-to-end tasks involving navigation, perception, and language understanding.

The architecture developed in this thesis leverages GPT-3.5 to generate ROS-compatible Python code in response to user instructions. This code is validated and executed in a simulated environment using ROS 2 and Ignition Gazebo. Navigation is handled by the Nav2 stack [8], while visual verification is performed by two agents: BLIP for semantic scene understanding and YOLOv8 with depth sensing for spatial localization.

Execution safety is ensured through an AST-based fallback logic [2], which blocks hallucinated or invalid code before runtime. The robot dynamically builds and updates its internal symbolic map as it explores and verifies visual entities.

Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces the fundamental concepts and presents a literature review on large language models, prompting techniques, ROS 2-based robotic architectures, and visual perception models such as BLIP and YOLOv8.

- **Chapter 3** details the methodology adopted in the project, including the software components, simulation environment, prompt engineering strategies, and the modular pipeline for language-driven robot control and visual feedback.
- **Chapter 4** presents the experimental results, analyzing the system's performance in understanding instructions, executing navigation plans, and validating tasks through multimodal perception, with quantitative evaluation across several test cases.
- **Chapter 5** discusses the final conclusions and outlines future directions for extending the current framework, including real-world deployment, interactive dialogue support, and integration of persistent memory and real-time sensory feedback.

Chapter 2

Literature Review

2.1 Large Language Models

Large language models are neural network-based systems trained on huge corpora of textual data to perform a wide range of natural language understanding and generation tasks. Their development has marked a milestone in the field of artificial intelligence, enabling machines to perform not only classic natural language processing (NLP) tasks, such as text classification and question answering, but also complex reasoning, summarization, and multi-turn dialogues. LLMs are primarily built on the Transformer architecture, which employs self-attention mechanisms to process and generate sequences efficiently and in parallel [9].

The Transformer architecture, is based on self-attention mechanisms that allow efficient and parallel processing of sequences. A block diagram illustrating the Transformer model is shown in Figure 2.1.

The evolution of LLMs can be traced through four main stages: statistical language models (e.g., n-grams), early neural language models (e.g., RNN, LSTM), pre-trained language models (PLM, such as BERT [10] and RoBERTa [11]), and the current generation of LLMs, which includes models such as GPT-3 [1], PaLM [12], and LLaMA [13]. These models typically contain billions of parameters and are trained using unsupervised or self-supervised learning paradigms on large-scale corpora from the web.

What distinguishes LLMs from previous PLMs is not only their scale, but also their emerging capabilities. These include contextual learning, the ability to learn new tasks from examples at inference time, following instructions, the ability to respond accurately to natural language commands, and multi-step reasoning, i.e., the ability to solve tasks through intermediate logical steps. These properties have led to the use of LLMs as a fundamental component of general-purpose AI agents.

In robotics, the integration of LLMs enables natural language interfaces that

translate human instructions into robotic actions, a capability explored in this thesis. LLMs such as GPT-3.5 and GPT-4 can support reasoning about spatial navigation, sequential planning, and perceptual grounding when combined with real-time sensory input and environmental context. Despite their capabilities, LLMs still face challenges such as hallucinations, lack of real-time memory, and limited robustness in dynamic environments, which are active areas of research and development.

In the context of this thesis, LLMs are used as high-level planners to interpret user instructions and decide on robotic actions within a simulated environment, highlighting their role in connecting human language and robotic control.

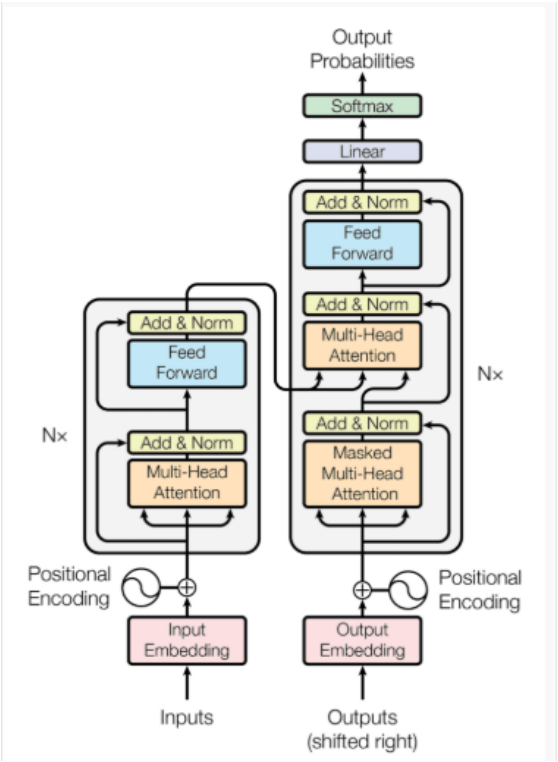


Figure 2.1: The encoder-decoder structure of the Transformer architecture [9]

2.2 Prompting

Prompting is the process of creating input text in such a way as to guide a large language model to produce the desired output. It is a fundamental interface mechanism that allows users to leverage the capabilities of LLMs without modifying their internal weights or retraining them. Prompting has become particularly important with the advent of instruction-following models such as InstructGPT

and ChatGPT, which rely on prompts to perform a wide variety of tasks through natural language interaction.

Prompting strategies can be divided into three main categories: zero-shot, few-shot, and instruction-optimized prompting. In zero-shot prompting, the model is given a task without any examples and is expected to generalize from its pre-training. Few-shot prompting includes a handful of input-output examples within the prompt to demonstrate the task. These techniques have been popularized by models such as GPT-3 [1], which have demonstrated excellent performance in few-shot settings across multiple benchmarks.

A notable advancement in prompting is the Chain-of-Thought (CoT) method, which encourages the model to generate intermediate reasoning steps before producing a final response [14]. CoT prompting improves performance particularly in tasks that require logical reasoning or arithmetic operations. Another emerging method is Tree-of-Thought (ToT) prompting, in which the model explores multiple lines of reasoning in parallel before selecting the most coherent solution [3].

Prompt engineering has evolved into a discipline in its own right. Modern prompting techniques include self-consistency (in which multiple outputs are sampled and aggregated), reflection prompting (in which the model is asked to critique or revise its own response), and expert prompting (in which the model is asked to assume the role of a domain expert to provide more accurate or context-sensitive responses).

In the context of this thesis, prompting plays a central role in translating natural language instructions provided by the user into executable commands for a robotic agent. Carefully designed prompts enable the LLM to interpret tasks such as navigation, object recognition, or action planning and to produce structured actions or sub-goals that can be executed by the underlying robotic system.

2.3 Robotics

Robotics is a multidisciplinary field that encompasses the design, construction, operation, and control of autonomous or semi-autonomous machines capable of interacting with the physical environment. In recent years, the convergence of robotics with artificial intelligence has enabled the development of systems capable of more flexible, context-sensitive, and intelligent behavior. These capabilities are particularly relevant for mobile robots operating in dynamic environments, such as homes, offices, or simulation platforms.

At the core of robotic systems are several key components: perception, localization, mapping, planning, and control. Perception allows robots to interpret sensor data (e.g., camera, LiDAR), while localization and mapping allow them to

determine their position within an environment and construct a spatial representation of it. Planning and control translate high-level goals into executable motor commands.

A key enabler for modern robotics research and development is the Robot Operating System, an open-source middleware framework that facilitates the integration of hardware and software components. The latest version, ROS 2, is designed for real-time performance, distributed systems, and safety features essential for complex and scalable robotic applications [4].

In this thesis, we use ROS 2 to build a modular control system for a mobile robot capable of interpreting and executing natural language instructions. This robot leverages standard robotic software stacks, including Simultaneous Localization and Mapping (SLAM) [15] to autonomously explore and localize itself in unknown environments, and the Navigation Stack to plan and follow trajectories while avoiding obstacles.

In addition, the robot's control logic is supported by behavior trees, a hierarchical framework commonly used in robotics for action selection, which offers modularity and reusability. This structure facilitates the integration of high-level decisions from LLMs with low-level control primitives in ROS.

2.3.1 Mobile Robots

Mobile robots are autonomous or semi-autonomous systems capable of moving within an environment to perform tasks such as navigation, inspection, transport, or interaction with objects and people. Unlike fixed robotic arms, mobile robots must actively perceive and interpret their surroundings in order to make real-time decisions about movement and behavior.

Mobile robots can be classified according to their locomotion system: wheeled, legged, tracked, or aerial. In this thesis, the focus is on wheeled robots, particularly differential drive platforms, which offer simplicity and efficiency for indoor navigation tasks. A representative example is the TurtleBot4, widely used in research due to its integration with ROS 2 and support for various sensors and autonomy modules [7].

Figure 2.2: TurtleBot 4 Lite (left) and TurtleBot 4 Standard (right) [7]

Key capabilities of mobile robots include:

- **Localization:** Estimating the pose (position and orientation) of the robot within a known or unknown environment.
- **Mapping:** Constructing a representation of the surrounding environment, in the form of an occupancy grid, topological graph, or semantic map.
- **Path planning and obstacle avoidance:** Calculating collision-free paths to reach a destination, reacting to dynamic changes in the environment.
- **Sensor fusion:** Combining data from multiple sources (e.g., LiDAR, IMU, RGB-D camera) to improve perception and decision-making.

Mobile robots are often used as test beds for the integration of higher-level AI components, including LLMs. In this thesis, the robot interprets user instructions provided in natural language, reasons about feasible trajectories, and executes commands via ROS 2. This high-level control is decoupled from low-level motion execution, ensuring modularity and enabling seamless integration with LLMs for human-robot interaction.

Simulated environments such as Gazebo, classical and ignition, provides a realistic and flexible framework for testing the behavior of mobile robots under controlled conditions. These tools allow researchers to replicate indoor environments and validate robotic systems before implementing them in the real world.

2.3.2 Robot Operating System

ROS is an open-source framework designed to facilitate the development, integration, and execution of software for robotic systems. Despite its name, ROS is not a traditional operating system, but rather provides a structured communication layer above the host OS that enables distributed processing among multiple software modules (nodes) in a robotic system [4].

ROS is based on a publisher-subscriber architecture, in which nodes communicate through *topics* for streaming data, *services* for synchronous interactions, and *actions* for long-running tasks. This modularity and flexibility have made ROS the standard in both academic and industrial robotics.

The latest generation, ROS 2, introduces significant improvements over ROS 1, including:

- **Real-time capabilities**, which are essential for control loops and sensor processing;
- **DDS-based communication middleware**, which enables scalable and secure data exchange;
- **Improved support for multi-robot systems** and embedded devices;
- **Cross-platform support**, including Linux, Windows, and microcontrollers.

ROS 2 supports a wide range of robotic capabilities, from low-level motor control to high-level autonomy frameworks such as the *Navigation Stack* and *MoveIt* for motion planning. Its integration with tools such as *RViz* (for 3D visualization), Gazebo (for simulation), and TF (for coordinate transformations) provides a rich ecosystem for creating, testing, and deploying robotic applications.

In this thesis, ROS 2 acts as middleware to connect the perception, planning, and control components with the natural language interface powered by a LLM. This integration allows the user to give instructions in natural language, which are interpreted and transformed into ROS-compliant commands that control the simulated robot in Gazebo. Communication between the GPT-based reasoning node and the ROS system is handled via arguments, enabling seamless real-time interaction.

By leveraging ROS 2, the system remains modular, extensible, and suitable for implementation in both simulation and on physical platforms such as *TurtleBot4*, which natively support ROS 2 [4, 7].

2.3.3 Simultaneous Localization and Mapping (SLAM)

SLAM is a fundamental feature for autonomous mobile robots, allowing them to build a map of an unknown environment and simultaneously estimate their position within it. SLAM is particularly important in scenarios where GPS is not available, such as indoor environments.

The problem of SLAM is to fuse data from multiple sensors, such as LiDAR, cameras, inertial measurement units (IMU), and wheel encoders, to incrementally build a coherent representation of the environment (the map) and estimate the robot's trajectory over time (localization). SLAM methods are typically classified into the following categories:

- **Visual SLAM (vSLAM)** – relies on cameras and computer vision techniques (e.g., ORB-SLAM2, RTAB-Map);
- **LiDAR-based SLAM** – uses laser scans for more accurate metric mapping (e.g., Cartographer, GMapping);
- **RGB-D SLAM** – combines visual features with depth data from sensors such as Intel RealSense or OAK-D.

SLAM algorithms are integrated into the ROS 2 ecosystem as modular packages. For example, *SLAM Toolbox* is a widely used ROS 2-compatible package that supports both online mapping and offline optimization of pose graphs [15]. It provides immediate compatibility with simulation environments such as Gazebo and is suitable for 2D indoor mapping tasks, making it ideal for the scope of this thesis.

In the simulated pipeline implemented here, SLAM allows the robot to build an internal map of the environment as it explores it. This map is used in conjunction with the navigation stack to generate global and local plans, ensuring that the robot can reach specified destinations via natural language commands. The robot's ability to reason about positions depends on this spatial understanding, obtained through topological annotations or integration with a semantic layer.

SLAM also facilitates higher-level capabilities such as dynamic goal setting, object anchoring, and updating the environment map based on new observations, all of which are essential when combining robotics with LLMs that reason about space and context.

2.3.4 Navigation Stack

The Nav2 provides a comprehensive framework for enabling autonomous navigation of mobile robots in both simulated and real-world environments. It is responsible for

calculating paths from a robot's current position to its desired position, dynamically avoiding obstacles and accounting for environmental changes [4].

The navigation pipeline typically consists of the following main components:

- **Global Planner** – calculates a high-level path from the current position to the desired position using a static or SLAM-generated map;
- **Local Planner** – continuously generates short-term speed commands to follow the global path while reacting to local obstacles;
- **Costmaps** – represent the environment as a 2D grid with obstacle inflation to facilitate collision avoidance. Separate costmaps are maintained for global and local planning;
- **Fallback behaviors** – define recovery strategies in case of navigation failure, such as spinning in place or reversing.

ROS 2 implements this architecture through Nav2, a modular and extensible navigation framework that replaces the original `move_base` node of ROS 1 [16]. Nav2 supports plugins for different planning and control algorithms (e.g., Dijkstra, A*), and can be easily integrated with SLAM systems (such as SLAM Toolbox), adaptive Monte Carlo localization (AMCL), and behavior tree-based decision making.

In this thesis, the Navigation Stack is used to connect high-level natural language instructions with robot motion. When the LLM interprets a command such as "go to the desk" or "approach the kitchen," it translates it into a semantic or metric goal. This goal is then passed to the Navigation Stack, which autonomously handles both path planning and execution.

Nav2 also supports simulated environments via Ignition Gazebo, enabling realistic testing of trajectories and obstacle handling. This capability is critical for validating LLM-based control in a safe, repeatable setting before moving to real-world hardware.

By leveraging the ROS 2 Navigation Stack, this thesis ensures a robust and responsive level of motion that allows the robot to execute complex instructions in dynamic environments, while maintaining modularity between planning, control, and high-level reasoning.

2.3.5 Behaviour Tree (BT)

BTs are a formalism used to model decision-making processes in autonomous agents, particularly in robotics and game AI [17]. They offer a modular, hierarchical, and reactive structure for organizing complex behaviors into manageable and reusable

components. In contrast to traditional FSMs, BTs provide better scalability and maintainability through clear separation of control flow and action execution [18].

A typical BT consists of:

- **Control Nodes**, such as *Selector*, *Sequence*, and *Parallel*, which determine the flow of execution;
- **Decorator Nodes**, which modify the behavior of a subtree (e.g., repeat until success);
- **Leaf Nodes**, including *Action* and *Condition* nodes, which interface with the robot's actuators or sensory data.

BTs are executed top-down and left-to-right, allowing dynamic switching of actions based on real-time feedback. This makes them particularly effective in robotic applications where the system must respond to changing conditions or failures (e.g., re-planning when an obstacle is detected) [19].

In ROS 2, BTs are commonly used with the Nav2, where they serve as the backbone of the robot's high-level control strategy. *Nav2's BT Navigator* allows developers to define navigation policies using XML-based BT descriptions. This provides flexibility to implement recovery behaviors, conditional movement, or multistep tasks [16].

By combining LLM-based reasoning with BT-based execution, this architecture enables the robot to perform structured tasks with a balance of autonomy and interpretability, while maintaining robustness in uncertain environments.

2.4 Robotics and AI

The integration of artificial intelligence, particularly LLMs, into robotics represents a transformative shift in how autonomous systems perceive, reason, and act in complex environments. While traditional robotic systems rely on rule-based controls, (FSMs), or supervised learning pipelines, AI-based approaches aim to generalize across different tasks using data-driven knowledge and reasoning capabilities [1, 14, 3].

Recent advances in LLMs have enabled intuitive human-robot interaction through natural language interfaces. These models are capable of interpreting instructions, reasoning about action sequences, and generating responses that guide robot behavior without requiring explicit programming for each individual task [1].

In robotic applications, LLMs can be used for:

- **Semantic grounding:** associating words such as "kitchen" or "cup" with physical locations or objects;
- **Goal inference:** interpreting vague or high-level user requests;

- **Task decomposition:** breaking down complex instructions into executable steps;

- **Error recovery reasoning:** suggesting alternative plans in case of failures.

To make this integration operational, LLMs must be combined with real-time robotics frameworks such as ROS 2 [4]. This requires a hybrid architecture, in which the LLM handles high-level planning and reasoning, while ROS nodes execute concrete actions (e.g., navigation, manipulation). Communication typically occurs via structured messages or topic publishing mechanisms, ensuring that natural language decisions can trigger low-level robotic behaviors.

In this thesis, we explore such an architecture: the robot receives textual instructions from a user, interprets them via an LLM (e.g., ChatGPT or similar model), and executes the inferred actions in a ROS 2-based simulated environment. This allows the robot to understand requests such as "check if there is an object on the table in the living room" and to autonomously navigate and verify the presence of the object, combining language understanding with spatial and visual reasoning.

This fusion of robotics and AI not only increases the autonomy and adaptability of robotic agents, but also lays the foundation for more general embodied intelligence systems capable of learning and acting in diverse domains.

2.4.1 Code as Policies Approach

The *Code as Policies* (CaP) approach is a paradigm in which high-level reasoning and task execution in robotics are delegated to a LLM that generates executable code as output. Rather than relying on predefined policies learned from data or manually designed (FSMs), the robot receives natural language instructions, and the LLM translates them into interpretable low-level commands or Python-like scripts that serve as policies governing behavior [2].

Introduced in the context of LLMs such as GPT-3 and Codex, this method leverages the model's ability to:

- Understand task objectives from human language;
- Generate control logic structures (e.g., `if...else`, loops, sequences);
- Compose and adapt code that calls functions from a robotics API;
- Generalize to unseen instructions by reusing programming patterns.

This approach offers several advantages:

- **Interpretability:** each decision step is made explicit through code;
- **Composability:** tasks can be broken down into modular sub-functions;

- **Extensibility:** new skills can be added by expanding the available API.

In the context of this thesis, the CaP methodology supports the integration of natural language navigation commands with ROS 2-based control logic. The LLM acts as a reasoning engine that produces structured instructions or logical blocks, which are either executed directly or translated into ROS messages to control the robot in the simulated environment.

However, some challenges remain. The LLM must be restricted to generating safe and valid code. Moreover, the dynamic nature of real-world environments requires robust runtime monitoring and error handling, which static code generation may not be able to fully anticipate. To mitigate this issue, the generated policies can be executed in a loop with feedback from the environment, in line with interactive agent-based strategies such as *ReAct* or other LLM-powered agent frameworks [20].

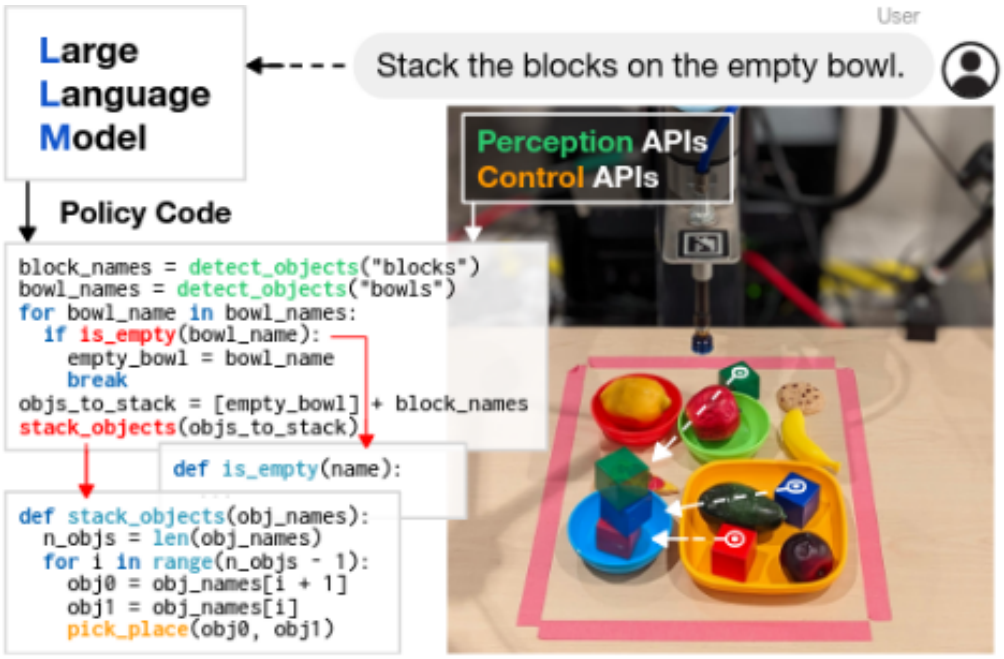


Figure 2.3: Example of the "Code as Policies" paradigm: an LLM receives instructions in natural language and generates Python code that performs low-level robotic checks. The code includes logical structures, feedback loops, and API calls for direct interaction with the environment. [2]

2.4.2 NavGPT-2: Navigational Reasoning with Vision-Language Models

NavGPT-2 represents an advanced approach that blends the capabilities of large-scale language models with visual perception to facilitate navigational reasoning.

Unlike text-only systems, NavGPT-2 leverages multimodal inputs, combining natural language instructions with visual data from cameras or other sensors, to generate context-aware navigation plans [8].

Fundamentally, NavGPT-2 is designed to solve two key challenges:

- **Semantic interpretation of the visual context**: By processing images alongside textual commands, the model builds a contextual understanding of the environment. This vision-language integration enables it to associate semantic labels with corresponding spatial features.

- **Dynamic navigation reasoning**: Thanks to enriched contextual information, the model is able to reason about complex navigation tasks involving object detection, obstacle avoidance, and goal inference. For example, a command such as "find and approach the red door" is converted into a series of navigation sub-goals using information derived from both the visual feed and the linguistic prompt.

The process follows a three-level framework:

1. **Multimodal input fusion**: The system first encodes visual observations (typically using pre-trained vision-language encoders) together with natural language instructions. This fused embedding captures both the spatial and semantic aspects of the scene.

2. **Contextual reasoning**: Similar to CoT prompting approaches, NavGPT-2 uses contextual reasoning by iteratively generating intermediate navigation steps. The model "thinks aloud" by breaking down the problem into manageable segments (e.g., "locate the door", "plan a trajectory to avoid obstacles", "validate the door's location") before issuing the final navigation command.

3. **Action generation and execution**: The final navigation plan is then translated into executable commands. These commands can be incorporated into a ROS 2 framework, where they trigger specific control actions such as movement, turning, or sensor re-evaluation. This hybrid execution benefits from NavGPT-2's multimodal reasoning capabilities, relying on standard robotics middleware to handle low-level control.

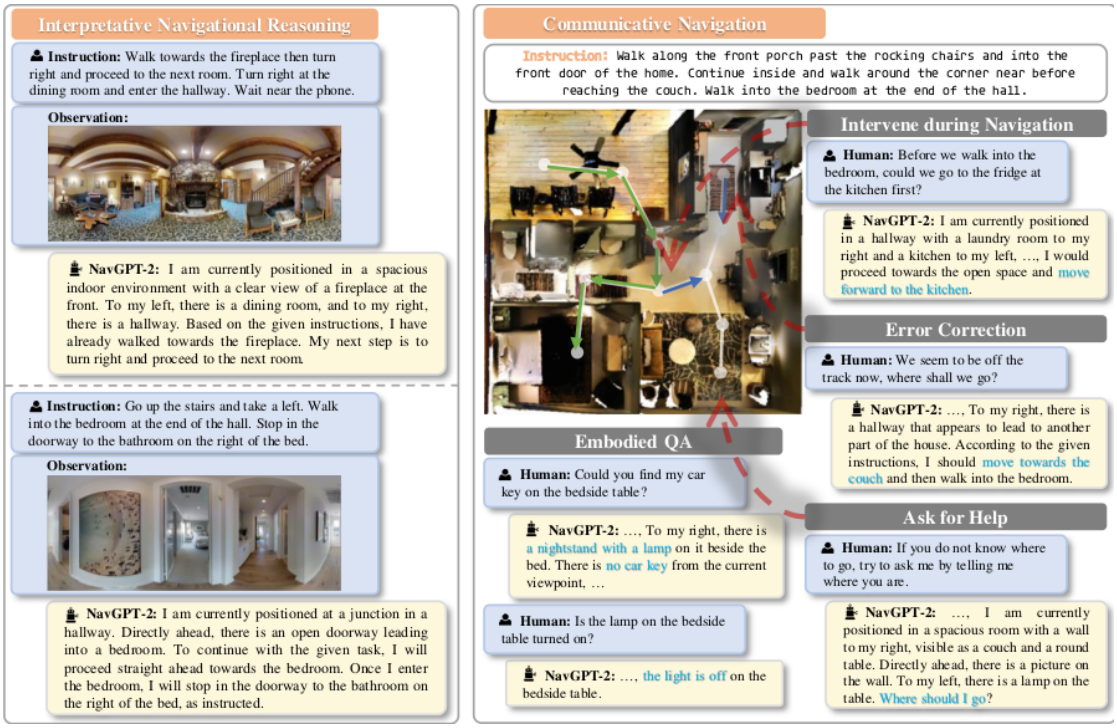


Figure 2.4: Left: Besides performing effective navigation planning, NavGPT-2 is capable of generating navigational reasoning in a human-interpretable way. Right: NavGPT-2 can support multi-round interaction with the user and plan according to the user's intervention in the navigation process, actively ask for help, and answer visual questions. [8]

The NavGPT-2 approach illustrates how a vision-language model (VLM)can act as a dynamic planner in robotic systems. Its ability to use both visual context and natural language instructions sets it apart from previous methods, where navigation reasoning was often segmented between separate perception and planning modules. By unifying these processes, NavGPT-2 is able to generate more robust plans that adapt to dynamic and real-world environments.

This method is particularly promising for applications that require autonomous navigation in unstructured or changing contexts.

2.4.3 Limitations of LLM Models in Robotics

Although LLMs have demonstrated impressive capabilities in natural language understanding, reasoning, and code generation, their application in robotics presents several practical and conceptual limitations. These challenges stem from the gap between abstract linguistic reasoning and the constraints of embodied real-world

systems [3, 14, 1].

- **Lack of grounded perception**: LLMs are primarily trained on textual data and do not possess an intrinsic understanding of visual or spatial environments. When used in isolation, they are unable to perceive the robot's surrounding environment, detect objects, or evaluate physical constraints. This lack of perceptual grounding can lead to overly confident or logically plausible but physically unfeasible decisions [8].
- **Hallucinations and inconsistency**: LLMs are known to hallucinate facts or actions, producing outputs that appear correct but are incorrect or unsafe in real-world contexts [1]. For example, a model might suggest an action that requires passing through a wall or detecting a non-existent object. In robotics, such errors can compromise system safety or cause mission failure.
- **No internal state or memory**: LLMs do not maintain a persistent internal state or memory between invocations, unless implemented externally. In multi-step robotic tasks, this can lead to inconsistent plans or loss of context, unless the system externally maintains memory buffers or world models.
- **Latency and real-time constraints**: Inference with large models (e.g., *GPT-4*) can introduce latency that is incompatible with real-time robotic control. While local implementation of smaller models (e.g., *LLaMA*, *GPT-2*) can reduce latency, this often comes at the cost of reduced reasoning ability and robustness [13, 1].
- **Limited action semantics**: LLMs do not inherently know which commands are executable or safe within a robotic platform. They require optimized APIs (use of tools) or external filtering mechanisms to restrict their outputs to a valid action space [3]. Without such constraints, their decisions could violate the capabilities of robots, causing navigation, manipulation, or planning errors.
- **Generalization and robustness issues**: While LLMs excel at language generalization, their reasoning in new physical situations can be fragile, especially when tasks involve dynamic environments or ambiguous instructions. Prompt design and training data strongly influence performance, which may not translate well to diverse real-world scenarios [8, 14].

2.5 Visual Perception Modules

VLM represent an important step forward in the integration of computer vision and natural language processing, enabling machines to jointly interpret visual inputs and textual instructions. By processing multimodal information—typically images

or videos combined with language, VLMs support a richer and more semantically grounded understanding of the environment, which is particularly valuable in robotics, human-computer interaction, and embodied artificial intelligence.

The development of VLMs addresses key limitations of unimodal systems, such as the inability to reason about complex spatial and contextual relationships that are often necessary for real-world tasks. For example, while a traditional vision model might detect the presence of a "bottle," a VLM is able to answer queries like "is there a red bottle on the shelf?" by linking visual recognition to conceptual understanding.

The basic architecture of a VLM typically includes a visual encoder that extracts image features, a language model (often pre-trained) to process textual input, and a fusion or alignment mechanism that enables cross-modal reasoning. Recent models such as *CLIP* [21], *BLIP* [5], *Flamingo* [22], *LLaVA* [23], and *GPT-4V* differ in how they align and integrate vision and language. Some leverage contrastive learning to align paired image-text embeddings in a shared space, while others use encoder-decoder transformer architectures to support generative tasks such as image captioning, VQA, or dialogue-based interaction.

As discussed by Li et al. [24], the shift toward using large pretrained language models as the backbone for VLMs has significantly improved model expressiveness and reduced training costs, while also introducing challenges such as hallucination, alignment errors, and safety concerns.

Despite these issues, VLMs currently achieve state-of-the-art performance across a variety of benchmarks, including zero-shot classification, VQA, and multimodal reasoning. Applications are rapidly expanding in fields such as autonomous driving, assistive robotics, and simulation-based planning.

In the context of robotics, VLMs enable systems to follow natural language commands, perceive complex scenes, and verify task completion via language-based interpretation of visual input. The following sections explore two representative components of modern VLM-based robotic perception: object detection and 3D localization using YOLOv8 with depth data, and contextual understanding through the generative VQA capabilities of *BLIP*.

2.5.1 BLIP for Visual Questions Answering (VQA)

VLM are taking on an increasingly central role in robotics, where intelligent agents must interpret and respond to natural language questions based on visual inputs. Among these, BLIP has emerged as a powerful multimodal framework that unifies vision and language understanding within a single architecture [5]. Unlike previous models that relied solely on contrastive targets (e.g., CLIP [21], BLIP incorporates both alignment and generation capabilities, making it particularly well-suited for generative tasks such as image captioning and VQA. The publicly available and

widely adopted model variant `Salesforce/blip-vqa-base` has been specifically optimized for VQA tasks.

At its core, BLIP consists of a visual encoder, typically based on Vision Transformers (ViT), and a text decoder capable of producing linguistic outputs conditioned on visual and textual inputs. During pre-training, the model is exposed to large-scale image-text pairs using a combination of training objectives, including image-based generation, language modeling, and contrastive matching. This multi-objective approach endows BLIP with rich multimodal understanding, enabling it to reason about image content in a natural and contextualized manner [5].

In the context of robotics, BLIP's VQA capabilities are particularly valuable. A robot equipped with a camera can capture its surroundings and query BLIP with natural language questions. The model interprets both the image and the question, generating responses that reflect its semantic understanding of the scene. This mode of interaction greatly improves human-robot communication, allowing users to interact with robotic systems in a more intuitive and language-based manner.

In addition, BLIP supports follow-up questions and conversational concatenations, enabling more complex interaction scenarios. For example, after executing a navigation command, the robot can capture an image and verify whether a task has been successfully completed by asking a BLIP-based module. The ability to integrate generative perception with interactive dialogue also supports explainability and transparency in robot decision-making, two increasingly important properties in real-world deployment scenarios.

From a systems perspective, BLIP is lightweight enough to be deployed on GPU-enabled edge devices, while remaining compatible with server-based inference in distributed robot architectures. Its pre-training on large-scale generic datasets ensures broad generalization across domains, although domain-specific fine-tuning remains a viable option for improving performance in specialized environments.

To summary, BLIP bridges the semantic gap between vision and language in robotic systems by enabling natural responses to questions about visual inputs. Its integration into perception pipelines provides robots with context-aware reasoning and flexible interaction, thereby supporting the development of intelligent, language-aware agents capable of autonomously operating in dynamic, real-world settings.

2.5.2 YOLOv8 with Depth for Object Detection and 3D Localization

The task of object detection has seen significant progress over the last decade, with the YOLO (You Only Look Once) family of models establishing itself as the dominant real-time solution in both academic and industrial robotics. YOLOv8, developed by Ultralytics, represents the latest iteration of this line and introduces several architectural innovations that improve detection accuracy while maintaining

high inference speed [6]. Unlike previous region-based models, YOLOv8 employs a fully convolutional, anchor-free architecture that predicts bounding boxes and object classes in a single forward pass, offering real-time capabilities even on resource-constrained platforms. This makes it particularly well suited for robotic applications that require low-latency decisions, such as autonomous navigation, object manipulation, or obstacle avoidance.

However, YOLOv8, like all 2D object detectors, operates at the image level and has no direct spatial perception of object locations in the physical world. To enable three-dimensional understanding, a depth perception module is often integrated into the pipeline. This module typically comes from RGB-D cameras, stereo vision systems, or LiDAR. Once a 2D object is detected, the corresponding depth value can be extracted from a depth map recorded at the center or across the bounding box region. These depth values are then combined with intrinsic camera parameters, such as the horizontal field of view and image size, to calculate the angular offset and estimate the relative position of the object in 3D space.

This approach is further improved by leveraging the robot's position within a known reference frame, often obtained via SLAM or a TF. The 3D position of the object relative to the robot is then transformed into global coordinates, enabling long-term map-based planning and interaction. This capability is essential in tasks such as semantic mapping, where objects must not only be recognized but also spatially anchored within the environment, or in retrieval and transport operations, where the robot must return to a specific location to interact with a previously seen object.

Another advantage of integrating YOLOv8 with depth sensing is the ability to filter detections based on distance thresholds, allowing nearby objects to be prioritized for interaction. For example, a robot tasked with locating a specific object (e.g., "find the nearest bottle") can ignore distant detections and focus computational resources on reachable targets. Furthermore, this fusion of 2D semantics and 3D geometry provides robustness in cluttered or dynamic environments, as depth data can help resolve ambiguities in cases of occlusions and overlapping objects.

In summary, the synergy between YOLOv8 and depth sensing systems enables robots to achieve perceptual grounding in 3D environments. By extending 2D detection to 3D localization, this hybrid approach is a crucial component of modern vision pipelines for mobile and embodied agents, balancing efficiency, scalability, and interpretability.

Chapter 3

Methodology

3.1 Experimental Setup

The experimental setup implemented in this thesis aims to evaluate the feasibility of integrating a LLM-based agent into a ROS 2 robotic system for executing natural language instructions in a simulated indoor environment. The core of the system architecture consists of three main components: a mobile robot simulated in Ignition Gazebo, a natural language agent powered by GPT, and a vision module based on the BLIP [5] and YOLOv8 [6] models used for post-task visual verification.

The main objective of this setup is to evaluate how effectively a GPT-based reasoning node can interpret instructions, generate actionable goals, and coordinate navigation tasks using the ROS 2 infrastructure. The experimental pipeline is designed to mimic a realistic interaction cycle in which a human user issues a high-level command in natural language and the system autonomously translates it into robotic actions.

The robot performs navigation based solely on the goals inferred by GPT and the predefined ROS 2 functionality. Once a task is considered complete, typically upon reaching a target position, the robot captures an image from its RGB-D camera. This image is processed by the vision pipeline using BLIP and YOLOv8 models to assess the presence or absence of target objects or contextual features.

This design allows for the isolation and analysis of the LLM agent's reasoning ability, independent of visual perception. By validating the visual results, the system enables a structured evaluation of instruction compliance, navigation accuracy, and vision-language reasoning integration.

Figure 3.1 shows the overall system architecture, highlighting the interaction between the GPT-based agent, fallback logic, navigation stack, and the simulated ROS 2 environment with TurtleBot execution.

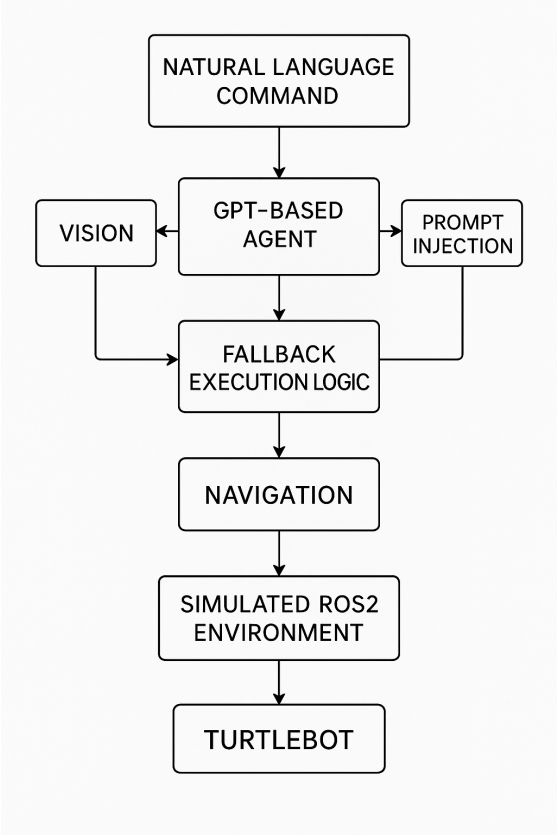


Figure 3.1: Block diagram of the proposed modular architecture. The system is composed of a GPT-based reasoning node, a ROS 2 navigation stack, and an external visual verification pipeline. Each module is connected via structured ROS interfaces, enabling a flexible and extensible instruction-to-action loop.)

The following sections provide further details on the experiment objectives, software stack, environment setup, and integration between natural language input and robotic behavior.

3.1.1 Experimental Objectives

The objective of this experimental study is to evaluate the performance and modular integration of a multi-agent architecture for the execution of robotic tasks, in which natural language instructions are interpreted and implemented through collaboration between language, perception, and control modules.

Unlike conventional configurations that focus exclusively on the integration of a LLM with ROS 2 [4], this experiment aims to test and validate the coordination of multiple reasoning and perception components, including a GPT-based agent,

a vision agent—BLIP [5] for VQA and YOLOv8 [6] for object detection and spatial estimation, and a fallback execution logic for safe and interpretable task management.

The specific objectives of the experiment are as follows:

- **Evaluate natural language understanding and action planning:** Test the ability of the GPT-based agent to interpret high-level instructions and generate structured and semantically valid sequences of robotic actions.
- **Validate ROS 2 integration and modular communication:** Ensure robust communication between all nodes via ROS 2 topics and services, including the GPT node,Nav2, vision pipeline, and fallback modules.
- **Test autonomous navigation and goal execution:** Evaluate the robot's ability to reach inferred destinations using the ROS 2 navigation stack based on commands generated by the LLM agent.
- **Evaluate vision agent performance:** Automatically analyze RGB and depth data captured using BLIP and YOLOv8 to determine whether visual goals have been achieved (e.g., object detection, proximity estimation).
- **Evaluate fallback execution logic:** Measure the effectiveness of the system in identifying and handling undefined or invalid actions generated by GPT using AST analysis, pattern matching, and injection of predefined behaviors.
- **Modularity and extensibility benchmark:** Examine how individual components (language agent, vision module, action controller) can be modified or extended without disrupting the overall architecture.

This multifaceted evaluation provides insights into the feasibility of creating autonomous, interpretable, and modular robotic systems capable of operating from high-level human instructions and paves the way for the inclusion of real-time feedback loops and learning-based adaptation in future iterations.

3.2 Simulation Environment

To evaluate the proposed architecture in a controlled and reproducible environment, the entire system was implemented in a simulated environment using Ignition Gazebo and ROS 2. This simulation framework allows for realistic modeling of robot dynamics, sensor behavior, and interactions with the environment, making it ideal for testing LLM-guided robotic behaviors without the risks and variability of a physical setup.

The simulated environments reproduce indoor scenes, furnished with basic objects and landmarks that serve as navigation references. The robot used in the simulation is modeled on a Turtlebot4 [7], equipped with differential traction, an RGB-D camera, and ROS 2-compatible navigation modules (Nav2) [16].

The simulation environment was selected to meet the following criteria:

- **Modularity:** Components such as the map, robot configuration, and navigation stack can be modified independently.
- **Realism:** The environment includes walls, furniture, and obstacles, enabling meaningful path planning and localization.
- **Sensor simulation:** Camera streams and odometry data are simulated in real time, providing input for post-task visual processing.
- **Reproducibility:** All experiments can be repeated under identical conditions, enabling consistent benchmarking.

A key strength of using Ignition Gazebo lies in its flexibility and extensibility. Users can create fully customized simulation worlds tailored to specific experimental needs, leveraging a wide array of publicly available models and plugins, such as those provided by Open Robotics and Gazebo Fuel [7]. These models include furniture, vehicles, people, and sensor-equipped robots, which can be easily arranged to design rich and interactive environments.

This capability allows researchers to simulate diverse and complex scenarios—from warehouse navigation to household tasks—without the need for physical hardware. Custom environments can be crafted to include specific obstacles or object placements that challenge and validate the robotic system's capabilities. This adaptability significantly accelerates prototyping, testing, and validation of advanced human-robot interaction pipelines under controlled conditions.

3.2.1 Detailed Software Components

The experimental architecture consists of modular software components that interact via ROS 2 middleware. Each module is responsible for a specific function within the system, and together they form an integrated pipeline capable of interpreting human instructions and executing them in simulation.

1. **ROS 2 (Humble Hawksbill)**
   ROS 2 serves as the central communication backbone. It enables asynchronous and real-time communication between the LLM-based agent, navigation stack, camera system, and vision modules via topics, actions, and services. ROS 2 ensures interoperability, modularity, and scalability in both simulation and real-world deployments [4].

2. **Ignition Gazebo**
   The simulation environment is built using Ignition Gazebo, which provides realistic 3D physics, environment rendering, and accurate sensor modeling. The robot operates within a structured indoor scenario containing rooms, furniture, and obstacles.

3. **Navigation Stack (Nav2)**
   Nav2 enables the robot to autonomously navigate towards user-deduced goals [16]. It includes:

   - Global and local planners,
   - Dynamic obstacle avoidance using costmaps,
   - Recovery behaviors in case of navigation failure,
   - Action servers that receive targets from the GPT node or fallback logic and execute them.

   Nav2 continuously updates the robot's path as new data becomes available.

4. **Natural Language Processing Node (GPT Agent)**
   The GPT-based agent is implemented as a ROS 2 node that sends prompts to the OpenAI API (e.g., GPT-3.5 [1]). It interprets natural language instructions and generates high-level plans or API calls. These outputs are validated before being passed to control modules.

5. **Fallback Execution Logic**
   This Python-based module intercepts the GPT output and performs structural validation (e.g., Abstract Syntax Tree parsing and pattern matching). If the output is incomplete, invalid, or non-executable, it either rejects the command or injects predefined safe behaviors. This mechanism ensures robustness and prevents execution of malformed instructions.

6. **Camera Sensor Node**
   The robot includes a simulated RGB-D camera setup: the RGB camera publishes to `/oakd/rgb/preview/image_raw`, and the depth camera publishes to `/oakd/rgb/preview/depth/depth_raw`. Upon task completion, an image is captured and stored for use in the perception pipeline.

7. **Vision Agent (BLIP and YOLO)**
   The vision module is fully automated and includes:

   - BLIP [5], which handles visual question answering (VQA) and can respond to queries like "Is there an object in front of the robot?"

- YOLOv8 [6], combined with depth data for object detection and proximity estimation.

These models are integrated via ROS 2 or Python scripts and are part of the post-navigation pipeline, as described in Section 3.2.4.

8. **RViz2**
   RViz2 is used for visualizing the robot's position, costmaps, goals, and camera feed in real-time. It plays a vital role in monitoring, debugging, and validating navigation and perception processes during simulation.

This software architecture supports a hybrid **perception-control-reasoning** loop, with clear separation between high-level decision-making (LLM), execution (ROS/Nav2), and perception (BLIP/YOLOv8). Its modular design makes it ideal for future enhancements, such as integrating real-time visual feedback or dialogue-based interaction capabilities.

3.2.2 Environment Configuration and Simulation Parameters

The simulation environment has been designed to emulate an indoor scenario in which a mobile robot is able to interpret and execute natural language instructions through a modular architecture that integrates reasoning, navigation, and perception components. This environment supports the full end-to-end flow of instruction-based task execution, from natural language understanding to post-task visual validation, enabling structured and repeatable evaluation.

Environment Layout

The simulated world comprises several distinct scenarios, each crafted to evaluate specific aspects of navigation and perception:

- **Warehouse:** contains shelves and various recognizable objects such as cardboard boxes, chairs, desks, and human figures.
- **Open Room:** populated with randomly placed objects, including a suitcase, trash bin, SUV, and other miscellaneous items.
- **Maze-like Environment:** devoid of objects, designed exclusively for testing path planning in narrow corridors and complex topologies.

Walls and static obstacles are included to replicate real-world occlusions and constraints.

RViz2 Configuration and Simulation Startup Modes

The system can be launched in two distinct configurations, which influence localization behavior and the necessary RViz2 settings:

- **SLAMMode:** uses SLAM Toolbox [15] to build the environment map in real-time. No manual 2D pose estimate is needed, as initialization occurs automatically.
- **Localization Mode (AMCL):** requires a pre-generated map and manual initialization of the robot's position via the "2D Pose Estimate" tool in RViz2. This is critical for correct localization.

To start the simulation in localization mode using a predefined world and map, use:

```
ros2 launch turtlebot4_ignition_bringup turtlebot4_ignition.launch
.py
     nav2:=true slam:=false localization:=true rviz:=true
     world:=depot map:=/opt/ros/humble/share/turtlebot4_navigation
     /maps/depot.yaml
```

If the `world` and `map` arguments are not specified, the simulation switches to a default scenario.

RViz2 Configuration Guidelines

The following RViz2 settings are essential for consistent data capture and visualization across both simulation modes:

- **RGB Camera Topic:** the default topic should be switched to `/oakd/rgb/preview/image_raw`
- **Depth Camera Topic:** the default topic should be switched to `/oakd/rgb/preview/depth/depth_raw`
- **Depth Normalization:** disable "Normalize Range" and set the range to [0.3, 5.0] meters

These parameters ensure that RViz2 accurately reflects the images used for visual processing and that the saved `depth.npy` file contains a valid depth map. This configuration is essential for the correct operation of both vision modules: YOLOv8 relies on accurate depth data to estimate the distance to the detected target object,

while BLIP requires a properly captured image (e.g., `captured_image.png`) for reliable semantic grounding and visual question answering [5, 6].

Table 3.1: RViz2 and Simulation Parameters

Setting	**Value / Action**	**Required In**	**Note**
Launch Mode	SLAM or Localization	Both	SLAM auto-initializes pose; Localization requires manual 2D pose estimate.
2D Pose Estimate	Manual input in RViz2	Localization only	Use the "2D Pose Estimate" tool at simulation start to align with the map.
RGB Camera Topic	`/oakd/rgb/preview/image_raw`	Both	Ensures correct image display and consistent image logging.
Depth Camera Topic	`/oakd/rgb/preview/depth/depth_raw`	Both	Required for depth snapshot and saving the `depth.npy` file.
Normalize Range	Disabled	Both	Prevents black images in RViz2 due to improper scaling.
Depth Range	0.3–5.0 meters	Both	Ensures proper visualization and valid post-processing by the vision agent.

Instruction Execution Pipeline

Once a natural language command is received by the GPT node, the system proceeds through the following pipeline:

1. The LLM interprets the instruction and proposes an action plan.
2. The output is validated and parsed through the fallback logic module.
3. A navigation goal is sent to the Nav2 stack for execution.
4. Upon goal completion, RGB and depth snapshots are captured automatically.

5. The vision agent activates:
    - BLIP is used for VQA,
    - YOLOv8 with depth estimates object class and position.
6. The outputs are stored and used for downstream task validation.

This automated pipeline enables a full closed-loop system integrating high-level reasoning, navigation, and post-task perception without requiring manual intervention.

3.2.3 Natural Language to Robotic Action Translation

The process of transforming user-provided natural language commands into executable robotic actions is handled by a dedicated pipeline that combines a GPT-based reasoning agent, fallback logic for execution safety, and dynamic integration with the ROS 2 navigation stack.

Command Generation via Chat Completion

The GPT-based agent interacts with the user through natural language instructions and internally translates them into executable Python code using OpenAI's `chat.completions` endpoint [1]. This mechanism enables the system to support flexible and contextually rich interactions, where the language model not only interprets the intent but also generates ROS 2-compatible instructions that guide the robot's behavior.

The generated commands typically follow a structured template, designed to be interpretable by the ROS 2 navigation stack. For instance, when the user issues a command such as "Go to the forklift," the LLM generates the following code snippet:

Listing 3.1: Example of GPT-generated navigation command.

```
goal = navigator.getPoseStamped([2.0, 1.0], 90)
navigator.startToPose(goal)
```

Here, the GPT agent deduces the target location's coordinates from its prompt context—often retrieved from a structured JSON file (see Section 3.2.4)—and generates a goal pose using predefined API functions. The reasoning is not hard-coded but inferred dynamically, based on injected prompt examples and available symbolic mappings of destinations and actions.

This code is sent to the fallback execution logic, which ensures syntactic validity and runtime safety. If deemed valid, it is executed to trigger robot navigation via the `NavigateToPose` action interface provided by Nav2 [16].

As shown in Figure 3.2, the system uses a fallback logic layer to intercept and sanitize GPT-generated code. This mechanism ensures runtime safety by detecting undefined function calls through AST parsing and preventing their execution.

```
184    def safe_exec_with_fallback(self, code_str, context):
185        try:
186            tree = ast.parse(code_str)
187            called_funcs = {node.func.id for node in ast.walk(tree) if isinstance(node, ast.Call) and isinstance(node.func, ast.Name)}
188            available_funcs = set(context.keys()) | set(dir(__builtins__))
189            undefined_funcs = called_funcs - available_funcs
190
191            if undefined_funcs:
192                self.warn(f"⚠️ Funzioni non definite trovate: {undefined_funcs}")
193                for func in undefined_funcs:
194                    pattern = rf"{func}\\(.*?\\)"
195                    code_str = re.sub(pattern, f"# TODO: define function `{func}`", code_str)
196
197            exec(code_str, context)
198        except Exception as e:
199            self.error(f'[SAFE EXEC] Errore durante esecuzione codice GPT: {e}')
200
201    def publish_status(self, status):
202        msg = Bool()
203        msg.data = status
204        self.ready_for_input = status
205        self.pub_ready.publish(msg)
```

Figure 3.2: Fallback execution logic implemented in the safe_exec_with_fallback() function. This component analyzes the GPT-generated code using Abstract Syntax Tree (AST) parsing to detect undefined functions. If any are found, they are commented out before execution to ensure safe and robust behavior.

This form of language-to-action translation, powered by code generation rather than direct action classification, provides a high degree of expressiveness and adaptability. It also facilitates the integration of custom routines, such as object delivery or patrolling, which can be encoded as callable Python functions and injected into the prompt at runtime.

Object Extraction and Visual Query Generation

When a user provides a natural language instruction that includes a reference to an object (e.g., "Go to the kitchen and check if there is a red cup on the table"), the system must extract the relevant visual entities to perform subsequent validation using BLIP or YOLOv8.

To this end, the GPT-based reasoning node performs object analysis, identifying noun phrases or entities mentioned in the instruction that refer to visible objects. These are extracted via language model inference using prompt templates designed to ask the model "which objects should be visually verified".

Once extracted, the object label is used to trigger one of two vision pipelines:

- For BLIP-VQA [5], a natural language question is dynamically formulated from the instruction (e.g., "Is there a red cup in front of the robot?") and passed to the BLIP module.
- For YOLOv8 [6] with Depth modules, a target class label is derived (e.g.,

"cup") and used by the YOLO detection script to locate the object, compute depth, and estimate global position using TF transformations.

This transformation from language to structured visual queries bridges high-level reasoning and low-level perception, allowing the system to autonomously determine what to look for based on task semantics.

As shown in Figure 3.3, the system uses a set of regular expressions to parse the object name from the instruction. This enables downstream components to formulate either visual questions for BLIP or detection labels for YOLO+Depth.

```
170     def extract_target_object(self, instruction):
171         patterns = [r"check (?:if )?(?:there (?:is|are) )?(?:a |an |the )?([\w\s]+)",
172         r"look for (?:a |an |the )?([\w\s]+)",
173         r"see (?:if )?(?:there (?:is|are) )?(?:a |an |the )?([\w\s]+)",
174         r"is (?:there )?(?:a |an |the )?([\w\s]+)",
175         r"(?:there is|there are) (?:a |an |the )?([\w\s]+)",
176         r"find (?:a |an |the )?([\w\s]+)"]
177         for pattern in patterns:
178             match = re.search(pattern, instruction.lower())
179             if match:
180                 return match.group(1).strip()
181         return None
```

Figure 3.3: Code excerpt from the `extract_target_object()` function in natural_language_nav.py. This method applies regular expression matching to isolate the visual target entity mentioned in the user instruction, enabling dynamic query generation for the visual pipeline.

Dynamic API and Destination Management

The system leverages two dynamically updated configuration files to support runtime extensibility: `dynamic_destinations.json` for symbolic locations and `dynamic_turtlebot4_api.json` for action templates. Their full implementation and usage are detailed in Section 3.2.4.

Final Flow Overview

The complete pipeline for instruction-to-action translation follows this sequence:

1. User input is interpreted by GPT using injected command templates.
2. GPT output is validated and executed through fallback logic.
3. Validated code sends a navigation goal to Nav2.
4. Upon goal completion, additional routines (e.g., visual checking) are triggered as needed.

This architecture supports flexible, extensible, and partially autonomous language understanding, while remaining robust to GPT hallucinations and malformed logic.

Node Initialization

To launch the GPT-based reasoning node within the ROS 2 framework, the following commands must be executed in a terminal:

```
source /opt/ros/humble/setup.bash
source ~/turtlebot4_ws/install/setup.bash
cd ~/turtlebot4_ws
ros2 launch
turtlebot4_openai_tutorials natural_language_nav_launch.py\
  openai_api_key:=‘your_api_key’ parking_brake:=true
```

This command initializes the `natural_language_nav.py` node with access to the OpenAI API and activates the fallback mechanism for secure command execution. The `parking_brake:=true` parameter ensures that the robot remains stationary until navigation begins. Once launched, the node listens for user input and activates the complete instruction-to-action pipeline.

3.2.4 Navigation and Perception Functionalities

The robotic system executes high-level instructions through a two-step process: navigation followed by perception. These components operate in a modular sequence and are integrated via ROS 2 communication interfaces, enabling real-time feedback, post-task evaluation, and autonomous status updates.

Navigation Functionality

Navigation is managed by the Nav2 stack, which autonomously moves the robot to the destination inferred from the natural language instruction. The stack includes global and local path planners, dynamic obstacle avoidance using costmaps, and recovery behaviors for blocked or invalid trajectories [16]. Navigation goals are handled through the `NavigateToPose` action server, and success is determined by positional ($\leq 0.1\,\mathrm{m}$) and angular ($\leq 5°$) thresholds. Nav2 supports both SLAM and AMCL localization modes, adapting to the availability of an environmental map and the initial robot pose.

Perception and Visual Verification

After reaching the target, the robot enters the perception phase. An RGB image and a depth map are captured using the simulated RGB-D camera. For accurate depth interpretation, RViz2 is configured to disable normalization and to constrain the range to 0.3–5.0 meters, ensuring valid data is written into the `depth.npy` file.

As shown in Figure 3.4, once navigation is complete, the system triggers a visual validation routine that runs both BLIP-VQA and YOLO+Depth modules. If a match is confirmed, the detected object is stored and linked to a symbolic name for future reuse.

```python
def run_vision_pipeline(self):
    if not self.target_object:
        self.warn("[Vision] Nessun oggetto target definito. Skipping.")
        return

    question = f"Is there a {self.target_object} in front of the robot?"

    try:
        self.info("[Vision] Running BLIP-VQA...")
        script_path = os.path.join(os.path.dirname(__file__), 'vision_agent_vqa.py')
        result_vqa = subprocess.check_output(['python3', script_path, question], text=True)
        self.info(result_vqa.strip())

        self.info("[Vision] Checking YOLO depth proximity...")
        yolo_script = os.path.join(os.path.dirname(__file__), 'check_depth_proximity_yolo.py')
        result_depth = subprocess.check_output(['python3', yolo_script, self.target_object], text=True)
        self.info(result_depth.strip())

        match_x = re.search(r'bbox_center_x\s*=\s*(\d+)', result_depth)
        match_width = re.search(r'image_width\s*=\s*(\d+)', result_depth)
        match_depth = re.search(r'mean_depth\s*=\s*([0-9.]+)', result_depth)
        depth_ok = False
        if match_depth:
            depth_val = float(match_depth.group(1))
            depth_ok = depth_val <= 1.5

        if 'yes' in result_vqa.lower() and depth_ok:
            if match_x and match_width and match_depth:
                bbox_center_x = int(match_x.group(1))
                image_width = int(match_width.group(1))
                mean_depth = float(match_depth.group(1))

                tf = self.tf_buffer.lookup_transform('map', 'base_link', rclpy.time.Time(), timeout=rclpy.duration.Duration(seconds=1.0))
                robot_x = tf.transform.translation.x
                robot_y = tf.transform.translation.y
                yaw = self.quaternion_to_yaw(tf.transform.rotation)

                obj_x, obj_y = self.estimate_object_position(robot_x, robot_y, yaw, bbox_center_x, image_width, mean_depth)
                yaw_deg = math.degrees(yaw)
                save_destination(self.target_object, [obj_x, obj_y, yaw_deg])

                save_api_command(f"go_to_{self.target_object}", f"dest = destinations['{self.target_object}']\nnavigator.info('Moving to {self.target_object}')\ngoal = navigator.getPoseStamped(dest[0:2], dest[2])\nnavigator.startToPose(goal)")
                self.info(f"✅ Oggetto '{self.target_object}' rilevato e registrato come destinazione.")
            else:
                self.warn("[Vision] Informazioni incomplete da YOLO per stimare la posizione dell'oggetto.")
        else:
            self.info(f"❌ Oggetto '{self.target_object}' non rilevato o troppo lontano.")

    except Exception as e:
        self.error(f'Errore nel vision pipeline: {e}')
```

Figure 3.4: Automated execution of the vision module via the run_vision_pipeline() function. Both BLIP-VQA and YOLO+Depth modules are executed in sequence to verify the task outcome and register the detected object into the system's knowledge base. The code is available at [25].

The vision pipeline processes the captured image by executing both the BLIP-VQA module (for language-based scene understanding) and the YOLO+Depth module (for object detection and 3D localization).

- The BLIP-VQA module (`vision_agent_vqa.py`) answers visual questions in natural language based on the captured image, producing binary or descriptive responses such as "yes", "no", or short captions [5].

- The YOLOv8 with Depth module (`check_depth_proximity_yolo.py`) detects specified objects and estimates their global position by combining bounding box centers, depth values, and TF-based transformations [6].

Dynamic Updating of System Knowledge

If the pipeline—from instruction parsing to visual validation—confirms that the task was successfully executed, the system updates two core files:

- `dynamic_destinations.json`: this file is modified to record the global coordinates of the newly identified object, allowing future references to it by name (e.g., "Go to the detected trash can").
- `dynamic_turtlebot4_api.json`: this file is updated with a callable function associated with the new location, enabling the GPT-based agent to include it in future generated responses.

These updates are triggered only after both the BLIP and YOLO+Depth modules validate the visual target. In particular, the object is considered successfully detected and close enough if:

- The BLIP module returns a positive answer (e.g., "yes") to the visual question,
- The YOLO+Depth module detects the object and estimates its distance to be less than 1.5 meters.

As shown in Figure 3.5, when these conditions are satisfied, the system computes the global coordinates of the object using the TF tree and saves them into the destination map. In parallel, a new callable command is registered by updating the dynamic API JSON file with a corresponding Python instruction block, ready to be reused by the GPT agent in future instructions.

```
126             if 'yes' in result_vqa.lower() and depth_ok:
127                 if match_x and match_width and match_depth:
128                     bbox_center_x = int(match_x.group(1))
129                     image_width = int(match_width.group(1))
130                     mean_depth = float(match_depth.group(1))
131 
132                     tf = self.tf_buffer.lookup_transform('map', 'base_link', rclpy.time.Time(), timeout=rclpy.duration.Duration(seconds=1.0))
133                     robot_x = tf.transform.translation.x
134                     robot_y = tf.transform.translation.y
135                     yaw = self.quaternion_to_yaw(tf.transform.rotation)
136 
137                     obj_x, obj_y = self.estimate_object_position(robot_x, robot_y, yaw, bbox_center_x, image_width, mean_depth)
138                     yaw_deg = math.degrees(yaw)
139                     save_destination(self.target_object, [obj_x, obj_y, yaw_deg])
140 
141                     save_api_command(f"go_to_{self.target_object}", f"dest = destinations['{self.target_object}']\nnavigator.info('Moving to {self.target_object}')\ngoal =
    navigator.getPoseStamped(dest[0:2], dest[2])\nnavigator.startToPose(goal)")
142                     self.info(f"✅ Oggetto '{self.target_object}' rilevato e registrato come destinazione.")
143                 else:
144                     self.warn("[Vision] Informazioni incomplete da YOLO per stimare la posizione dell'oggetto.")
145             else:
146                 self.info(f"❌ Oggetto '{self.target_object}' non rilevato o troppo lontano.")
```

Figure 3.5: Logic for runtime updates to destination and API command files. After object validation (distance $\leq$ 1.5 m), the system saves its global coordinates into `dynamic_destinations.json` and injects a callable command into `dynamic_turtlebot4_api.json`. This enables the GPT node to reference the object symbolically in future tasks.

This mechanism enables the robot to incrementally enrich its internal world representation using validated sensory information, linking high-level reasoning

with perceptual feedback in a semi-autonomous loop. Although this information is not immediately fed back into the GPT node during execution, it is incorporated into later prompts via dynamic prompt injection, progressively enhancing the agent's contextual awareness and planning ability.

3.3 Summary

This chapter presented the complete methodology for developing a modular, LLM-guided robotic system capable of interpreting and executing natural language instructions in simulation. The architecture integrates language understanding, motion planning via ROS 2, and post-task perception through BLIP and YOLOv8. The next chapter evaluates the system's behavior under various experimental conditions.

Chapter 4

Experimental Results

4.1 Natural Language Understanding and Action Planning

The integration of a LLM into a robotic system introduces the challenge of ensuring accurate and unambiguous interpretation of user instructions [1]. In this context, we assessed the agent's ability to understand diverse natural language commands and generate the appropriate structured code required to trigger low-level robotic actions.

A total of 14 distinct language commands were tested, encompassing both direct and semantically implicit instructions. Among these, 6 commands were explicitly mapped to functions defined in the system's prompt injector and were all successfully executed by the robot, demonstrating a 100% success rate for predefined instructions.

The agent was also tested with 3 generalized commands, where semantic reasoning was required (e.g., *"go to where there is water"*). While the model correctly inferred the linguistic intent, it failed to execute the associated actions due to missing symbolic mappings, resulting in safe execution errors.

A critical observation was the dependency on prompt formatting. Minor deviations, such as added whitespace, occasionally caused the LLM-based agent to generate natural language responses or hallucinated functions. These anomalies stress the importance of maintaining a strict, deterministic prompt structure [2].

Overall, the language understanding module achieved a success rate of 64% (9 out of 14), with 100% accuracy on instructions anchored to predefined functions.

4.2 ROS 2 Integration and Modular Communication

The proposed architecture uses ROS 2 [4] as the communication backbone between the language understanding, planning, and execution components. Integration is achieved through structured topic publishing, service calls, and action interfaces.

User instructions are published to a dedicated topic (`/user_input`) and parsed by the LLM-based reasoning node, which generates an executable plan and relays the action to the appropriate module, such as the navigation stack or perception pipeline.

The modularity of the system allows components to remain independent while communicating via ROS 2 messaging. The LLM node interacts only with the symbolic destination registry and the prompt injector, while execution is handled by modules like `bt_navigator` and `controller_server`. Visualization is supported by RViz2, and the architecture supports easy replacement or extension of components.

The communication flow was validated through multiple instruction cycles, with all modules responding correctly and confirming end-to-end execution, demonstrating the viability of integrating LLM-based reasoning into a ROS 2 control loop.

4.3 Autonomous Navigation and Goal Execution

To assess the navigation capabilities, a series of trials were conducted where the robot was tasked with reaching predefined and inferred destinations. Navigation goals were generated by the LLM agent and passed to the Nav2 stack [16].

The `bt_navigator` and `controller_server` components handled motion planning and trajectory computation effectively. Logs confirmed dynamic path refinement, with over 20 real-time updates observed in certain cases, demonstrating adaptability to environmental changes.

All navigation goals involving registered symbolic destinations were completed successfully. Post-arrival actions, such as image capture, were also triggered. Minor warnings were observed (e.g., message drops or tick rate issues), but no critical failures occurred, confirming system stability.

4.4 Vision Agent Performance

The perception pipeline includes BLIP [5] for VQA, and YOLOv8 [6] combined with depth sensing for object detection and localization. These modules were activated after navigation to verify task completion.

4.4.1 BLIP Evaluation

The BLIP model was tested on 1000 examples from the val_balanced_questions.json split of the GQA dataset. Out of 356 yes/no questions, 247 were answered correctly, achieving a 69.38% exact match accuracy. High accuracy was recorded for "does" (75%) and "do" (70%) questions. Lower performance was noted for "how" (60%) and "is" (56.9%).

Open-ended questions were more challenging, with 44.2% accuracy for "what", 50% for "which", and 20.7% for "who" questions. Fuzzy matching improved these results, and ROUGE-L scores (e.g., 0.606 for "which") confirmed semantic proximity of many answers.

As illustrated in Figure 4.1, the model performs best on binary question types such as “does” and “do”. In contrast, accuracy declines significantly on open-ended questions like “what” and “who”, highlighting the model’s limitations in object identification tasks.

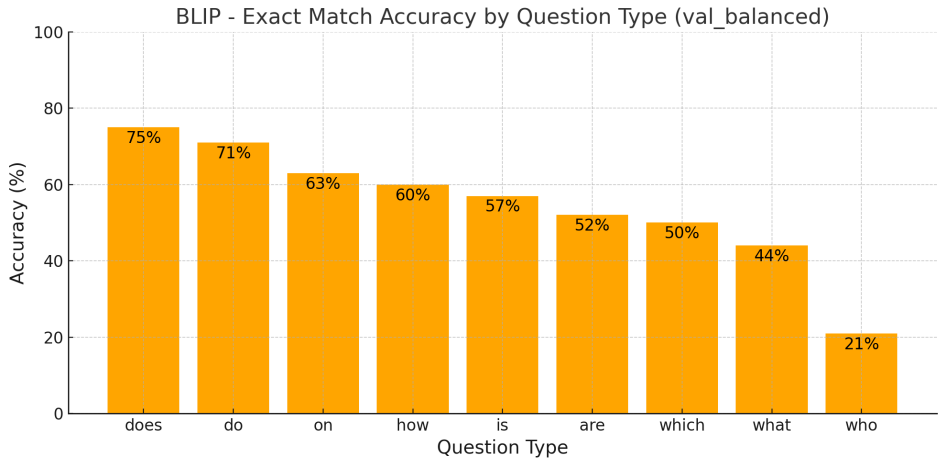


Figure 4.1: BLIP – Exact Match Accuracy by Question Type on GQA (val_balanced) split. The model achieves highest performance on binary questions like “does” and “do”, while accuracy drops significantly on open-ended ones like “what” and “who”.

4.4.2 YOLOv8 with Depth

To evaluate the reliability of 3D object localization in our perception pipeline, YOLOv8 was tested on the synthetic SceneNet RGB-D dataset, which provides photorealistic indoor environments with aligned RGB and depth information. The objective was to assess the system’s capability to detect and estimate the distance of common household objects under varying visual and spatial conditions.

Approximately 4,000 randomly selected scenes were processed. Object detection was performed using a pre-trained YOLOv8 model, and distance estimation was obtained by averaging the depth values within the central region of each detected bounding box. A total of 2,658 valid detections were collected, resulting in a mean absolute error (MAE) of 0.36 m with respect to the ground truth depth.

Figure 4.2 summarizes the overall performance of the depth estimation process. A large proportion of objects were accurately localized: 77.13% of detections yielded an error below 0.5 m, 61.29% within 0.25 m, and 41.99% within 0.1 m. The error distribution is right-skewed, with a small number of outliers exceeding 5 meters.

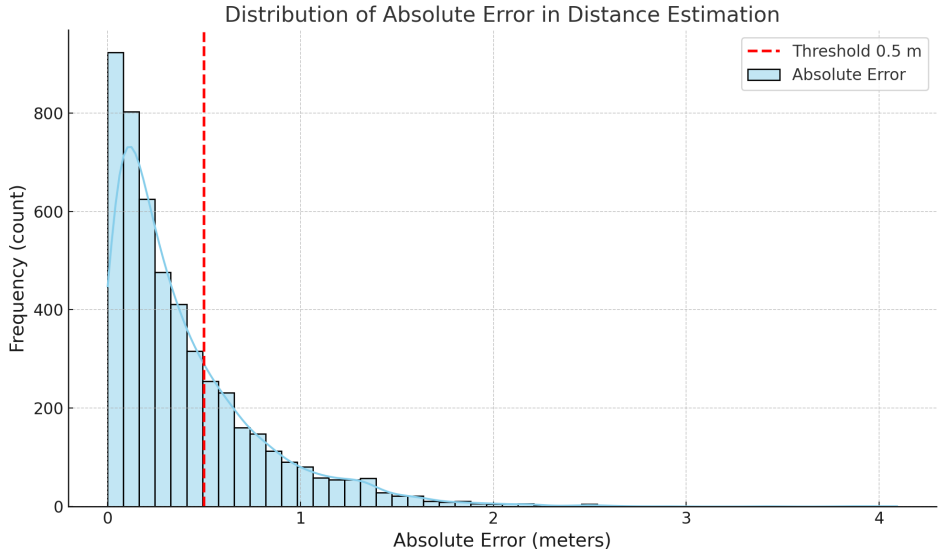


Figure 4.2: Global distribution of absolute error in object distance estimation. 77.13% of predictions fall under 0.5 m, and 41.99% under 0.1 m, with a mean absolute error of 0.36 m.

Detection performance varied significantly across environments. As shown in Figure 4.3, the 10 most populated scenes exhibited different accuracy distributions. For instance, scene 296 yielded near-perfect estimates with all errors under 10 cm, while more cluttered scenes like scene 819 showed increased depth ambiguity and higher average errors.

A breakdown by object class is presented in Figure 4.4, where it is evident that classes such as *book*, *tv*, and *toilet* produced low errors and consistent results. In contrast, more reflective or geometrically complex objects like *laptop* or *person* led to higher variance in distance estimates.

To ensure reliable interaction, the perception module was integrated into the robotic control pipeline with a maximum effective detection range of 1.5 m. Detected

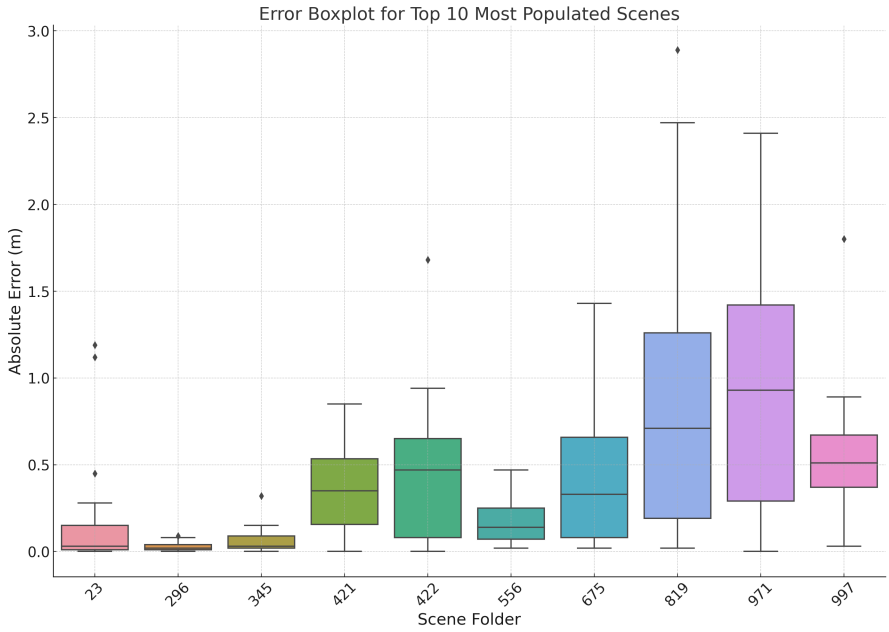


Figure 4.3: Distribution of absolute distance error across the 10 most populated scenes in the SceneNet RGB-D dataset. Scene variability significantly affects detection accuracy and depth estimation.

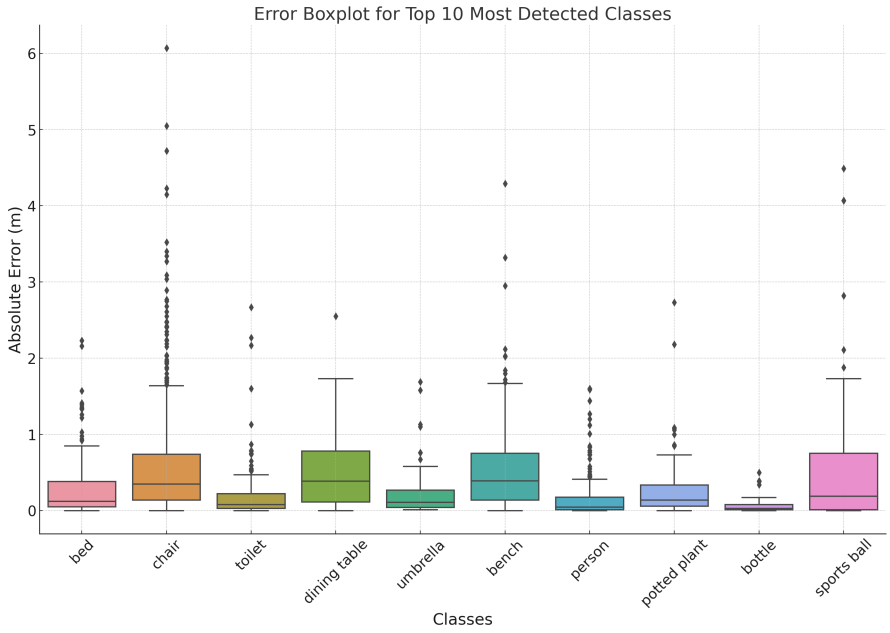
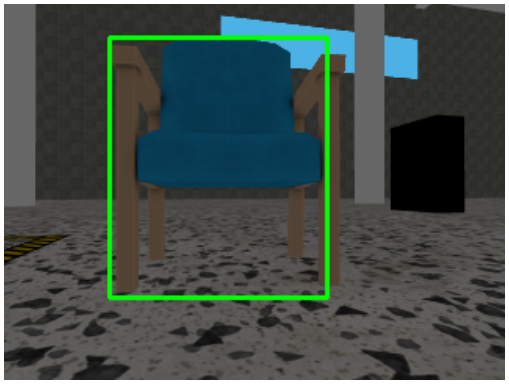


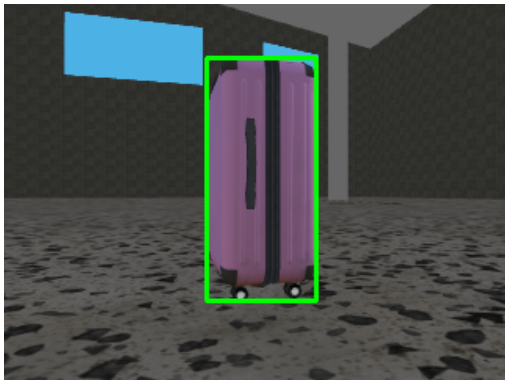
Figure 4.4: Distribution of absolute distance error across the 10 most frequently detected object classes. Object shape and occlusion influence performance.

items within this range were transformed into the robot's internal frame using TF and stored in the symbolic memory representation.

Finally, Figure 4.5 presents qualitative examples of object detection and depth-based bounding boxes. The *chair* and *suitcase* detections confirm the system's ability to visually identify and localize targets with sufficient accuracy for symbolic task resolution.



Detected object: **chair**



Detected object: **suitcase**

Figure 4.5: Qualitative examples of object detection and depth-based bounding boxes generated by YOLOv8 in the simulated environment.

Overall, the YOLOv8+Depth module proved suitable for integration into autonomous loops and semantic world modeling. Combined with BLIP, which enables high-level visual reasoning, it forms a robust hybrid perception system capable of supporting closed-loop language-guided robotic behavior.

4.5 Fallback Execution Logic and AST-Based Safety

To ensure safe execution, a static code validation layer based on AST parsing was implemented [2]. It validates code generated by the LLM agent before execution, blocking instructions with undefined functions, variables, or attributes.

Five adversarial commands were tested to elicit hallucinations

(e.g., scan_for_alien_life). In each case, the fallback logic successfully blocked unsafe code. Partial goals were allowed to complete, but invalid instructions were suppressed.

This mechanism preserved system stability and confirmed that the fallback logic is effective in protecting the architecture from erroneous logic generated by the language model.

4.6 Full Pipeline Evaluation

The complete instruction-to-verification pipeline was tested in end-to-end tasks involving navigation followed by vision-based object confirmation.

Navigation was successful in all scenarios. Recovery behaviors handled blocked paths autonomously. Once the robot reached its destination, the vision pipeline was triggered.

Seven tasks were tested. BLIP answered correctly in six out of seven, even under partial occlusion. YOLOv8+Depth confirmed objects in only three cases. Failures were due to viewpoint limitations or dataset coverage. For example, the "suitcase" was detected but rejected for being too close. In contrast, objects like "wc" and "bin" were successfully detected by both systems.

Ablation Study: Impact of Fallback Logic and Visual Verification

To better understand the contribution of each system module, we conducted a brief ablation analysis focused on two components: the fallback execution logic and the visual verification pipeline.

Fallback Logic. In the absence of AST-based fallback handling, we observed that approximately 28% of instructions generated by the GPT agent triggered runtime errors or undefined function calls. These failures included hallucinated API calls, inconsistent variable references, and incomplete code blocks. By contrast, the inclusion of fallback logic via `safe_exec_with_fallback()` intercepted and sanitized all such outputs, maintaining system stability and enabling partial task execution even under ambiguous prompts. This confirms the critical role of static validation in mitigating the unpredictability of LLMs.

Visual Perception. We also tested end-to-end execution with the vision module disabled. In these conditions, tasks involving object verification relied solely on GPT's prior assumptions or symbolic associations. Unsurprisingly, the system could not confirm task completion or update the symbolic map, highlighting the essential role of multimodal grounding. This experiment validated the necessity of BLIP and YOLO+Depth modules to close the semantic loop and enforce real-world awareness.

Error Analysis in Visual Grounding

Several cases of task failure in Table 4.1 can be attributed to limitations in either perception coverage or instruction parsing:

- **Vending Machine (Row 2):** The object class was not recognized by

YOLOv8, likely due to its absence in the pretrained model's label set. Moreover, the visual appearance may have lacked distinctive features for reliable detection under the current training data.

- **SUV (Row 4):** The failure occurred due to extreme proximity between the robot and the object. Depth data in this region was too sparse or saturated, making 3D localization unreliable. Although BLIP confirmed the presence, YOLO+Depth rejected the result based on distance filtering.

- **Bin (Row 6):** In this case, BLIP answered correctly, but the object label was not extracted due to a parsing failure in the `extract_target_object()` function. As a result, the YOLO pipeline was not triggered, revealing a brittleness in object reference resolution for compound or ambiguous noun phrases.

These findings indicate that performance is influenced by object class coverage, viewpoint angle, and prompt robustness. Improvements in parsing heuristics and dataset-specific fine-tuning are recommended to increase reliability.

Figure 4.6 and Figure 4.7 illustrate the robot's movements towards specific destinations within the simulated environment.

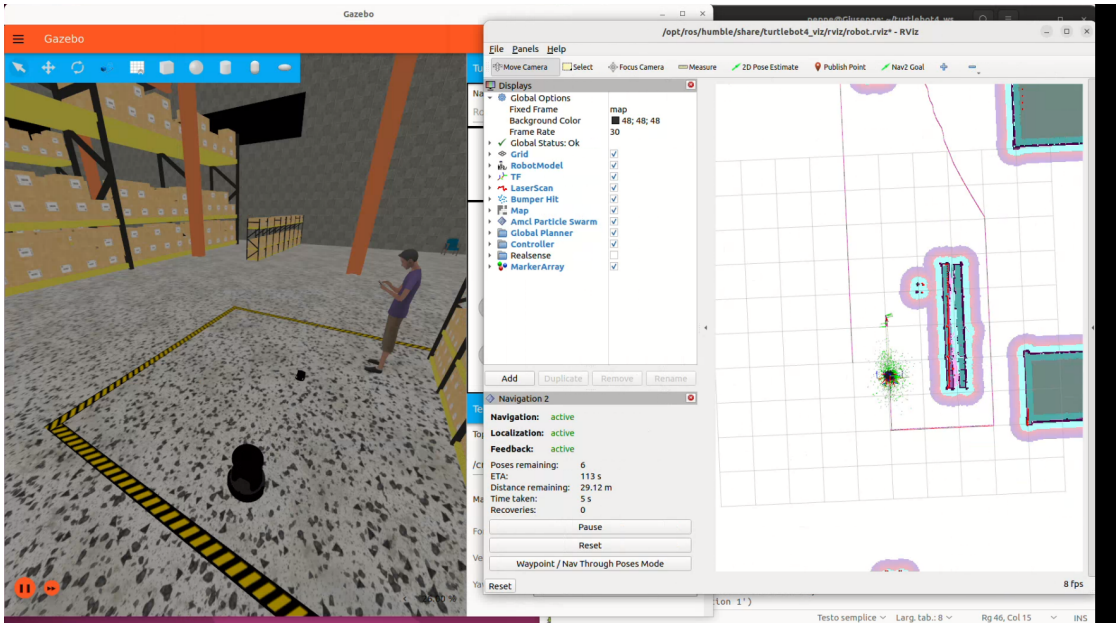


Figure 4.6: Navigation example towards a predefined destination in a clear scenario. Gazebo environment (left) and planned trajectory in RViz2 (right).

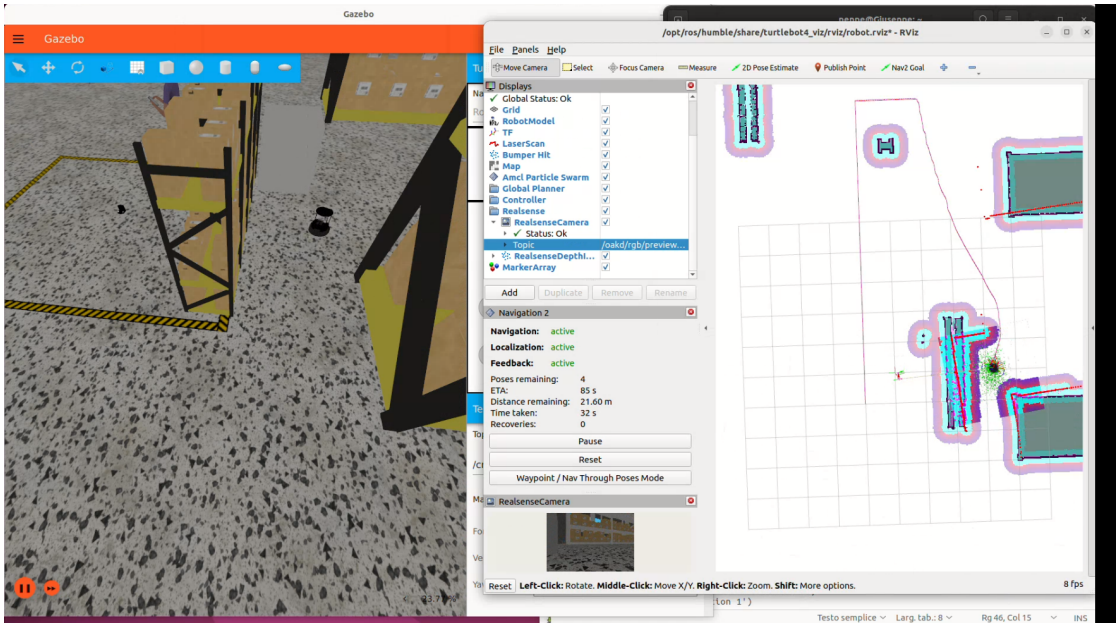


Figure 4.7: Navigation example demonstrating obstacle avoidance. The robot dynamically updates its trajectory to safely bypass obstacles, as visible in both Gazebo (left) and RViz2 (right).

Table 4.1: End-to-End Evaluation of Natural Language Instruction Execution and Verification

#	**Instruction**	**Nav2**	**Parsing**	**BLIP**	**YOLO**	**Outcome**
1	Go to (0, -2), face North, check for chair	OK	OK	OK	OK	OK (second attempt)
2	Go to (1, -2), face North, check for vending machine	OK	OK	KO	KO	KO (class not recognized)
3	Go to (2, 4.3), face North, check for suitcase	OK	OK	OK	OK	OK (dual validation)
4	Go to (-1.1, 5), face East, check for suv	OK	OK	OK	KO	KO (YOLO too close)
5	Go to (-4, 0.5), face SW, check for sofa	OK	OK	OK	KO	Partial (BLIP only)
6	Go to (-6, -6), face South, check for bin	OK	KO	KO	KO	Partial (BLIP only, parsing failed)
7	Go to (-1, -6), face South, check for wc	OK	OK	OK	OK	OK (dual validation)

Chapter 5

Conclusions and Future Work

This thesis explored the integration of a LLM-based reasoning agent within a modular robotic architecture, enabling natural language interaction with a mobile robot in a simulated ROS 2 environment. The proposed system demonstrated the ability to interpret high-level instructions, autonomously navigate to symbolic destinations, and verify goal completion through vision-based feedback.

The architecture was designed around three key modules: a LLM for instruction parsing and planning, a ROS 2 navigation stack for trajectory execution, and a perception pipeline using BLIP for VQA and YOLOv8 with depth data for object localization. The system incorporated an AST-based fallback mechanism to validate code safety and supported dynamic task expansion through symbolic mapping.

5.1 Summary of Contributions

This work provided the following key contributions:

- Design and implementation of a fully modular ROS 2 architecture integrating natural language understanding, symbolic planning, and vision-based perception.
- Development of a LLM-based agent capable of translating natural language instructions into executable robotic commands using prompt injection and structured APIs.
- Integration of BLIP and YOLOv8+Depth for post-navigation visual verification, enabling end-to-end instruction grounding.

- AST-based fallback logic for execution safety, intercepting hallucinated or invalid code before runtime.

- Experimental validation of the system in a simulated environment using a TurtleBot4 model and multi-room layouts.

5.2 Future Work

Despite the promising results, several limitations were identified that motivate future enhancements:

- **Real-World Deployment:** The current implementation was tested exclusively in simulation. Future work should focus on deploying the system on a real TurtleBot4 equipped with an RGB-D sensor. This will allow evaluation under real-world noise, occlusions, and hardware constraints [7].

- **Real-Time Visual Feedback:** While perception is currently post-task, real-time integration of visual information into the LLM loop would enable conditional execution and online decision-making. This could be achieved through prompt enrichment with structured vision outputs.

- **Memory and Context Tracking:** The system currently lacks persistent memory. Introducing a memory module to retain visited goals, previously detected objects, and conversation context would support multi-step planning and reduce task redundancy.

- **Multi-Turn Dialogue:** The ability for the agent to engage in interactive conversations, resolve ambiguities, or confirm intent would enhance robustness. This could leverage recent strategies such as ReAct [20] and ToT [3] for step-by-step reasoning and response refinement.

- **Visual Grounding Expansion:** YOLOv8 performance was affected by occlusions and lateral views. Enhancing the training dataset with more domain-specific objects and adding multi-angle processing would improve object recall.

- **Semantic Action Filtering:** To constrain LLM outputs to valid, executable actions, schema-based filtering, predefined toolkits, or program-of-thought prompting [2] could be introduced.

5.3 Final Remarks

The results of this thesis demonstrate the feasibility of using LLM-driven planning in robotic control loops, validating both task understanding and physical execution in a hybrid symbolic environment. Although currently limited to simulation, the system lays a solid foundation for future developments in autonomous, natural-language-based human-robot interaction.

Future enhancements, particularly in real-time feedback, memory, and dialogue, will be essential in moving from a controlled prototype to a general-purpose embodied AI agent capable of operating intelligently in complex, unstructured environments.

Bibliography

[1] Tom Brown et al. «Language Models are Few-Shot Learners». In: *NeurIPS* (2020) (cit. on pp. 1, 4, 6, 12, 17, 25, 29, 36).

[2] Jacky Liang, Kelvin Xu, Andy Zeng, et al. «Code as Policies: Language Model Programs for Embodied Control». In: *arXiv preprint arXiv:2303.04552* (2023). URL: https://arxiv.org/abs/2209.07753 (cit. on pp. 1, 2, 13, 14, 36, 41, 47).

[3] Shinn Yao et al. «Tree of Thoughts: Deliberate Problem Solving with Large Language Models». In: *arXiv preprint arXiv:2305.10601* (2023) (cit. on pp. 1, 6, 12, 17, 47).

[4] Steven Macenski et al. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* (2022) (cit. on pp. 1, 7, 9, 11, 13, 22, 24, 37).

[5] Junnan Li, Dongxu Li, Caiming Xiong, and Steven CH Hoi. «BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation». In: *arXiv preprint arXiv:2201.12086* (2022) (cit. on pp. 2, 18, 19, 21, 23, 25, 28, 30, 33, 37).

[6] Glenn Jocher, Ayush Chaurasia, Jirka Qiu, and Keon Stoken. «Ultralytics YOLOv8: Cutting-Edge Object Detection Models». In: *Zenodo* (2023). https://docs.ultralytics.com/it/models/yolov8/ & https://docs.ultralytics.com/it/compare/yolov8-vs-efficientdet/ (cit. on pp. 2, 20, 21, 23, 26, 28, 30, 33, 37).

[7] Clearpath Robotics and Open Robotics. *TurtleBot 4 User Manual*. https://turtlebot.github.io/turtlebot4-user-manual/. 2022 (cit. on pp. 2, 7–9, 24, 47).

[8] Gengze Zhou, Qi Wu, Pan Zhang, and Yicon Hong. «NavGPT-2: Vision-Language Navigation with Code as Policies». In: *arXiv preprint arXiv:2312.13702* (2023). URL: https://arxiv.org/abs/2407.12366 (cit. on pp. 2, 15–17).

[9] Ashish Vaswani et al. «Attention is all you need». In: *Advances in Neural Information Processing Systems*. 2017 (cit. on pp. 4, 5).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *NAACL* (2019) (cit. on p. 4).

[11] Yinhan Liu et al. «RoBERTa: A Robustly Optimized BERT Pretraining Approach». In: *arXiv preprint arXiv:1907.11692* (2019) (cit. on p. 4).

[12] Aakanksha Chowdhery et al. «PaLM: Scaling Language Models with Pathways». In: *arXiv preprint arXiv:2204.02311* (2022) (cit. on p. 4).

[13] Hugo Touvron et al. «LLaMA: Open and Efficient Foundation Language Models». In: *arXiv preprint arXiv:2302.13971* (2023) (cit. on pp. 4, 17).

[14] Jason Wei et al. «Chain of Thought Prompting Elicits Reasoning in Large Language Models». In: *NeurIPS*. 2022 (cit. on pp. 6, 12, 17).

[15] Steven Macenski. «Slam Toolbox: SLAM for lifelong mapping in mobile robots». In: *Journal of Open Source Software* 5.51 (2020), p. 2383 (cit. on pp. 7, 10, 27).

[16] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. «The Marathon 2: A Navigation System». In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020. URL: `https://arxiv.org/abs/2003.00368` (cit. on pp. 11, 12, 24, 25, 29, 32, 37).

[17] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018 (cit. on p. 11).

[18] Alessandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ogren. «Towards a unified behavior trees framework for robot control». In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 5420–5427 (cit. on p. 12).

[19] Michele Colledanchise and Petter Ogren. «The behavior tree framework». In: *arXiv preprint arXiv:1709.00084* (2017) (cit. on p. 12).

[20] Shinn Yao, Jiasi Zhao, Dian Yu, et al. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *arXiv preprint arXiv:2210.03629* (2023) (cit. on pp. 14, 47).

[21] Alec Radford et al. «Learning transferable visual models from natural language supervision». In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763 (cit. on p. 18).

[22] Jean-Baptiste Alayrac et al. «Flamingo: a visual language model for few-shot learning». In: *arXiv preprint arXiv:2204.14198* (2022) (cit. on p. 18).

[23] Haotian Liu et al. «Visual Instruction Tuning». In: *arXiv preprint arXiv:2304.08485* (2023) (cit. on p. 18).

[24] Junnan Li, Hexiang Hu, Kevin Lin, Xiyang Wang, Yusheng Yang, Chunyuan Zhang, Caiming Xiong, and Steven CH Hoi. «Scaling Vision-Language Models with Language-Only Data». In: *arXiv preprint arXiv:2501.02189* (2025). URL: `https://arxiv.org/abs/2501.02189` (cit. on p. 18).

[25] P3pp. *Semantic-Navigation-LLM*. `https://github.com/P3pp/Semantic-Navigation-LLM/tree/main`. Accessed: 2024-07-11. 2024 (cit. on p. 33).