Politecnico di Torino

Master of Science in Mechanical Engineering

Department of Mechanical and Aerospace Engineering

Master's Degree Thesis

Development of an LLM-driven Framework for Intelligent Robotic Manipulation in Self-Driving Labs: A Case Study on Autonomous Plant Care

Supervisors:
Prof. Stefano Mauro
Dr. Matteo Melchiorre
Dr. Laura Salamina

Candidate:
Francesco Re
ID: 324988

A.Y. 2025/2026

Abstract

This thesis explores the development of a Self-Driving Lab (SDL) controlled by a Large Language Model (LLM) to perform plant care tasks, serving as a case study for non-cooperative and morphologically evolving environments.

The proposed architecture integrates a cognitive layer based on an LLM with a 7-Degree of Freedom (DoF) robotic manipulator controlled via Robot Operating System 2 (ROS2) and MoveIt.

Plants were selected as the biological target to investigate robotic interactions that go beyond the limits of rigid automation. Their morphologically evolving nature requires adaptive perception and decision-making, providing an ideal case study to test the reasoning capabilities of the LLM-driven architecture.

Following a comparative analysis of different Artificial Intelligence (AI) integration strategies, a cloud-based model integration was adopted, refined through system instruction tuning. In this framework, the LLM acts as the high-level decision-maker: it interprets environmental data from Internet of Things (IoT) sensors (soil moisture, light and air temperature) to determine the necessary interventions. The LLM also orchestrates the processing of RGB-D data using computer vision algorithms for object identification and pose estimation. Moreover, the system interfaces with MoveIt to generate collision-free trajectories, transforming high-level decisions into robotic manipulation.

Initially, the architecture was validated through the development of a virtual environment in CoppeliaSim. This served as a primary test-bed to refine the control pipeline, validate the grasping strategy and optimize the target geometry. These preliminary tests provided an assessment of the framework logic and the decision-making capabilities of the LLM.

Subsequently, the system was deployed in a real-world experimental robotic cell featuring a Franka Research 3 arm. This phase evaluated the transferability of the methods to a physical environment. The experimental campaign focused on the execution of grasping and plant care tasks in unstructured settings and allowed for an observation of how the system adapts to real-world scenarios.

Lastly, a systematic analysis of diagnostic capabilities and corrective actions was performed. This investigation explored the ability of LLMs to identify anomalies and perform diagnostic analysis when faced with non-standard scenarios. While the results highlighted the potential of LLMs to manage environmental uncertainty, the study also identified critical challenges regarding the robustness and reliability of cognitive agents.

Future works will focus on the characterization of the LLM capabilities through an experimental campaign with different plants, different environmental conditions and robotic cell configurations. Another aspect is the study of the integration of a local LLM module in an embedded unit, enabling the system to be completely autonomous.

Contents

List of Figures vi

List of Source Codes vii

List of Acronyms viii

1 Introduction 1

1.1 The shift in robot autonomy: from rigid to adaptive systems 2

1.1.1 Conventional robotics and rigid automation 2

1.1.2 The transition to collaborative and dexterous systems 2

1.1.3 The advent of AI-driven intelligent manipulation 2

1.1.4 The laboratory as an autonomous frontier 3

1.2 State of the art: self-driving labs and autonomous experimentation 3

1.2.1 The evolution of laboratory autonomy 3

1.2.2 The integration of LLMs and vision-language models 4

1.3 Problem statement: the limitation of rigid automation with dynamics objects 4

1.3.1 The "blindness" of predefined setups 4

1.3.2 The perception and full-state information gap 4

1.3.3 The failure of rigid automation in anomaly handling 5

1.4 Contribution of this work 5

2 Self-Driving Lab Case Study: Autonomous Plant Care 6

2.1 Task description: the biological variability of plants 6

2.1.1 The plant as a non-deterministic target 6

2.1.2 Perception challenges in semi-structured environments 7

2.1.3 The biological closed-loop cycle 7

2.2 Requirements analysis: what does a robot need to manage a plant? 7

2.2.1 Multi-modal perception and sensing 8

2.2.2 AI-driven interpretation and feature extraction 10

2.2.3 Precision action and manipulation 11

2.3 Trade-offs on hardware and software 11

2.3.1 Robotic platform and manipulation trade-offs 12

2.3.2 Multimodal sensing and vision strategy 12

2.3.3 Software architecture and cognitive integration 12

3 Large Language Models for Robotics 14

3.1 From chatbots to robot controllers 14

3.2 Methodologies for LLM-driven control 15

3.2.1 Prompt engineering and logical reasoning . . . 15
3.2.2 Task decomposition and function libraries . . . 15
3.2.3 Feedback integration and closed-loop control . . . 16
3.2.4 Spatial reasoning and 3D trajectory generation . . . 16
3.3 Architectural constrains: cloud vs edge AI . . . 17

4 System Design and Architecture 18
4.1 System architecture overview . . . 18
4.2 The perception layer: sensors and data handling . . . 20
4.2.1 Vision subsystem and context-aware perception . . . 20
4.2.2 Plant telemetry data handling . . . 23
4.3 The cognitive layer (AI) . . . 24
4.3.1 Multimodal context aggregation . . . 24
4.3.2 Prompt engineering and structured querying . . . 25
4.3.3 Output parsing and safety check . . . 28
4.3.4 System transparency and Graphic User Interface . . . 29
4.4 The motion layer . . . 30
4.4.1 Mission dispatching and dynamic scene reconstruction . . . 31
4.4.2 Motion planning strategies . . . 33
4.4.3 Kinematic logic and motion primitives . . . 34
4.4.4 High-level mission sequences . . . 39
4.4.5 Fault tolerance and recovery mechanism . . . 41

5 Modelling and Simulation 43
5.1 Environmental setup and ROS2 interfacing . . . 43
5.1.1 The simulated workspace . . . 43
5.1.2 The Coppelia bridge node . . . 44
5.1.3 Trajectory execution and custom robot control . . . 45
5.2 Validation of the control and cognitive pipeline . . . 46
5.2.1 Use case 1: moving . . . 46
5.2.2 Use case 2: watering . . . 48
5.3 Perception strategy evaluation: AI vs local processing . . . 49
5.3.1 Test 1: prompt A vs prompt B . . . 49
5.3.2 Test 2: local perception . . . 50
5.3.3 Comparisons and conclusions . . . 50

6 Experimental Validation 51
6.1 Experimental setup . . . 51
6.1.1 The robotic cell and the hardware components . . . 51
6.1.2 Multi-camera system . . . 53
6.1.3 Custom extrinsic calibration using ArUco markers . . . 54

6.1.4 Telemetry hardware integration 55
6.2 Sim-to-real transfer 57
6.2.1 Mission planner tuning 57
6.2.2 Perception pipeline refinement 58
6.3 Real-world case studies 60
6.3.1 Autonomous object relocation 60
6.3.2 Context-aware plant care 62
6.3.3 Non-standard scenarios and workspace anomalies 65
6.3.4 Advanced cognitive diagnostics and corrective reasoning 68

7 Conclusions and future works **72**
7.1 Summary of contributions 72
7.2 System limitations 73
7.3 Future developments 74

List of Figures

2.1 Remote sensing phenotype features and their corresponding technologies [14] 9
2.2 Main proximal sensors for precision agriculture [14] 10

4.1 High-level system architecture outlining the Cognitive, Perception and Motion layers, along with the ROS 2 data flow 19
4.2 Logical workflow of the SDL framework 20
4.3 Perspective projection of the 3D point cloud clusters onto the 2D RGB frames 22
4.4 Example of the Graphic User Interface (GUI) displaying the AI response to a low soil moisture condition 30
4.5 Dynamic scene reconstruction in RViz. Difference between raw volumetric representation and primitive collision objects 32
4.6 The grasping primitive sequence 37
4.7 The placing primitive sequence 38
4.8 The pouring action sequence 41

5.1 The simulated workspace in CoppeliaSim containing the robotic manipulator, the two cameras, the plant and the cup 44
5.2 Simulated execution of the moving sequence 47
5.3 Simulated execution of the watering sequence 48

6.1 The experimental setup 52
6.2 Hardware synchronization topology utilizing the Sync Hub Pro 53
6.3 Eye-to-Hand calibration setup 54
6.4 Schematic of the telemetry hardware integration detailing the electrical connections and data types between the environmental sensors, the Arduino microcontroller and the PC 56
6.5 Side-by-side comparison of object detection and geometric refinement 60
6.6 The GUI displaying the AI cognitive reasoning and the generated executive JSON for the relocation task 61
6.7 Detailed real-world execution of the autonomous relocation task by the Franka manipulator 62
6.8 The GUI displaying the autonomous AI decision-making process triggered by low soil moisture telemetry 63
6.9 Real-world execution of the watering task. Note: while the mission was triggered by live telemetry, the physical moisture sensor was temporarily removed from the pot during this specific photographic sequence to prevent hardware damage during repeated kinematic captures 64
6.10 Fully-integrated autonomous researcher loop 65

6.11 System detection to unmapped anomalies 65
6.12 The GUI displaying the autonomous AI decision-making process triggered by the obstacle presence in the workspace 66
6.13 Dynamic injection of unmapped anomalies as red collision bounding boxes in the RViz environment 66
6.14 Visual profile of the watering vessel intentionally laid horizontally on the workspace 67
6.15 The GUI displaying the AI failing to recognize the laid cup, classifying it as an obstacle and aborting the mission 68
6.16 The GUI displaying the AI diagnostic reasoning under optimal telemetry conditions 69
6.17 The GUI displaying the AI diagnostic reasoning when explicitly prompted to identify the plant species 70
6.18 The GUI displaying the AI diagnostic reasoning when prompted to suggest corrective actions to clear the workspace 71

List of Source Codes

1 RANSAC table filtering algorithm 21
2 DBSCAN clustering of remaining points 21
3 Algorithm selection based on LLM semantic category 22
4 Custom interface used to transmit geometric data 23
5 The system instructions defining the cognitive constraints and behavioral rules for the LLM 26
6 The JSON schema imposed on the Gemini API to guarantee deterministic output parsing 27
7 The safety override logic and the construction of the streamlined executive payload dispatched to the motion layer 29
8 Orientation constrains settings 34
9 Grasping orientation computing 35
10 Pre-grasping pose computing 35
11 Allowed Collision Matrix (ACM) manipulation 39
12 Recovery service sequence 42
13 Trajectory down-sampling strategy 45
14 Custom proportional controller implemented in the bridge node 46
15 Example of the JSON-formatted plant telemetry 57

List of Acronyms

SDL	Self-Driving Lab
LLM	Large Language Model
DoF	Degree of Freedom
ROS2	Robot Operating System 2
AI	Artificial Intelligence
IoT	Internet of Things
ML	Machine Learning
Cobot	Collaborative Robot
API	Application Programming Interface
CoT	Chain of Thought
RAG	Retrieval-Augmented Generation
VLM	Vision-Language Model
RANSAC	Random Sample Consensus
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
GUI	Graphic User Interface
OMPL	Open Motion Planning Library
PTP	Point-To-Point
ACM	Allowed Collision Matrix
TCP	Tool Center Point

1. Introduction

The field of robotics is undergoing a paradigm shift from traditional and rigid automation towards more flexible systems. For decades robotic systems have been confined to highly structured industrial environments where they perform repetitive tasks with high precision and repeatability. In these conditions, robots are programmed to execute pre-defined sequences of actions and are incapable of adapting to unexpected changes in the workspace. However, the modern applications of robotics are moving towards more dynamic and unstructured environments such as homes, hospitals and space research. In these contexts, robots need to be able to perceive and understand their surroundings and adapt their behavior.

This transition is being accelerated by the recent improvements in Artificial Intelligence (AI) and Machine Learning (ML). These technologies contributed to the development of intelligent agents capable of learning from data, making decision and adapting to new situations. The emergence of Large Language Models (LLMs) opened up new possibilities for natural language understanding and reasoning in robotic systems. By integrating LLMs into robotic architectures it is possible to create system capable of understanding complex instructions, interpret contextual information and make decisions based on the environment.

Within this landscape, the concept of Self-Driving Lab (SDL) has emerged as a promising paradigm for accelerating scientific discovery. In a SDL, robotic systems are required to function as autonomous researchers dealing with experimental design, execution and data analysis. To achieve this and overcome traditional industrial paradigm, robots must have reasoning capabilities and the ability to interact with non-cooperative and evolving targets.

To provide the necessary context for this work, this chapter explores the evolution of robotic autonomy from rigid to adaptive reasoning. It further analyzes the state of the art in autonomous experimentation highlighting the limits of current automation when dealing with dynamic objects. Finally, the chapter outlines the proposed LLM-driven framework and its contribution to the development of intelligent manipulator in SDLs.

1.1 The shift in robot autonomy: from rigid to adaptive systems

Over the years, robotic systems have evolved from simple machine, performing repetitive tasks, to more complex and intelligent agents. This evolution is not only about the physical capabilities of robots but also about the way they perceive and interact with their surroundings.

1.1.1 Conventional robotics and rigid automation

For decades, the main usage of robots has been in automation of manufacturing processes. In these settings, robots are designed and programmed to perform specific tasks in highly structured environments where the conditions are predictable and controlled. These systems typically operate within a deterministic framework, interacting with cooperative targets: objects with predefined geometries and fixed positions.

While this approach increased production efficiency and reduced the risk for human workers [1], it also limited the scope of robotic applications to environments where the conditions are well-defined and static. They are unable to handle unexpected changes in the workspace or interact with dynamic objects.

1.1.2 The transition to collaborative and dexterous systems

The introduction of Collaborative Robots (Cobots) marked the first step towards more flexible and adaptable robotic systems. Unlike traditional manipulator confined to safety cells, Cobots are engineered to work alongside human operators in shared workspaces.

To ensure safe interaction, these systems incorporate advanced sensors and strict safety protocols, including force and torque limits, smoother physical structures and reduced dimensions.

Furthermore, programming Cobots is more intuitive and user-friendly allowing operators to easily implement new tasks and adapt to changing production requirements. This flexibility makes Cobots suitable for a wider range of applications beyond traditional manufacturing, including assembly, packaging and quality control [2].

1.1.3 The advent of AI-driven intelligent manipulation

The current frontier in robotic autonomy involves integrating AI and ML to create agents capable of autonomous reasoning. A major challenge in traditional modeling is the requirement for "full-state information": exact physical parameters that are often impossible to obtain from raw perception in real-world scenarios. Learning-based dynamics models provide a solution by deriving state transitions directly from interaction data, capturing complex factors such as frictional contact or material deformation.

Robots are thus moving beyond "primitive goals" toward "mission competence", where the agent optimizes a policy to achieve a high-level objective over a defined time horizon. In this framework, LLMs act as high-level decision-makers, orchestrating learning-based capabilities and providing the reasoning necessary to handle non-cooperative targets that exhibit morphological changes or complex multi-object interactions [3].

1.1.4 The laboratory as an autonomous frontier

This technological progression finds a critical application in the development of SDLs. In these environments, robots are no longer merely supporting tools but function as autonomous researchers capable of "closing the loop" between experimental design and physical execution. By integrating cognitive robotics, SDLs can redefine the boundaries of autonomous experimentation, managing biological specimens and evolving targets that traditional automation cannot process.

1.2 State of the art: self-driving labs and autonomous experimentation

The concept of the SDL represent the pinnacle of laboratory autonomy, where robotic systems evolve from task executors to autonomous closed-loop researchers. In a SDL, AI and robotics cooperate to autonomously conduct experiments, gather and analyze data and iteratively refine decision-making processes in research with minimal human intervention [4]. This requires a level of autonomy and adaptability that goes beyond traditional automation.

1.2.1 The evolution of laboratory autonomy

The laboratory automation began in the 1940s when the rise of industrial automation began influencing the field of chemistry. The first automated systems were designed to perform simple task such as sample handling and preparation. This process reached a milestone in 2004 when King et al. introduced the idea of a "robot scientist" that could determine gene functions by using ML-driven experiments, interpreting results and providing feedback for future experiments. The team developed a fully automated system named "Adam" which identified 12 new gene functions [4].

In the last decade, rapid advancements in robotics and AI, including LLM and computer vision, have further accelerated the development of SDLs toward more modular and flexible systems.

A notable example is "Ada" (2020) a flexible and modular SDL capable of autonomously synthesizing, processing and characterizing organic thin films. "Ada" trains itself how to find target parameters without any prior knowledge, enabling iterative experimental designs that maximize the information gain per sample [5].

However, these systems often operate in predefined setups where the targets are cooperative and static, limiting their applicability in more dynamic research contexts.

1.2.2 The integration of LLMs and vision-language models

The most recent advancements have integrated LLMs and computer vision to overcome the limitations of traditional automation. A frontier in this direction is "ORGANA" (2024), a system that combines vision-language models, task and motion planning with scheduling and an LLM-based interface. These features give the robot a greater flexibility to plan and execute complex tasks autonomously while interacting with humans [6][4].

Despite these advancements, the current state of the art in SDLs remains largely focused on chemical synthesis and material science, where experiments involve cooperative and predictable targets like liquids, powders and solid samples. However, as the focus shift towards more complex research domains involving morphologically evolving or non-cooperative targets, the current automation faces significant limitations in perception, reasoning and adaptability.

1.3 Problem statement: the limitation of rigid automation with dynamics objects

A major challenge in the development of a SDL is the need to handle dynamic and non-cooperative targets like biological specimens or multi-structured objects. Traditional automation relies on predefined sequences of actions and assumes a static environment where every interaction is predictable.

1.3.1 The "blindness" of predefined setups

A critical limitation of many current SDLs and industrial robotic systems is their reliance on predefined and fixed environments. As highlighted previously, these systems are often "blind" to their surroundings because they lack the ability to dynamically detect and locate objects and are, thus, limited to predefined setups [4].

1.3.2 The perception and full-state information gap

Conventional robotic systems often relies on analytical physical-based models to predict the behavior of objects ad plan actions. These models typically require full-state information, which can be difficult or impossible to extract from perception data in complex, real-world scenarios [3].

Dynamic objects present a significant challenge because of different factors such as morphological changes, deformability and partial observability. These factor introduce a significant gap between the information available from perception and the requirements of

traditional modeling approaches, making it difficult for robots to interact effectively with dynamic targets.

1.3.3 The failure of rigid automation in anomaly handling

Standard automation follows a linear logic when the system performs a predefined sequence of actions. When an anomaly occurs, the system triggers a safety stop or continues to execute movements, risking to produce damage to the robot, the target or the environment.

While humans have an intuitive grasp on physics that lets us interact with the environment and predict its evolution, robots lack this common sense and are unable to adapt their behavior to plan corrective actions autonomously [3]. This is a critical limitation in the context of the development of SDLs.

1.4 Contribution of this work

This thesis proposes an integrated framework that leverages an LLM to orchestrate intelligent robotic manipulation in a SDL. By utilizing a plant care scenario as a case study, this work demonstrates how an LLM-driven system can effectively handle dynamic and non-cooperative targets, overcoming the limitations of traditional automation.

The contribution of this work can be summarized as follows:

- development of an LLM-driven cognitive layer: this layer serves as the high-level decision-maker, bridging the gap between mission competence and low-level robotic execution. It interprets environmental data, orchestrates data processing and plan the interventions;
- integration of multi-source perception: the framework orchestrates the processing of RGB-D data and environmental IoT sensors (soil moisture, light and air temperature). By relying on the reasoning capabilities of the LLM, the system can interpret complex scenes and handle partial observability;
- development of a robust motion planning pipeline: the system incorporates a motion control layer based on MoveIt [7] and ROS2 [8] that translates high-level decision into collision-free trajectories;
- sim-to-real transfer: the architecture was validated through the development of a virtual environment in CoppeliaSim [9]. This served as a primary test-bed to refine control pipelines and validate grasping strategies before deployment in a physical experimental cell featuring a Franka Research 3 arm [10];
- anomaly identification and diagnostic capabilities: a systematic analysis of the LLM ability is performed to identify anomalies and suggest corrective actions.

2. Self-Driving Lab Case Study: Autonomous Plant Care

This chapter defines the application scenario that serves as the experimental foundation for this thesis: autonomous plant care within a Self-Driving Lab (SDL) framework. While recent literature has documented the success of SDL in materials science and chemistry, this case study address the challenge of managing biological specimens. These targets represent a significant point of discontinuity compared to traditional automation due to their deformable, non-cooperative and morphologically evolving nature.

Rather than detailing the software implementation, which is discussed in subsequent chapters, the following section focus on analyzing the biological variability of plants as a robotic target, the requirements for their autonomous management and the high-level trade-offs adopted to bridge the gap between abstract reasoning and physical intervention.

2.1 Task description: the biological variability of plants

The objective of this section is to define the challenges inherent to the autonomous management of biological specimens by applying the precision agriculture paradigm at a laboratory scale. The primary obstacle in automating plant care lies in the intrinsic variability of the biological target, which differs significantly from the deterministic components handled in conventional industrial automation. Unlike rigid automation systems, plant care requires a data-driven management strategy to perform "the right thing, in the right place, and at the right time" [11].

2.1.1 The plant as a non-deterministic target

In industrial automation, robots interact with objects possessing known geometric and physical properties. In contrast, a plant is a dynamic living system characterized by complex phenotyping and intricate anatomical and morphological traits [12].

The robotic task is therefore not a predefined sequence of actions but a complex process of continuous assessment. The non-deterministic nature of the target stems from its constant morphological evolution: its shape, volume and orientation change over time due to growth and health status. Consequently, the required interventions, such as localized irrigation or

nutrient delivery, must be dynamically adjusted to meet these evolving physiological needs.

2.1.2 Perception challenges in semi-structured environments

The manipulation of plants introduces technical challenges that exceed the requirements of standard robotics. The laboratory environment, while controlled, presents several semi-structured challenges documented in current literature [13]:

- target occlusion: the complex morphology of foliage and the heterogeneous distribution of branches often create significant self-occlusion phenomena. This hinders vision sensors in correctly localizing points of intervention, such as the soil surface for irrigation or a specific symptomatic leaf;
- environmental interference: sensor precision is highly influenced by ambient conditions, such as rapidly varying light, which can degrade the performance of RGB cameras used for color and shape perception;
- deformable and fragile targets: biological tissues are fragile and deformable. This imposes strict constraints on manipulation precision and contact force to avoid permanent tissue damage during robotic interaction.

2.1.3 The biological closed-loop cycle

The ultimate goal is to "close the loop" of autonomous experimentation. In the context of biological SDLs, this cycle is divided into three fundamental phases that integrate smart sensors and decision support systems [13][14]:

- monitoring and data collection: non-invasive assessment of the plant status and environmental parameters;
- analysis and planning: interpretation of multisensory data to identify specific stress factors and plan required treatments;
- precision intervention: execution of the physical actions, such as irrigation, pruning or nutrient administration, delivered only where and when necessary to optimize resource usage.

2.2 Requirements analysis: what does a robot need to manage a plant?

To effectively implement a closed-loop control cycle for biological specimens, a robotic system must integrate advanced technical capabilities across perception, reasoning and physical interaction. In line with the precision agriculture paradigm, the system goal is to perform the correct action at the optimal place and time, requiring a high level of autonomy.

2.2.1 Multi-modal perception and sensing

The core of plant management lies on the ability to accurately assess both soil and vegetation health through remote and proximal sensing [14].

Remote sensing encompasses non-invasive and non-destructive approaches for analyzing plant conditions from a distance. These technologies, primarily based on optical-visible and near-visible spectrum sensors, are used for phenotyping and can be classified into three main groups [14]:

- structural characterization: estimating traits such as canopy volume, plant height and biomass to optimize resource distribution;
- plant or fruit detection: identifying objects of interest using features like color, shape and temperature for automated actions like harvesting or pruning;
- physiology assessment: analyzing spectral patterns to assess nitrogen levels, chlorophyll content and water stress.

Vision-based and range sensors are two main types of remote sensing technologies.

Vision-based sensors primarily utilize RGB or color cameras to determine texture and geometric characteristics, though their performance is significantly affected by fluctuating ambient lighting conditions. Advanced systems employ stereo vision or structured light cameras to reconstruct the environment in three dimensions, providing the 3D point clouds necessary for identifying structural parameters. Due to sensitivity to lighting, structured light cameras are predominantly utilized in controlled environments like laboratories or greenhouses. Thermal cameras provide a critical indicator of water stress by measuring leaf temperature, while spectrometers (multispectral and hyperspectral) analyze radiation absorption to infer vital physiological variables [14].

Range sensors provide the spatial context required for navigation and structural modeling. Ultrasonic range sensing uses high-frequency acoustic pulses to estimate crop metrics such as volume and density. Time-of-Flight cameras integrate arrays of detectors to provide both 3D distance and intensity data, allowing for the detailed modeling of individual leaves. Light Detection and Ranging (LiDAR) is the most prominent range-sensing technology, utilizing laser pulses to generate high-resolution 3D maps of the canopy [14].

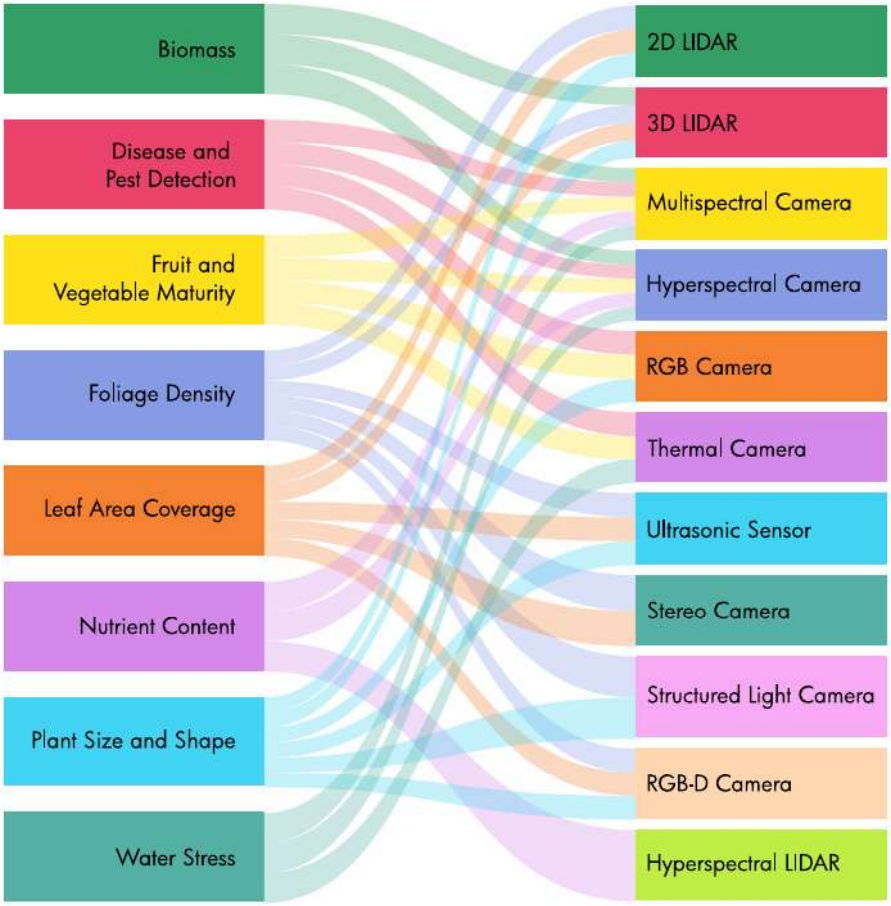


Figure 2.1: Remote sensing phenotype features and their corresponding technologies [14]

While remote sensing provides extensive coverage, it is often restricted to surface evaluations and may lack the resolution necessary to determine localized soil variability. Proximal sensing addresses this gap by utilizing sensors in proximity to or in direct contact with the ground to measure soil properties with higher precision. Methods such as electromagnetic induction and electrical resistivity are widely used to measure soil electrical conductivity, which provides data on soil texture, compaction, pH and nutrient composition. Furthermore, ground-penetrating radar and time-domain reflectometry are employed to map soil depth, structure and volumetric water content [14].

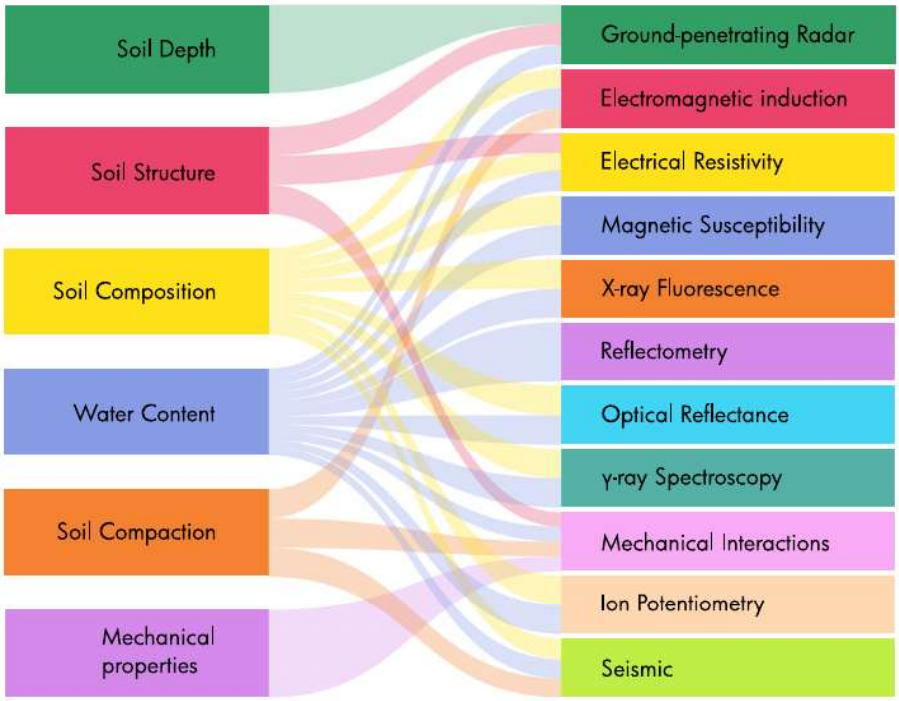


Figure 2.2: Main proximal sensors for precision agriculture [14]

2.2.2 AI-driven interpretation and feature extraction

Artificial Intelligence (AI) serves as the essential bridge between multi-modal perception and autonomous decision-making, transforming raw sensory data into actionable information for plant management.

Advanced computer vision and deep learning models are employed to execute specific interpretation tasks [12]:

- target detection and monitoring: utilizing real-time architectures like YOLO to identify specific plant organs, fruits or pests even under challenging conditions such as foliage occlusion;
- health diagnostics and disease detection: assessing complex traits like plant stress and physiological parameters to identify pathogens before they exhibit widespread symptoms;
- yield and performance prediction: analyzing phenotypic traits in relation to spatially and temporally dynamic environments to estimate future productivity.

By fusing phenotypic features with environmental variables, AI models enable the system to autonomously select the most appropriate intervention strategies. This analytical phase effectively bridges the gap between raw perception and physical response, identifying not only "the right thing" to do but also precisely where the intervention is required.

However, for these high-level decisions to result in effective plant care, they must be translated into localized physical commands, shifting the challenge from abstract reasoning to the requirements of high-dexterity interaction and precision movement.

2.2.3 Precision action and manipulation

Physical action represents the final execution stage in the precision agriculture cycle, where algorithmic decisions are translated into localized physical interventions. To effectively manage biological specimens, robotic hardware must be designed with high adaptability to operate within versatile and complex environments.

Based on recent technological advancements, the requirements for an effective action layer include [15]:

- flexible manipulation: modern systems are increasingly integrating bionic principles to develop flexible manipulators capable of interacting with fruits and vegetables without causing permanent tissue damage;
- soft grippers and mechanical constraints: because biological targets are inherently fragile, the use of soft robotic grippers is essential for enabling gentle physical interaction;
- vision-action integration for occlusions: the effectiveness of autonomous intervention is frequently limited by self-occlusions caused by dense foliage. The integration of advanced vision systems, such as thermal cameras, has become fundamental for detecting hidden specimens and guiding the robotic manipulator to the exact point of intervention. This multi-modal approach improves overall system efficiency and allows for harvesting or maintenance in low-light environments;
- specialized arm architectures: robotic configurations vary significantly depending on the task, ranging from independent Cartesian arms for high-speed picking to redundant 7-DoF collaborative arms that can dynamically adapt their workspace to the anatomical constraints of the plant.

In summary, the action layer must bridge the gap between abstract autonomous reasoning and physical reality through high-dexterity manipulators and sensitive end-effectors capable of navigating the complex structural architectures of living plants.

2.3 Trade-offs on hardware and software

This section analyzes the technical trade-offs adopted for the experimental setup. Moving from the broad requirements of precision agriculture to a controlled laboratory scale, specific hardware and software choices were made to implement an autonomous closed-loop system for plant watering and manipulation.

2.3.1 Robotic platform and manipulation trade-offs

The primary hardware requirement for interacting with evolving biological specimens is high dexterity and safety. For this case study, a Franka Research 3 robot was selected. This platform is a redundant 7-DoF collaborative arm, which allows for superior flexibility in navigating complex workspaces and adapting to different interaction scenarios.

While the latest advancements in harvesting robots emphasize the use of flexible end-effectors for handling soft tissues, this study focuses on watering and manipulation rather than fruit picking or pruning. Consequently, a standard Franka Hand [16] with parallel fingers was deemed sufficient for handling a cup with modified geometry, optimized for robust robotic grasping.

This trade-off simplifies the mechanical complexity of the end-effector while maintaining the high precision required for targeted resource delivery.

2.3.2 Multimodal sensing and vision strategy

To assess the plant physiological and environmental status, a multimodal sensing suite was integrated, focusing on parameters identified as critical for precision plant care:

- soil and environmental monitoring: the setup includes a capacitive soil moisture sensor to provide data on plant water needs. This is integrated with ambient light, air temperature and humidity sensors to track the growth conditions of the specimen;
- vision system architecture: to overcome the challenges of occlusion and spatial perception in a semi-structured environment, two fixed RGB-D cameras are employed. These sensors provide the color and depth data necessary for identifying the plant and locating the target for intervention.

2.3.3 Software architecture and cognitive integration

The software stack is built on Linux using ROS2, serving as a modular framework for integrating sensing and control. MoveIt is used for trajectory planning and kinematics, ensuring stable movement within the laboratory workspace.

A key technical trade-off in this architecture is the functional split between local perception and the high-level cognitive layer. While the LLM is integrated via Application Programming Interface (API) to act as the primary engine for object recognition and state assessment, it does not process spatial coordinates. Instead:

- semantic assessment: the LLM receives images and raw sensor data to perform non-invasive diagnostics, identifying the "what" and "why" of the required action;
- geometric pose estimation: the precise "where", the geometric localization of poses for the pot and the watering cup, is calculated locally using vision algorithms.

The integration of the LLM via cloud API enables the system to leverage advanced object recognition and logical reasoning capabilities without the need for significant investment in expensive and high-performance edge hardware. This architectural partitioning optimizes system performance by delegating semantic interpretation to the LLM, whereas geometric pose estimation is executed locally to ensure low-latency performance through standard deterministic algorithms. Furthermore, maintaining spatial data processing locally significantly reduces token consumption and avoids the transfer of resource-intensive 3D point clouds to the Cloud.

Through this hybrid framework, the robotic system moves away from traditional manual-based methods toward an intelligent, data-driven management strategy capable of interpreting biological needs with high reliability.

3. Large Language Models for Robotics

The integration of LLMs into robotics represent a fundamental shift, moving the focus to system capable of interpreting and interacting with the physical world. While traditional robotic controllers are often limited by rigid programming, LLMs serve as reasoning engines capable of handling natural language instructions and high-level semantic abstractions. This chapter explores the transitions of these models from text-based interfaces to the cognitive core of physical agents.

3.1 From chatbots to robot controllers

The integration of LLMs into robotics represents a significant expansion beyond the boundaries of natural language processing, transforming these models into reasoning engines for robotic task planning [17]. This evolution marks the transition to text-only processing to "Embodied AI", where the model acts as the cognitive core of a physical agent.

A robotic system, unlike text-only applications, require a deep understanding of real world physics and environmental context. The fundamental advantage lies on the vast "common-sense knowledge" possessed by the LLMs, which allows them to interact with users to interpret and execute commands [18].

Unlike traditional control system, which require a rigid programming for every task or the constant intervention of an expert engineer, the use of LLMs enables a shift towards a "user-on-the-loop" interacting with the LLM through high-level language commands [18].

A practical implementation of this approach is illustrated in [18], where the connection between the model reasoning and the robot physical actions is forged through a high-level function library. By equipping the LLM with a "vocabulary" of descriptive APIs for complex tasks, the system bypasses the need for generating low-level motor commands, which are often impractical for text-based models. This methodology enables the model to interpret user intent directly from natural dialogue and convert it into structured code or a logical sequence of function calls specifically tailored to the scenario.

This paradigm shift overcomes the inherent rigidness of systems that typically require manual reprogramming or extensive retraining to handle even minor perturbations. By leveraging the LLM capacity to decompose abstract instructions into actionable sub-goals while integrating real-time sensory feedback, physical agents gain the necessary agility

to operate within uncertain environments. Consequently, the robot is no longer just a tool following deterministic trajectories, but evolves into an autonomous agent capable of evaluating its interactions and adapting its behavior to the shifting needs of its surroundings [19].

3.2 Methodologies for LLM-driven control

Utilizing LLMs as orchestrators for robotic systems requires specific methodologies to translate natural language into safe physical commands. Current research focuses on different aspects such as optimizing the prompting process, implementing task decomposition using high-level function libraries or generating dynamic code to bridge the gap between reasoning and execution.

3.2.1 Prompt engineering and logical reasoning

Prompt engineering is the fundamental methodology used to guide LLMs behavior and ensure a safe alignment with physical world. An effective prompt must accurately define the system role, the environmental context and the operational constraints. Two of the main prompting techniques are:

- Zero-shot and Few-shot prompting: LLMs demonstrate a great ability to solve tasks without specific examples or following a minimal number of demonstrations, leveraging on their intrinsic knowledge;
- Chain of Thought (CoT): this technique encourages the model to articulate its reasoning process step by step before providing the definitive command. This approach improves accuracy in decomposing complex objectives and reduces the likelihood of logical errors or "hallucinations" within the action plan.

3.2.2 Task decomposition and function libraries

Task decomposition is the fundamental technique for bridging the gap between natural language instructions and low-level physical control. This process consists in subdividing a complex task into a logical sequence of executable primitives creating a hierarchy structure. The model uses its "common-sense knowledge" to interpret user request and select appropriate functions from a library that acts as a vocabulary of the system capabilities. For this reasoning to be effective, APIs must have highly descriptive names, allowing the model to understand functional connections and generate action plans consistent with real-world physics.

An example of this methodology is presented in [18], where the authors formalize a strategy based on a high-level function library acting as a wrapper for existing perception

and planning systems. This framework enables the generation of Python code to solve various tasks interfacing with different simulators and platforms.

3.2.3 Feedback integration and closed-loop control

Integrating sensory feedback and implementing closed-loop control systems are essential to enable LLMs to interact with uncertain physical environments. "Closing the loop" involves transforming perception into a continuous feedback stream that informs the LLM decision-making process, allowing the system to adapt actions and correct trajectories in response to unexpected perturbations. In this architecture, Retrieval-Augmented Generation (RAG) plays a fundamental role because, acting as a dynamic knowledge base, it allows the model to retrieve contextually relevant code examples, motion primitives and safety protocols.

A representative case of this strategy is the ELLMER (Embodied LLM-enabled Robot) framework [19], which integrates the cognitive capabilities of an LLM with a RAG infrastructure to execute complex, long-horizon tasks in unpredictable settings. ELLMER closes the control loop by combining visual data from depth sensors with force feedback from multi-axis transducers. This integration allows the robot to manage physical disturbances in real-time and to perform delicate manipulations by dynamically adapting movement based on received sensory data.

3.2.4 Spatial reasoning and 3D trajectory generation

While LLMs excel at semantic reasoning and task decomposition, they lack of spatial awareness and physical geometry. In standard methodologies the spatial planning is delegated to local controller while the LLM handles the high-level logic. Instead of predicting raw motor commands or low-level trajectories, which require a massive amount of robotic training data, the LLM is tasked with generating mathematical objective functions or "value maps". By understanding the natural language instruction, the model infers what parts of the environment are attractive and what parts are repulsive. These spatial rules are then translated into dense 3D maps that guide standard model-based motion planners to compute zero-shot trajectories.

The VoxPoser [20] framework is an example that this strategy: the LLM is prompted to write Python code that interacts with Vision-Language Models (VLMs) to ground language concepts into the robot visual space. The LLM orchestrates perception calls to obtain spatial-geometric information from RGB-D observations and manipulates 3D arrays to prescribe costs or rewards at relevant locations. Consequently, VoxPoser generates a sequence of 3D value maps that serve as objective functions for a motion planner, effectively translating a set of free-form language instructions into complex, collision-free trajectories.

3.3 Architectural constrains: cloud vs edge AI

In the deployment of autonomous robotic agents, the choice of computational infrastructure represents a critical architectural constraint, typically divided between Cloud AI and Edge AI paradigms. The Cloud-based approach offers nearly unlimited computational and memory resources, which are ideal for processing massive and complex models; however, it introduces significant challenges related to network latency, connection reliability and data privacy. On the other hand, Edge AI aims to shift intelligence directly aboard the physical device. This paradigm resolves latency and bandwidth dependency issues, enabling real-time deterministic control and secure offline operations.

However, Edge processing is severely restricted by hardware constraints like reduced computational capacity, limited physical memory and a strict power envelope. To cope with these limitations and implement AI on embedded devices, the literature proposes various optimization techniques such as model compression, pruning and quantization, which specifically reduces the numerical precision of model weights, to drastically decrease memory footprint and accelerate inference times [21].

These computational limitations become particularly severe when attempting to execute LLMs entirely locally. As detailed in recent evaluations of edge computing performance [22], these massive architectures are heavily constrained by memory bandwidth limitations rather than sheer computational capacity. Consequently, running models containing billions of parameters on modern embedded platforms requires aggressive weight quantization techniques to ensure the system fits within the restricted physical memory while maintaining energy consumption below operational thresholds. Despite the application of these advanced software and hardware optimizations, the compromise among inference speed, power expenditure and the degradation of cognitive accuracy caused by compression remains a significant hurdle preventing the exclusive reliance on edge artificial intelligence for advanced reasoning tasks.

In light of these architectural constraints, the most effective robotic systems adopt a hybrid partitioning approach delegating high-level semantic interpretation and logical reasoning to Cloud APIs (such as Gemini or ChatGPT). Simultaneously, deterministic tasks requiring low latency, such as geometric pose estimation via local cameras and manipulator kinematics, are processed on the Edge. This functional division optimizes performance avoiding transmitting heavy 3D point clouds over the network and leveraging local infrastructure for safe and real-time execution, bypassing the computational bottlenecks of language models on the Edge.

4. System Design and Architecture

This chapter details the system architecture design and development to implement the SDL framework for autonomous plant care. The primary objective of the project is to create a system capable of bridging the gap between high-level reasoning and low-level robotic manipulation.

The system is designed following a hybrid Cloud-Edge paradigm: complex cognitive decisions and context analysis are delegated to a Cloud API powered by a LLM while geometric perception, safety and real-time control are managed locally on the Edge. This functional division enhances the performance of the system because processing the data in the edge and using the LLM only for high-level decisions reduce the latency, reduce the costs linked to token usage and avoid AI hallucination during heavy data processing.

To manage these different functions and ensure a high modularity, the entire framework is based on ROS2. The use of this middleware guarantees inter-process communication and, most importantly, a clear hardware abstraction: the logical architecture can drive both a digital twin in a simulated environment and a robotic arm in the real world.

This chapter is structured following the interpretation of the robotic Sense-Think-Act paradigm which provides a natural and modular framework for autonomous operations. After an initial overview of the system architecture and its corresponding ROS2 implementation, the discussion moves through the three core phases of the control loop. It first introduces the perception layer (Sense) responsible for environmental and phenotypic data acquisition, followed by the cognitive layer (Think) where the LLM process these multimodal data to generate semantic action plans. Finally, the control and motion layer (Act) describes the translation of such directives into safe and executable trajectories.

4.1 System architecture overview

The system architecture is designed as a modular, distributed framework based on ROS2. To effectively manage the asynchronous nature of cloud communications and the strict real-time requirements of robotic hardware, the system is logically partitioned into three interconnected layers. This decoupling ensures that high-level semantic reasoning does not interfere with the low-level kinematic control loops.

The overall data flow and the interactions between the main hardware and software components are illustrated in figure 4.1.

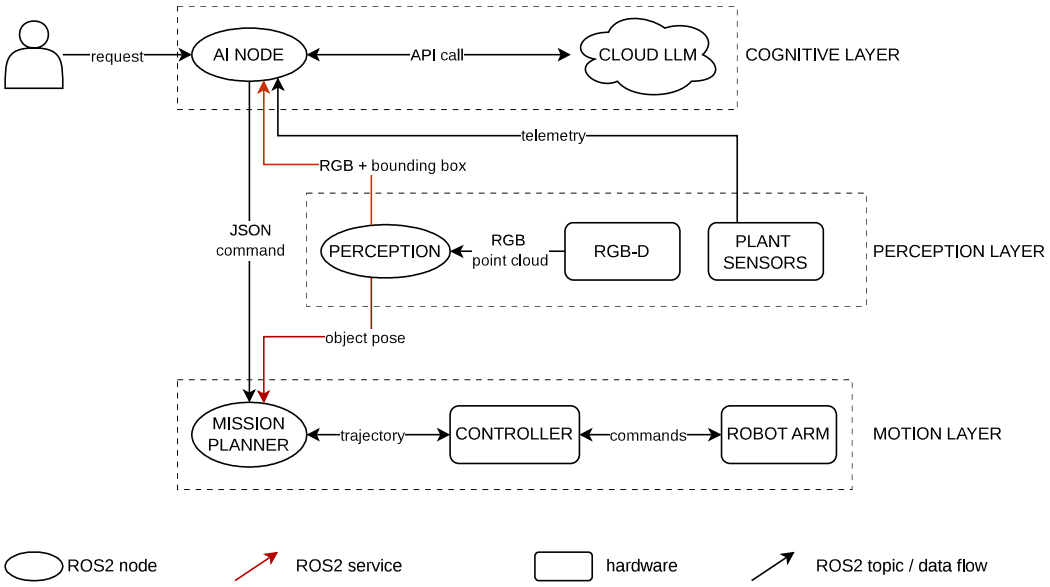


Figure 4.1: High-level system architecture outlining the Cognitive, Perception and Motion layers, along with the ROS 2 data flow

As illustrated in the diagram, the framework operate through the continuous interaction of three fundamental layers:

- perception layer (Sense): this layer acts as the sensory interface of the system. It gathers data from the physical world, including telemetry from the plant sensors and visual and depth information from RGB-D camera. To optimize resource usage, these raw data are made available to the upper layers via both continuous streams (topics for telemetry) and on-demand ROS2 services (for single image acquisition). Furthermore, the point clouds are processed by the *perception_node* to compute exact object poses in the 3D space;

- cognitive layer (Think): serving as the core decision-making unit of the system, this layer contains the *ai_agent_node* which acts as a bridge between the local robotic network and the Cloud LLM. Triggered by user request, it aggregates a set of system instructions, the requested visual data and the current plant telemetry to query the generative AI. The LLM is used both for decision-making and 2D object identification. The output, provided in a structured JSON format, is then parsed and dispatched to the motion layer;

- motion layer (Act): this layer translates semantic intent generated by the AI into physical robotic actions. The *mission_planner_node* receives the parsed JSON, containing the action required and the object recognition results, and, when necessary, requests the exact 3D object poses from the perception layer via a ROS2 service.

Once the objects are localized, the node leverages MoveIt to compute collision-free trajectories and dispatches the execution commands to the low-level robot controllers.

A key architectural choice is the distinction in communication paradigms. Continuous data streams, such as camera frames and telemetry, are broadcasted asynchronously using ROS2 topics to avoid blocking the system. Conversely, event-driven interactions, such as the planner requesting the exact 3D pose of an object, are implemented using ROS2 services, ensuring synchronous execution and reliability only when a specific task is active.

To better clarify the temporal execution of a robotic task, the operational workflow is illustrated in figure 4.2. This diagram outlines the sequential logic of the Sense-Think-Act loop, highlighting how the asynchronous requests to the Cloud LLM are synchronized with the deterministic requirements of the local perception and motion pipelines.

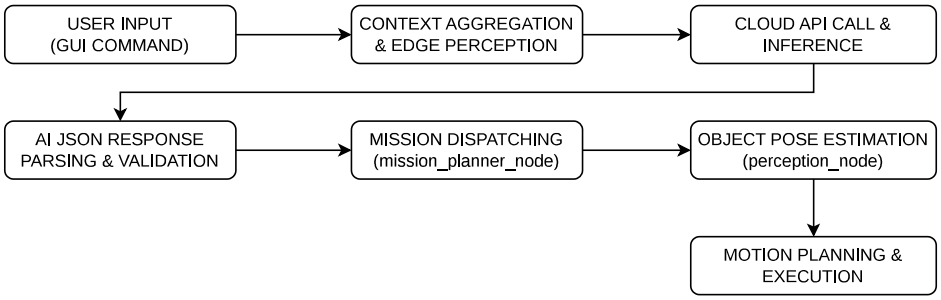


Figure 4.2: Logical workflow of the SDL framework

4.2 The perception layer: sensors and data handling

The perception layer is designed to be highly modular and hardware-agnostic. By leveraging ROS2 abstraction capabilities, this layer provides a standardized interface between the physical environment and the higher-level cognitive and motion layers. Whether the data originates from a simulated environment in CoppeliaSim or from physical sensors in a real-world laboratory, the data structures and communication protocols remain identical.

The perception pipeline is divided into two primary subsystems: the semantic vision unit and the telemetry data bridge.

4.2.1 Vision subsystem and context-aware perception

The vision subsystem manages the acquisition and processing of multimodal visual data. The *perception_node* subscribes to standard ROS2 topics to receive RGB frames (via `sensor_msgs/Image`) and point clouds (via `sensor_msgs/PointCloud2`) from up to two different camera perspectives, allowing for multi-view aggregation. The point cloud the node uses is the one filtered by MoveIt: knowing the configuration and the status of the robot arm, it deletes from the raw point cloud the data of the robot structure.

To optimize computational resources and network bandwidth, the system avoids continuous processing of heavy point clouds. Instead, the perception architecture implements a deterministic, two-phase service-oriented approach.

Phase 1: scene processing and AI input generation

Triggered via the `/process_ai_scene` service, the *perception_node* aggregates the active point clouds and applies a Random Sample Consensus (RANSAC) algorithm to identify and remove the planar surface of the table.

```
# RANSAC Table Segmentation & Smart Cut
pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(xyz_clean)
plane_model, inliers = pcd.segment_plane(distance_threshold=self.ransac_dist_thresh,
↪  ransac_n=3, num_iterations=1000)
pcd_table = pcd.select_by_index(inliers)
pcd_objects = pcd.select_by_index(inliers, invert=True)
pts_table = np.asarray(pcd_table.points)
pts_candidates = np.asarray(pcd_objects.points)
if len(pts_table) > 0:
    z_table_mean = np.mean(pts_table[:, 2])
    mask_objects = pts_candidates[:, 2] > (z_table_mean + self.table_z_margin)
    final_pts = pts_candidates[mask_objects]
else:
    self.get_logger().warn("Table not found by RANSAC! Using all remaining points.")
    final_pts = pts_candidates
```

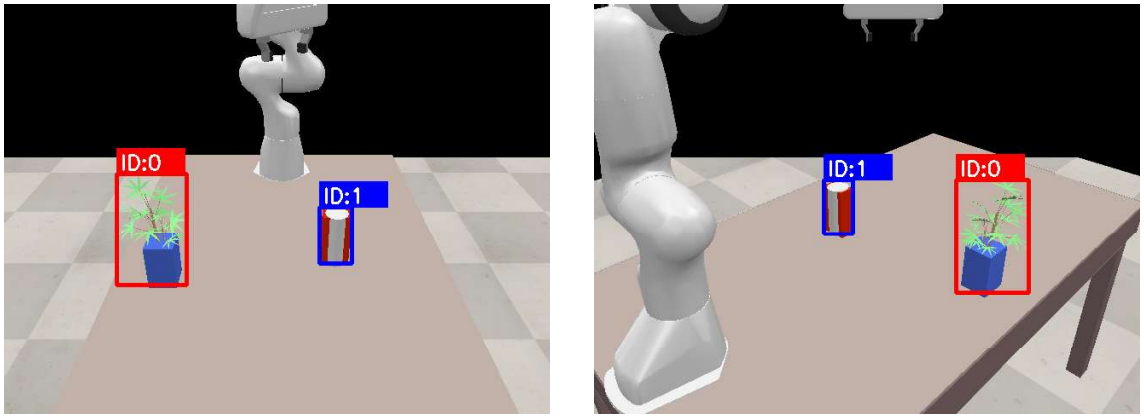
Listing 1: RANSAC table filtering algorithm

The remaining spatial data is processed using the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm to isolate individual physical objects into distinct point clusters.

```
# DBSCAN clustering
pcd_final = o3d.geometry.PointCloud()
pcd_final.points = o3d.utility.Vector3dVector(final_pts)
labels = np.array(pcd_final.cluster_dbscan(eps=self.dbscan_eps,
↪  min_points=self.dbscan_min_points))
...
# Memorization of clusters
self.memorized_clusters.clear()
```

Listing 2: DBSCAN clustering of remaining points

These 3D clusters are mapped to unique numerical IDs and saved in the node local memory. Simultaneously, the node performs a 3D-to-2D perspective projection using the camera intrinsic and extrinsic matrices to draw bounding boxes and ID labels directly onto the captured RGB frames. These annotated images are then saved to a local directory, providing the Cloud LLM with a spatially quantified visual context. The figure 4.3 shows an example of the results of the process in the simulated environment.



(a) View from Camera 1 (b) View from Camera 2

Figure 4.3: Perspective projection of the 3D point cloud clusters onto the 2D RGB frames

Phase 2: Context-aware pose estimation

Once the LLM has analyzed the annotated images, it outputs the target object ID, its semantic label, its category (e.g., "plant" or "vessel") and its physical state (e.g., "standing" or "laid"). The *mission_planner_node* forwards these parameters to the *perception_node* via the `/GetClusterPose` service.

Instead of treating all objects as simple geometric primitives, the node retrieves the corresponding 3D cluster from its memory and dynamically selects a specific geometric estimation algorithm based on the AI semantic category, as shown in listing 3.

```
# Algorithm selection based on label and state
if category_lower == "plant":
    if state_lower == "standing":
        pose_msg = self.estimate_standing_plant(pts, req_id, req_label)
    else:
        pose_msg = self.estimate_laid_plant(pts, req_id, req_label)

elif category_lower == "vessel":
    if state_lower == "standing":
        pose_msg = self.estimate_standing_cup(pts, req_id, req_label)
    else:
        pose_msg = self.estimate_laid_cup(pts, req_id, req_label)

else:
    # Fallback generic estimation for unknown categories
    pose_msg = self.estimate_generic_pose(pts, req_id, req_label)
```

Listing 3: Algorithm selection based on LLM semantic category

To ensure safe and reliable robotic manipulation, the pose estimation logic is highly specialized based on the object semantic category and physical state. Computing the grasping pose for a standing plant, for instance, presents a unique challenge: the robotic gripper must interact exclusively with the rigid pot, avoiding the fragile canopy.

To achieve this, the `estimate_standing_plant` algorithm implements a Z-axis slicing

technique. The 3D point cluster is sectioned horizontally into thin layers and the algorithm dynamically evaluates the cross-sectional area and point density of each slice. A sudden variation in these metrics indicates the morphological transition from the rigid pot to the leaves. Once the subset of points belonging to the pot is isolated, the algorithm computes a safe Z-grasping height. The pot points are then projected onto the XY plane, where a Minimum Area Rectangle is fitted over their convex hull to extract the precise planar centroid, the bounding dimensions and the primary rotation angle required to align the end-effector.

Conversely, for objects like a standing cup or a generic rigid cylinder, the geometry is more uniform. The `estimate_standing_cup` algorithm focuses on filtering out residual table noise at the base and calculates the volumetric Z-center, applying a predefined safety offset to prevent the gripper from colliding with the work surface. The planar coordinates and orientation are similarly derived using the convex hull of the projected XY points.

Regardless of the specific geometric algorithm executed, the computed physical parameters are structured and returned via a custom ROS2 message interface, defined as `ObjectPose.msg`. This standardized payload, shown in listing 4, encapsulates all the spatial data required by the *mission_planner_node* to orchestrate the physical manipulation: the 3D centroid coordinates, the primary axis orientation for end-effector alignment and the bounding box dimensions.

```
int32 id
string label
float64 x
float64 y
float64 z
float64[] main_axis_vector
float64 rotation_angle_deg
float64 long_side
float64 short_side
float64 height
```

Listing 4: Custom interface used to transmit geometric data

4.2.2 Plant telemetry data handling

To monitor the biological status of the plant, the system gathers key environmental and soil parameters. Instead of interacting directly with raw electrical signals within the core robotic framework, a dedicated bridge node (*arduino_bridge_node*) handles the low-level serial communication with the microcontroller.

This node continuously reads the incoming serial data stream, validates its structure to prevent corrupted readings and encapsulates the payload into a standard `std_msgs/String` containing a formatted JSON object. The validated data is then published at a high frequency (e.g., 10 Hz) to the `/hardware/sensors_data` topic. This continuous, lightweight stream ensures that the latest plant conditions are always available in the background,

ready to be sampled by the cognitive layer when a diagnostic task is triggered.

4.3 The cognitive layer (AI)

The cognitive layer is the decision-making unit of the architecture. Traditional robotic systems rely on pre-programmed state machines or hard-coded decision trees, which struggle to adapt to unstructured environments or ambiguous human requests. To overcome these limitations, this layer delegates high-level reasoning to a Cloud-based LLM (Google Gemini 2.5 Flash), accessed via an API call.

In this architecture, the generative model acts as the "brain" of the system and is entrusted with three primary cognitive tasks:

- semantic perception: identifying the objects within the workspace, assigning them to strict categories (e.g., `plant`, `vessel` or `obstacle`), and inferring their physical state (e.g., `standing` or `fallen`);
- contextual evaluation: reading the hardware telemetry to determine the biological needs of the plant;
- task assignment: translating a natural language user command into a strict, executable robotic mission, identifying object and scanning the scene for anomalies.

The integration between the ROS2 environment and the generative AI is managed by the *ai_agent_node* which implements a `MultiThreadedExecutor` to ensure non-blocking operations during API call. The node acts as a bidirectional bridge orchestrating the data gathering, the prompt generation and the AI response parsing. The cognitive pipeline is triggered by asynchronous human request and unfold through three main execution steps.

4.3.1 Multimodal context aggregation

When the user issues a command, the *ai_agent_node* cannot rely only on the text prompt, it must provide to the LLM a comprehensive and updated understanding of the current workspace. To achieve this, it gathers two types of data:

- environmental data: the node maintains an internal state variable constantly updated via a subscriber to the `/hardware/sensors_data` topic. When a request is triggered, it extracts the last data containing real-time soil moisture and ambient parameters;
- visual context: the node acts as a ROS2 client, calling the `/process_ai_scene` service, described in the perception layer. This triggers the *perception_node* to perform the point cloud processing and return the file paths of the annotated RGB frames (figure 4.3) where each object is bounded and marked with a unique numerical ID.

4.3.2 Prompt engineering and structured querying

Once the multimodal context is gathered, the *ai_agent_node* composes a highly structured query to be sent to the Cloud LLM. To ensure the AI to have a complete set of information the prompt is rigidly engineered into distinct components using the Google GenAI SDK.

System instructions

The first component is a static set of rules defining the AI role, its available tools and its operational constraints. As shown in listing 5, the model is instructed to act as the "brain" of the robotic arm and is forced to classify objects and user intents into specific, predefined semantic categories.

```
sys_instruction_text = """ROLE & OBJECTIVE
    You are the Cognitive Reasoning Agent (the "Brain") for a Franka Emika Panda robotic arm in a self-driving lab.
    Your low-level geometry and kinematics are handled by a separate local subsystem.
    Your sole responsibility is semantic perception: analyzing hardware sensor telemetry, understanding the visual scene and dispatching high-level missions.
    You can act on explicit user commands OR autonomously decide the best action based on the plant's needs.
    INPUT DATA
    1. RGB images: the current visual scene. Detected objects are highlighted with bounding boxes and an integer ID.
    2. Context message: a text containing both the [SENSOR TELEMETRY] and the USER COMMAND.
    DIRECTIVES
    1. Object identification: analyze the images. ONLY list objects that have a visible bounding box and an integer ID. Do not invent objects.
    - CLASSIFICATION: You MUST classify each detected object into one of three strict categories:
      * "plant": for potted plants, flowers, or plant pots containing soil.
      * "vessel": for cups, beakers, or open containers.
      * "obstacle": for ANY other unexpected item or unknown shapes.
    2. Autonomy and telemetry evaluation: If the user gives a direct explicit command (e.g., "water the plant"), you want to execute it. If the user gives a generic prompt, AUTONOMOUSLY choose the mission. HOWEVER, this is always subject to the Safety Override below.
    3. Mission assignment: choose strictly between: "watering", "moving", or "info".
    4. Target ID Selection: The `target_id` MUST be the exact integer ID of the object to grasp first.
    - If "watering", the target is the vessel.
    - If "moving", the target is the objects specified by the user.
    - If "info", use -1.
    5. CRITICAL SAFETY OVERRIDE (Anomaly Detection):
    - If ANY object is classified as an "obstacle", or if a plant/vessel is "fallen", this is an ANOMALY.
    - If an anomaly is present, you MUST populate the "anomalies" JSON field describing the issue.
    - IF AN ANOMALY IS PRESENT, YOU CANNOT EXECUTE A "WATERING" OR "MOVING" MISSION. You MUST override the user's command, default the mission to "info", and explain in the reasoning that the workspace is unsafe.
    - The ONLY exception to this block is if the user explicitly commands you to "move" the specific obstacle (in which case, mission is "moving").
    TARGET DESTINATION (EXTRACTION ONLY):
    - You DO NOT have spatial awareness. Do not invent coordinates.
    - IF the user EXPLICITLY provides X and Y coordinates in their command (e.g., "move to x=0.4, y=0.1"), extract them and output the exact string "[X, Y]".
    - IF the mission is "moving" but NO coordinates are provided, output exactly "safe_zone".
    - IF the mission is "watering" or "info", output "none".
    CONSTRAINTS & TONE
    - Language: English only.
    - Conversational reasoning: in the 'reasoning' field, act as a helpful robotic assistant. Explain to the user what you see, what the sensors say and what autonomous or commanded action you are taking. Avoid deep technical jargon.
    - Safety First: If a command is dangerous or ambiguous, default to the "info" mission and explain why."""
```

Listing 5: The system instructions defining the cognitive constraints and behavioral rules for the LLM

Dynamic multimodal input

The second component of the prompt is the dynamic input containing the real-time state of the system. The user's raw text command is concatenated with the serialized string of the plant telemetry. This string is then packed together with the annotated RGB frames and send to the Cloud LLM through API call.

Structured JSON output

A crucial architectural choice at this stage is the enforcement of a strict JSON schema response. Because the subsequent motion layer requires deterministic inputs to plan the kinematics, the system cannot risk receiving conversational text, Markdown formatting or unpredictable structures from the generative model.

Through the API configuration, the LLM is programmatically constrained to reply exclusively with a predefined JSON object. As detailed in listing 6, this schema enforces required keys, specific data types and restricts categorical fields.

```
response_schema = types.Schema(type=types.Type.OBJECT,
        required=["reasoning", "mission", "target_id", "target_destination", "anomalies", "objects"],
        properties={
            "reasoning": types.Schema(type=types.Type.STRING,
                description="A polite, natural-language explanation directed at the user. Describe what you see, interpret the plant's health from the sensors and clearly state what you are about to do and why. Be helpful and conversational, avoiding technical jargon."),
            "mission": types.Schema(type=types.Type.STRING,
                description="The mission chosen based on the command.",
                enum=["watering", "moving", "info"]),
            "target_id": types.Schema(type=types.Type.INTEGER,
                description="Integer ID of the main object. Use -1 if the mission is 'info' or not applicable."),
            "target_destination": types.Schema(type=types.Type.STRING,
                description="Extracted coordinates as string '[X, Y]' or 'safe_zone' or 'none'."),
            "anomalies": types.Schema(type=types.Type.STRING,
                description="Describe obstacles or write 'none'."),
            "objects": types.Schema(type=types.Type.ARRAY,
                description="List of detected objects.",
                items=types.Schema(
                    type=types.Type.OBJECT,
                    required=["id", "label", "category", "state"],
                    properties={
                        "id": types.Schema(type=types.Type.INTEGER),
                        "label": types.Schema(type=types.Type.STRING,
                        description="Free text description, 2 words max."),
                        "category": types.Schema(type=types.Type.STRING,
                            description="The strict semantic category.",
                            enum=["plant", "vessel","obstacle"]),
                        "state": types.Schema(type=types.Type.STRING, enum=["standing", "fallen"])
                    })})})
```

Listing 6: The JSON schema imposed on the Gemini API to guarantee deterministic output parsing

This schema forces the AI to output a machine-readable payload, translating natural language reasoning into deterministic parameters. Each field serves a specific architectural purpose:

- `reasoning`: before outputting any command the model is forced to generate a textual explanation of its thought process;
- `mission`: the LLM must classify the user's intent by choosing exclusively from a predefined enum array: `watering`, `moving` or `info`, used for generic questions or safe fallbacks;
- `target_id` and `target_destination`: these fields extract the exact numerical ID of the object to be manipulated and the spatial coordinates, if provided, feeding the required geometric parameters to the motion layer,
- `anomalies`: a dedicated string where the AI acts as a safety monitor, reporting any unexpected or hazardous elements in the workspace;
- `objects`: an array that enriches the numerical IDs provided by the perception layer with human-readable labels, bounding their semantics to strictly supported categories to prevent grasping errors.

4.3.3 Output parsing and safety check

Upon receiving the generated output, the *ai_agent_node* decodes the JSON payload and, before forwarding any commands to the *mission_planner_node*, it acts as a high-level safety supervisor.

At first, it evaluates the `anomalies` field and if the AI detects an unexpected obstacle or a hazardous workspace configuration, a safety override protocol is automatically triggered. Regardless of the user's initial request, any physical execution intent (such as `watering` or `moving`) is downgraded to an `info` mission. This intervention blocks the robot from moving and logs a warning to the system.

If the environment is deemed safe, the node filters the output, generating a streamlined executive JSON. This final payload discards the verbose textual reasoning and retains only the deterministic parameters required for planning: the selected mission, the target ID, the target destination for moving missions and the semantic object list. This executive command is then published on a dedicated ROS2 topic (`/ai/json_output`), ready to be intercepted by the motion layer.

To illustrate this process, listing 7 shows the core parsing and filtering logic implemented within the *ai_agent_node*. The code evaluates the presence of anomalies and, if necessary, overrides the assigned mission to prevent hazardous behaviors. Subsequently, a streamlined dictionary is constructed, stripping away the AI's verbose textual reasoning, serialized into a strict JSON string and published to the motion layer.

```
mission = ai_data.get("mission", "info")
    # Anomalies Handling - Safety override
    anomalies = ai_data.get("anomalies", "none").lower()
    if anomalies != "none":
        self.get_logger().warn(f" SCENE ANOMALY DETECTED: {anomalies}")
        # Block movement
        if mission in ["watering", "moving"]:
            self.get_logger().warn("SAFETY OVERRIDE: mission changed to info.")
            mission = "info"
    # Reconstruct a "clean" JSON with only the executive data for MoveIt
    planner_payload = {
        "mission": mission,
        "target_id": ai_data.get("target_id"),
        "target_destination": ai_data.get("target_destination", "safe_zone"),
        "objects": ai_data.get("objects", [])
    }
    msg_out = String()
    msg_out.data = json.dumps(planner_payload)
    self.json_pub.publish(msg_out)
```

Listing 7: The safety override logic and the construction of the streamlined executive payload dispatched to the motion layer

Before evaluating the semantic payload for robotic execution, the *ai_agent_node* performs a data logging operation. The full cognitive output, including the initial user command, the aggregated hardware telemetry and the complete AI JSON response, is automatically serialized and saved locally as a discrete log file. This logging mechanism ensures a persistent and structured trace of the LLM decision-making process, which is essential for offline debugging, system diagnostics and evaluating the autonomous performance of the model over time.

4.3.4 System transparency and Graphic User Interface

To monitor this complex cognitive pipeline and facilitate human-robot interaction, a custom GUI was developed. While the data aggregation and API querying operate transparently in the background, the interface serves as a front-end tool to expose the "black-box" decision-making process of the generative model.

From a software architecture perspective, the GUI runs as a completely independent ROS2 node (*ai_gui_node*), providing a chat terminal where the user can easily type natural language requests. The node captures this input and publishes it to the `/user_command` topic, effectively triggering the *ai_agent_node*.

Simultaneously, the interface subscribes to the `/ai/gui_output` topic. Once the Cloud LLM finishes processing the multimodal prompt, the interface intercepts the serialized JSON response and performs a front-end specific parsing. To maximize human readability, it extracts the `reasoning` string and displays it as a natural, conversational chat message while the remaining parameters are cleanly formatted and confined within a dedicated "DEGUB INFO" section. This dual-display approach allows the human operator to intuitively understand the AI intentions while verifying the deterministic parameters before the robot

begins any physical execution, guaranteeing both user-friendliness and operational safety.

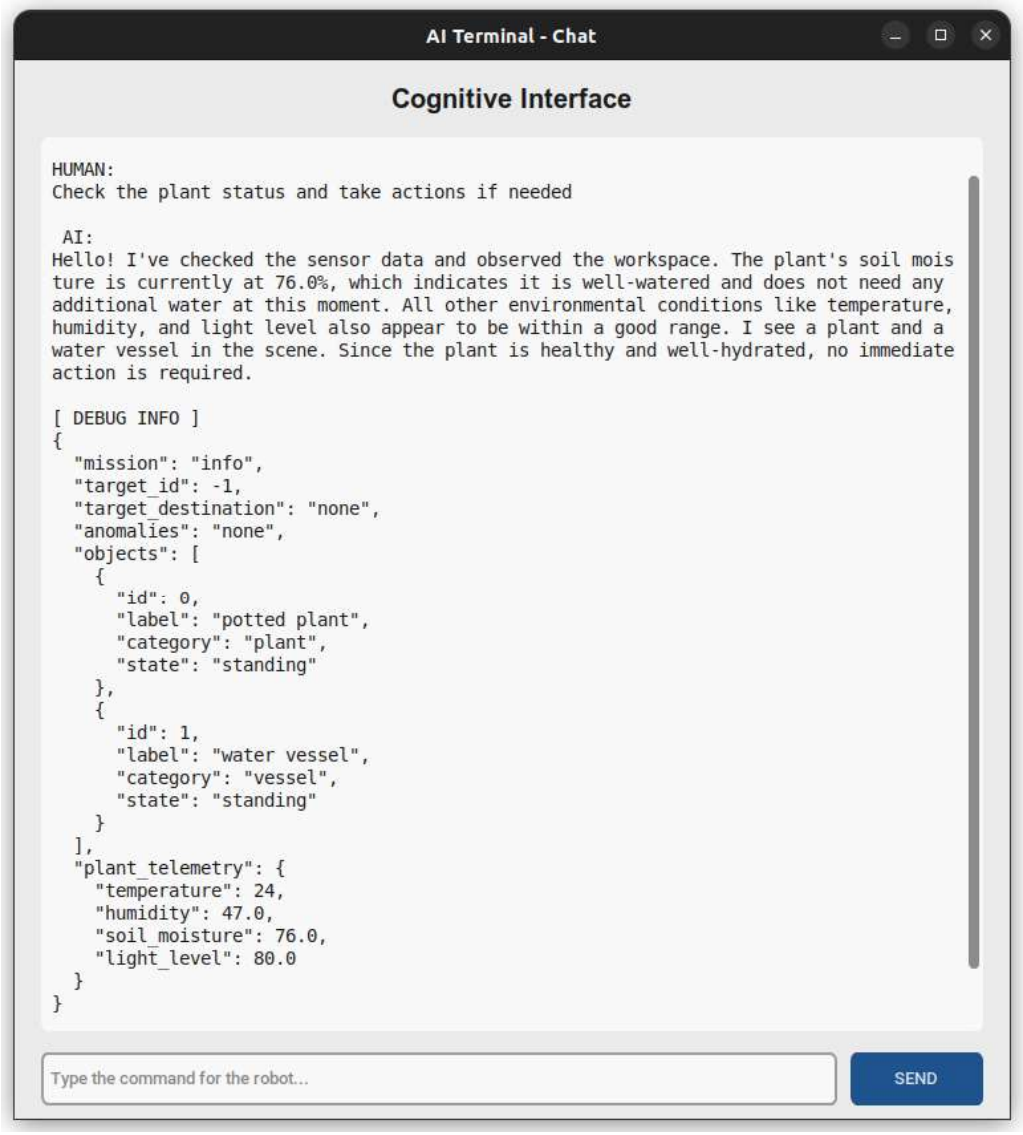


Figure 4.4: Example of the GUI displaying the AI response to a low soil moisture condition

4.4 The motion layer

The motion layer represents the lowest tier of the proposed architecture. Its primary responsibility is translating the high-level semantic directives generated by the AI into collision-free trajectories and robotic tasks. This layer acts as the bridge between the cognitive reasoning and the physical hardware relying on MoveIt framework, implemented through its python API [23], used to solve inverse kinematics and motion planning problems.

The core component of this layer is the *mission_planner_node*. Unlike traditional robotic state machines, this node does not follow rigidly hard-coded paths, but it dynamically computes grasp poses, approach vectors and pouring orientations based on the

real-time geometric data retrieved from the perception layer. To ensure smooth, non-blocking execution while waiting for lengthy service responses or action servers, the node is implemented using a `MultiThreadedExecutor`. The operational pipeline is modularly structured, unfolding through data parsing, dynamic scene reconstruction, motion planning and sequential task execution.

4.4.1 Mission dispatching and dynamic scene reconstruction

The kinematic pipeline starts when the *mission_planner_node* receives the streamlined JSON from the cognitive layer (listing 7) via the dedicated `/ai/json_output` topic. The node immediately deserializes this message to extract the core executive parameters: the specific `mission` type, the `target_id` representing the primary object to manipulate, the `target_destination` coordinates if a relocation is requested and the `objects` info. This data extraction phase acts as the primary mission dispatcher, setting the internal state of the orchestrator to route the subsequent kinematic sequence appropriately.

However, while this payload contains the semantic intent and the object identification results, it lacks the spatial awareness needed to plan and move the robotic arm. To obtain these data the node acts as a ROS2 client and calls the `/GetClusterPose` service exposed in the perception layer. By transmitting the specific numerical IDs of the semantic targets, the planner queries the vision system to retrieve their exact 3D coordinates, bounding box dimensions and the orientation of their primary spatial axes onto XY plane.

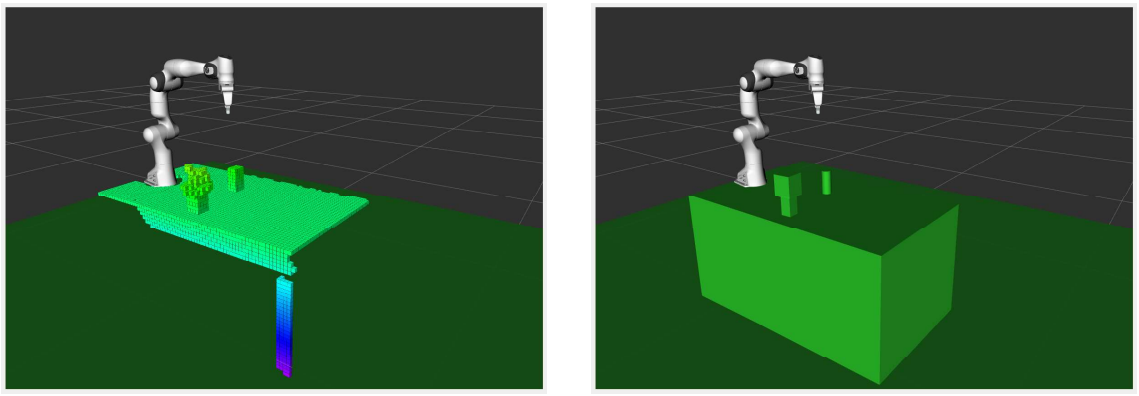
Once the spatial data is acquired, the system must translate the unstructured physical environment into a mathematical collision model that MoveIt can safely navigate. This critical task is handled by a dedicated `SceneManager` module. The manager reconstructs the MoveIt planning scene by spawning primitive collision objects that match the estimated sizes, locations and yaw orientations of the real physical items. The work surface is also geometrically instantiated as a static collision box to prevent the end-effector from crashing into the table during object grasping.

In addition to the discrete semantic objects, the robot must be aware of the unstructured surrounding environment to guarantee completely collision-free trajectories. MoveIt incorporates a 3D perception pipeline that processes the raw point cloud data and maintains a volumetric representation of the workspace using an `octomap`, a 3D occupancy grid based on voxels.

To integrate this voxel grid efficiently with the kinematic planner, the framework performs real-time self-filtering, removing the points belonging to the robot own geometry to prevent self-collision false positives. Furthermore, when the `SceneManager` spawns a known collision object, its volume is inherently considered by the planner, allowing the system to distinguish between a manipulable target and an obstacle.

Building this volumetric map relies on the fusion of point clouds originating from two distinct RGB-D sensors, ensuring a multi-view perception that mitigates blind spots and

object occlusions. However, continuously fusing dense point clouds from multiple sources at high frequencies introduces a severe computational bottleneck. To guarantee real-time performance and prevent CPU saturation, an Octomap throttling strategy is implemented at the launch configuration level. The integration pipeline sub-samples the incoming point clouds and limits the maximum volumetric update rate. This optimization provides an ideal compromise, maintaining a responsive spatial awareness without compromising the computational stability of the ROS2 network.



(a) Raw volumetric representation

(b) Reconstructed collision scene

Figure 4.5: Dynamic scene reconstruction in RViz. Difference between raw volumetric representation and primitive collision objects

The specific geometric shape of each object is deterministically assigned based on the semantic category provided by the cognitive layer:

- vessels: modeled as uniform collision cylinders using predefined dimensions;
- plants: modeled using a composite, predefined geometry consisting of two stacked bounding boxes. This custom shape is designed to safely encapsulate both the rigid pot at the base and the fragile canopy extending above it;
- obstacles: generic unknown items are instantiated as simple rectangular bounding boxes, sized using the exact geometric dimensions (length, width, and height) extracted from the point cloud clustering.

To successfully manipulate these generated objects, the system must balance general obstacle avoidance with the need for close-proximity interaction. Assigning a global safety margin to the entire robot would cause the motion planner to detect false collisions between the gripper and the target object before the physical grasp could even occur. To solve this, the `SceneManager` implements a "Hybrid Collision Padding" strategy: it assigns a conservative collision padding to the main kinematic links of the robotic arm to ensure safe navigation across the workspace, while applying a near-zero margin to the end-effector links, allowing the fingers to safely approach and enclose the target geometry.

In a dynamic manipulation scenario, physically moving an object leaves a crucial challenge: sensor latency or occlusions can cause the old voxels of the grasped item to remain in the `octomap`, creating "ghost obstacles" that block future trajectory planning. To overcome this, the *mission_planner_node* implements a strict voxel-reset policy. Before any critical motion planning phase, it actively calls the `/clear_octomap` service. This effectively deletes the volumetric grid, forcing the framework to rebuild the spatial occupancy from the latest sensor data, ensuring that the calculated paths are based on the updated state of the workspace.

4.4.2 Motion planning strategies

The *mission_planner_node* relies on a dynamically selected dual-pipeline motion planning strategy. Depending on the specific kinematic requirements of the task phase, the system alternates between algorithmic approaches to guarantee both reachability and precision.

For standard Point-To-Point (PTP) transfers across the workspace, the system utilizes the Open Motion Planning Library (OMPL), a pipeline that excels at finding complex, non-linear and collision-free paths, making it the primary choice for navigating between distant operational zones. Under the hood, the OMPL pipeline is configured to use the `RRTConnectkConfigDefault` algorithm. This bidirectional Rapidly-exploring Random Tree approach is particularly efficient in high-dimensional configuration spaces, ensuring fast computing convergence when planning the global reach-and-avoid motions of the manipulator.

However, global PTP movements are unsuitable for the delicate phases of physical interaction. During the final approach to grasp an object or the vertical lift required to detach it from the work surface, the end-effector trajectory must be strictly rectilinear to prevent lateral collisions with the target or the accidental spilling of its contents. To satisfy this requirement, the planner switches to the Pilz Industrial Motion Planner, specifically using its linear (`LIN`) algorithm. The Pilz pipeline computes deterministic, straight-line paths and, to enhance operational safety during these critical insertions and retreats, the planner applies tighter path constraints, operating at significantly reduced velocity and acceleration scaling factors.

Fluid manipulation demands strict postural control throughout the entire transit phase. If an open vessel is moved using unconstrained PTP planning, the inverse kinematics solver might rotate the end-effector to optimize the joint-space trajectory causing the fluid to spill. To prevent this, the node leverages MoveIt `OrientationConstraint` interface during transit missions. The planner mathematically locks the absolute angular tolerances on specific axes, typically restricting the roll and pitch of the end-effector, while allowing the necessary positional freedom. This forces the OMPL planner to generate a trajectory that maintains the grasped object perfectly upright from the pick location to the final destination, fully securing the integrity of the payload.

Specifically, since the manipulator employs a side-grasping approach, strict absolute tolerances are imposed on the global X and Y axes (local Z and Y axes) to prevent any tilting of the payload. Conversely, the rotation around the global Z axis (local X axis) is left completely unconstrained, providing the inverse kinematics solver with the necessary spatial flexibility to successfully compute a valid joint-space solution. The exact `OrientationConstraint` parameters are detailed in Listing 8.

```
# Constrain configuration
constraint = OrientationConstraint()
constraint.header.frame_id = frame
constraint.link_name = self.tcp_link
# Keep the same orientation as the  current pose to maintain gripper's X-axis consistent
↪  with the grasp
current_pose = self.get_current_tcp_pose()
...
constraint.orientation = current_pose.orientation
# Z and Y axis tollerances (tight)
constraint.absolute_z_axis_tolerance = 0.35 # 20°
constraint.absolute_y_axis_tolerance = 0.35
# X axis tollerance (free)
constraint.absolute_x_axis_tolerance = 3.14
constraint.weight = 1.0 # max importance
constraints_msg = Constraints()
constraints_msg.orientation_constraints.append(constraint)
self.panda_arm.set_path_constraints(constraints_msg)
```

Listing 8: Orientation constrains settings

4.4.3 Kinematic logic and motion primitives

The motion planning and execution are decoupled into modular components so, rather than computing single trajectories for a given task, the *mission_planner_node* relies on a structured hierarchy of functions: geometric pose generation, elementary motion abstractions and aggregate manipulation primitives. This separation of concerns allows for high code reusability and significantly improves fault tolerance during physical interactions.

Geometric pose generation

Before invoking any motion planner, the system computes the exact 6-DoF target frames based on the spatial data provided by the perception layer. The geometric functions rely on linear algebra to transform 3D centroids and main axes into executable poses for the end-effector.

To calculate a valid grasping orientation, the system extracts the object primary directional vector on the XY plane. By computing the cross products between this vector and the global vertical axis, the algorithm constructs an orthonormal basis localized for the gripper. This rotation matrix is subsequently converted into a quaternion, ensuring the end-effector approaches the object orthogonally to its longest dimension.

```python
def calculate_grasp_orientation(self, main_axis_vector):
    y_axis = np.array([main_axis_vector[0], main_axis_vector[1], 0.0])  # y_axis for
    ↪  gripper
    norm = np.linalg.norm(y_axis)   # normalize
    if norm == 0:
        return [0.0, 0.0, 0.0, 1.0] # avoid crash if zero vector
    y_axis = y_axis / norm
    z_axis = np.array([-y_axis[1], y_axis[0], 0.0]) # z_axis for gripper
    # Verify z_axis point on global +x direction (from robot to object)
    if z_axis[0] < 0:
        z_axis = -z_axis
    # x_axis for gripper: X = Y x Z
    x_axis = np.cross(y_axis, z_axis)
    rotation_matrix = np.column_stack((x_axis, y_axis, z_axis)) # rotation matrix
    r = R.from_matrix(rotation_matrix)
    quat = r.as_quat() # convert into quaternion [x, y, z, w]
    return quat
```

Listing 9: Grasping orientation computing

Similarly, the retraction and approach trajectories are calculated dynamically relative to the object local frame. To define a pre-grasp pose, the node extracts the local Z-axis from the calculated grasp quaternion, representing the axis coming out of the robotic hand, and translates the target coordinates backward along this specific vector by a chosen distance.

```python
def calculate_pre_grasp_pose(self, grasp_pose, distance=0.070):
    # Extract orientation quaternion
    q = [grasp_pose.orientation.x, grasp_pose.orientation.y,
     grasp_pose.orientation.z, grasp_pose.orientation.w]
    # Convert quaternion to rotation matrix
    rotation = R.from_quat(q)
    matrix = rotation.as_matrix() # 3x3 rotation matrix
    # Extract local Z axis (third column of rotation matrix)
    z_axis_local = matrix[:, 2]
    # Calculate pre-grasp position by moving back along local Z axis
    pre_grasp_pose = Pose()
    pre_grasp_pose.orientation = grasp_pose.orientation
    pre_grasp_pose.position.x = grasp_pose.position.x - (distance * z_axis_local[0])
    pre_grasp_pose.position.y = grasp_pose.position.y - (distance * z_axis_local[1])
    pre_grasp_pose.position.z = grasp_pose.position.z - (distance * z_axis_local[2])
    return pre_grasp_pose
```

Listing 10: Pre-grasping pose computing

Recognizing that a single calculated pose might be unreachable, the geometric module systematically generates arrays of candidate poses. For placing operations, the `calculate_place_candidates` method creates multiple candidates by rotating the target frame around its global Z-axis. For watering tasks, the `calculate_watering_poses` algorithm radially distributes up to four distinct pouring locations around the plant centroid using the `get_pouring_orientations` method to assign two gripper orientations to each position.

Elementary motion abstractions

The node implements specific MoveIt functions for elementary movements. These abstractions automatically configure the planning pipelines, velocity and acceleration scaling factors and path constraints based on the requested motion profile.

The architecture provides a dual interface for each kinematic profile: a planning function (e.g., `plan_ptp`, `plan_linear`) that computes and returns a robot trajectory for future evaluation and a direct execution function (e.g., `move_ptp`, `move_linear`) that immediately plans and dispatches the movement to the hardware controllers.

As detailed previously, standard transfers across the workspace are handled by OMPL algorithms, providing rapid configuration-space exploration to navigate around obstacles, implemented in the `plan_ptp` and `move_ptp` methods.

Conversely, precise insertions and vertical lifts are delegated to the Pilz Industrial Motion Planner (`LIN`), used in the `plan_linear` and `move_linear` methods.

Furthermore, to handle open vessels containing fluids, an orientation-constrained motion profile is implemented in `plan_guarded` and `move_guarded`. These methods leverage the `OrientationConstraint` interface as detailed in listing 8.

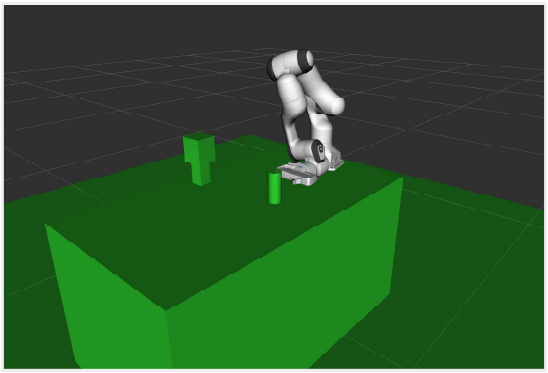
Finally, the pouring action is implemented in the `plan_pour_trajectories` and `perform_pour` methods. Executing a standard 6-DoF rotation to tilt a vessel often results in unpredictable elbow movements or kinematic singularities. To bypass this, the pouring abstraction isolates the rotation exclusively to the manipulator final joint. By directly modifying the `RobotState`, the node computes a precise +90° or -90° joint-space rotation based on the evaluated geometric relationship between the vessel orientation and the plant centroid, ensuring a controlled and collision-free pouring motion.

Aggregate manipulation primitives

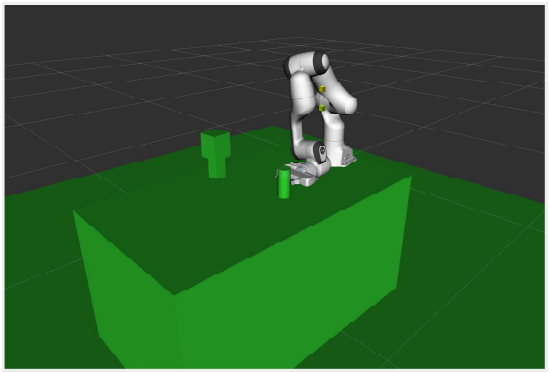
The geometric candidates and the elementary motion functions are ultimately orchestrated within aggregate manipulation primitives, specifically the `perform_grasp` and `perform_place` methods. These functions manage the complex sequencing of physical interactions by introducing a robust planning strategy paired with dynamic collision management.

Practically, the `perform_grasp` sequence guides the manipulator to a pre-grasp clearance location, opens the gripper, performs a linear insertion along the object geometry, closes the fingers to secure the payload and finally executes a linear vertical lift.

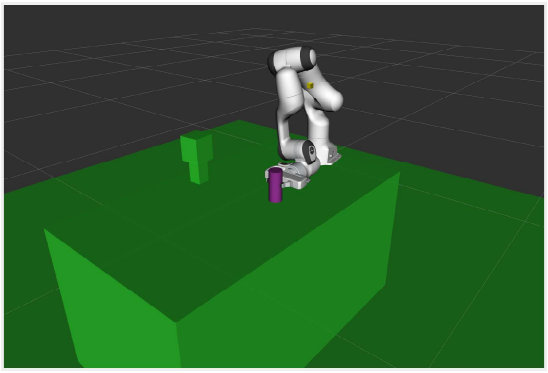
During the insertion, it explicitly evaluates the success of the grasping action: if the object is not successfully secured, the system prevents the execution of the lift phase, autonomously opens the gripper, restores the environmental collisions and performs a safe retraction to the pre-grasp waypoint.



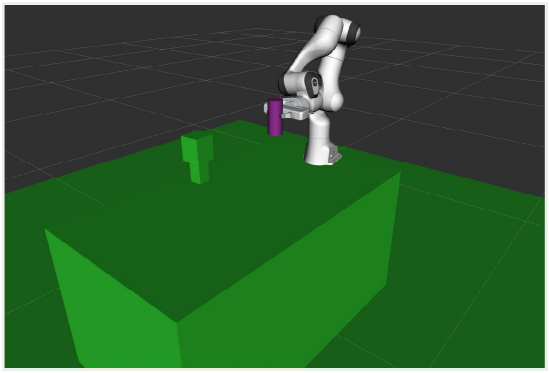
(a) Pre-grasping phase



(b) Linear insertion and grasping



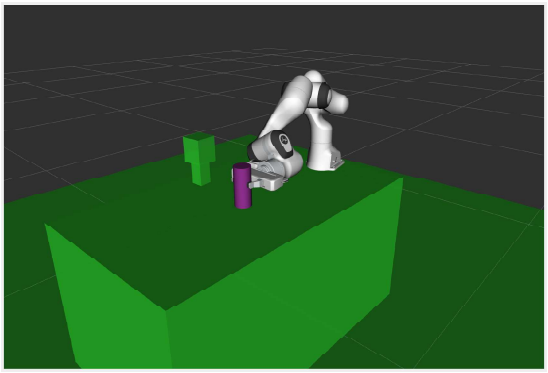
(c) Object attached to the TCP



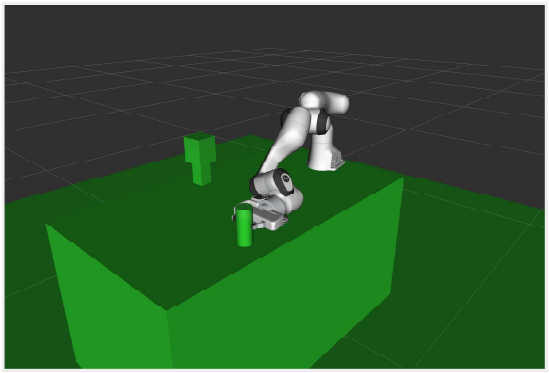
(d) Linear lift execution

Figure 4.6: The grasping primitive sequence

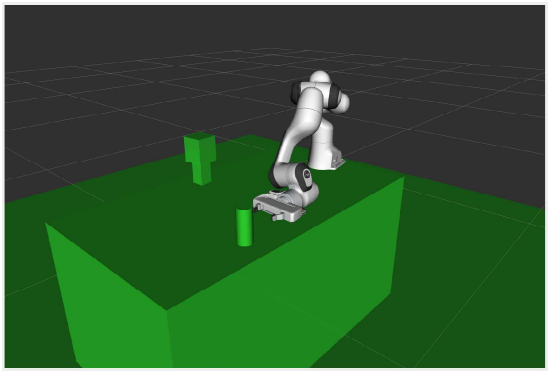
On the other hand, the `perform_place` method handles the deposit phase: it navigates the secured object to a target spatial coordinate, executes a linear descent, releases the payload and linearly retreats to clear the workspace.



(a) Pre-placing positioning



(b) Linear descent, release and detach



(c) Linear retreat

Figure 4.7: The placing primitive sequence

Instead of planning and executing sub-movements sequentially, the primitive iterates through the generated geometric candidate poses and attempts to pre-calculate the entire trajectory sequence in memory using the `plan_*` functions. To achieve this, the end-state of each planned trajectory is mathematically extracted and fed as the starting state for the subsequent planning phase. If a candidate sequence incurs a kinematic singularity or collision at any intermediate step, it is entirely discarded and only when all consecutive trajectory segments are successfully validated the robot begin the physical execution. This "plan-memorize-execute" paradigm ensures the manipulator never commits to a sequence that would result in a mid-task deadlock.

While a successful planning phase guarantees that a theoretical, collision-free path sequence exists, physical execution can still be aborted by the low-level hardware controllers due to real-world factors such as communication latency, strict trajectory tracking tolerances or transient execution delays. Consequently, the robot actual physical state at the onset of a subsequent trajectory segment might slightly diverge from the idealized starting point previously computed. If blindly forced to track this outdated path, the manipulator would attempt a sudden joint compensation to realign itself. This abrupt movement would violate the strict safety limits imposed on joint velocity and acceleration, triggering a protective

hardware stop. To mitigate this issue, the aggregate primitives incorporate a robust fallback mechanism. If a pre-planned segment is rejected, the system does not immediately abort the mission; instead, it seamlessly degrades to the direct `move_*` abstractions. By re-planning the single failing step from the current joint state, the fallback mechanism absorbs any spatial discrepancies, ensuring smooth task continuation.

Executing these primitives also requires precise control over the Allowed Collision Matrix (ACM). A strict collision-checking system fails during a grasp as the robot fingers must intersect the object bounding volume. Therefore, in these functions the ACM is dynamically modified during the execution to avoid collision errors during grasping phases. As showed in listing 11, a custom function of the `SceneManager` is used to temporarily allow intersections, effectively disabling collision checking, during the insertion phase.

```
# Disable collisions
self.scene_manager.set_allowed_collision(object_id, self.left_finger_link, allow=True)
self.scene_manager.set_allowed_collision(object_id, self.right_finger_link, allow=True)
self.scene_manager.set_allowed_collision(object_id, self.hand_link, allow=True)
...
# Restore collisions
self.scene_manager.set_allowed_collision(object_id, self.left_finger_link, allow=False)
self.scene_manager.set_allowed_collision(object_id, self.right_finger_link, allow=False)
self.scene_manager.set_allowed_collision(object_id, self.hand_link, allow=False)
```

Listing 11: Allowed Collision Matrix (ACM) manipulation

Once the physical gripper successfully closes, verified by evaluating the joint efforts to confirm the minimum grasping force, the object is programmatically attached to the end-effector using the `attach_object` method of the `SceneManager`. This operation alters the robot kinematic model: the object becomes a rigid extension of the robotic arm, meaning MoveIt will compute collisions between the grasped item and the surrounding environment during all subsequent trajectory planning.

Conversely, during placing operations, the `detach_object` method is called upon releasing the gripper. This releases the bounding volume back into the static scene, restoring the manipulator original end-effector geometry before the retreat phase.

4.4.4 High-level mission sequences

The uppermost logic of the motion layer acts as a mission orchestrator: upon receiving the streamlined executive JSON from the cognitive layer, the *mission_planner_node* parses the requested task and triggers the corresponding kinematic state machine. This architectural layer combines the previously defined robust primitives to accomplish complex, multi-stage physical tasks, ensuring that the semantic intent is safely mapped to the spatial constraints of the workspace.

The moving sequence

The moving sequence is designed to relocate an identified object from its current position to a designated target area.

The pipeline begins by extracting the `target_id` and the corresponding spatial geometry of the object. The orchestrator also evaluates the `target_destination` field: if the cognitive prompt provides explicit cartesian coordinates, requested by the user, the system parses the array to establish the precise place location. If no coordinates are provided, the system defaults to a predefined safe zone situated away from the primary biological targets.

Once all the data are available, the `moving_sequence` starts delegating the pick operation to the `perform_grasp` primitive which computes and executes the entire sequence of actions. When the grasping is completed, the `perform_place` primitive is invoked to deposit the payload: this method calculates the entire placing sequence and executes it.

Upon successful release, the sequence commands a system homing procedure, cleans the dynamic scene of attached objects and forces an Octomap voxel reset, leaving the robot in a nominal state awaiting further commands.

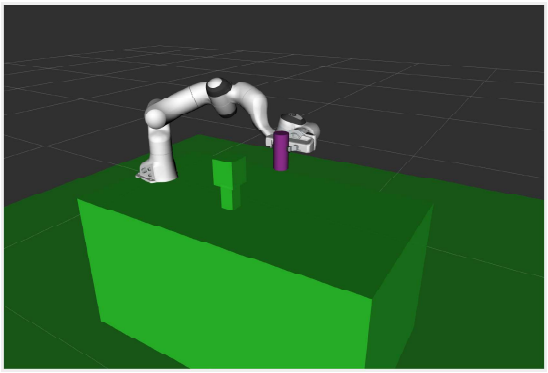
The watering sequence

The watering sequence represents the most complex kinematic state machine within the architecture as it requires the interaction between two distinct semantic objects: a vessel and a plant.

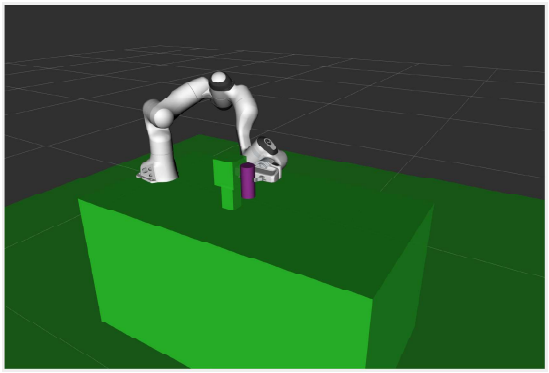
The orchestration begins by parsing the object map to verify the presence of both categories. Once validated, the system commands the `perform_grasp` primitive specifically on the cup, converting it into an extension of the robot arm using the `attach_object` method.

With the vessel secured, the system computes an array of valid pre-watering candidate poses dynamically generated around the plant perimeter. This pose acts as a pre-watering waypoint, located at a predefined vertical and radial offset from the plant centroid, where the manipulator halts to safely allow the `SceneManager` to explicitly disable collisions between the plant and the vessel, as well as between the plant and the gripper links.

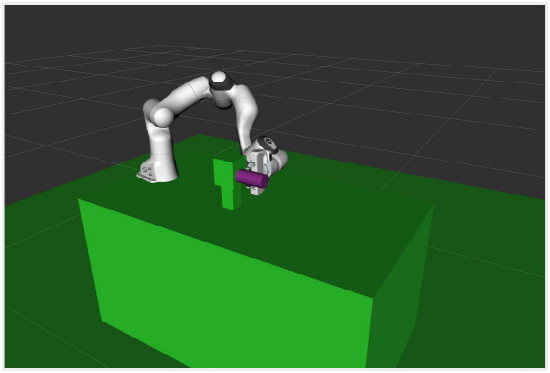
Once the ACM is correctly modified, the system performs a linear approach to the real watering pose using the Pilz planner and then executes the custom pouring action rotation around the 7th joint. Upon completion of the pouring action, the vessel is straightened, and a linear retreat is performed.



(a) Pre-watering pose



(b) Watering descent



(c) Pouring rotation

Figure 4.8: The pouring action sequence

Finally, once the vessel is safely retracted and the ACM is restored, the sequence invokes the `perform_place` primitive to return the cup to its original coordinates, concluding the agricultural task.

4.4.5 Fault tolerance and recovery mechanism

Autonomous robotic systems operating in semi-structured environments must account for unpredictable physical failures, such as a grasped object slipping, inverse kinematics solver deadlock or severe sensor latency. To ensure system resilience and prevent cascading errors, the *mission_planner_node* exposes dedicated fault tolerance mechanisms through asynchronous ROS2 services.

The most fundamental safety feature is the homing service. When triggered, the system invokes the `go_to_home` method, which commands the manipulator to return to a predefined, safe joint-space configuration. This method defines a strict joint-angle dictionary for all seven degrees of freedom: the node applies this configuration to a target `RobotState` and utilizes the OMPL pipeline to compute a global, collision-free extraction path. This ensures that the end-effector can safely navigate out of cluttered areas and return to a known nominal posture, concluding the movement by flushing the residual

Octomap voxels.

However, a simple homing trajectory is insufficient if the system experiences a critical desynchronization between the physical hardware and its mathematical representation in MoveIt. To handle these critical faults, the system provides a comprehensive `perform_recovery` protocol that acts as a complete hardware and software reset, executed sequentially to restore operational safety. As illustrated in listing 12, the recovery protocol follows a sequential logic to safely disengage the manipulator and resynchronize the system.

```
def perform_recovery(self):
    self.command_gripper(open_gripper=True)
    self.scene_manager.detach_all_objects()
    self.scene_manager.clean_scene(keep_floor=True, keep_table=True)
    self.scene_manager.restore_initial_acm()
    self.reset_voxels()
    success = self.go_to_home()
    if success:
        self.scene_manager.clean_scene(keep_floor=True, keep_table=False)
    return success
```

Listing 12: Recovery service sequence

The recovery sequence begins by commanding the gripper to open, releasing any grasped object. Subsequently, the system logically detaches all semantic items from the end-effector. The `SceneManager` then removes the dynamically generated primitive objects from the workspace. Static collision volumes, such as the work table, the floor and the camera mounts, are temporarily preserved to prevent the arm from colliding with the environment during the extraction maneuver.

Next, the ACM is restored to its default state, re-enabling standard self-collision and obstacle avoidance constraints. The system also calls the `/clear_octomap` service to clear the volumetric grid of any residual sensor noise or ghost obstacles. Once the planning scene is reset, the manipulator executes the `go_to_home` trajectory.

After reaching the home configuration, the procedure concludes by removing the remaining static structures, including the table. Removing the work table is a necessary preparatory step for future perception tasks: if left in the scene, MoveIt real-time self-filtering would subtract its volume from the raw point cloud, unintentionally erasing the base geometry of the objects resting on it.

This procedure restores the system to a nominal state, ready to receive new commands from the cognitive layer.

5. Modelling and Simulation

This chapter details the validation of the proposed hardware-agnostic architecture within a virtual environment, serving as a highly reliable simulated test-bed. Developed in CoppeliaSim, this environment allows for the safe evaluation of the system cognitive reasoning, the refinement of the kinematic control pipelines and the validation of the grasping and pouring strategies prior to real-world deployment.

The discussion is structured into three main sections, following the logical progression of the simulation development and testing. The first section introduces the virtual workspace setup and the custom ROS2 interfacing, detailing the specific software bridges and control solutions engineered to overcome the communication bottlenecks of the simulator. The second section provides a qualitative validation of the entire cognitive and control pipeline, illustrating the execution of the high-level robotic missions. Finally, the third section presents a quantitative assessment of the perception strategy by comparing the spatial reasoning capabilities of the Cloud LLM against the deterministic local point cloud processing. This final analysis definitively justifies the architectural distribution of tasks between the AI and the edge computing layers.

5.1 Environmental setup and ROS2 interfacing

To safely validate the proposed hardware-agnostic architecture, a virtual test-bed was established. This environment allows for the preliminary assessment of the cognitive reasoning, the visual perception pipeline and the kinematic trajectory planning before transitioning to the physical robot.

5.1.1 The simulated workspace

The virtual environment is built within CoppeliaSim, a versatile physics-based simulator. As shown in figure 5.1, the scene reproduces a standard laboratory workspace containing a table, a robotic arm, two RGB-D cameras, a plant and a cup.

The manipulator deployed in the simulation is a Franka Emika Panda robotic arm equipped with its standard parallel gripper. At the physics engine level, the manipulator joints are explicitly configured to be controlled in velocity, while the gripper mechanism is constrained to exert a maximum grasping force of 50 N, considered adequate for a

successful grasp.

The workspace is monitored by two simulated RGB-D sensors positioned to provide multi-view point clouds and overcome object occlusions to enhance pose estimation accuracy.

The table is populated with specifically modeled semantic items: a rectangular plant pot and a composite vessel. The vessel is designed as the union of a primary cylinder and a smaller bounding box, providing a clear geometric feature to test the precision of the grasping orientation algorithms. This design choice is particularly advantageous for the parallel gripper as establishing a stable contact on two flat and parallel surfaces is significantly more reliable than grasping a cylindrical profile.

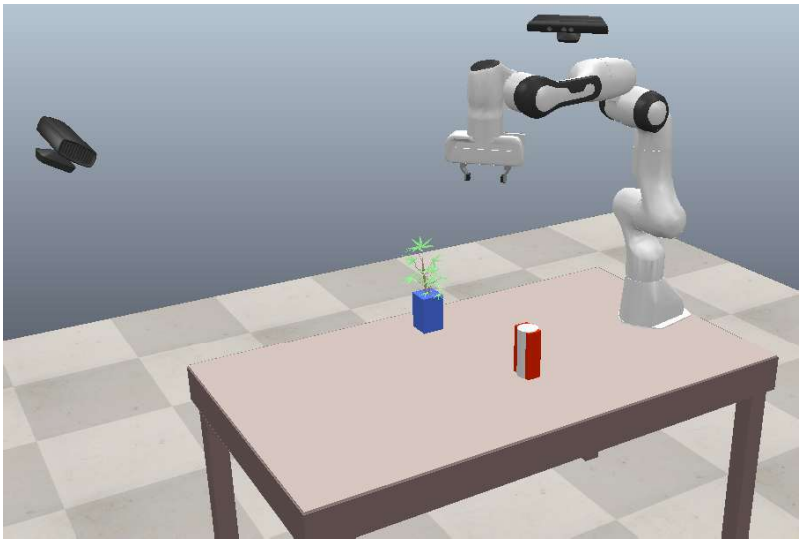


Figure 5.1: The simulated workspace in CoppeliaSim containing the robotic manipulator, the two cameras, the plant and the cup

5.1.2 The Coppelia bridge node

To guarantee full hardware abstraction, the ROS2 architecture must not differentiate between the simulator and the physical robot. To achieve this, a custom bidirectional bridge node, *coppelia_api_node*, was developed using the ZMQ remote API.

This node acts as a transparent middleware: it continuously reads the simulated sensors and joint values, publishing standard ROS2 messages such as `sensor_msgs/Image`, `sensor_msgs/PointCloud2` and `sensor_msgs/JointState`. Simultaneously, it acts as a command interface by exposing the `/panda/set_gripper` service and instantiating an action server for the `FollowJointTrajectory` interface, mirroring the actual hardware controllers. This bidirectionality ensures that while the node streams real-time environmental states to the ROS2 network, it concurrently receives and translates high-level planning requests into low-level simulator commands. The specific mechanisms governing this trajectory execution and control are detailed in the subsequent section.

5.1.3 Trajectory execution and custom robot control

A critical challenge in robotic simulation is the temporal mismatch between the kinematic trajectory generation and the physics engine execution. When MoveIt computes a trajectory, it often generates a highly dense sequence of waypoints and streaming these raw, dense trajectories directly to a simulator running at a fixed integration step causes the physics engine to stutter, accumulating significant tracking delays and failing the trajectory execution.

To resolve this bottleneck, the bridge node actively intercepts the incoming MoveIt trajectory and performs a spatial down-sampling by computing the Euclidean distance in the joint space between consecutive waypoints. The algorithm discards points that fall below a minimum distance threshold, retaining only the strictly necessary kinematic points, as demonstrated in listing 13.

```
# Downsampling
waypoints = []
waypoints.append(points[0])
last_kept_pos = np.array(points[0].positions)
MIN_DIST_THRESHOLD = 0.1    # rad, threshold for downsampling
for i in range(1, len(points) - 1):
    curr_pos = np.array(points[i].positions)
    dist = np.linalg.norm(curr_pos - last_kept_pos) # Euclidean distance in joint space
    if dist > MIN_DIST_THRESHOLD:
        waypoints.append(points[i])
        last_kept_pos = curr_pos
waypoints.append(points[-1])    # Always keep the last point as goal
```

Listing 13: Trajectory down-sampling strategy

Once the optimized waypoints are extracted, the node cannot simply enforce the theoretical joint positions, as forcing instantaneous teleports between coordinates would cause the robot to move in a rigid and jerky manner, bypassing the physics engine and compromising the dynamic realism of the simulation. Instead, it implements a custom closed-loop proportional controller: for each waypoint the algorithm calculates the positional error for all seven degrees of freedom. This error is then multiplied by an adaptive proportional gain (K_p) to compute a requested joint velocity. Specifically, the controller dynamically adjusts this gain based on the overall trajectory length: shorter paths, involving finer movements, employ a lower K_p to prevent overshoots, while longer paths use a higher gain. To ensure stability, this velocity is dynamically clipped against both the physical hardware limits and a predefined safe velocity boundary. These continuous velocity commands are then sent to the CoppeliaSim engine until all joints fall within a strict angular tolerance, signaling the reaching of the waypoint.

The core logic of this adaptive velocity control is detailed in listing 14.

```
# Control loop for each waypoint
for wp_idx, wp in enumerate(waypoints):
    ...
    target_pos = wp.positions
    ...
    is_last_point = (wp_idx == len(waypoints) - 1)
    current_tolerance = FINAL_TOLERANCE if is_last_point else INTERMEDIATE_TOLERANCE
    arrived = False
    ...
    while not arrived and rclpy.ok():
        ...
        joints_within_tolerance = 0
        max_error_display = 0.0
        with self.sim_lock:
            for i, handle in enumerate(self.joint_handles):
                current = self.sim.getJointPosition(handle)
                target = target_pos[i]
                error = target - current
                if abs(error) > max_error_display: max_error_display = abs(error)
                # Compute proportional velocity
                req_vel = KP_VAL * error
                # Dynamic physical limit
                phys_limit = self.MAX_VELOCITIES[i]
                # Effective limit
                limit_used = min(SAFE_VEL_LIMIT, phys_limit)
                # Clip to limits
                if req_vel > limit_used: req_vel = limit_used
                if req_vel < -limit_used: req_vel = -limit_used
                # Send command
                self.sim.setJointTargetVelocity(handle, req_vel)
                # Check tolerance
                if abs(error) < current_tolerance:
                    joints_within_tolerance += 1
        if joints_within_tolerance == len(self.joint_handles):
            arrived = True
```

Listing 14: Custom proportional controller implemented in the bridge node

5.2 Validation of the control and cognitive pipeline

Before deploying the architecture on the physical Franka manipulator, the complete Sense-Think-Act loop was qualitatively validated within the CoppeliaSim test-bed. The primary objective of these simulated scenarios is to confirm that the semantic reasoning of the Cloud AI translates accurately into collision-free, stable trajectories executed by the local motion layer. The validation was conducted through predefined use cases, testing the system ability to handle the two core robotic missions.

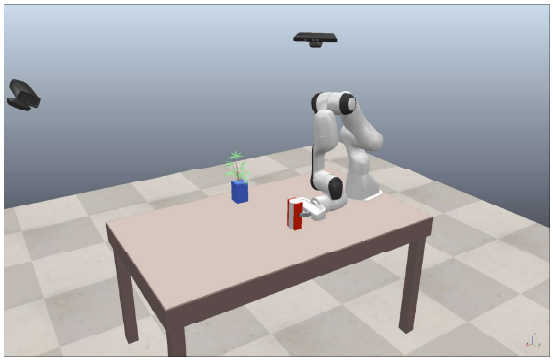
5.2.1 Use case 1: moving

The first simulated scenario evaluates the system ability to interpret explicit spatial commands and relocate known semantic objects. The cognitive layer receives a direct user prompt instructing the robot to move the cup to a specific set of spatial coordinates. The LLM processes the annotated visual input and the text command, classifying the item as a `vessel`. It correctly extracts the numerical target coordinates from the prompt,

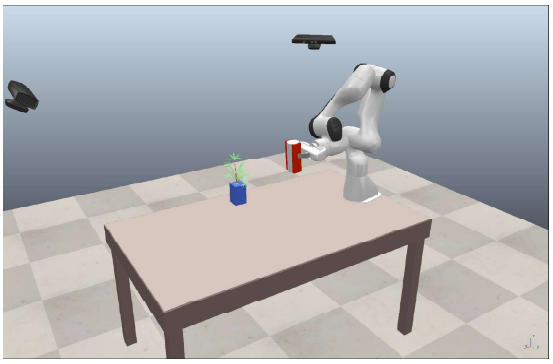
demonstrating accurate parameter parsing and dispatches a `moving` mission with a defined `target_destination`.

During physical execution, the geometric module extracts the grasping pose from the cup point cloud cluster. The `perform_grasp` primitive initiates the sequence by planning a collision-free approach to a pre-grasp waypoint. To secure the target, the system dynamically disables the ACM for the gripper links, allowing a strict linear insertion along the object geometry. Once the fingers close and the vessel is logically attached to the kinematic chain, a linear vertical lift is executed.

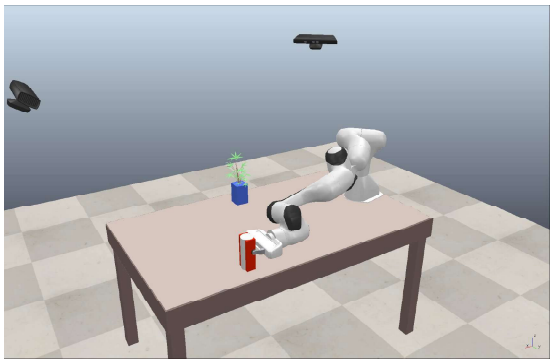
To transport and deposit the open vessel, the system invokes the `perform_place` primitive, which manages the entire relocation sequence. It initially executes a constrained transit phase, keeping the cup upright while navigating to a raised waypoint directly above the user-requested destination. Once positioned, the primitive proceeds with the deposit: it executes a linear descent to the table surface, opens the gripper to release the payload and performs a linear retreat to clear the workspace before fully restoring the ACM constraints. Upon completion of this sequence, the system executes a final homing procedure, returning the manipulator to its nominal resting posture.



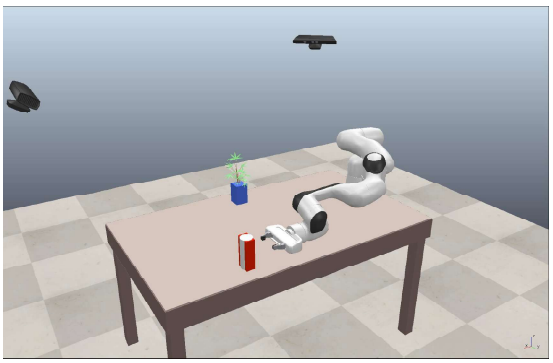
(a) Object grasping



(b) Linear lift



(c) Place



(d) Retreat

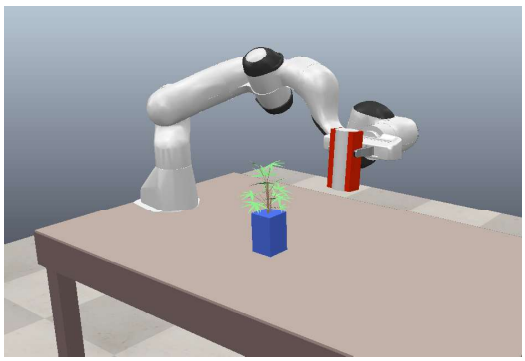
Figure 5.2: Simulated execution of the moving sequence

5.2.2 Use case 2: watering

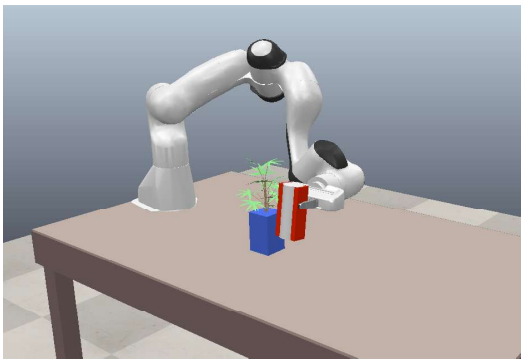
The second scenario evaluates the system ability to execute the most complex kinematic state machine of the architecture. Instead of relying on a continuous telemetry stream, the cognitive layer receives a user prompt stating that the hardware sensors are currently offline, but the plant soil appears completely dry. The LLM successfully processes this contextual information alongside the visual input: it identifies both the `vessel` and the `plant`, deduces the necessity of hydration despite the missing telemetry and autonomously dispatches a `watering` mission.

The physical execution orchestrates the interaction between the two objects. Following the `perform_grasp` of the cup, the manipulator executes a constrained transit to reach a pre-watering waypoint, dynamically calculated around the plant perimeter. At this stage, the simulation demonstrates the critical ACM toggling: by explicitly disabling collision checking between the plant and the cup-gripper system, MoveIt permits the subsequent linear approach towards the canopy.

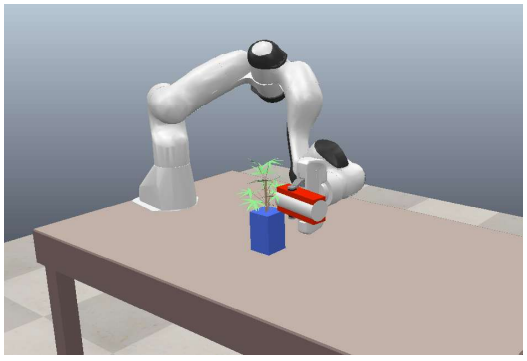
The custom pouring action is then performed by isolating the rotation exclusively to the 7th joint, tilting and straightening the vessel. Finally, the robot performs a linear retreat to the pre-watering waypoint, restores the environmental collisions and invokes the `perform_place` primitive to return the cup to its initial coordinates.



(a) Pre-watering waypoint



(b) Linear descent



(c) Joint 7 pouring rotation

Figure 5.3: Simulated execution of the watering sequence

5.3 Perception strategy evaluation: AI vs local processing

A core architectural decision of this thesis is the hybrid Cloud-Edge paradigm, which delegates semantic reasoning to the Cloud LLM while keeping spatial and geometric computations on the local edge. To justify this division of labor, an experimental campaign was conducted to evaluate whether a state-of-the-art multimodal LLM could autonomously process raw sensory data, RGB images and point cloud coordinates, to extract deterministic grasping poses and how this compares to traditional, deterministic local processing.

The initial design philosophy aimed to maximize the delegation of tasks to the generative model, attempting to centralize both reasoning and spatial computation within a single cognitive entity. However, iterative testing revealed that certain geometric operations were not only computationally expensive and token-intensive for the LLM, but also prone to inaccuracies. Consequently, the architecture evolved to offload deterministic spatial tasks back to the local Edge, where traditional algorithms already offer precision, robustness and lower latency at a fraction of the computational cost.

5.3.1 Test 1: prompt A vs prompt B

The first phase of the evaluation tested the spatial reasoning capabilities of the generative AI. The models were provided with an annotated RGB image for semantic context and a serialized CSV file containing the raw X, Y and Z coordinates of the point cloud. The LLM was instructed to generate and execute a Python script to compute the exact centroid and the main grasping axis vector for both the cup and the plant.

To evaluate the impact of prompt engineering on spatial accuracy, two distinct cognitive strategies were tested:

- prompt A (procedural logic): tested on Gemini Robotics-ER 1.5, this prompt forced the model to act as a rigid script generator. It provided explicit, hard-coded rules (e.g., "if height $>$ 0.16m, it is a plant") and forced the use of specific OpenCV functions on predefined Z-axis slices;

- prompt B (autonomous geometric reasoning): tested on Gemini 2.5 Pro, this prompt removed hard-coded thresholds. Instead, it instructed the model to autonomously evaluate the Z-axis density histograms to choose between two dynamic strategies ("Solid Body Isolation" or "Primitive Reconstruction"), effectively asking the AI to reason about the object morphology before applying mathematical fitting (e.g., least-squares circle fitting vs. bounding boxes).

The results of the AI-driven perception demonstrated a high degree of variability. While positional errors (X, Y) were generally bounded within a 20 mm range, the orientation estimation (yaw angle) proved critically inadequate for robotic manipulation. As detailed in the subsequent comparison, prompt A produced mean orientation errors up to 75° for

complex shapes like the plant. Prompt B improved the cognitive reasoning by avoiding basic centroid calculations, yet still produced mean angular errors exceeding 28° for the cup. These significant discrepancies highlight that, while the LLM successfully writes syntactically correct code, it lacks robust spatial awareness and struggles to adapt its generated algorithms to the noise and occlusions inherent in point clouds.

5.3.2 Test 2: local perception

To establish a baseline for comparison, the exact same spatial configurations were processed using the local *perception_node* described in section 4.2.

Unlike the AI-generated scripts, this local pipeline relies on an optimized and deterministic Python backend. By applying RANSAC for table removal and DBSCAN for cluster isolation, followed by custom density-slicing algorithms tailored to the specific semantic category, the local node filters out sensor noise before applying geometric fitting.

The local processing demonstrated high robustness, maintaining high accuracy for spatial positioning and an orientation error of near one degree, outperforming the AI-based estimations regardless of the object morphological complexity.

5.3.3 Comparisons and conclusions

To quantify the performance disparity, the mean errors in position (ΔX, ΔY) and orientation (Δyaw) were evaluated across multiple capture scenarios. The results are summarized in Table 5.1.

Methodology	**Mean Error Cup**			**Mean Error Plant**		
	ΔX **[m]**	ΔY **[m]**	Δyaw [°]	ΔX **[m]**	ΔY **[m]**	Δyaw [°]
AI Prompt A	0.003	0.001	22.44	0.009	0.008	75.16
AI Prompt B	0.005	0.000	28.84	0.006	0.005	8.17
Local edge node	0.002	0.001	1.23	0.004	0.004	1.32

Table 5.1: Quantitative comparison of spatial perception errors between Cloud AI strategies and local processing

The experimental data justifies the proposed architectural design. While LLMs exhibit strong capabilities in semantic labeling and contextual decision-making, they are still unreliable for high-precision spatial calculation. The orientation errors produced by the AI would inevitably cause collisions during a physical grasping phase, failing the entire task.

Conversely, the local perception pipeline guarantees the mathematical determinism required for safe kinematic planning. Therefore, the hybrid Cloud-Edge architecture proves to be the optimal and safest approach for autonomous robotic manipulation.

6. Experimental Validation

Following the validation of the cognitive and kinematic pipelines within the virtual environment, this chapter details the deployment of the proposed architecture onto a physical robotic cell. The primary objective of this experimental campaign is to evaluate the sim-to-real transferability of the developed SDL framework. Operating in a physical environment inevitably introduces unpredictable variables, such as sensor noise and the complex dynamics of the real robot controller.

To assess the framework performance under these unstructured conditions, the system was deployed on a real-world experimental setup featuring a Franka Research 3 manipulator. The discussion is structured into three main focal points. The first section details the physical experimental setup, outlining the hardware components, the multi-camera integration and the custom spatial calibration procedures. The second section discusses the sim-to-real transfer phase, highlighting the necessary algorithmic refinements applied to the motion planning and perception pipelines. Finally, the third section demonstrates the system operational capabilities through concrete case studies, specifically focusing on autonomous object relocation and context-aware plant care.

6.1 Experimental setup

To evaluate the real-world performance and the sim-to-real transferability of the proposed SDL architecture, a physical experimental test-bed was established. While this setup functionally mirrors the layout of the virtual environment developed in CoppeliaSim, it introduces the complex hardware infrastructure necessary for physical data acquisition, low-level kinematic control and Edge computing. The following subsections detail the mechanical, visual and environmental sensory subsystems that constitute the physical robotic cell.

6.1.1 The robotic cell and the hardware components

The core of the experimental setup is built around a Franka Research 3, a 7-DoF collaborative robotic arm equipped with its standard parallel gripper. The manipulator is firmly mounted on a dedicated base adjacent to the main operational table, which houses the agricultural targets along with the associated cabling for the sensory and telemetry

infrastructure.

A spatial adjustment was applied to the physical workspace geometry during the initial deployment phase. Unlike the idealized simulated environment, operating the physical manipulator on a standard-height table forced the end-effector to frequently reach downward, operating near the lower boundaries of its reachable envelope. This configuration forced the inverse kinematics solver to position the proximal joints near their physical limits, causing the controller to trigger its safety brakes. To mitigate this, the primary work surface was structurally elevated. This spatial optimization effectively shifts the target objects into the robot dexterous workspace ensuring smoother and safer trajectory execution.

The operational table is populated with the physical semantic items: a pot and a blue watering vessel. Unlike the simulated environment, the physical vessel was specifically designed as a pure rectangular parallelepiped ensuring a complete, planar surface contact with the parallel gripper's fingers. The physical grasp failure rate was reduced to near zero during the experimental campaign.

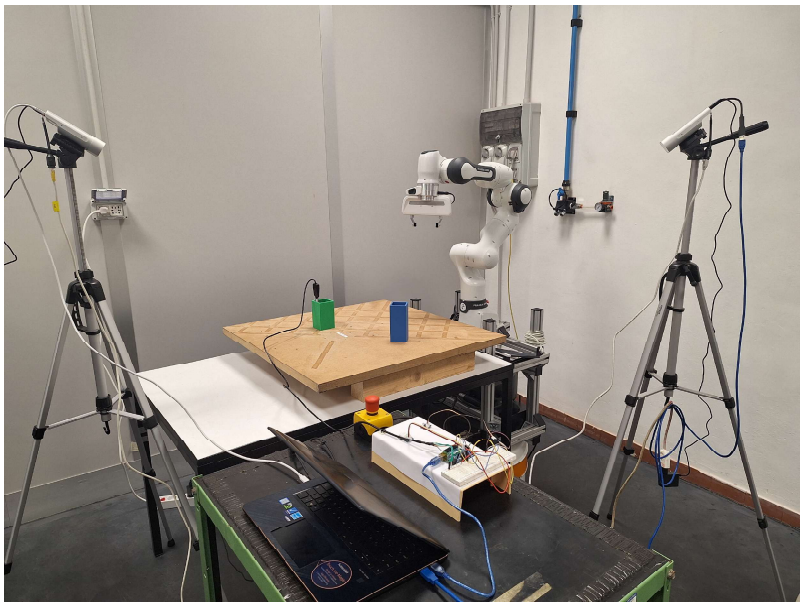


Figure 6.1: The experimental setup

The entire software architecture is executed on a standard computer running Ubuntu 22.04 and the ROS2 Humble middleware. Acting as the central computing hub of the experimental cell, this machine is physically interfaced with the Franka manipulator controller, the two RGB-D cameras and the Arduino microcontroller. It serves as the local Edge platform hosting the complete ROS2 network: it runs the MoveIt motion planning pipelines, processes the sensor data through the local perception algorithms and manages the asynchronous API calls to the Cloud-based LLM.

6.1.2 Multi-camera system

The environmental perception relies on a multi-camera vision system composed of two Orbbec Femto Mega RGB-D sensors [24]. These cameras are strategically mounted around the workspace to provide overlapping 3D views, reducing blind spots and mitigating object occlusions.

Deploying multiple active depth sensors in the same physical space introduces a critical physical challenge: since both cameras project structured infrared light to compute depth maps, their overlapping projection patterns interfere with one another, resulting in severe degradation of the point cloud quality, spatial noise and "ghost" geometries. To completely eliminate this problem at the hardware level, the cameras are physically tethered using an Orbbec Sync Hub Pro. This external module acts as a hardware trigger mechanism, distributing a unified synchronization signal across the connected devices. By designating one camera as the "primary" device and the other as "secondary", the hub interleaves their optical emissions with microsecond precision, ensuring that the two sensors never project their infrared patterns simultaneously.

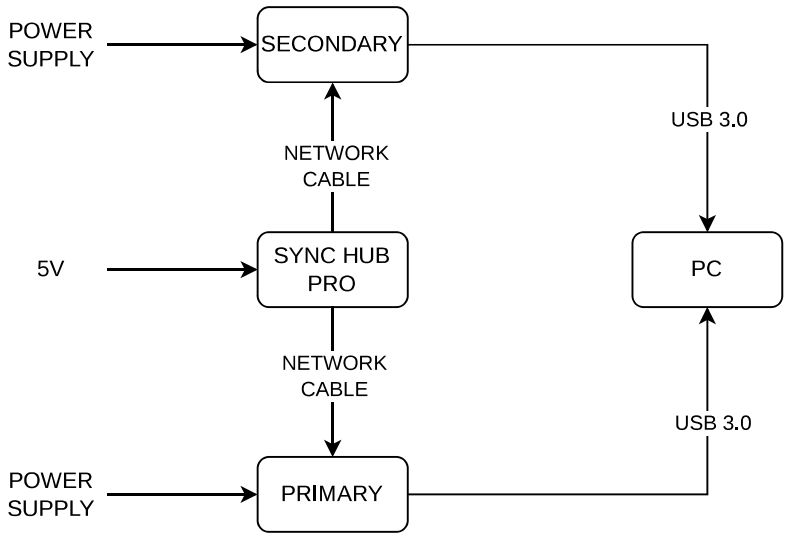


Figure 6.2: Hardware synchronization topology utilizing the Sync Hub Pro

The low-level hardware communication and data streaming are handled by the official OrbbecSDK V2 ROS2 Wrapper provided by the manufacturer [25]. However, integrating this generic multi-camera driver into the specific SDL architecture requires parameter tuning to balance perception quality with the CPU and bandwidth limitations. To achieve this, a custom launch configuration was engineered to enforce these system-specific optimizations at startup.

Firstly, to prevent the saturation of the data bus and to allocate the Edge processing power exclusively to spatial reasoning, all non-essential data streams, such as raw infrared

feeds, internal accelerometers and gyroscopes, are explicitly disabled. The active streams are limited to the aligned RGB frames (1280x720) and the spatial Depth maps (640x576), capped at a functional frequency of 15 frames per second.

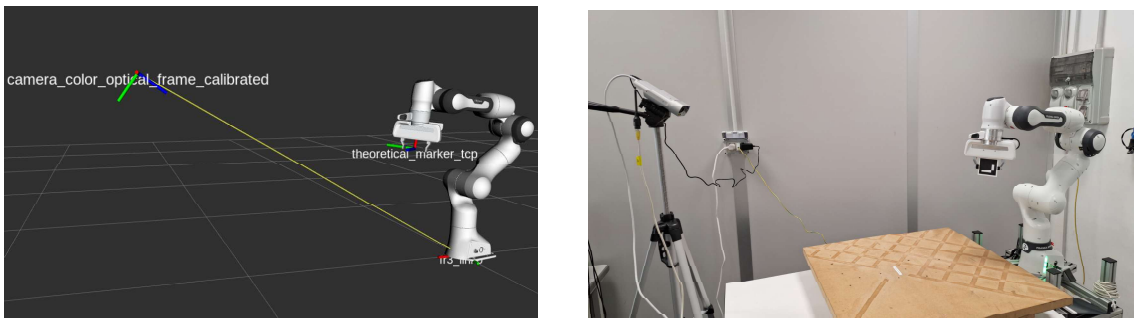
Secondly, point cloud preprocessing is largely offloaded to the cameras internal hardware. The launch parameters activate built-in spatial, temporal and noise-removal filters, along with a decimation filter to down-sample the raw cloud before transmission. This significantly reduces the payload size without compromising the object geometric features.

Finally, the launch sequence manages the hardware synchronization protocol: the primary camera is initialized first to establish the master clock while the secondary camera is launched using a ROS2 `TimerAction` with a 5.0 seconds delay, ensuring the primary sensor is fully operational. When the secondary sensor boots, it is configured with a strict `depth_delay_us` of 160 microseconds: this programmatic micro-delay shifts its optical emission phase relative to the primary trigger, physically executing the temporal interleaving required to prevent optical interference.

6.1.3 Custom extrinsic calibration using ArUco markers

To effectively use the multi-camera vision system for robotic manipulation, the isolated point clouds generated by the sensors must be spatially fused and projected into a unified coordinate space. Since the MoveIt motion planner operates relative to the robot origin, it is mandatory to determine the exact 6-DoF extrinsic calibration, the rigid body transformation consisting of translation and rotation, between the robot base frame (`fr3_link0`) and each camera optical frame.

Since the cameras are rigidly mounted in the environment while the manipulator moves, this problem falls under the "Eye-to-Hand" calibration paradigm. To solve it, a custom and deterministic calibration algorithm was engineered: the procedure relies on a known ArUco marker, from the `DICT_ARUCO_ORIGINAL` dictionary, rigidly attached to the robot Tool Center Point (TCP) (figure 6.3b).



(a) RViz visualization

(b) Physical laboratory setup

Figure 6.3: Eye-to-Hand calibration setup

The calibration mathematics relies on the resolution of a closed-loop spatial chain:

at any given moment, the forward kinematics of the manipulator, retrieved in real-time via the ROS2 `tf2` library, provide the transformation matrix from the base to the TCP, denoted as T_{base}^{tcp}. Knowing the physical mounting offset of the marker relative to the gripper, modeled as a static 90° rotation matrix T_{tcp}^{marker}, the absolute pose of the marker with respect to the robot base is calculated as:

$$T_{base}^{marker} = T_{base}^{tcp} T_{tcp}^{marker}$$

Simultaneously, the camera captures the visual scene and the algorithm processes the RGB frame. It detects the corners of the ArUco marker and leverages the Perspective-n-Point (PnP) algorithm (`cv2.solvePnP`) along with the camera intrinsic matrix to compute the marker pose relative to the camera lens, defined as T_{camera}^{marker}. By closing the kinematic loop, the unknown static transformation from the robot base to the camera optical frame (T_{base}^{camera}) is mathematically isolated:

$$T_{base}^{camera} = T_{base}^{marker} (T_{camera}^{marker})^{-1}$$

While a single image capture could theoretically resolve the equation, relying on a single sample is highly susceptible to sensor noise and algorithmic estimation errors. To guarantee a higher precision, the calibration script implements a robust statistical approach through human-in-the-loop multi-pose averaging.

During the calibration routine, the operator manually moves the robotic arm through a series of N different spatial configurations within the camera field of view. For each pose, instead of capturing a single frame, the user triggers a "burst capture" (e.g., 30 consecutive frames). The script aggregates these local frames, calculating the instantaneous T_{base}^{camera} for each iteration.

At the end of the procedure, the algorithm processes the entire dataset. The translational vectors are averaged linearly, while the rotational quaternions are correctly averaged using spherical geometric methods (`scipy.spatial.transform.Rotation.mean`). To quantify the reliability of the calibration, the algorithm computes the mean translational and rotational deviations across all sampled poses. It is crucial to note that these metrics do not represent an absolute accuracy error against a physical ground truth, which is inherently unknown, but rather the internal consistency of the estimation across different kinematic configurations. Finally, the algorithm outputs a formatted JSON file containing the precise `-x, -y, -z, -qx, -qy, -qz, -qw` arguments. These parameters are ready to be injected directly into the ROS2 `static_transform_publisher` launch files, reducing setup times and preventing human transcription errors.

6.1.4 Telemetry hardware integration

To provide the Cloud LLM with real-time data regarding the plant state, the experimental setup integrates a dedicated telemetry acquisition pipeline. Since the Edge nodes and

the ROS2 middleware are not designed for direct low-level analog signal processing, the hardware layer utilizes an Arduino UNO microcontroller as a localized data acquisition unit.

The microcontroller interfaces with three specific environmental sensors:

- capacitive soil moisture sensor: connected via an analog pin, it measures the dielectric permittivity of the surrounding soil to estimate water content;
- ambient light sensor: also connected via an analog pin, used to evaluate the workspace illumination levels;
- DHT22 module: a digital sensor, providing accurate readings for both ambient air temperature and relative humidity.

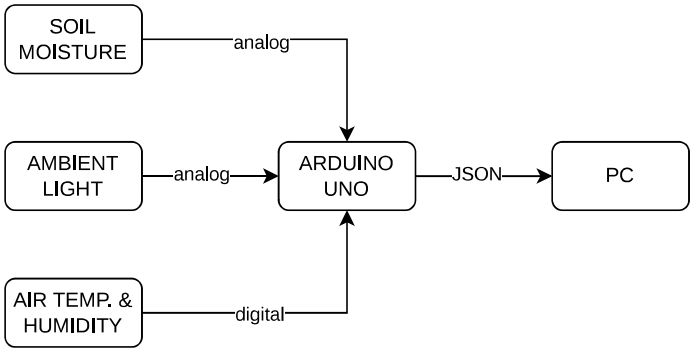


Figure 6.4: Schematic of the telemetry hardware integration detailing the electrical connections and data types between the environmental sensors, the Arduino microcontroller and the PC

The custom firmware deployed on the Arduino is specifically engineered to preprocess the raw data before transmission. To ensure semantic consistency for the generative AI, the raw analog readings (ranging from 0 to 1023) are mathematically mapped to standardized percentages (0-100%) using calibrated bounds for dry/wet and dark/bright conditions. Furthermore, the firmware implements safety checks to catch hardware misreads (e.g., `isnan` evaluations on the DHT22), dynamically aborting and retrying the read cycle if data is missing. The validated telemetry is then formatted into a JSON string and transmitted over the USB serial connection. Listing 15 provides an example of the serialized payload generated by the microcontroller.

```json
{
  "temperature": 24.50,
  "humidity": 52.10,
  "soil_moisture": 65,
  "light_level": 82
}
```

Listing 15: Example of the JSON-formatted plant telemetry

To inject this serial stream into the robotic software architecture, a custom ROS2 node, named *arduino_bridge_node*, runs on the Edge computing machine. Operating in an unstructured physical environment, serial communications can occasionally suffer from dropped bytes or interferences, resulting in truncated strings. To guarantee system robustness, the *arduino_bridge_node* acts as a data validator: it intercepts the incoming serial lines and attempts to decode them. If the string is malformed or incomplete, the node catches the `JSONDecodeError`, discards the data and waits for the next cycle without crashing. Only validated and structured JSON strings are published to the `/hardware/sensors_data` ROS2 topic, ensuring that the *ai_agent_node* always receives reliable and machine-readable telemetry.

6.2 Sim-to-real transfer

Transitioning a robotic architecture from a deterministic virtual test-bed to a physical experimental cell introduces a fundamental challenge in robotics known as the "reality gap". While the CoppeliaSim environment validated the high-level cognitive reasoning of the LLM and the core kinematic logic, physics engines inherently idealize environmental variables. They operate with perfect sensory data, mathematically flat surfaces and simplified dynamic interactions, which do not accurately represent the complexities of the physical world.

Deploying the SDL framework on the real Franka Research 3 manipulator exposed the system to unpredictable physical constraints: noisy depth point clouds, optical infrared reflections and strict joint torque limits enforced by the hardware controllers. To guarantee operational safety and achieve a successful sim-to-real transfer, the core software architecture required parametric tuning and algorithmic refinements.

The following subsections detail the critical adaptations implemented to bridge this gap. The discussion focuses primarily on the dynamic scaling constraints applied to the motion planning layer to prevent hardware faults and the robust filtering strategies integrated into the perception pipeline to handle real-world sensor noise.

6.2.1 Mission planner tuning

While the CoppeliaSim environment validated the motion logic, deploying the generated trajectories onto the physical Franka controllers required substantial modifications to the

mission_planner_node. The physical hardware introduces strict safety limits on joint torques and is highly sensitive to the arm physical inertia, necessitating a more conservative and adaptive kinematic approach.

The most critical architectural upgrade was the transition from the blind execution of pre-calculated trajectories to a dynamic, state-based execution paradigm. In the simulated environment, aggregate primitives could be fully planned in memory as a sequence of trajectories and executed sequentially. However, physical robots accumulate micro-delays and tracking inaccuracies. Executing a pre-calculated sequence often resulted in a state mismatch, where the physical robot joint positions at the end of the first movement slightly deviated from the idealized start state of the subsequent trajectory, causing the hardware controllers to abort the execution.

To resolve this without losing the safety guarantee of the pre-validation, the real-world planner implements dedicated functions. Instead of commanding the robot to reach a spatial Cartesian pose $(X, Y, Z, \alpha, \beta, \gamma)$, which could yield multiple valid inverse kinematic solutions, the system extracts the exact 7-DoF joint configuration from the end of the successfully pre-planned trajectory. The orchestrator then actively samples the real-time physical joint states and dynamically forces MoveIt to re-plan a new trajectory directly to that specific joint target. This approach guarantees that the manipulator arrives at the exact posture required to safely begin the subsequent movement, preventing mid-task kinematic singularities, joint limit violations or orientation constraint failures.

A second major refinement was applied to the geometric pose generation. Forcing an unnatural wrist orientation or an overly extended elbow posture triggers a torque fault from the Franka safety monitors. To prevent this, the real-world node expands the candidate generation to eight distinct spatial approaches, encompassing both the object primary and perpendicular axes. Crucially, these candidates are mathematically sorted based on their 2D Euclidean distance from the robot base. This forces the solver to prioritize the closest posture significantly reducing the mechanical stress on the proximal joints.

Finally, the dynamic parameters of the motion pipelines were strictly throttled. While velocity scaling was maintained at 15% to ensure smooth transits, the maximum acceleration scaling factor was capped at 5%. This significant reduction prevents sudden jerks and inertial spikes, avoiding the triggering of the sensitive collision thresholds of the collaborative arm.

6.2.2 Perception pipeline refinement

Transferring the vision system to the physical SDL cell highlighted the severe limitations of simulated sensors. While CoppeliaSim generates mathematically perfect, ultra-dense point clouds, the physical Orbbec RGB-D cameras introduced significant spatial noise, infrared reflections, depth shadows and background laboratory clutter. Consequently, the local *perception_node* required heavy algorithmic restructuring to prevent system failures

during semantic clustering and pose estimation.

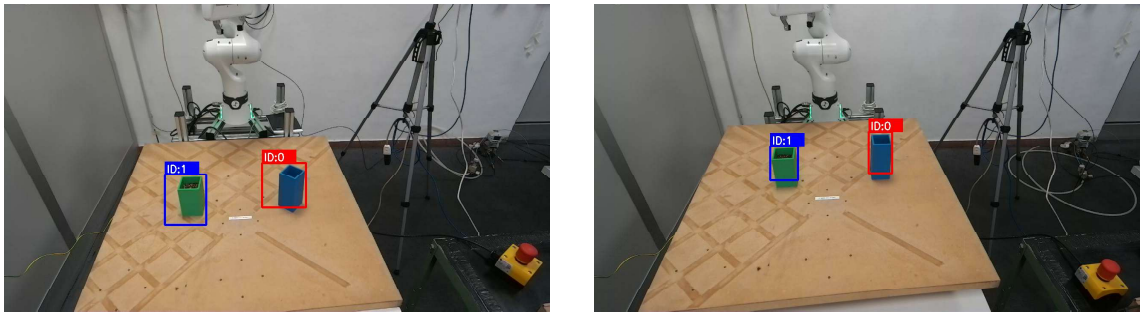
The first critical refinement targeted the preliminary scene filtering. In the simulation, applying a simple global Z-axis cutoff was sufficient to isolate the table. In the physical setup, the cameras capture surrounding infrastructure, floor surfaces and table edges. To counter this, a Cartesian bounding-box crop (constrained by $X_{min}, X_{max}, Y_{min}, Y_{max}$) was introduced prior to any processing, effectively isolating the functional workspace. Furthermore, the RANSAC `table_z_margin` threshold had to be increased from 5 mm to 15 mm. Physical tables are rarely perfectly flat and extrinsic calibration inherently carries inaccuracies.

Additionally, the physical depth sensors generate "ghost" floating points due to optical reflections and IR multi-path interference. To prevent the DBSCAN algorithm from erroneously classifying this noise as physical obstacles, a statistical `remove_radius_outlier` filter was injected into the pipeline immediately after the table removal phase, aggressively purging low-density artifacts.

The most substantial upgrade was implemented in the core geometric pose estimation logic (`estimate_standing_plant` and `estimate_standing_cup`). The simulated architecture relied on OpenCV Minimum Area Rectangle (`cv2.minAreaRect`) applied over the 2D convex hull of the clusters. Applying a standard minimum area rectangle to these imperfect point clouds produced heavily skewed bounding boxes, yielding completely inaccurate yaw orientation angles that would cause the parallel gripper to collide with the object corners.

To guarantee deterministic grasping orientations in the physical world, the standard convex hull approach was replaced with a custom, internal 2D RANSAC line-fitting algorithm. This specialized logic iterates through random pairs of points within the cluster to identify the densest, flattest visible wall of the target object. Once this dominant surface is mathematically isolated, its normal vector is extracted. The algorithm then normalizes this vector (handling the 90° symmetry of the rectangular objects), temporarily rotates the point cloud to align with the global axes, extracts an orthogonal bounding box to find the true spatial centroid and mathematically projects it back into the world frame. This robust geometric reasoning ensures high orientation accuracy despite the inherent sparsity and noise of the physical depth sensors.

To visualize the improvements of the algorithm, figure 6.5 presents a direct side-by-side comparison of the object detection bounding boxes. In the original image (6.5a), applying the simulated algorithm to real data results in wide and loose boxes. In contrast, the refined image (6.5b) clearly demonstrates the precision achieved by the new RANSAC-based geometric reasoning, showcasing tight, centered and deterministic bounding boxes.



(a) Original detection

(b) Refined detection

Figure 6.5: Side-by-side comparison of object detection and geometric refinement

6.3 Real-world case studies

With the reality gap bridged through the algorithmic refinements detailed in the previous section, the experimental campaign transitioned to the functional validation of the entire SDL framework. The primary goal of this phase is to demonstrate the end-to-end capabilities of the hybrid Cloud-Edge architecture, proving that the system can reliably translate high-level natural language commands and multimodal sensory data into safe, physical robotic interactions.

To evaluate the robustness of the complete Sense-Think-Act loop, the setup was subjected to a series of comprehensive experimental scenarios. These tests were explicitly designed to stress both the cognitive reasoning of the generative AI and the dynamic kinematic orchestration of the physical Franka manipulator.

The following subsections detail three specific experimental case studies. The first evaluates the system ability to interpret and execute explicit user commands for object manipulation and relocation. The second demonstrates full autonomy, testing the AI capacity to autonomously analyze real-time sensor telemetry and independently trigger and execute a watering task. Finally, the third case study explores non-standard scenarios, verifying whether the AI successfully detects workspace anomalies and enforces safety protocols by blocking any hazardous robot motion.

6.3.1 Autonomous object relocation

The first experimental scenario evaluates the system ability to process an explicit user command and physically relocate an identified target object. This test directly validates the integration between the cognitive intent extraction and the kinematic primitives developed for real-world manipulation.

The operational sequence was initiated by submitting a natural language command via the custom GUI, instructing the system to move an item within the workspace. The *ai_agent_node* aggregated the visual context and queried the Cloud LLM. As depicted in

figure 6.6, the model parsed the intent, identified the target from the perception data and outputted the `moving` mission along with the correct object ID.

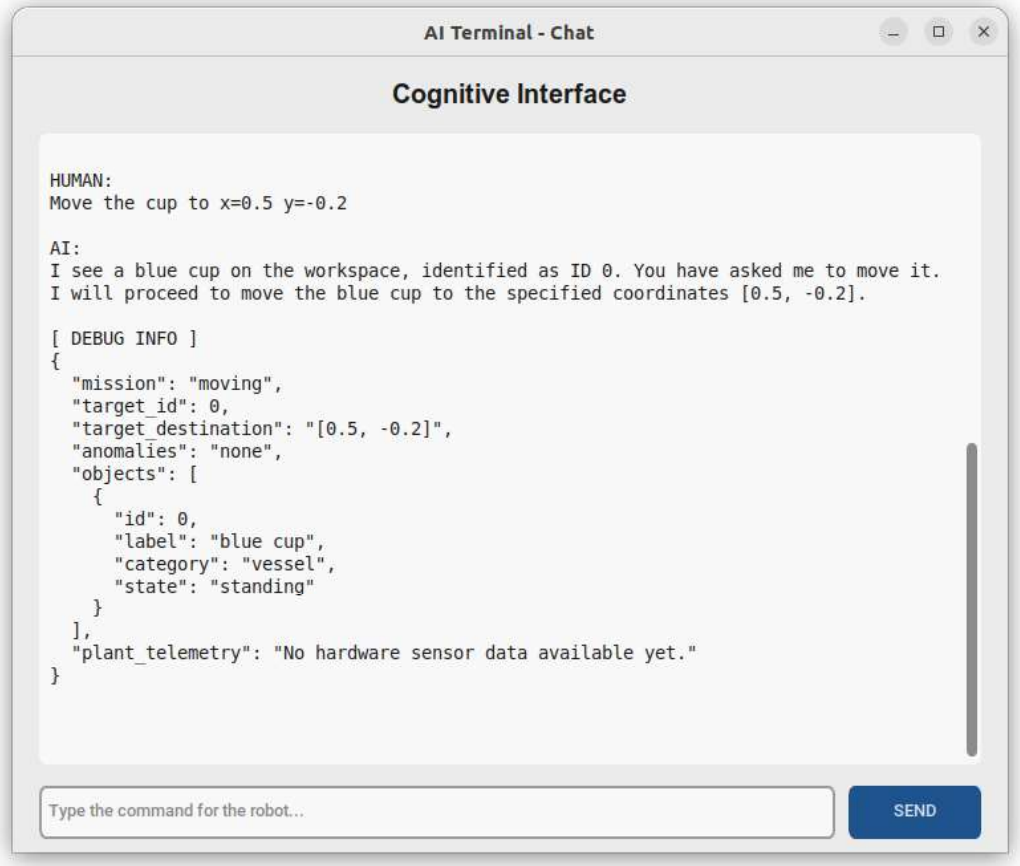
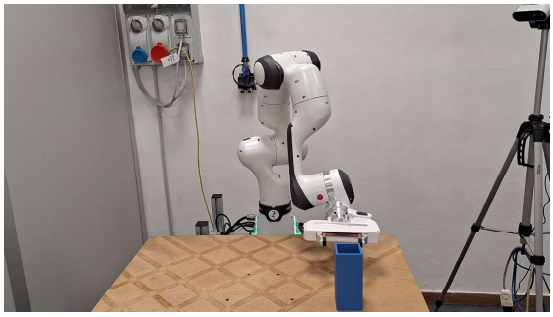
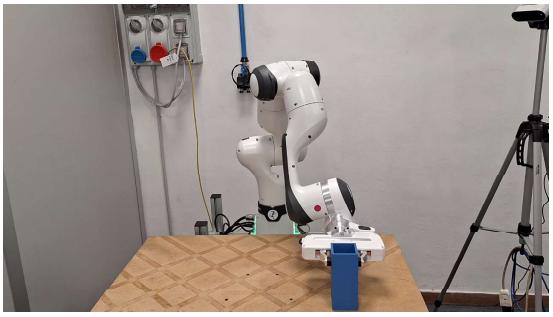


Figure 6.6: The GUI displaying the AI cognitive reasoning and the generated executive JSON for the relocation task

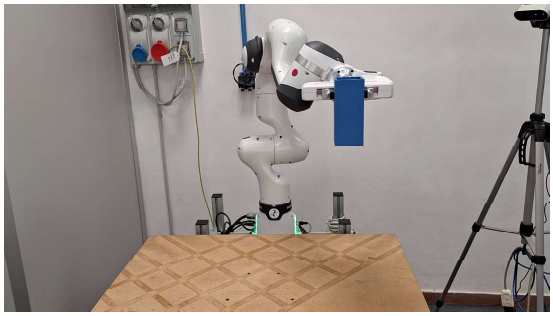
Figure 6.7 illustrates the complete execution of the task by the Franka manipulator, showcasing the transition between different kinematic primitives.



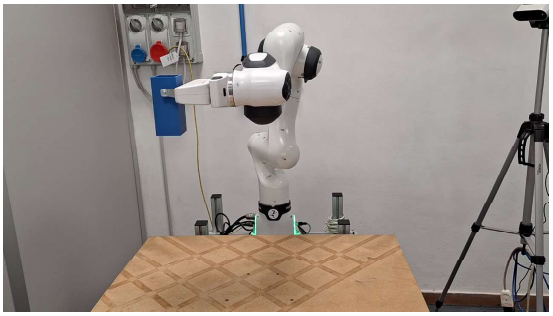
(a) Collision-free approach to pre-grasp waypoint



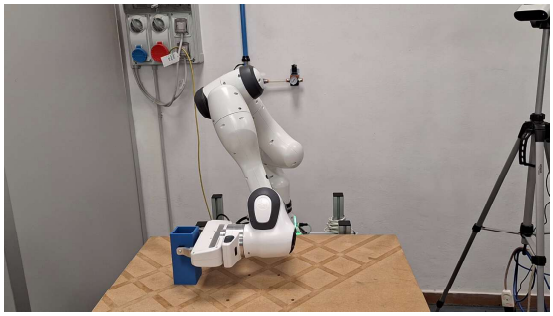
(b) Linear insertion and parallel grasping



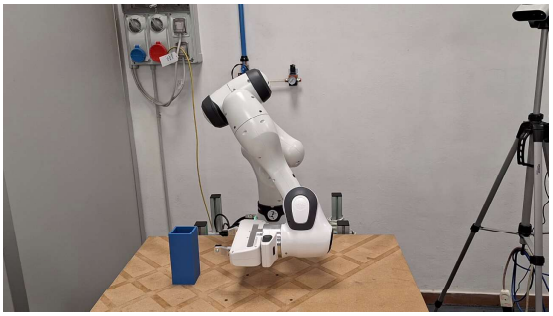
(c) Vertical extraction



(d) Transit to target coordinates



(e) Controlled descent and payload release



(f) Linear retreat

Figure 6.7: Detailed real-world execution of the autonomous relocation task by the Franka manipulator

The physical completion of this task demonstrates that the custom 2D RANSAC algorithm provides sufficient geometric accuracy to enable reliable grasping, while the motion planner effectively navigates the physical workspace without triggering hardware safety faults.

6.3.2 Context-aware plant care

The second experimental scenario elevates the complexity of the validation by testing the framework full autonomy and multi-object interaction capabilities. Unlike the previous task, which relied on explicit spatial commands, this scenario evaluates the system capacity to autonomously analyze physical hardware telemetry and trigger a biological care mission.

The sequence was initiated by a generic user prompt. Instead of executing a direct

command, the *ai_agent_node* aggregated the visual scene and the real-time serial data from the Arduino microcontroller. The Cloud LLM evaluated the telemetry, identifying a critical drop in the soil moisture percentage. As shown in figure 6.8, the AI generated a structured reasoning explaining the biological need and dispatched a `watering` mission, correctly identifying the target vessel ID from the visual context.

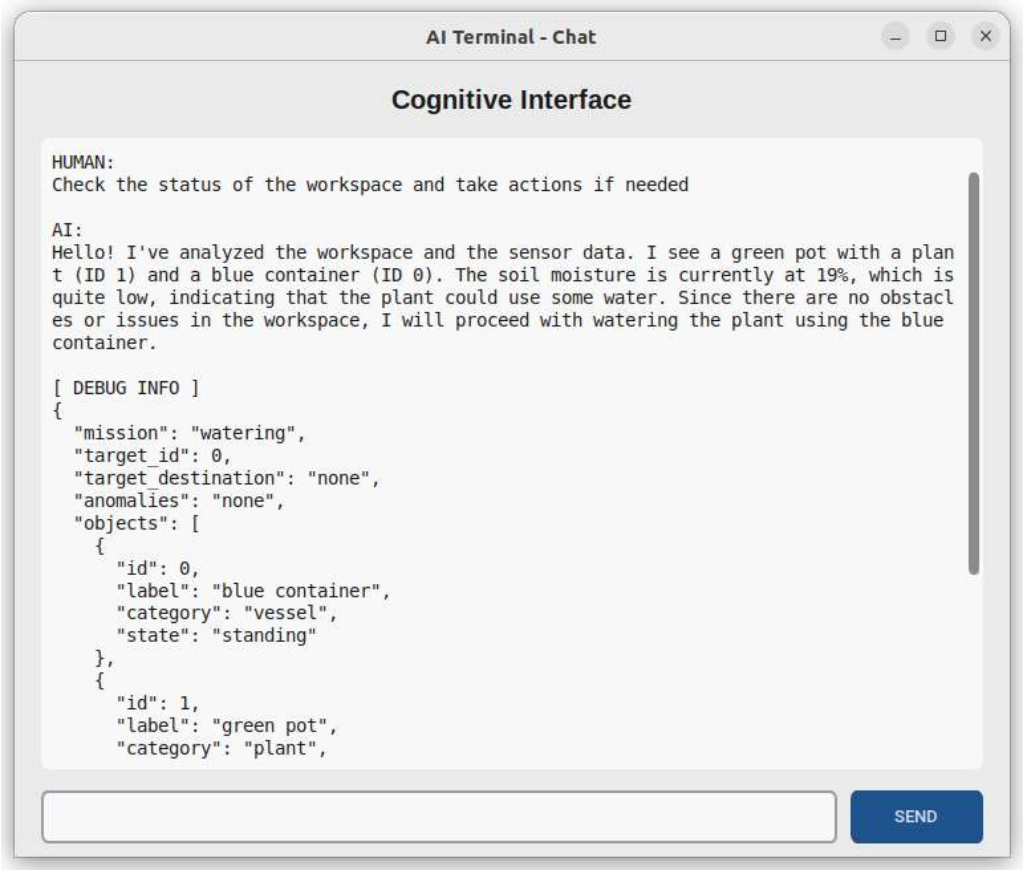
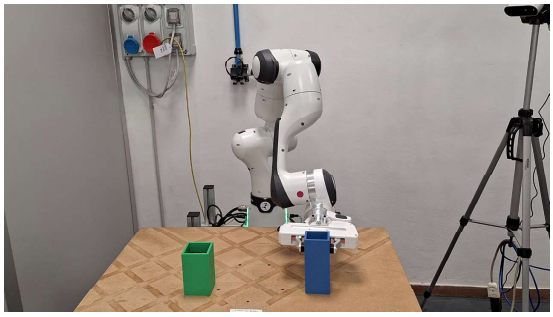
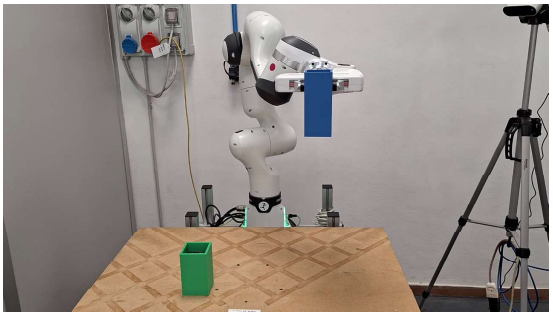


Figure 6.8: The GUI displaying the autonomous AI decision-making process triggered by low soil moisture telemetry

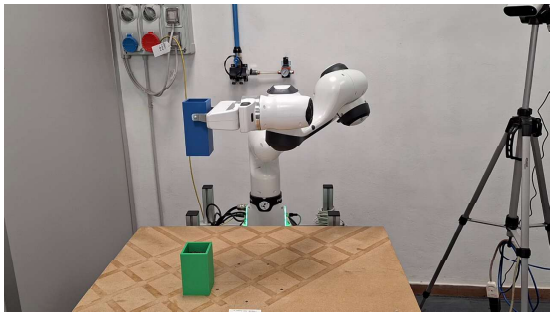
The physical execution of the watering sequence represents the most demanding kinematic challenge of the experimental campaign, requiring strict collision management between two semantic objects and dynamic constraints to prevent fluid spilling. Figure 6.9 focuses on the complex interaction phase between the two objects, illustrating the precise transition between different kinematic primitives.



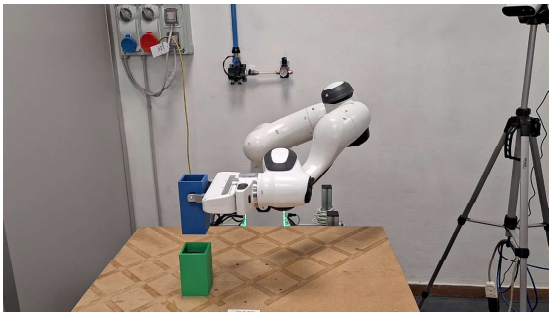
(a) Securing of the vessel



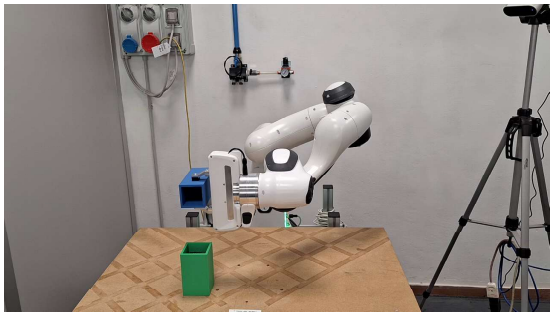
(b) Linear lift



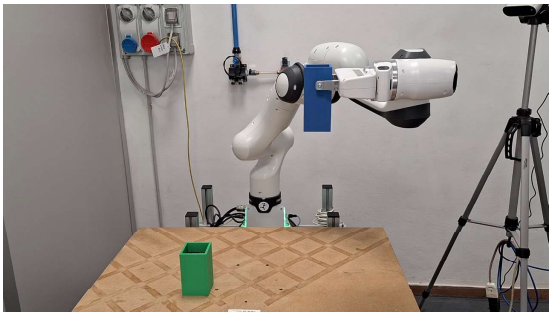
(c) Approach to the pre-watering waypoint



(d) Linear descent to the watering point



(e) Custom pouring via the 7th joint



(f) Vessel straightening and guarded retreat

Figure 6.9: Real-world execution of the watering task. Note: while the mission was triggered by live telemetry, the physical moisture sensor was temporarily removed from the pot during this specific photographic sequence to prevent hardware damage during repeated kinematic captures

The completion of this scenario validates the core hypothesis of the SDL framework: the architecture can bridge Cloud-based semantic reasoning and strict, low-level robotic control loops to autonomously perform delicate agricultural operations in the physical world.

To further validate the system robustness over extended experimental timescales, additional watering iterations were conducted after a period of biological growth. Figure 6.10 illustrates the manipulator operating on the evolved plant morphology, with the capacitive soil moisture sensor visibly integrated into the setup.

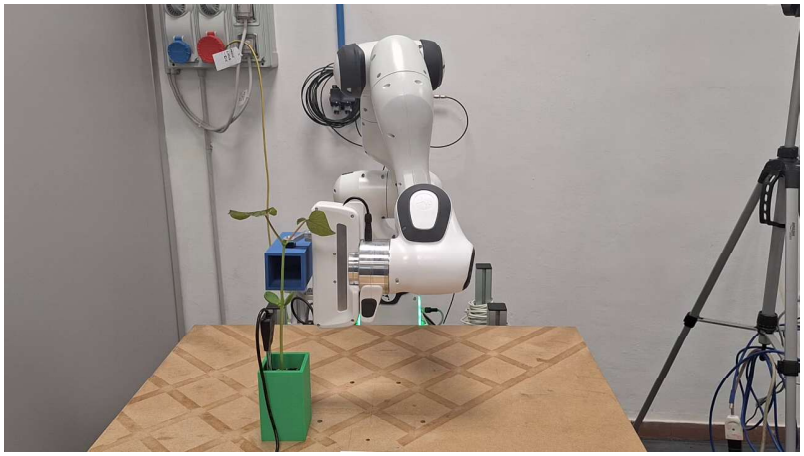
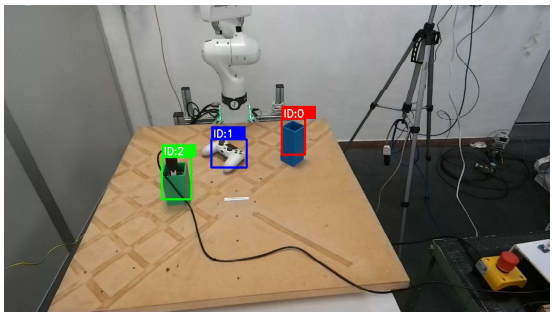


Figure 6.10: Fully-integrated autonomous researcher loop

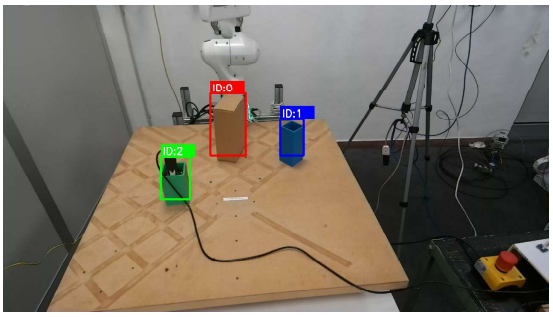
6.3.3 Non-standard scenarios and workspace anomalies

To conclude the experimental campaign, the framework fault tolerance and safety mechanisms were evaluated by exposing the system to non-standard scenarios and unexpected workspace anomalies.

During this testing phase various random items, such as a cardboard box, a gamepad controller and scattered electronic boards, were intentionally placed on the operational table alongside the standard targets. When a user command was issued, the local *perception_node* segmented these geometries, passing their spatial constraints and visual crops to the Cloud LLM.



(a) Detection of gamepad controller



(b) Detection of cardboard box

Figure 6.11: System detection to unmapped anomalies

As depicted in figure 6.12, the generative model classified the unfamiliar items as workspace anomalies overriding the task execution, outputting a safe state and returning an informative alert to the user.

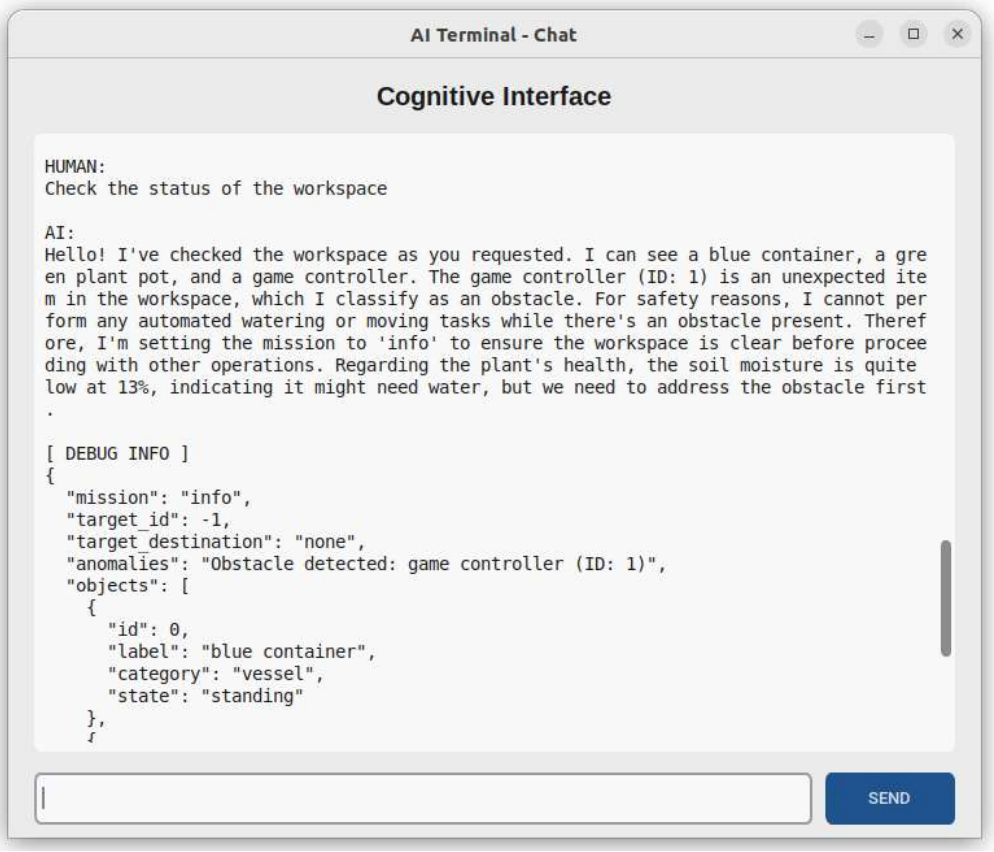
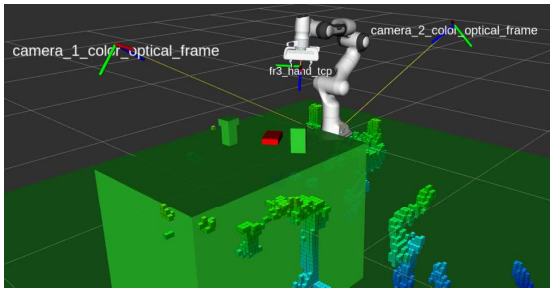
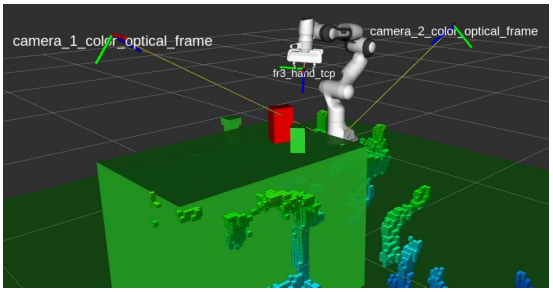


Figure 6.12: The GUI displaying the autonomous AI decision-making process triggered by the obstacle presence in the workspace

To physically enforce this safety override at the motion planning level, any object classified by the AI as a generic anomaly is dynamically injected into the MoveIt planning scene. As illustrated in figure 6.13, the local `SceneManager` renders these unidentified items as red collision bounding boxes within the RViz environment. This visual representation confirms that the robotic workspace is actively updated, explicitly segregating the operational areas from the unmapped clutter to guarantee collision-free states.



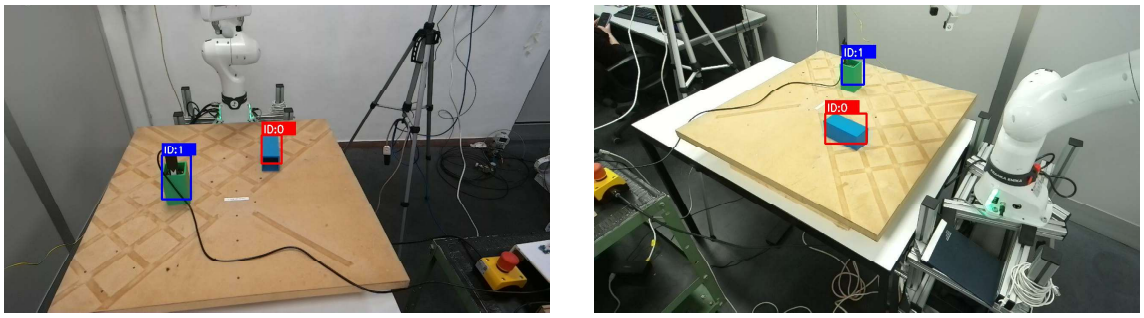
(a) RViz planning scene with the gamepad



(b) RViz planning scene with the cardboard box

Figure 6.13: Dynamic injection of unmapped anomalies as red collision bounding boxes in the RViz environment

An insightful edge case occurred when the standard watering vessel was intentionally laid horizontally on the table (figure 6.14). Due to the drastic change in its visual profile, the AI failed to recognize it as the designated target, classifying it instead as a generic `obstacle` (figure 6.15).



(a) Camera 1 view (b) Camera 2 view

Figure 6.14: Visual profile of the watering vessel intentionally laid horizontally on the workspace

This misclassification occurs because the Cloud LLM processes each user request statelessly. Lacking an episodic memory of previous interactions, the cognitive agent evaluates the scene entirely from scratch; therefore, it cannot infer that the newly detected geometry belongs to the custom cup used in earlier tasks.

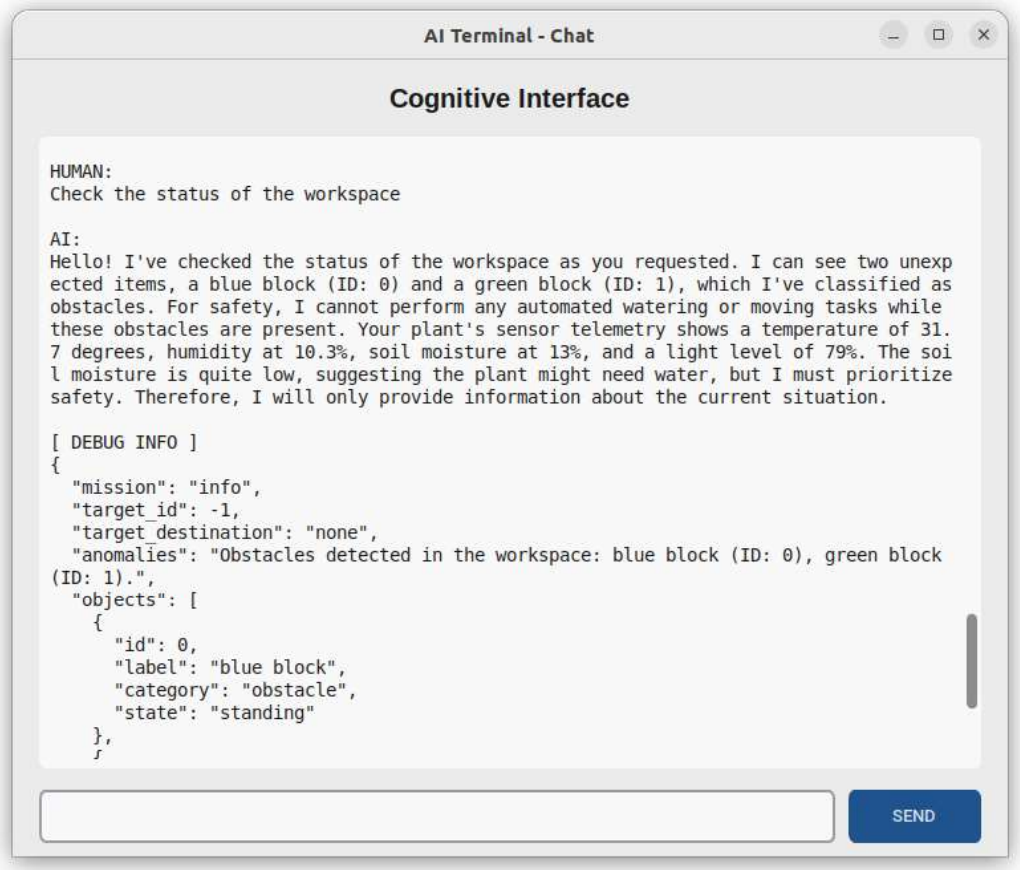


Figure 6.15: The GUI displaying the AI failing to recognize the laid cup, classifying it as an obstacle and aborting the mission

While this highlights a perceptual limitation regarding severe orientation changes, it demonstrates the system "Fail-Safe" design. By defaulting to an obstacle classification, the SDL framework wrapped the item in a collision bounding box and actively aborted the grasping mission. This prevented the motion planner from attempting an improper execution that would have caused hardware collisions or gripper faults.

These tests validate the safety and cognitive robustness of the hybrid architecture. The system evaluates the physical feasibility of the environment and enforces safety boundaries, completely avoiding the execution of routines under anomalous conditions.

6.3.4 Advanced cognitive diagnostics and corrective reasoning

To conclude the experimental validation, the cognitive limits of the LLM were tested beyond mission execution and basic fail-safes. While the previous scenarios evaluated physical manipulation and safety overrides, an advanced SDL framework must also serve as a diagnostic agent. To assess these capabilities, specific cognitive queries were submitted without requesting immediate physical movement.

The first phase evaluated the environmental diagnostic capacity of the system. When

provided with non-optimal telemetry conditions (low soil moisture and humidity), the model correlated the numerical data with water stress and advised immediate hydration (figure 6.8). Conversely, when tested under optimal parameters, the AI evaluated the environment as healthy, confirmed that no immediate physical actions were required and autonomously maintained an observational state (figure 6.16).

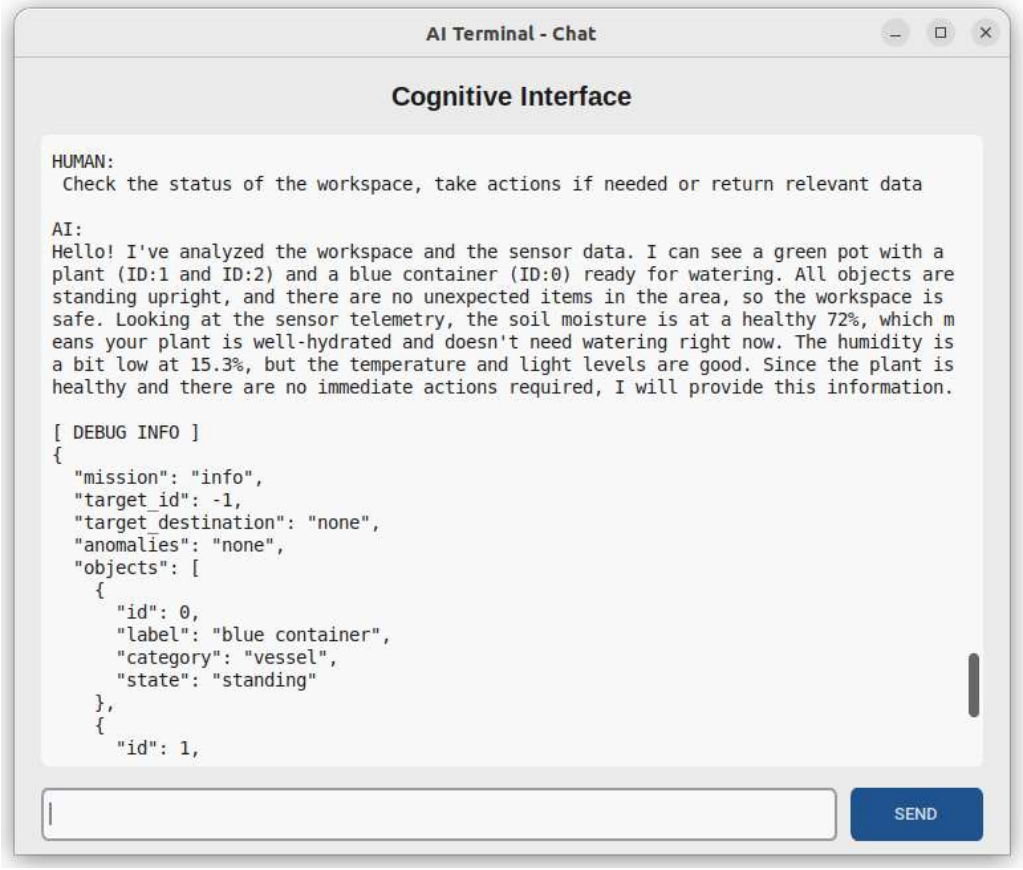


Figure 6.16: The GUI displaying the AI diagnostic reasoning under optimal telemetry conditions

This analysis highlighted a specific perceptual limitation. When explicitly prompted to identify the exact plant species, the LLM defaulted to a generic classification, stating its inability to determine the botanical family (figure 6.17). This limitation can be attributed to the physical sensor deployment: the RGB-D cameras are positioned to maximize the volumetric coverage of the workspace for kinematic collision avoidance, lacking the resolution and close-up perspective required for fine botanical feature extraction. The vision-language model possesses the inherent capacity for botanical classification, but is constrained by the quality of the input data.

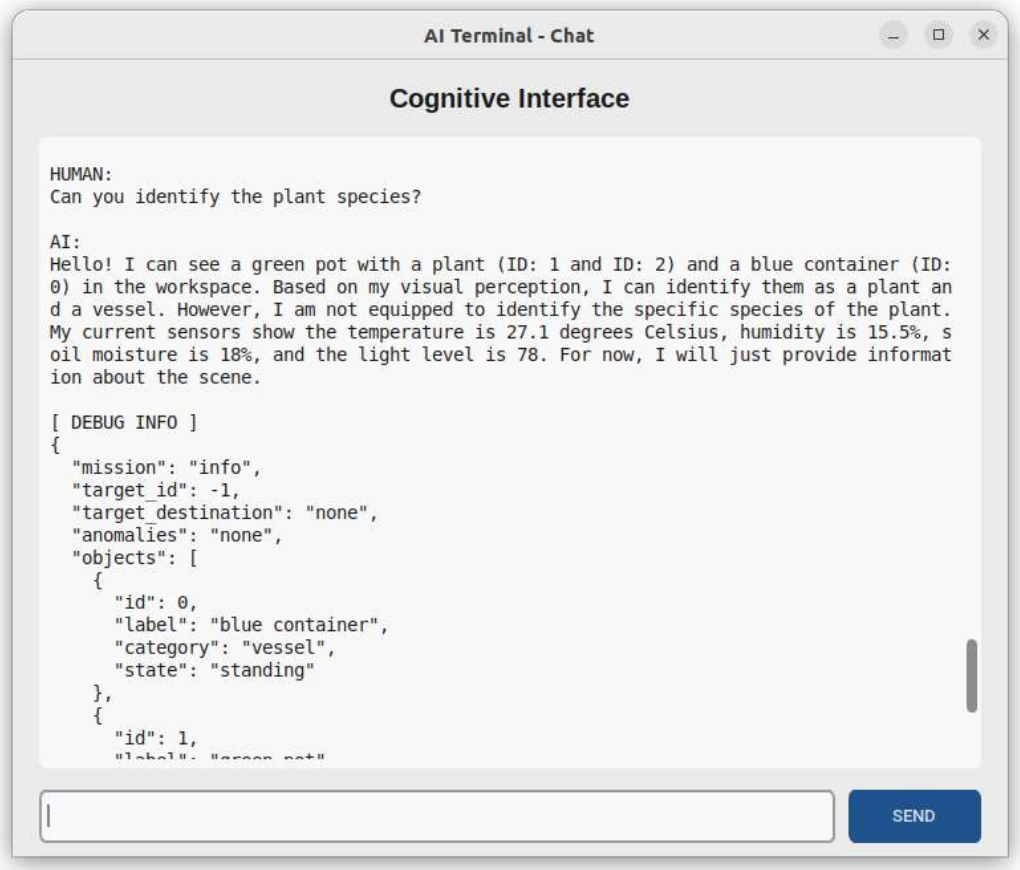


Figure 6.17: The GUI displaying the AI diagnostic reasoning when explicitly prompted to identify the plant species

The second testing phase focused on proactive corrective logic. Building upon the anomaly detection scenario discussed in the previous section, the cognitive orchestrator was explicitly queried to suggest corrective actions regarding an unmapped "game controller" obstacle blocking the operational area. As detailed in the system output (figure 6.18), the generative model did not merely abort the execution, but performed a complete prioritization assessment. It analyzed the entire workspace, read the telemetry data (noting a 19% soil moisture) and inferred the biological need for watering. However, adhering to its safety protocols, the AI prioritized the physical anomaly over the agricultural task. It blocked the automated execution, defaulted the system to a safe `info` mission and presented the human operator with specific resolution pathways: manual removal of the object or the issuance of an explicit command to have the manipulator relocate the hazard.

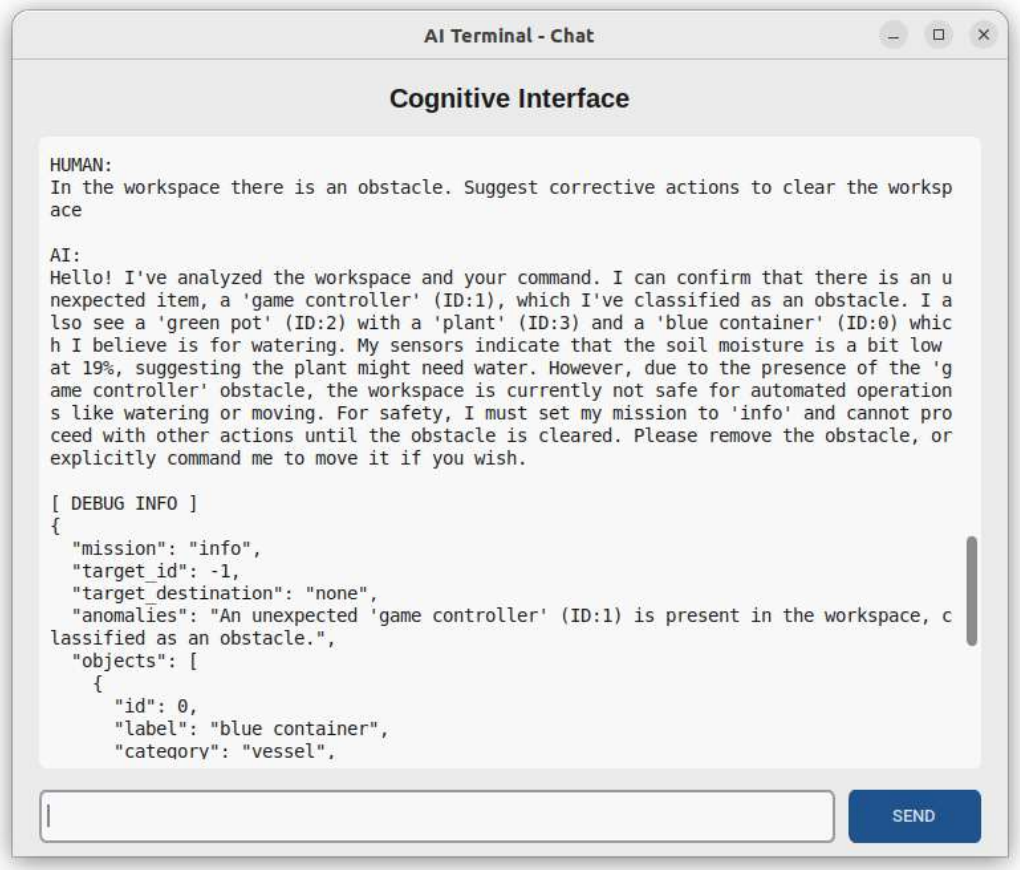


Figure 6.18: The GUI displaying the AI diagnostic reasoning when prompted to suggest corrective actions to clear the workspace

This response demonstrates how the system prioritizes physical safety over agricultural tasks: by halting the execution, the architecture prevents potential hardware collisions. Furthermore, the AI suggestion to relocate the obstacle outlines a path for future developments. Instead of pausing to wait for human intervention, this reasoning provides the baseline for an autonomous workspace recovery function. Future iterations could leverage this logic to automatically plan a relocation mission, clear the operational area from unmapped clutter and subsequently resume the main tasks. These diagnostic results validate the Cloud-Edge integration and provide the foundation for future developments discussed in the concluding chapter.

7. Conclusions and future works

The transition of robotics from rigid industrial automation to unstructured, dynamic environments represents one of the most complex challenges in modern engineering. This thesis addressed this transition by investigating the integration of cognitive Artificial Intelligence (AI) with low-level kinematic control, focusing on the autonomous management of biological specimens within a Self-Driving Lab (SDL).

Handling morphologically evolving and non-cooperative targets, such as plants, exposes the limitations of traditional automation. To overcome the problem, this work developed and validated a LLM-driven robotic framework capable of bridging high-level semantic reasoning with precise physical manipulation.

7.1 Summary of contributions

The core contribution of this work is the design and implementation of a hybrid Cloud-Edge robotic architecture. By decoupling cognitive decision-making from geometric spatial planning, the system maximizes the strengths of both computational paradigms while mitigating their respective bottlenecks.

Through the comparative analysis conducted in the simulated environment, this study demonstrated that, while modern LLMs exhibit strong capabilities in natural language understanding and contextual reasoning, they lack the spatial determinism required for safe kinematic planning. Consequently, the architecture delegates the semantic interpretation of the scene, analyzing user intents, processing RGB visual context and evaluating real-time IoT hardware telemetry to a Cloud-based generative model. Simultaneously, the heavy computational tasks requiring low latency and high precision, such as multi-camera point cloud processing, geometric pose estimation and trajectory generation via MoveIt, are strictly confined to the local ROS2 Edge network.

The robustness of this SDL framework was validated through an experimental campaign using a 7-DoF Franka Research 3 manipulator. The system successfully executed the complete "Sense-Think-Act" loop in real-world conditions, overcoming inherent sim-to-real challenges such as sensor noise, infrared interference and strict hardware safety limits.

The experimental case studies confirmed two major operational achievements:

- context-aware autonomy: the framework demonstrated the ability to act as an

autonomous researcher. By interpreting a drop in soil moisture telemetry, the cognitive layer independently deduced the biological need of the plant and orchestrated a complex, dual-object watering sequence without requiring explicit human spatial commands;

- cognitive safety and anomaly handling: the system overcame the "blindness" of rigid automation. When exposed to unmapped objects or morphological changes, the LLM identified the workspace anomalies. Acting as a high-level safety supervisor, it triggered a "Fail-Safe" override, injecting collision boundaries into the planning scene and aborting hazardous motions, guaranteeing hardware integrity.

In conclusion, this thesis proves that integrating an LLM as the cognitive core of a physical agent is not only feasible but highly effective. By coupling a structured vocabulary of motion primitives with an on-demand multimodal context aggregation triggered by user requests, the manipulator evolves from a rigid deterministic tool into an intelligent and adaptive assistant capable of safely managing non-cooperative laboratory environments.

7.2 System limitations

While the experimental validation demonstrated the viability of the proposed architecture, deploying a Cloud-based LLM as a physical robot controller highlighted several intrinsic limitations regarding operational latency and task rigidity.

The most prominent limitation is the stateless nature of the cognitive layer: the Cloud LLM evaluates each user request independently, entirely lacking an episodic memory of previous interactions. Consequently, the system suffers from contextual blindness over time. As demonstrated in figure 6.15, if the robot uses a blue watering vessel, but the object falls over, the AI evaluates the scene from scratch. Lacking a temporal context, it fails to recognize the object identity through its morphological change, classifying it instead as a generic unmapped obstacle.

Furthermore, the system lacks a persistent spatial memory: once an object is placed, the architecture relies entirely on the visual *perception_node* to find it again during the next cycle, rather than "remembering" its previously assigned coordinates.

A second critical bottleneck is the Cloud dependency and operational latency. Relying on an external API introduces a non-deterministic latency into the Sense-Think-Act loop, highly dependent on the internet bandwidth. In a strictly controlled laboratory environment, this dependency not only prevents real-time continuous feedback loops but also poses significant data privacy and security concerns, as sensitive experimental images and telemetry must be transmitted to external servers.

Finally, the current framework exhibits a rigid task orchestration. While the LLM acts autonomously in deciding what mission to execute (e.g., `watering`), the actual sequence of that mission (`grasp`, `transit`, `pour`, `place`) is hard-coded into the logic of the

mission_planner_node. The generative model is currently acting as a high-level classifier and intent extractor, rather than a true motion orchestrator.

7.3 Future developments

To overcome these limitations and fully realize the potential of SDLs, future works will focus on transitioning the architecture from a reactive, stateless framework to a fully autonomous and dynamic robotic researcher. The main development includes:

- integration of Local Edge AI and RAG: to eliminate internet dependency, reduce latency and guarantee absolute data privacy, the Cloud API will be replaced with a localized, quantized Vision-Language Model running on a dedicated Edge GPU. Furthermore, integrating a RAG architecture will allow the local agent to access a database of past laboratory experiments, effectively learning from its previous physical execution errors and iteratively optimizing its strategies;

- continuous workspace monitoring: unlike the current event-driven architecture, where the cognitive layer is triggered on-demand by specific user prompts or discrete telemetry checks, future iterations will implement a continuous perception-action loop. By persistently monitoring the experimental environment, the system will evolve into an active supervisor. This will enable the AI to autonomously detect spontaneous biological events, unmapped physical anomalies or safety hazards in real-time, proactively initiating corrective missions or data collection without requiring any human initiation;

- dynamic task generation and orchestration: instead of relying on hard-coded sequences for complex missions, the LLM will become the true orchestrator of the robot "body". The system will provide the AI with a vocabulary of highly modular aggregate primitives (e.g., `perform_grasp`, `perform_pour`, `perform_place`). The AI will autonomously assemble and sequence these blocks based on the goal, writing the execution logic rather than simply triggering a pre-packaged script;

- multi-agent diagnostic architecture: to enhance system robustness, a secondary "Diagnostic Agent" will be introduced alongside the primary operational AI, being deeply integrated with the ROS2 diagnostic streams and the system codebase. If a physical execution fails or the ROS2 network crashes, the diagnostic agent will autonomously read the terminal errors, cross-reference them with its knowledge of the code and assist the human operator in debugging the software or resolving the physical fault;

- advanced error recovery and workspace clearing: building upon the current "Fail-Safe" logic that aborts missions when an obstacle is detected, future iterations will implement an active recovery. The geometric pose estimation algorithms for laid and fallen

objects will be finalized, allowing the addition of a specific `pick_and_straighten` mission. This will enable the robot to autonomously recover overturned targets without requiring human intervention. Furthermore, this logic could be extended to manage generic workspace clutter: by evaluating the physical feasibility of a grasp, such as matching the obstacle dimensions against the maximum gripper stroke and identifying valid contact points, the perception system could authorize the autonomous relocation of unmapped items to clear the operational area;

- contextual and episodic memory: the cognitive layer will be upgraded to maintain a persistent state of the workspace. By implementing an episodic memory, the AI will track objects across multiple interactions. This will solve the statelessness issue: if a known vessel is knocked over, the context memory will inform the AI of its true identity, preventing misclassification and enabling dynamic spatial tracking of items even when temporarily occluded.

Bibliography

[1] Bruno Siciliano et al. *Robotics: modelling, planning and control*. Springer, 2009.

[2] Mariusz Olszewski. "Modern industrial robotics". In: *Pomiary Automatyka Robotyka* 24.1 (2020), pp. 5–20.

[3] Bo Ai et al. "A review of learning-based dynamics models for robotic manipulation". In: *Science Robotics* 10.106 (2025), eadt1497.

[4] Shi Xuan Leong et al. "Steering towards safe self-driving laboratories". In: *Nature Reviews Chemistry* 9.10 (2025), pp. 707–722.

[5] Benjamin P MacLeod et al. "Self-driving laboratory for accelerated discovery of thin-film materials". In: *Science Advances* 6.20 (2020), eaaz8867.

[6] Kourosh Darvish et al. "ORGANA: A robotic assistant for automated chemistry experimentation and characterization". In: *Matter* 8.2 (2025).

[7] MoveIt Project. *MoveIt: ROS 2 Motion Planning Framework*. 2026. URL: https://moveit.ros.org/ (visited on 03/02/2026).

[8] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[9] E. Rohmer, S. P. N. Singh, and M. Freese. "CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework". In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. www.coppeliarobotics.com. 2013.

[10] Franka Robotics GmbH. *Franka Research 3 datasheet*. Version 2.2.1. Franka Robotics GmbH. 2025. URL: https://franka.de/hubfs/Datasheet%20Franka%20Research%203_R02212_2.2.1_EN.pdf (visited on 03/02/2026).

[11] Francis J Pierce and Peter Nowak. "Aspects of precision agriculture". In: *Advances in agronomy* 67 (1999), pp. 1–85.

[12] Yongliang Qiao et al. *AI, sensors and robotics in plant phenotyping and precision agriculture*. 2022.

[13] Luiz FP Oliveira, António P Moreira, and Manuel F Silva. "Advances in agriculture robotics: A state-of-the-art review and challenges ahead". In: *Robotics* 10.2 (2021), p. 52.

[14] Andrea Botta et al. "A review of robots, perception, and tasks in precision agriculture". In: *applied mechanics* 3.3 (2022), pp. 830–854.

[15] Chao Cheng et al. "Recent advancements in agriculture robots: Benefits and challenges". In: *Machines* 11.1 (2023), p. 48.

[16] Franka Emika GmbH. *Franka Hand datasheet*. Version 2.2.1. Franka Emika GmbH. 2022. URL: https://download.franka.de/documents/220010_Product%20Manual_Franka%20Hand_1.2_EN.pdf (visited on 03/02/2026).

[17] Jiaqi Wang et al. "Large language models for robotics: Opportunities, challenges, and perspectives". In: *Journal of Automation and Intelligence* 4.1 (2025), pp. 52–64.

[18] Sai H Vemprala et al. "Chatgpt for robotics: Design principles and model abilities". In: *Ieee Access* 12 (2024), pp. 55682–55696.

[19] Ruaridh Mon-Williams et al. "Embodied large language models enable robots to complete complex tasks in unpredictable environments". In: *Nature Machine Intelligence* 7.4 (2025), pp. 592–601.

[20] Wenlong Huang et al. "Voxposer: Composable 3d value maps for robotic manipulation with language models". In: *arXiv preprint arXiv:2307.05973* (2023).

[21] Xubin Wang and Weijia Jia. "Optimizing edge AI: A comprehensive survey on data, model, and system strategies". In: *arXiv e-prints* (2025), arXiv–2501.

[22] Mayank Arya and Yogesh Simmhan. "Understanding the performance and power of LLM inferencing on edge accelerators". In: *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2025, pp. 1108–1111.

[23] Peter David Fagan. *MoveIt 2 Python Library: A Software Library for Robotics Education and Research*. 2023. URL: https://github.com/moveit/moveit2/tree/main/moveit_py.

[24] Orbbec Inc. *Femto Mega Datasheet*. Orbbec Inc. 2023. URL: https://d1cd332k3pgc17.cloudfront.net/wp-content/uploads/2023/04/ORBBEC_Datasheet_-Femto-Mega.pdf (visited on 03/29/2026).

[25] Orbbec 3D Technology International, Inc. *OrbbecSDK_ROS2: Official ROS 2 Wrapper for Orbbec 3D Cameras*. https://github.com/orbbec/OrbbecSDK_ROS2. Accessed: March 2026. 2024.