Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.a. 2024/2025

Graduation Session July 2025

Natural Language Control of Robotic Arms Using Large Language Models and ROS2

Supervisor: Guido Albertengo

Candidate: Miriam Ivaldi

Acknowledgements

Table of Contents

List of Figures

Chapter 1

Introduction

In recent years, the fusion of natural language processing and robotic control has paved the way for more intuitive and user-friendly interfaces in robotics. In particular, Vision-Language Action (VLA) models have allowed systems to understand the user's intentions expressed in natural language and convert them into executable robotic behaviors. In this thesis, we present the development of an intelligent agent that uses such models to control a UR10e robotic arm equipped with a Robotiq 2F-140 gripper, using the RAI framework (RobotecAI) and a conversational interface based on a Large Language Model (LLM) via Ollama.

This work was carried out in the industrial context of Reply Concept and is based on a pre-existing RAI-based architecture, originally designed for the Franka Emika Panda robotic arm. The project therefore involved the adaptation and extension of this framework to effectively integrate with the UR10e robotic arm with Robotiq2f-140 gripper, focusing on command interpretation, motion planning and simulated execution within the Open 3D Engine (O3DE).

1.1 Goal

The main objective of this thesis is to develop a robotic control system capable of performing high-level tasks defined through natural language. Specifically, the goals are as follows.

- Integrating a VLA model using an LLM (via Ollama) within the RAI framework to interpret natural language instructions.
- To enable the robotic agent to autonomously generate and execute motion

trajectories based on user commands.

- Adapt existing RAI tools and control logic for compatibility with the UR10e and Robotiq 2F-140 hardware.
- To use simulation in O3DE for testing, visual feedback, and interaction in dynamic environments.
- To evaluate the performance of different LLMs—both multimodal and non-multimodal—in robotic task execution scenarios.

These objectives aim to demonstrate a robust foundation for interactive and adaptable robotic control systems with potential for future deployment in real-world applications.

1.2 Thesis Structure

The remainder of this thesis is organized as follows:

- Chapter 2: Background and Related Technologies Provides an overview of the core technologies, including the RAI framework, VLA models, the UR10e robotic platform, and the simulation tools.
- Chapter 3: Intelligent Agent Architecture Describes the internal structure of the agent, focusing on how natural language commands are parsed, interpreted, and converted into robotic actions.
- Chapter 4: Adapting RAI to UR10e + Robotiq Details the modifications made to the RAI system to support the new hardware configuration and the challenges encountered during integration.
- Chapter 5: Simulation and testing in O3DE Outlines the simulation setup, task scenarios, and evaluation criteria used to validate the system's performance.
- Chapter 6: Evaluation of Language Models Compares multiple LLMs (including multimodal ones) in terms of their effectiveness in guiding the robot based on natural language commands.
- Chapter 7: Discussion Reflects on the results, identifies limitations, and proposes directions for future work.

- Chapter 8: Conclusion Summarizes the contributions of the thesis and highlights potential industrial applications.

Chapter 2

Background and Related Technologies

This chapter describes the foundational technologies and frameworks underlying the implementation of the intelligent agent for robotic control: the RAI framework, robotic hardware, the simulation environment, and the conversational LLM pipeline using Ollama.

2.1 Robot Operating System (ROS 2)

ROS (Robot Operating System) is an open-source software development kit for robotics applications. ROS provides developers with a standard software platform that guides them from research and prototyping to implementation and production.

This system was introduced in 2007 and serves as a flexible framework for writing robot software. It is a collection of libraries and tools. Hobbyists, researchers, and engineers can contribute to this open-source platform through various communities. With easy access to the code, robots become accessible to more people around the world.

Contrary to what the acronym suggests, it is not an operating system: ROS runs on Linux Debian and Ubuntu. In fact, ROS serves as a middleware and provides a set of plug-and-play libraries to accelerate robot development.

ROS 2 bases its communication on the Data Distribution Service (DDS). It eliminates the dependency on a master node of its predecessor. This standard, combined with the ROS 2 intra-process API, provides users with a much improved

transmission mechanism.

ROS provides functionality to abstract hardware, device drivers, inter-process communication across multiple machines, tools for testing and visualization, and much more.

The core feature of ROS is the way the software runs and communicates, which allows you to design complex software without knowing how the hardware works. ROS provides a way to connect a network of processes (nodes) with a central hub. Nodes can run on multiple devices and connect to the hub in various ways.

The main ways to create the network are by creating services that can be requested or by defining publisher/subscriber connections to other nodes. Both methods communicate via specific message types. Some types are provided by the core packages, but message types can also be defined in individual packages [1].

In the context of this project, ROS 2 serves as the **communication** backbone between different system components:

- Sensor data generated in **O3DE simulation environment** is published over ROS 2 topics.
- The RAI agent, which is the intelligent system used in this project, listens to these data and sends commands through ROS to control the robot.
- Tools like MoveIt 2 are also inside the ROS system, and they use the target poses to calculate the robot's movements.
- RViz 2 is used to watch what the robot is doing, to see the sensor data and the motion plans, and to check that everything is working correctly.

These parts together show why ROS 2 is powerful: it is **modular**. This means you can change or update some parts (for example change the planner or add a new sensor) without breaking the rest of the system, because the communication between parts is very clear [1].
ROS 2 operates using a decentralized, message-passing architecture composed of **nodes** (independent computational units), **topics** (for asynchronous pub/sub messaging), **services** (for synchronous calls), and **actions** (for long-duration tasks with feedback). All these components are organized in packages. In this project, ROS 2 topics are used to send camera images from the simulation, services are used to control the gripper, and actions are used for moving the arm. Chapter 4 explains this system in more detail.

2.1.1 MoveIt 2

MoveIt 2 is the robotic manipulation platform for ROS 2, it provides high-level capabilities for robotic manipulators. It includes tools for:

- Inverse kinematics (IK)
- Collision checking
- Path planning (using OMPL, STOMP, CHOMP)
- Trajectory execution
- Task-space constraints

In this thesis, MoveIt 2 is responsible for **generating motion trajectories** for the UR10e arm in response to high-level commands issued by the LLM. These plans are validated against simulated environments in O3DE to ensure feasibility and safety.

MoveIt 2 also integrates tightly with ROS 2's TF (transform) system and can be used to visualize motion plans and collision objects, making it easier to debug and test robotic behaviors [2].

2.1.2 RViz 2

RViz 2 is a 3D visualization tool that helps to see and check the robot status, sensor data, and the environment in real time. It is a very useful tool when working with ROS 2, especially for simulation and testing.

With RViz it is possible to:

- See TF frames and the robot joint positions
- Check live camera images and point clouds
- Show collision objects, planned motions, and the position of the robot's end-effector
- Use interactive markers to set goals or help debugging

In this project, RViz is used mainly to show the robot's movements, to check that MoveIt generated correct trajectories, and to visualize the camera and sensor data from the O3DE simulation. This visual feedback helped to be sure that the RAI agent understood the environment and selected the correct commands [3].

2.1.3 ROS 2 Jazzy Distribution

The system used in this thesis is based on ROS 2 Jazzy Jalisco, which is the 13th official release of ROS 2, published in May 2024. Compared to the older versions like Humble or Foxy, Jazzy has many improvements, such as:

- new and better APIs that support modern C++ (like C++17 and C++20),
- better performance and memory usage in `rclcpp` and `rclpy`,
- more stable support for real-time execution and node scheduling,
- and better compatibility with new hardware and simulators.

Jazzy also brings better security, more powerful CLI tools, and improved launch systems. For this project, Jazzy was a good choice because it works well with both MoveIt 2 and RViz 2, and it is supported by the ROS and O3DE communities. Using Jazzy made sure that the software is compatible with recent libraries and will continue to work in the future. This helps to keep the system stable and easy to maintain [4].

2.2 RAI framework

RAI (RobotecAI) is a flexible agent framework designed for embodied AI systems in robotics, designed to integrate large language models with robotic software stacks like ROS2. It offers tools for perception, planning, and action execution in both real and simulated environments [5].

The RAI framework is built around three main concepts: Agents, Connectors, and Tools.

- **Agents** are not just normal controllers. They include the logic and intelligence of the robot (or a digital twin). An agent in RAI can understand language commands, see what happens in the environment, use its memory, and interact

with planning tools. This behavior is written in the Agent.run() method, which controls how the agent works step by step.

- **Connectors** are like bridges that connect the agent to other systems such as ROS 2, O3DE, or different types of sensors and actuators. Thanks to these connectors, the agent can receive real-time data (like images or LiDAR), send movement commands (for example, joint positions), and communicate through topics or services in ROS [5]. ROS 2 integration is native, enabling plug-and-play capabilities with existing robotic ecosystems.

- **Tools** are the callable modules or actions that an agent can use in response to natural language queries. These include perception pipelines (e.g., object recognition, semantic scene parsing), motion planning (e.g., via MoveIt 2), and state estimators. Tools are registered with the agent and exposed through LLM-based reasoning, meaning that a language model can "call" a tool to execute a function when formulating a response [5] [6].

One of the distinguishing aspects of RAI is its seamless integration with large language models (LLMs). Using natural language as the primary interface, RAI enables agents to interpret high-level human instructions (e.g., "Pick up the red block to the left of the green one") and autonomously decide which tools to invoke to fulfill the task. In the RAI framework, the LLM works like a central planner. It keeps calling tools, checking the results, and updating the task plan in a loop based on the conversation[5].

RAI also focuses on making the agent aware of its own body and limits. Using tools like `whoami`, the agent can understand its own shape, abilities, and constraints. It reads this information from files like the URDF, sensor descriptions, and kinematic models. This helps the agent make decisions that are possible in the real world and adapt them to the situation. This type of "self-awareness" can also improve how the LLM reasons, and it can support advanced techniques like Retrieval-Augmented Generation (RAG) or memory-based planning [5].

RAI has already been tested in different scenarios:

- A mobile robot doing navigation tasks in indoor environments, both real and simulated.

- A robotic arm in O3DE that grabs and places objects based on what it sees.

- An agricultural robot that plans tasks using tools.

These examples show how RAI connects high-level reasoning (using LLMs) with real robot control. It solves some of the problems of traditional robotic systems by using natural language and visual feedback, while still working with standard platforms like ROS 2 and MoveIt.

To sum up, RAI is a flexible and powerful framework for building smart and interactive robots. It allows agents to understand complex instructions, reason about their physical structure, and take action in dynamic environments.

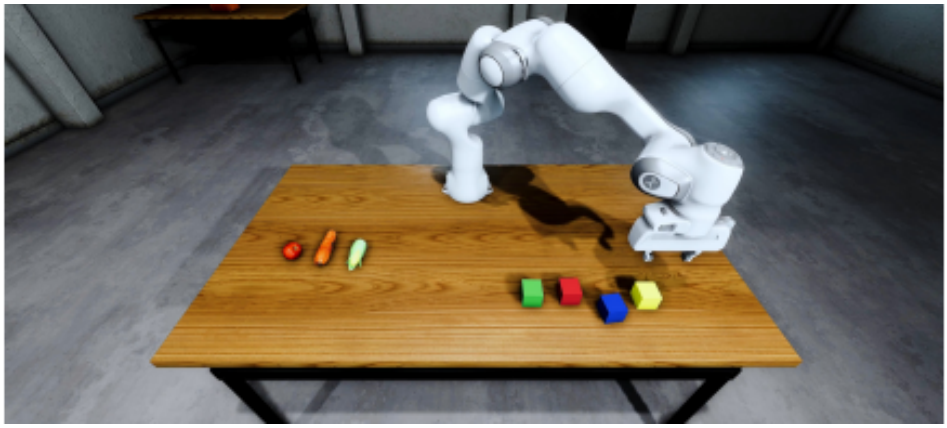


Figure 2.1: RAI framework demonstration using the Franka Emika Panda robot in O3DE. This setup served as a baseline reference for the development of the UR10e-based implementation presented in this thesis.

As shown in Figure 2.1, the original RAI demonstration integrates a Franka Panda arm in a simulated scene using O3DE and ROS 2. This project extended that foundation by adapting the framework to the UR10e robot.

2.2.1 Configuration Interface

To facilitate setup and user interaction, the RAI framework includes a web-based configuration interface built with Streamlit. This tool guides the user through all the steps required to initialize the agent, select models, and enable optional services such as speech recognition or tracing. The configuration wizard makes the system more accessible and helps avoid common setup errors.

The configuration is divided into intuitive steps: model selection, tracing, speech recognition (ASR), text-to-speech (TTS), and review. The final output is a structured configuration file that the RAI agent uses at runtime.

Figure 2.2 shows the welcome screen of the RAI Configurator, while Figure 2.3 displays the model selection interface where the user can choose a language model such as `qwen2.5:3b` for both simple and complex tasks.

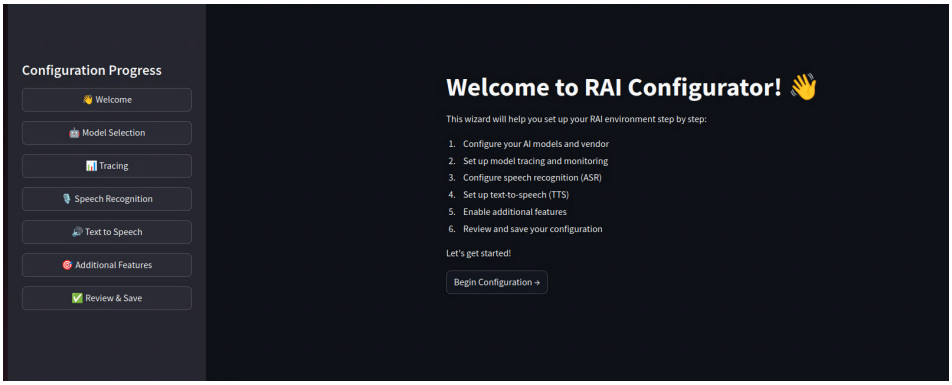


Figure 2.2: Start page of the RAI Configurator interface (Streamlit-based).

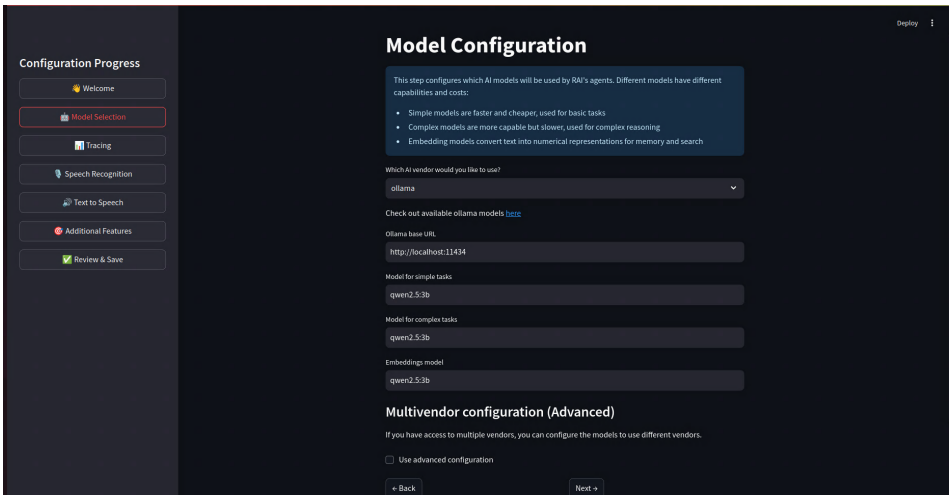


Figure 2.3: Model selection screen showing the use of the Qwen LLM through Ollama.

At the end of the process, the full configuration file is generated and shown (Figure 2.4), allowing the user to review or save it. Before completing the setup, a test step validates that all required components are accessible and properly configured. Figure 2.5 confirms that all modules passed the test successfully.

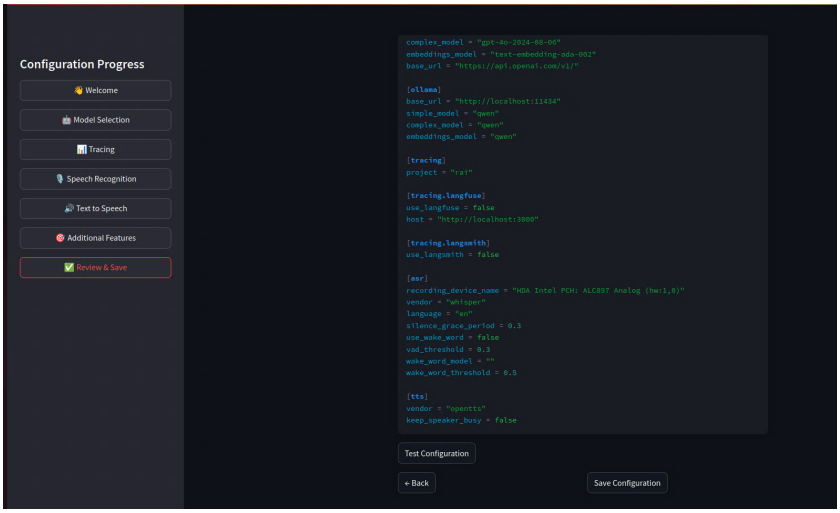


Figure 2.4: Final configuration file generated by the interface.

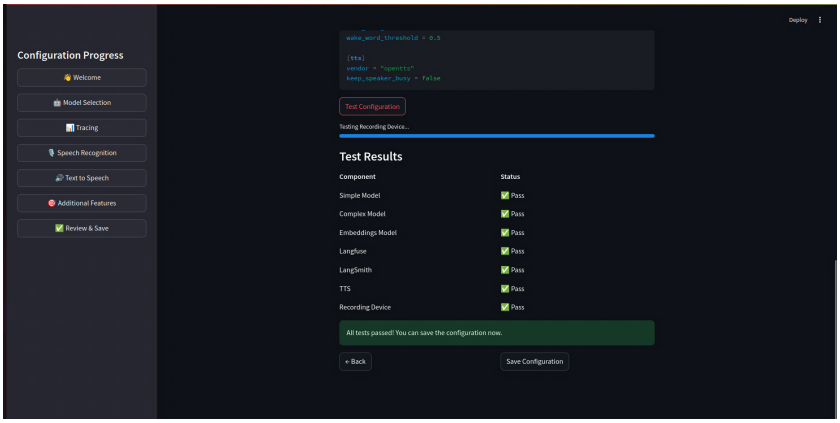


Figure 2.5: Validation screen confirming that all setup steps have passed.

2.3 UR10e Robotic Arm and Robotiq 2F-140 Gripper

UR10e

The UR10e is a 6-axis industrial robotic arm made by Universal Robots, and it is part of the e-Series. It is placed between the smaller UR5e and the bigger UR16e. The UR10e has a good balance between reach, weight it can carry, and

safety. It can lift up to 12.5 kg and has a reach of 1300 mm, so it is useful for tasks like manipulation in factories, warehouses, or research.

It also has force and torque sensors, some flexible settings, and a very precise movement with repeatability of 0.03 mm. The UR10e respects safety standards like ISO 10218-1 and PLd Category 3, so it can work close to people without needing protection barriers in most cases.

Another good thing is that the UR10e works well with ROS and is compatible with MoveIt, which makes it easier to use in research projects or simulations, like with the RAI framework or O3DE. [7] [8].

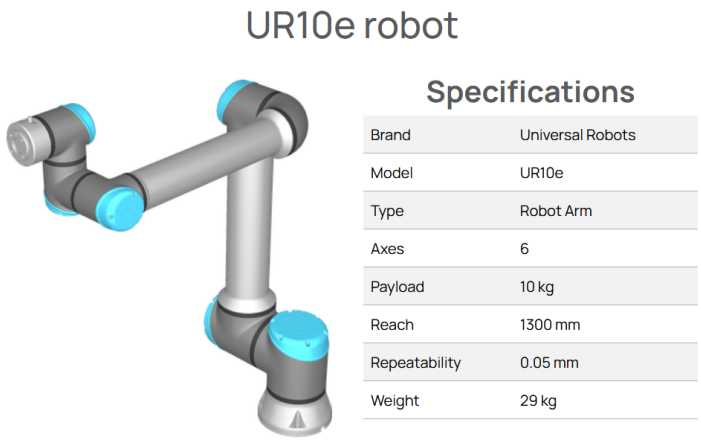


Figure 2.6: UR10e robot arm by Universal Robots, featuring 6 degrees of freedom, 1300 mm reach and 10 kg payload.

Robotiq 2F-140 Gripper

The Robotiq 2F-140 is a two-finger gripper that can adapt to different situations and is made for general robotic tasks. It has an adjustable stroke up to 140 mm, so it can take objects of different shapes and sizes — from small and delicate parts to bigger ones. It can hold up to 2.5 kg and lets you control position and speed with good precision. It also has internal sensors that help to check if the object was taken correctly.

One important feature is that it works in a plug-and-play way with Universal Robots, using the URCaps software, which makes it easier to program from the UR teach pendant. Because of its strong design and flexibility, the 2F-140 is often

used in industry and research when the robot needs to grasp different kinds of objects or do manipulation in dynamic environments [9].

Figure 2.7: Robotiq 2F-140 adaptive gripper. Two-finger design with a 140 mm stroke, suitable for diverse object sizes.

2.4 Vision–Language Action (VLA) Models

Vision–Language Action (VLA) models are a new type of system that combines natural language, visual input, and robot control. The idea is to connect the words used by humans with the real-world actions of a robot, by using images and spatial information to understand what the user wants. For example, a VLA model can understand a command like "put the red cube on the green platform" and convert it into robot movements that make sense depending on the current situation.

In classic robotics systems, perception, planning, and control are separated and programmed by hand. VLA systems are different—they use machine learning models trained on mixed data (text and images) to understand the scene and decide what to do. These models can work end-to-end or in a combination with traditional methods.

In this thesis, the VLA method is done by using a large language model (LLM) that runs locally with Ollama, inside the RAI agent. The LLM reads the natural language command, uses RAI tools to get information (like camera data from O3DE), and sends the correct commands to plan the robot's action. This way, the system is flexible and can adapt if the environment changes or the user gives a new task.

With live visual feedback from the O3DE simulation, the agent can check if the object is still in the same place, if the task was done correctly, or if something changed (like an object moved or is blocked). This feedback makes the robot

smarter—it can ask for more info ("which red block?") or try again if something goes wrong.

So, VLA models make robots easier to use, even for people who are not experts. They are also useful when the environment is not perfect, because they help manage uncertainty and complex situations. [5] [6].

2.5 O3DE for Robotic Simulation

The Open 3D Engine (O3DE) is a free and open-source real-time 3D engine. It was first created by AWS and now it is managed by the Linux Foundation. O3DE is made for high-quality graphics and has a modular system. It includes many tools for rendering, physics, animation, and simulation, so it's not only for video games, but also very useful for robotics and digital twin applications.

One of the key reasons why O3DE was used in this project is because it has native support for ROS 2 through the ROS 2 Gem. This allows direct connection with the Robot Operating System without using external bridges. This is different from other platforms like Unity or NVIDIA Isaac Sim, which need extra plugins or bridges to work with ROS. With O3DE, the simulation can directly publish and subscribe to ROS topics, send RGB and depth images, and share TF transforms and sensor data—all using normal ROS messages.

Thanks to the ROS 2 Gem, it's possible to import robot models written in URDF or XACRO files. This makes it possible to see the real joint movements, kinematics, and collisions inside the simulation. Also, developers can create interactive environments where the robot reacts to physics, moving objects, and dynamic elements—while keeping good timing that is important for testing planning and perception [10].

In this thesis, O3DE wa selected as the main simulator for the UR10e robot with the Robotiq 2F-140 gripper. The scene included 3D objects, a table, and a static camera. The camera sent RGB and depth data using ROS topics, and the RAI agent used that data for perception. This way, it was possible to test the full vision-language control system without a real robot [11].

O3DE was very helpful during development and integration, because it gave real-time visual feedback and good physics simulation. Since it's open-source and works well with ROS 2, it was a perfect choice for a flexible and realistic robotics simulator [12].

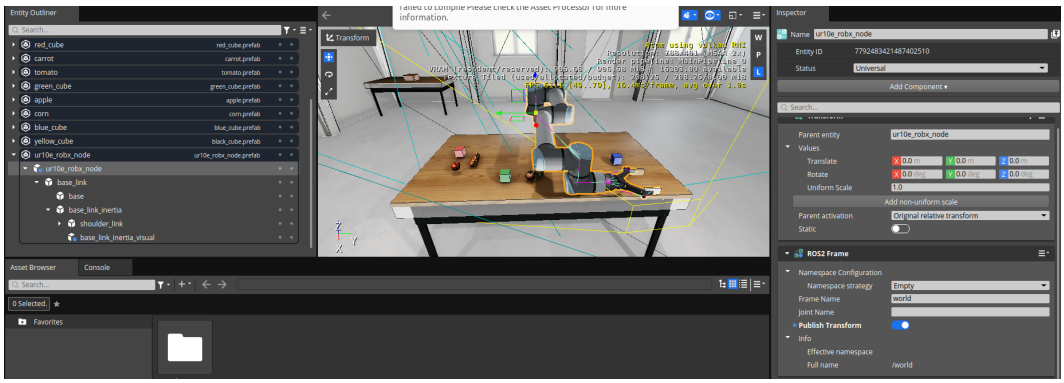


Figure 2.8: Screenshot of the UR10e robotic arm within the RAI-integrated O3DE simulation environment. The editor view shows the robot, objects on the table, and the ROS 2 transform and namespace configuration.

2.6 Conversational Agent via Ollama

Ollama is an open-source platform that helps to run large language models (LLMs) on your own computer. It is different from cloud-based AI services, which need internet all the time, user authentication, and sometimes have problems with speed or privacy. Ollama instead runs the LLMs locally, using the CPU or GPU, and this is very useful for robotics, where fast response and offline use are important.

The main idea in Ollama is the Modelfile. This is a configuration file that tells the system how to download, prepare, and start a specific LLM. The Modelfile includes the model weights, but also other information like the type of tokenizer, the backend used (for example, llama.cpp or GGML), and hardware limits. This design makes it easy to switch from one model to another (for example, from Mistral to LLaMA 2), and also helps to keep the system easy to update and to use on different machines or by different people.

Ollama provides a built-in REST API that allows external tools and agents—such as those written in Python, C++, or JavaScript—to interact with a model via

HTTP requests. This API is used in this thesis to connect the RAI Conversational Agent with a locally hosted LLM. In this architecture, natural language commands from the user are sent to the LLM through Ollama, which returns structured reasoning steps or tool invocation plans. The RAI framework then uses these outputs to trigger robotic actions or query sensor data, forming a closed control loop that is both conversational and reactive.

Ollama supports a large number of models. It works with:

- **Text-only language models** like LLaMA 2, Mistral, DeepSeek, Phi-2, and Gemma.
- **Multimodal models** Multimodal models that can also understand images, like Llava and BakLLaVA, which will be useful in the future for doing vision and language tasks directly on the device.

In this project, Ollama offers many advantages:

- **Privacy and reproducibility**: Everything runs locally, so no data goes to the cloud, and the results can be repeated on other machines.
- **Real-time performance**: The system replies fast, which helps for smooth communication with the robot.
- **Customizability**: Developers can change the prompt style, adjust how the model answers, and even switch models easily without changing all the code.

In general, Ollama is light and easy to use. It is a good choice for using LLMs in robotics, because it allows natural language understanding and works well with physical actions [13] [14] [15].

2.7 Dockerized Deployment of the RAI Robotic System

Modern robotic systems are becoming more and more complex because they use many different software tools, special hardware, and constantly changing middlewares. For this reason, it is not easy to make everything work the same way on different computers. To solve this problem, the robotics community started using

Docker. Docker is a tool that puts all the software and its environment inside a container that is light, separated from the rest of the system, and reliable.

Docker was first created for cloud applications, but its design is very flexible and works well also in robotics. With Docker, developers can run the same robotic system on different computers without problems, because everything is already prepared inside the container.

2.7.1 Fundamentals of Docker

Docker is a platform for defining and running containers

Containers are isolated processes for each of the components of a project. Each component runs in its own isolated environment, completely isolated from everything else on your machine [16].

Here's what makes them useful. Containers are:

- Self-contained. Each container has everything it needs to function with no reliance on any pre-installed dependencies on the host machine.
- Isolated. Since containers are run in isolation, they have minimal influence on the host and other containers, increasing the security of your applications.
- Independent. Each container is independently managed. Deleting one container won't affect any others.
- Portable. Containers can run anywhere! The container that runs on a certain development machine will work the same way in a data center or anywhere in the cloud [17].

Key Docker components relevant to robotics include:

- Images: read-only template with instructions for creating a Docker container [18];
- Containers: runnable instance of an image [17];
- Dockerfiles: text file containing instructions for building the source code. It automate the setup of environments [19];
- Volumes and Networking: persistent data stores for containers, essential for ROS-based workflows and simulation engines [20].

Docker offers clear advantages for robotics, such as dependency management because it ensures consistent setups across machines, reproducibility by simplifying testing and collaboration in research, isolation because it avoids software conflicts and hardware access enabling GPU acceleration via NVIDIA Container Toolkit [21].

In this thesis, a single Docker container was created to encapsulate the entire robotic development stack. It included ROS 2 Jazzy, MoveIt 2, O3DE, the RAI agent and its tools and Ollama.

Chapter 3

System Architecture

This chapter presents the overall architecture of the robotic system developed in this thesis. The system is designed to enable an intelligent agent to interpret natural language commands, perceive its environment, and execute physical tasks using a UR10e robotic arm with a Robotiq 2F-140 gripper. The architecture combines several advanced software components—including the RAI framework, a locally hosted LLM via Ollama, the ROS 2 middleware, the MoveIt 2 motion planning library, and the O3DE simulation engine—into a cohesive pipeline that enables high-level, language-driven robotic control.

3.1 System Components

The system is composed of the following interconnected modules:

- **User Interface** – Accepts natural language input in free text form.
- **LLM via Ollama** – A large language model running locally via Ollama, used to semantically interpret commands and propose structured actions [13] [5].
- **RAI Agent Framework** – It uses the LLM and connects it with the tools for perception, planning, and control [5].
- **RAI Tools** – These are small modules that the agent can use to see the scene, plan motions, and execute robot actions.
- **ROS 2** – It is the communication system that connects everything using topics, services, and actions [1].

- **MoveIt 2** – It helps with robot motion: it calculates trajectories, checks for collisions, and solves kinematics [2].
- **O3DE** – It is the 3D simulator where the robot acts. It gives the world, the camera input, and physics simulation [10].
- **RViz 2** - A tool to see what the robot is doing in real time: its position, trajectories, and sensor data [3].

Each of these modules plays a specific role in a **perception–cognition–action loop**: commands are interpreted by the LLM, reasoning and planning are handled by the RAI agent, and execution is performed through ROS-integrated simulation.

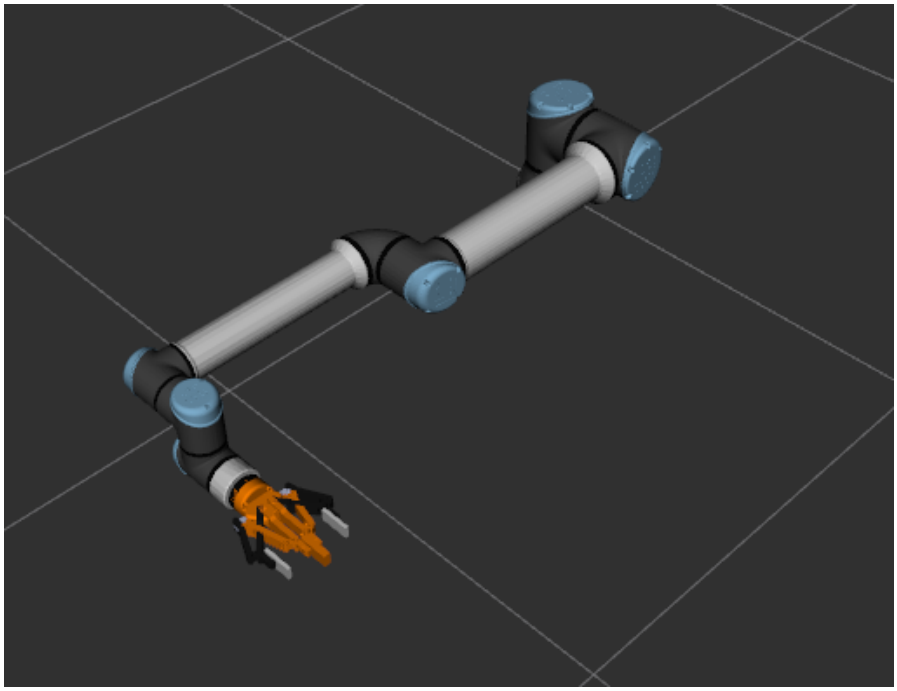


Figure 3.1: The UR10e robotic arm visualized in RViz 2. This interface is used to monitor joint states, trajectories, and coordinate frames in real time during execution.

3.2 Functional Flow

The control pipeline can be summarized in six main steps:

1. **Command Input**
   The user issues a natural language instruction, e.g., "Place the red cube on the green platform."

2. **Semantic Interpretation**
   The text from the user is sent to the LLM that runs locally with Ollama. This model understands what the user wants and changes the command into a specific tool call—for example, calling `get_object_positions("red cube")` and then `move_to_point(x, y, z, task="grab")`.

3. **RAI Agent Reasoning and Tool Invocation**
   The RAI agent receives the result from the LLM and decides which tools to use. It chooses the right tool for perception or manipulation, depending on the goal. The communication happens using ROS 2, and the modular system helps the agent to work with different robot setups.

4. **Perception and Scene Understanding**
   When the agent calls `get_object_positions`, it tries to find the object in the environment. Inside, this tool uses `GetGrabbingPointTool`, which reads the RGB image, the depth image, and camera info coming from the O3DE simulator. Then, it uses Grounding DINO to detect the object by the name given in natural language, and Grounded SAM to create a mask of that object. After that, it uses the camera calibration and the depth data to create a 3D point cloud. It calculates the center of the object (centroid) and converts it into the robot's coordinate frame using TF2. This final 3D pose is used later to plan the robot's movement.

5. **Motion Planning and Execution**
   After the position of the object is found, the agent uses the `move_to_point` tool. This tool sends a request to the `ManipulatorMoveTo` service, with a 3D position and the type of task (like "grab" or "drop"). Then, the robot's motion planner (using MoveIt 2 or similar system) calculates the correct trajectory. The movement is done using ROS 2, and the gripper opens or closes depending on the task [2].

6. **Feedback and Visualization**
   The movement is shown inside the O3DE simulator. ROS 2 gives feedback,

and this is visualized in RViz 2 [3], so we can see if the task was done correctly. The agent checks the result and verifies if the action worked.

This system is modular, so the tools can be reused or changed without touching the main logic of the agent. All parts talk to each other using ROS 2 topics and services, which helps to keep the performance in real time and the system flexible.

Figure 3.2 provides a visual summary of the modular pipeline described above.

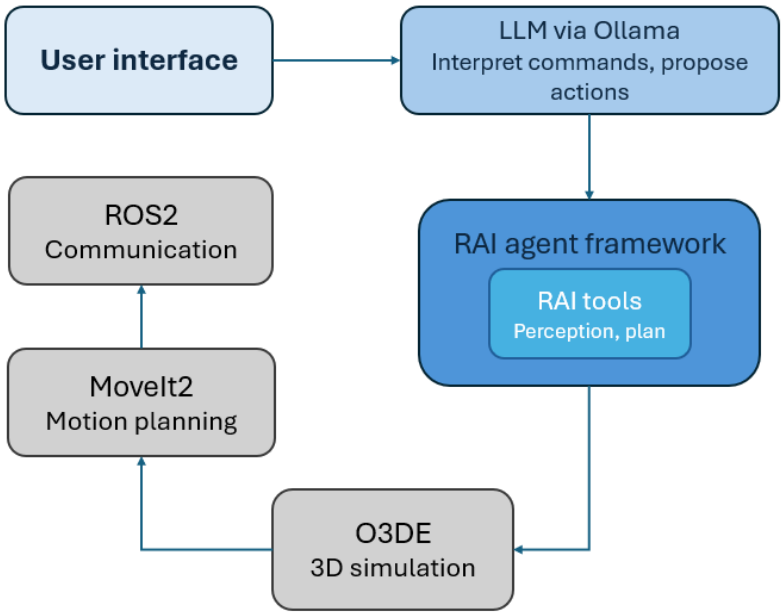


Figure 3.2: Block diagram of the system architecture. The user interface sends natural language commands to the LLM via Ollama, which interprets the intent and triggers the appropriate tools through the RAI agent. The agent interacts with ROS 2, MoveIt 2, and O3DE to simulate and execute the robotic action.

3.3 Example Execution Loop

A typical execution loop might proceed as follows:

1. **User Input**: The user provides a high-level instruction in natural language, for example: *"Pick up the red cube and place it on the green platform."*

2. **LLM Interpretation**: The embedded large language model (LLM), running locally via Ollama, interprets the intent of the instruction and converts it into a sequence of structured tool calls. For example:
   - `get_object_positions("red cube")`
   - `move_to_point(x1, y1, z1, task="grab")`
   - `get_object_positions("green platform")`
   - `move_to_point(x2, y2, z2, task="drop")`
3. **Perception Pipeline Execution**: The `get_object_positions` tool calls the internal `GetGrabbingPointTool`, which uses **Grounding DINO** to detect the object in the RGB image based on the provided class name. It then invokes **Grounded SAM** to segment the object and projects the mask onto the depth image. The result is a set of 3D centroids, which are transformed into the robot's base frame using TF2 and returned to the agent.
4. **Motion Planning and Control**: Using the retrieved 3D coordinates, the agent calls `move_to_point(...)`, specifying both the position and the type of action ("grab" or "drop"). This tool packages the information into a ROS 2 service request to the `ManipulatorMoveTo` server. The robot's planner computes a valid trajectory and executes it, including the appropriate gripper action (open/close) at the start or end of the motion.
5. **Simulation and Feedback**: The motion is simulated in O3DE, where the robot arm performs the desired action. The full interaction is visualized in RViz 2, which displays detected object frames, planned trajectories, and real-time robot pose updates. If any inconsistency is detected (e.g., failure to localize an object), the agent can re-invoke the perception tool and retry the operation.

The system architecture is designed to integrate high-level natural language understanding with low-level robotic execution in a modular and scalable fashion. The RAI framework manages reasoning and tool invocation, ROS 2 handles communication, and O3DE simulates the physical environment. Thanks to the use of a local LLM (via Ollama), the system supports fast, offline, and privacy-preserving language processing. This architecture serves as a foundation for future deployment on real hardware.

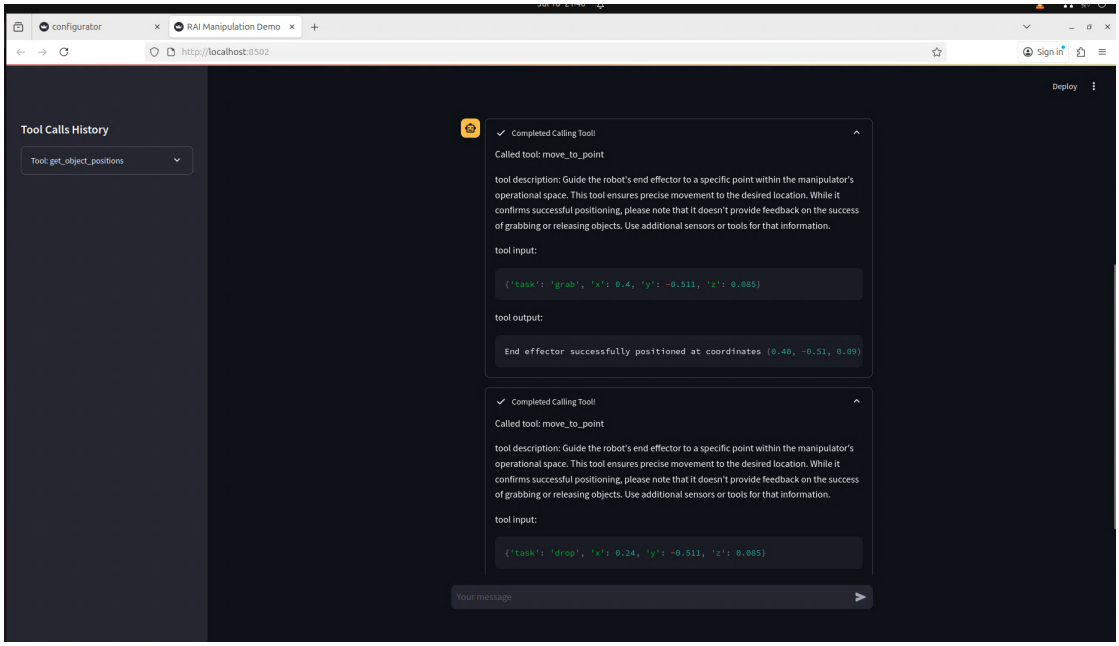


Figure 3.3: Example of tool invocation sequence. The RAI agent calls the `move_to_point` tool twice during a pick-and-place task, based on the reasoning generated by the local LLM.

Chapter 4

System Implementation

This chapter explains the general structure of the robotic system made in this thesis. The goal of the system is to let an intelligent agent understand natural language commands, see what is happening in the environment, and do real physical actions using a UR10e robot arm with a Robotiq 2F-140 gripper.

The system uses different software parts that work together: the RAI framework, a large language model (LLM) running locally with Ollama, the ROS 2 middleware, the MoveIt 2 library for planning movements, and the O3DE engine for simulation. All these components form one complete system that allows the robot to be controlled with high-level language instructions.

4.1 System Setup

The implementation of this project is based on the RAI (Robotic Agent Interface) framework. RAI is a modular system that helps robots to understand and follow high-level commands written in natural language. It gives a clear way to connect large language models (LLMs), like the ones running with Ollama, to real robot actions using ROS 2.

To integrate the UR10e robotic arm with a Robotiq 2F-140 gripper into the RAI ecosystem, several software layers were configured or adapted:

- **Middleware**: ROS 2 Jazzy Jalisco, used to let the system parts communicate to each other.

- **Motion Planning**: MoveIt 2, for calculating robot movements, checking collisions, and generating trajectories.
- **Simulation**: Open 3D Engine (O3DE), which simulates both the robot and its environment.
- **Perception Stack**: Grounding DINO and Grounded SAM for multimodal object detection and segmentation.
- **LLM Interface**: A local large language model running via Ollama.
- **RAI Framework**: The core conversational agent, tool manager, and infrastructure linking perception and control.

The whoami package

A central element in integrating a new robot with RAI is the creation of a `whoami` **configuration** package, that includes its looks, purpose, ethical code, equipment, capabilities and documentation. As outlined in RAI's official guide, this package defines the robot's identity and operational parameters in a declarative format.

More specifically, `whoami` includes:

- **The name and model of the robot** (e.g., `"ur10e_gripper"`).
- **ROS namespace** used by the robot's topics and services.
- **End-effector link name** used for sending target positions.
- **Planning configuration**, such as the motion planning group name for MoveIt.
- **Camera and sensor topics** that the vision tools can use.
- **Available RAI tools**, that the robot can , like `get_object_positions` and `move_to_point`.

This kind of modular file allows the RAI agent to work with different robots, just by reading the information inside whoami package.

Human-Robot Interface (HRI)

After the robot identity is defined using whoami, the Human-Robot Interface (HRI) must be created. This part connects the high-level commands from the agent to the ROS services, topics, and actions. According to the RAI HRI documentation, this layer includes:

- ROS 2 service servers for robot actions like motion (e.g., `/manipulator_move_to`) and controlling the gripper
- Subscriptions to topics used for perception like `/camera/color/image_raw`, `/camera/depth/image_raw`, and `/camera/color/camera_info`.
- ROS 2 nodes that contain the logic specific to this robot and how it should execute the commands.

This interface is like a bridge between the abstract tool calls from RAI and the real operations that move the robot or control the simulation. In this project, the HRI was modified to work correctly with the UR10e robot, including its joint limits, coordinate frames, and how the gripper should behave.

Core RAI ROS Packages

The integration of RAI into the ROS 2 workspace required the inclusion of the following packages (source):

- `rai_interfaces`: Contains .srv definitions (e.g., `ManipulatorMoveTo.srv`) and message types used for agent-to-robot communication.
- `rai_open_set_vision`: Implements the perception pipeline using Grounding DINO and Grounded SAM, including `GetGrabbingPointTool` and other vision tools.
- `rai_tools`: Provides reusable tool wrappers for motion commands and semantic tool definitions usable by the RAI agent.
- `rai_node`: The ROS node that runs the agent's execution environment and routes tool calls to the appropriate backend logic.

Each of these packages was cloned, built, and sourced within a ROS 2 workspace using colcon build. Parameters specific to the UR10e were set via launch files and

ROS 2 parameters, often initialized from YAML configuration files stored alongside the whoami package [5].

Summary of Setup Flow

1. Clone and build the rai workspace.
2. Create the whoami package with robot identity.
3. Implement or adapt the Human-Robot Interface for UR10e.
4. Ensure that all ROS services (e.g., motion planning) and topics (e.g., camera feeds) are correctly connected.
5. Launch the simulation (O3DE), the agent (RAI node), and the LLM backend (via Ollama).
6. Test perception and motion tools individually, then verify agent-level interactions via natural language prompts.

4.2 Adapting RAI to the UR10e Robotic Arm and Robotiq 2F-140 Gripper

The original implementation of RAI was tailored for the Franka Emika Panda robot. To adapt this framework for the UR10e robotic arm paired with a Robotiq 2F-140 gripper, a complete reconfiguration was necessary across multiple layers of the system: robot description, motion control, tool integration, and semantic identity. This section describes all the steps involved in this adaptation process.

4.2.1 Creating the whoami Package

At the core of every RAI-compatible robot lies its whoami package, which defines the robot's semantic identity. This package is not merely descriptive—it informs the RAI agent of the robot's physical structure, operational capabilities, documentation, and behavior rules. It is used during the reasoning process, especially in interactions involving natural language queries about the robot's characteristics or

constraints.

In this project, a new `whoami` package named `ur10e_gripper_whoami` was created by following the RAI developer guide [22]. The package includes a URDF model describing the robot and gripper, an image showing the physical layout, and documentation in PDF format for both the UR10e and the Robotiq 2F-140. A constitution file was also added to define the robot's operational safety constraints—such as payload limits and acceptable working ranges.

Once the folder was complete, the package was parsed using RAI's `parse_whoami_package.sh` script. This command generated a vector database from the content, producing structured files like `robot_identity.txt`, `robot_constitution.txt`, and `robot_description.urdf.txt`. These are queried by the language model during runtime to answer contextual questions and to guide planning in ways that are compatible with the robot's design.

This process also supports multimodal interaction and dynamic knowledge retrieval. For example, if a user asks, "Can you lift 5 kg?", the agent does not rely on pre-coded logic but instead queries the internal documentation parsed during the whoami setup. This allows the robot to give answers based on its own specifications and capabilities, as defined in its identity package.

An example of this interaction is shown in Figure 4.1, where the user asks about the robot's arm and gripper. The agent responds with a detailed explanation retrieved from the documentation of the UR10e and the Robotiq 2F-140 gripper that was used to create the whoam package, demonstrating context-aware and role-specific behavior.

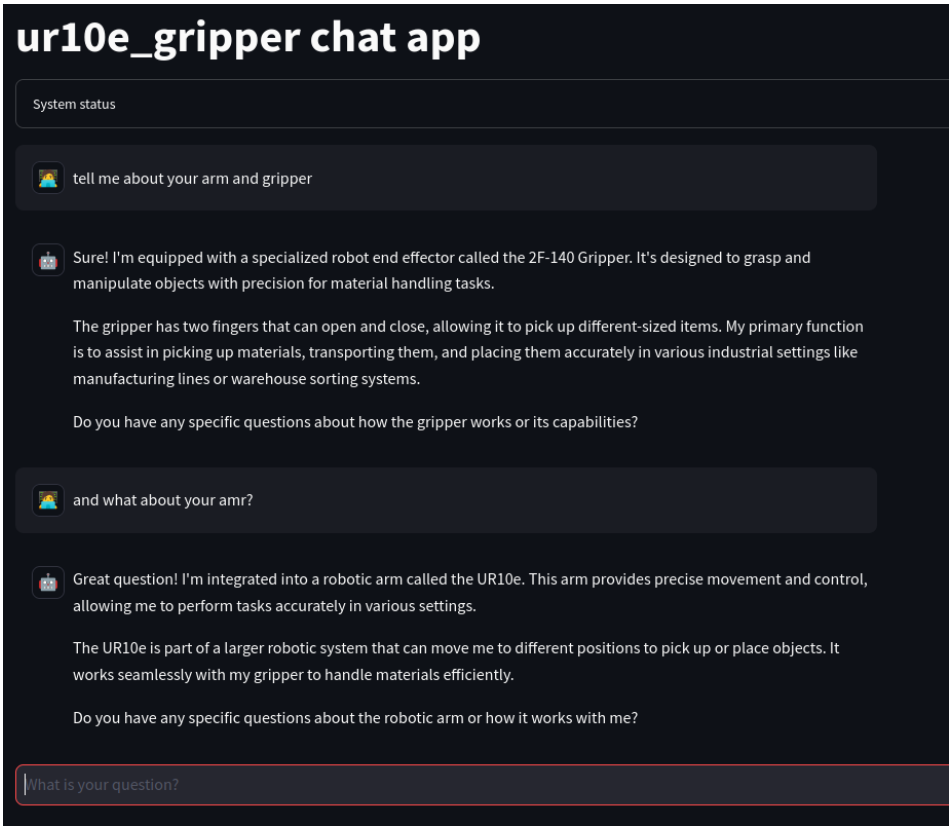


Figure 4.1: Example conversation with the RAI agent using the whoami package. The agent provides a description of its arm and gripper by consulting internal identity and documentation.

4.2.2 Implementing the Motion Interface

To allow the RAI agent to control the robot, it is essential to expose a service named `/manipulator_move_to`, which is the expected interface for high-level motion commands across all supported robots. This design allows RAI tools such as `move_to_point` to be robot-agnostic: rather than implementing a different planning strategy for each platform, they simply call the same service with a pose and a task intent (e.g., "grab" or "drop"), and rely on the robot-specific node to execute the action.

To do this, I created a new ROS 2 node called `ur10e_motion_node.py`. This node receives the `ManipulatorMoveTo` service requests and executes them by planning or forwarding the movement. It manages the full action, including going to a

pre-grasp position, using the gripper, and then moving back to a safe pose after the task is finished. The node also coordinates gripper state changes based on the task type, ensuring that a "grab" begins with an open gripper and ends with it closed, and vice versa for a "drop".

This abstraction layer is critical to keeping the tools reusable and consistent across robot types. Instead of modifying tool logic, only this interface layer needs to change when adapting RAI to new hardware.

4.2.3 Robot Model and URDF Modifications

To simulate and plan the robot movements correctly, the URDF of the UR10e was modified by adding the Robotiq 2F-140 gripper. This was done using a XACRO file called `ur10e_adapter_robotiq_2f_140_urdf.xacro`, which extends the original UR10e model by adding the gripper as a child link connected to the wrist joint. This file also contains the joints for the fingers, and the collision and visual elements needed for MoveIt 2 and O3DE.

This URDF was used both in the `whoami` file (so the agent knows the robot structure) and in the launch files to load the model into the simulation and planning environment. The URDF and the robot state publishers were also configured to send all joint states to MoveIt 2, so it can calculate correct motions and avoid collisions.

An image of the full robot in RViz was added to the documentation, to help check and debug the robot's virtual setup quickly.

4.2.4 Controller Adaptation

The default joint state controller for the UR10e was not enough to support the full configuration, especially when the robot arm and the gripper need to work together during complex actions. To solve this, I modified the file `ur10e_state_controller.cpp`, which is used to manage how joint states are published and updated.

With these changes, the system could give real-time feedback while the robot was moving, improve the timing, and send correct joint states to the motion planner.

Having the controller well configured was very important for tasks like pick-and-place, because even small errors in joint feedback can cause problems like failed

grasps or collisions.

4.2.5 Launch and System Integration

To connect the robot with the simulation and the motion planning environment, I edited two launch files:
`max_demo_arm_gripper.launch.py` and
`ur10e_manipulation.launch.py`.

I changed these files to load the correct robot model, start the MoveIt 2 planning system, launch the gripper controller, and run the motion interface node I made before.

These launch files help to start all the needed parts—like the O3DE simulation, MoveIt 2, controllers, and the RAI agent—in a synchronized and repeatable way. I tested each part one by one, and then I ran complete scenarios to check that everything was working correctly.

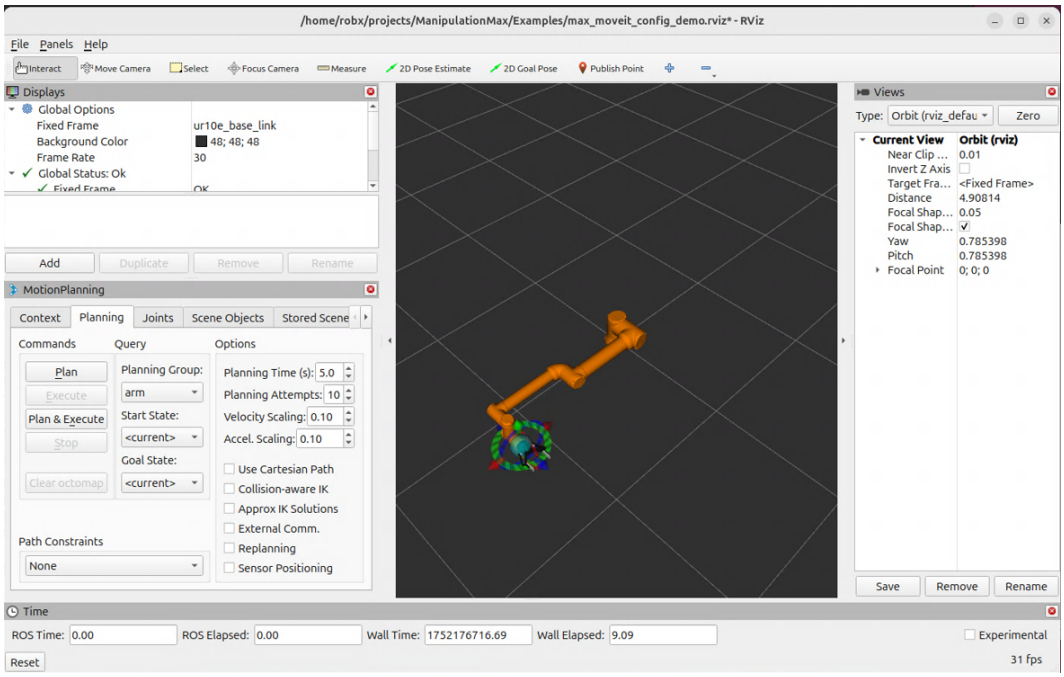


Figure 4.2: UR10e arm visualized in RViz 2 with planning group loaded. The robot model is published via ROS 2 and controlled through MoveIt 2.

4.3 Perception Pipeline

The perception system is very important because it helps the robot to understand the environment and do tasks that need to detect and manipulate objects using natural language commands.

In this project, the perception is connected to the RAI framework. It uses images from ROS 2 and two advanced models: Grounding DINO and Grounded SAM. These models allow the robot to detect and segment objects in the scene, even if they were not seen before (open-set detection).

4.3.1 High-Level Overview

The primary perception tool used in this setup is get_object_positions, implemented in the file `manipulation.py`. This tool allows the agent to localize objects in the environment by name—for instance, when given a command like "pick up the red cube", the agent will internally call `get_object_positions("red cube")`. This tool serves as a high-level wrapper that internally relies on the more specialized `GetGrabbingPointTool`, defined in `segmentation_tools.py`.

The pipeline integrates with the robot's RGB-D camera setup in O3DE via ROS 2 topics. Specifically, the tool subscribes to:

- /camera/color/image_raw for RGB images,
- /camera/depth/image_raw for depth images, and
- /camera/color/camera_info for intrinsic parameters.

These inputs are used to perform visual grounding and segmentation of the target object class.

4.3.2 Grounding DINO and Grounded SAM Integration

The perception pipeline uses two state-of-the-art models:

- **Grounding DINO** (GDINO), which matches textual object prompts to image regions by producing bounding boxes for class names given in natural language.

- **Grounded SAM**, which segments the object masks based on the bounding boxes returned by GDINO.

The `GetGrabbingPointTool` initiates the pipeline by calling the GDINO service `RAIGroundingDino`. It constructs a request that includes the image message and the object name passed by the user or LLM. After receiving a response with bounding boxes, the tool forwards these detections to the `RAIGroundedSam` service, which returns segmentation masks for each detected object.

The relevant services are defined in:

- `rai_interfaces/srv/RAIGroundingDino.srv`
- `rai_interfaces/srv/RAIGroundedSam.srv`

These services are called asynchronously and their responses are awaited via `rclpy.spin_once`.

4.3.3 3D Position Estimation

After the segmentation masks are received, the pipeline finds possible 3D positions to grasp the object. The mask is projected on the depth image, and with the camera parameters (from the camera info topic), a point cloud is created using the `depth_to_point_cloud` function. This point cloud is then cleaned to remove low-confidence areas and background.

To find a good point to grasp, the tool takes the top points based on the Z value (height), and calculates the average position. This becomes the grasp centroid. The tool also calculates the best angle for the gripper using the shape of the object in the image, with `cv2.minAreaRect`.

Then, this centroid is transformed from the camera frame to the robot's planning frame using `TF2TransformFetcher`, which uses the TF logic in ROS 2.

At the end, the `get_object_positions` tool gives a list of 3D poses (centroids), already in the base frame of the robot, and ready to be used by the `move_to_point` tool.

4.3.4 Node and Topic Architecture

The perception logic is divided into different parts:

- The RAI tool `get_object_positions` (in the file manipulation.py) (defined in manipulation.py)
- The segmentation tool `GetGrabbingPointTool` (in `segmentation_tools.py`)
- ROS 2 nodes that run the GDINO and SAM services
- The camera topics published from O3DE

This modular setup lets the robot detect and understand objects it never saw before, using only language and vision. It doesn't need specific training for each object. Also, the perception tool is not directly connected to the robot hardware—it just uses the camera data and TF transforms to find object positions.

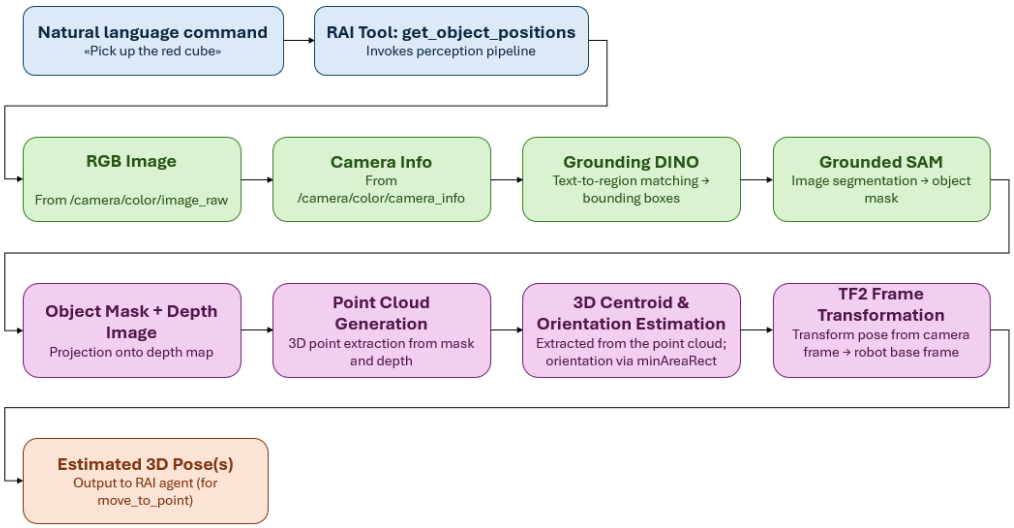


Figure 4.3: Perception pipeline architecture. A natural language command triggers the get_object_positions tool, which invokes the vision pipeline. The system uses Grounding DINO for grounding object names to image regions, Grounded SAM for segmentation, and depth projection to estimate 3D object poses in the robot base frame.

4.4 Motion Planning and Execution

After the perception system gives the position of one or more target objects, the robot has to plan and execute a correct movement to reach, grab, and maybe move the objects. This step is very important because it goes from "knowing where the object is" to "doing something with it." It needs to generate a valid path, respect some limits, and control the gripper, all based on natural language commands.

In this project, the motion planning and execution were done using a mix of ROS 2 services, RAI agent tools, and MoveIt 2 to calculate the trajectories.

4.4.1 Motion Planning with Pose Targets

The robot receives target positions from the tool `get_object_positions`, which outputs 3D poses in the robot's base frame. These poses are then used by the RAI agent to construct a motion command that is passed to the tool `move_to_point`.

The `move_to_point` tool, defined in `manipulation.py`, accepts a Cartesian pose and a task intent ("grab" or "drop"). This tool does not plan the motion by itself. Instead, it sends the execution to the robot using a standard interface: the `/manipulator_move_to` service.

This service takes as input:

- A 3D position (x, y, z)
- A fixed orientation in quaternion (already set in the tool class)
- The task type (like "grab" or "drop"), which is used to decide how the gripper should move

This setup keeps the planning part separate from the agent logic. The RAI agent only needs to think about "where" and "why" to move, while the tool and ROS system handle the "how" to actually do it.

4.4.2 Execution via `/manipulator_move_to`

The `/manipulator_move_to` service is implemented inside a custom node called `ur10e_motion_node.py`. This node manages both the robot movement and the gripper control. It creates a `ManipulatorMoveTo.Request` message using the target pose, and takes care of the full execution of the task.

For a **"grab"** task:

- The gripper is opened before the move
- The robot moves to the target pose
- The gripper is closed upon arrival

For a **"drop"** task:

- The gripper remains closed during movement
- The robot places the object at the destination
- The gripper is opened at the end

This state machine makes sure that the robot always does the same thing when grabbing or releasing an object, without needing to write new code for each task. It also works well with the standard RAI tools, because they expect the robot behavior to change depending on the task type, not on special instructions.

The node uses rclpy and keeps spinning until it gets a response or the timeout is reached. If the service fails or the target position can't be reached (for example because of kinematic limits), the tool sends back a failure message. This lets the RAI agent try again or choose a different plan.

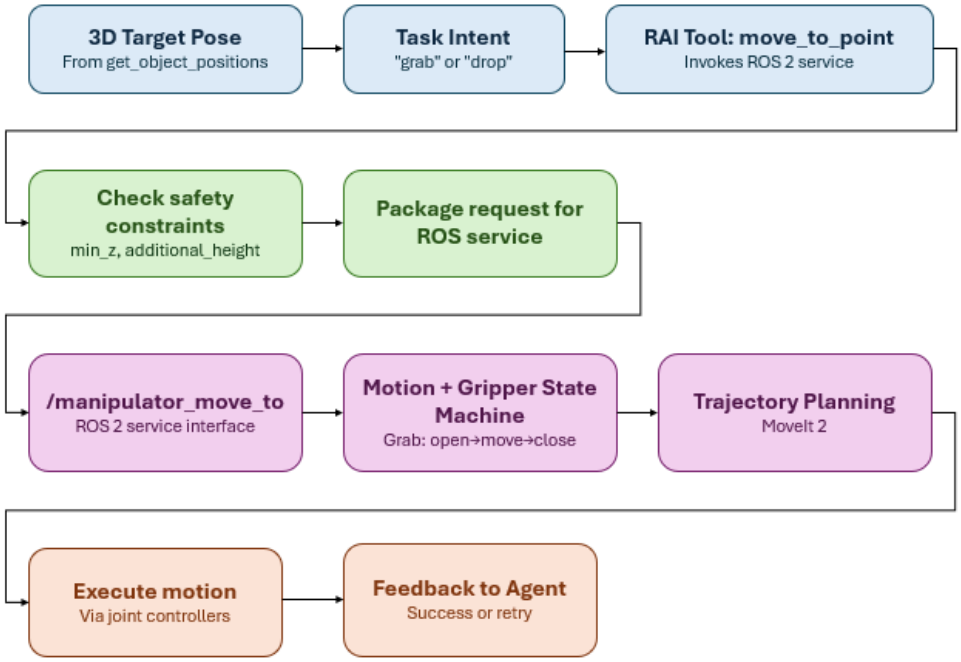


Figure 4.4: Motion planning and execution flow. The `move_to_point` tool transforms a 3D target pose and task intent into an executable motion via the ROS 2 service `/manipulator_move_to`. The motion node handles gripper actions and trajectory planning using MoveIt 2, ensuring safety constraints are met. The result is fed back to the agent for verification or retry.

4.4.3 Orientation and Safety Constraints

In this project, the robot's orientation is fixed using a quaternion that is already set in `move_to_point`. This makes the control easier and is good enough for most tasks on a table.

The Z value (height) of the position is managed more carefully:

- A `min_z` value is used so the robot never goes too low and hits the table.
- An `additional_height` is added when doing a "drop" to avoid collisions with the object or the surface.

These limits are checked inside the tool before calling the service. If needed, the position is adjusted to avoid dangerous movements. This helps to make the system more safe and stable.

4.4.4 Integration with MoveIt 2

Even if MoveIt 2 is not called directly by the RAI tool, it is still an important part of the system that supports the motion node. The UR10e robot is configured with a planning group in the MoveIt 2 setup (called `ur10e_moveit_config`), and the motion node uses this to create the trajectories using MoveIt services or its own planning code.

The movement execution is synchronized with the joint controllers that are started by the launch files. The gripper is also controlled using joint states, so the grasp and release actions work the same in both simulation and on the real robot.

4.5 Tool Integration with the RAI Agent

The RAI system is based on the idea of a conversational agent that understands the user's high-level commands written in natural language. Then, it uses specific "tools" to do actions with the robot, the environment, or some internal knowledge.

These tools are like small functional blocks that are added to the agent's configuration and can be used by the LLM during the reasoning process.

In this project, the integration of tools focused mainly on two things: motion control and object perception. Both tools were added to the agent through the `whoami` package and were written in Python using the `BaseTool` API from RAI.

In this project, tool integration focused on two main areas: motion control and object perception. Both tools were configured and exposed to the agent through the `whoami` package and implemented in Python using RAI's `BaseTool` API.

4.5.1 Tool Registration and Configuration

In the RAI framework, tools are created as Python classes that extend from `BaseTool`, and they must be imported and registered when the system starts. Unlike other systems that use `.yaml` or `.json` files for configuration, RAI uses registration directly in the code. This gives more control to the developer to decide which tools are available to the agent.

In this project, I added the tools `move_to_point` and `get_object_positions` by

creating their classes in the files `manipulation.py` and `segmentation_tools.py`. These tools are loaded when the agent starts and are ready to be used by the LLM using prompts or function calls.

Each tool has:

- A name, used by the system and the LLM
- A description, that helps the LLM understand what the tool does
- An args_schema (using Pydantic) to check the input arguments and format the prompt correctly

With this method, the agent can decide which tool to use based on the user's request. For example, if the LLM wants to move the robot to grab something, it uses `move_to_point`. If it needs to find an object in the scene, it uses `get_object_positions`. All this works without needing to manually change the configuration.

4.5.2 Motion Tool: `move_to_point`

The `move_to_point` tool is the connection between natural language commands like "place the cube here" and the ROS 2 motion service `/manipulator_move_to`. It hides the low-level details of planning and movement, and only gives the agent a simple interface with the target position and the type of task.

The tool uses a fixed orientation quaternion (always the same for all tasks), and includes some calibration offsets and height limits to avoid collisions.

The task type—like "grab" or "drop"—is understood by the system and used to control the gripper in the right way. This is managed inside the motion node.

This separation between logic and execution means the LLM doesn't need to know about kinematics or how to plan motions. It just needs to know which tool to call and when

4.5.3 Perception Tool: `get_object_positions`

The `get_object_positions` tool handles all vision-related tasks. Given a class name (e.g., "red cube"), it invokes the full grounding and segmentation pipeline.

Internally, this tool delegates to the `GetGrabbingPointTool`, which performs:

- The call to the Grounding DINO service using the class name
- The call to Grounded SAM using bounding boxes from GDINO
- Projection of masks onto depth data
- Grasp pose estimation and transform to robot base frame

The tool returns one or more object poses as strings formatted for readability. These outputs are parsed or reasoned about by the agent (or filtered using tool logic) before being passed to motion tools. In this way, the agent can autonomously link object detection and manipulation without requiring the user to provide exact coordinates.

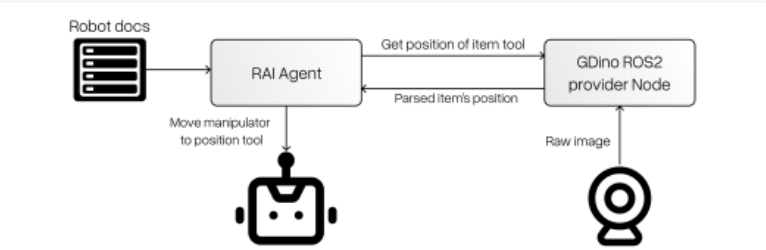


Figure 4.5: Simplified pipeline: the RAI agent interprets a natural language command, invokes the perception tool to identify object poses via Grounding DINO and Grounded SAM, and issues movement commands using the motion tool.

4.5.4 Tool Invocation Process

During a reasoning cycle, the agent processes the user's instruction using the LLM hosted via Ollama. The model first interprets the intent and identifies whether a tool should be called. If so, it selects the tool and populates its arguments from the context.

For example, given the command:

"Put the red cube on the green cube."

The agent might issue:

1. `get_object_positions`("red cube") → returns [x1, y1, z1]

2. `get_object_positions`("green cube") → returns [x2, y2, z2]
3. `move_to_point`(x1, y1, z1, task="grab")
4. `move_to_point`(x2, y2, z2, task="drop")

Each invocation is logged and can be monitored for debugging or analysis. This architecture makes it possible to follow the decision-making process and trace the full chain from command to action.

4.6 Challenges and Solutions

Adapting the RAI framework to a new robotic platform inevitably introduced technical challenges at various levels, particularly in ensuring compatibility with the expected interfaces and achieving stable motion and perception behaviors.

One of the main challenges was the implementation of the `/manipulator_move_to` service. RAI's motion tools are designed to be generic and expect this service to be exposed regardless of the robot model. Since this interface did not exist for the UR10e by default, I had to create a custom ROS 2 node (`ur10e_motion_node.py`) that received requests with target poses and translated them into valid robot motions, including gripper actuation. This was necessary to allow RAI's motion tools to work out-of-the-box with the new hardware, without modifying the internal logic of the tools themselves.

Another issue involved motion execution. Some target poses returned by the perception system were outside the UR10e's workspace or too close to the table surface. To address this, I applied constraints on the Z-axis within the `move_to_point` tool, setting a `min_z` threshold and adjusting placement height with an `additional_height` parameter. These safety buffers prevented collisions and improved stability.

Finally, during initial tests, the synchronization between arm motion and gripper state transitions was not reliable. This was fixed by carefully managing the gripper commands within the motion node, ensuring that the gripper state was set before and after the robot moved, depending on the task type.

Despite these challenges, the modular design of RAI and its reliance on standard interfaces allowed for a smooth adaptation process without having to rewrite core agent logic.

Chapter 5

Simulation and Testing in O3DE

The system developed in this project was tested within a simulated environment using Open 3D Engine (O3DE). The simulation scene was based on the original RAI demo with the Franka Emika Panda robotic arm that was replaced by the UR10e arm + Robotiq 2F-140 gripper. The workspace configuration, lighting, and camera placement were preserved, allowing the integration of the new robot without major modifications to the virtual scene.

The UR10e robot was added to the simulation with its proper kinematics, visual model, and control configuration. ROS 2 topics for the RGB and depth camera feeds were reused to ensure compatibility with the existing perception pipeline.

The interaction with the system occurred through a Streamlit-based chat interface, where the user entered natural language commands that were then interpreted by the RAI agent. For visual validation were used 2 different tools: RViz 2, for monitoring trajectories and transformation frames, and O3DE, for observing the robot's behavior in the simulated environment.

5.1 System Setup

The intention behind this testing phase was also to compare the behavior of different language models when interpreting and generating robotic commands. However,

due to the limitations of available models in Ollama, only Qwen could be used for complete task execution, as it was the only one to support function calling natively.

Despite initial attempts to test several models—including `llama3:8b`, `mistral`, `mixtral`, `phi-mini`, `phi-medium`, `command-R`, `gemma`, and `openchat`—none of these provided direct support for function-based tool invocation within the RAI agent, making them unusable in the current setup. Likewise, multimodal models such as `llava`, `deepseek`, and `bakllava` were evaluated, but they lacked function calling capabilities and thus could not be integrated for active testing.

As a result, the evaluation focused entirely on **Qwen** in different variants. Four categories of commands were issued during testing:

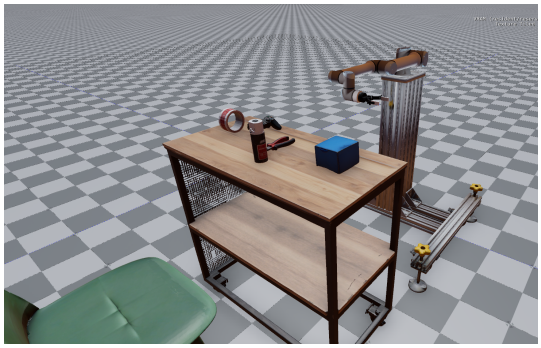
1. **Simple motion commands**, such as "move the arm to the right", to verify basic understanding and execution flow.

2. **Grasping commands**, like "pick up the red cube", to assess perception accuracy, execution timing, and number of attempts.

3. **Pick-and-place tasks**, involving compound actions such as "pick the red cube and place it on the green one", to evaluate planning and fluency.

4. **Multi-step sequences**, such as "sort the cubes by color" or "place the red cube on the green one and then move the apple to the right", to test autonomous task decomposition.

Each command was evaluated qualitatively, with attention to the correctness of the toolchain generated, the fluency of the execution, and the system's ability to retry after partial failures. Qwen performed reliably on most simple and intermediate tasks, and showed limited but functional capacity to handle decomposed sequences.



(a) Simulated manipulation environment created in O3DE. The scene includes a UR10e robot, table, and various tools and objects.

(b) Alternative view of the same environment with a focus on camera angle and background layout.

Figure 5.1: Custom O3DE simulation environments designed for this thesis. The scenes include physical elements like a table, chair, and tools to create realistic manipulation scenarios.

5.2 Hardware Setup

The physical hardware used in this thesis includes a UR10e robotic arm from Universal Robots and a Robotiq 2F-140 parallel gripper. This hardware setup is the same as the one used in the simulation. The robot was connected to the development PC using a direct USB cable, which allowed fast communication for sending commands and checking the robot's status. The real workspace was arranged to look as close as possible to the virtual one made in O3DE, with similar objects and layout.

Component	**Specification**
Device	Desktop Workstation
Operating System	Ubuntu 24.04.2 LTS
CPU	Intel Core Ultra 9 (Meteor Lake, 16 cores)
GPU	NVIDIA GeForce RTX 5080, 16 GB VRAM
GPU Driver and CUDA	NVIDIA Driver 570.xx, CUDA 12.4
Memory (RAM)	64 GB DDR5 @ 5200 MHz
ROS 2 Distribution	ROS 2 Jazzy
O3DE Version	O3DE 24.02 (custom Linux build)
Robot Connectivity	USB 2.0 interface for UR10e
Gripper Connectivity	Ethernet interface for Robotiq 2F-140
Containerization	Docker with GPU support (NVIDIA Container Toolkit)

Table 5.1: Hardware and software specifications of the development workstation

Hardware Specifications of the Development Environment All the development and tests were done on a powerful workstation that was made to support real-time simulation, GPU rendering, and fast communication with the robot. The system specs are shown in Table 5.1.

This setup had enough resources to run real-time physics, generate 3D scenes with AI, and plan robot motion with ROS 2. The NVIDIA RTX 5080 GPU helped to keep good graphics and physics performance inside O3DE. The full system was containerized using Docker, so it was easy to reproduce and to keep it working the same on different machines during development and testing.

Chapter 6

Evaluation of Language Models

A core objective of this project was to test the feasibility of using large language models (LLMs) to control a robot through high-level natural language commands. Given the modular structure of RAI, the conversational agent can in principle be paired with any LLM that supports function calling—allowing the model to decide when and how to invoke available robotic tools. This chapter documents the language models evaluated during the project, their level of compatibility with RAI, and their performance across different types of tasks.

Although multiple open-source LLMs were tested, only one—Qwen—was able to fully support the required function-calling mechanism via Ollama. All other models lacked this functionality or required significant adaptation, which would go beyond the scope of this work.

6.1 Comparison of Language Models

The following table summarizes the language models evaluated, their type, function-calling support, and whether they were usable in the final implementation:

Model Name	Type	Parameters	Function Calling	Usable	Notes
Qwen (various)	Non-multimodal	1.8B–14B	✓	Yes	Used for all tests; supports tool calls; robust in multi-step tasks.
LLaMA 3 (8B)	Non-multimodal	8B	✗	No	Lacks function calling support in Ollama.
Mistral	Non-multimodal	7B	✗	No	Good performance in NLP, but not integrable with RAI tools.
Mixtral	Non-multimodal	12.7B (MoE)	✗	No	Not compatible with tool invocation.
Phi-1.5 / Phi-2	Non-multimodal	1.3B–2.7B	✗	No	Lightweight and fast, but lacks required APIs.
Command-R	Non-multimodal	~35B	✗	No	Not callable via function interfaces in current setup.
Gemma	Non-multimodal	2B–7B	✗	No	Tested, but lacks ROS and tool support.
OpenChat	Non-multimodal	7B	✗	No	Natural-sounding responses but not tool-aware.
LLava	Multimodal	~13B	✗	No	No function calling; vision-only usage.
BakLLaVA	Multimodal	~12B	✗	No	Multimodal but lacks function-call support.
DeepSeek	Multimodal	~13B	✗	No	Not usable for structured tool chaining.

Table 6.1: Comparison of tested language models and their compatibility with RAI

6.2 Observations on Qwen's Behavior

All the commands described in Chapter 5 were executed using Qwen models with varying numbers of parameters. Even the most lightweight versions were able to correctly interpret simple instructions and select the appropriate tools. For object manipulation operations—for example, commands like "pick up the red cube"—the model consistently invoked first `get_object_positions` and then `move_to_point`, following the expected sequence.

With more complex instructions, which required spatial reasoning and multi-step

decomposition, the behavior was less linear: in some cases, one or more attempts were needed to complete the task. However, the requested action was generally completed in most cases. Reliability tended to improve as the model size increased, although this resulted in longer inference times.

The results show that, at present, the choice of language model is heavily constrained by the availability of native function-calling support. Although many LLMs perform well in traditional NLP tasks, they cannot be integrated with robotic tools unless they are able to reason over structured function schemas and trigger calls dynamically. Qwen proved to be a practical choice due to its compatibility with Ollama and its built-in support for tool-oriented reasoning.

While the integration of multimodal models would open the door to even more advanced capabilities—such as vision-language reasoning without hardcoded pipelines—this remains out of scope for the current system due to limitations in tool interfacing.

Chapter 7

Discussion

This thesis set out to investigate whether it is feasible to control a robotic arm using high-level natural language commands, leveraging the RAI (RobotecAI) framework in combination with a large language model. Throughout the project, the focus remained on integrating perception, planning, and execution into a single tool-based agent architecture that could operate in simulation using the UR10e robotic arm and Robotiq 2F-140 gripper.

The overall structure of RAI proved to be well-suited for this kind of extension. Although originally built around the Franka Emika Panda, the framework's use of abstracted services and modular tools made it possible to adapt the pipeline to new hardware without rewriting the core logic [5]. Creating a dedicated `whoami` package, implementing the required ROS 2 services, and adjusting the robot description files were sufficient to enable compatibility.

The agent, based on a conversational LLM using Ollama, interfaced naturally with the available tools. Of the tested models, Qwen was the only one to support function invocation in a manner aligned with the expectations of the RAI tool system. Although other LLMs were evaluated (e.g., LLaMA 3, Mistral, LLava), none supported dynamic tool invocation at runtime [**rai_docs**]. Qwen's structured output allowed the agent to correctly generate a sequence of tools for object manipulation and basic spatial reasoning.

The simulated environment, adapted from the original RAI demo, allowed for

comprehensive integration testing using O3DE and ROS 2. RViz 2 was used to validate the joint trajectories and sensor data in parallel. The various tests performed confirmed the overall feasibility of the system in handling pick-and-place tasks, perception-driven actions, and multi-step natural language instructions.

Some limitations were noted. The environment, while functional, was not completely realistic. Depth noise, occlusion, and latency, common in physical systems, were not replicated. Furthermore, only one usable LLM with native tool support was available through Ollama, which limited broader evaluation. Finally, the system was not implemented on a physical robot, which would have been necessary for real-world validation. Nonetheless, the results confirm the feasibility of LLM-driven robot control using a modular, tool-based framework like RAI. This work lays the groundwork for future research into open-set robotic manipulation guided by natural language.

7.1 Personal Contributions and Framework Reuse

This thesis is based on the existing RAI framework, which was first created for the Franka Emika Panda robot. However, to use it with a new industrial robotic platform (UR10e with Robotiq 2F-140 gripper), it was necessary to create some new components and also change some of the existing ones to make everything work correctly.

The table below gives a summary of the main components used in the system. It shows which ones were already available in the original RAI framework, which were modified, and which were developed from zero during this project.

Component	Pre-existing	Modified	Created from Scratch
RAI Agent Core	✓	X	X
`whoami` package for UR10e	X	X	✓
`ur10e_motion_node.py`	X	✓	✓
`/manipulator_move_to` service interface	X	✓	✓
O3DE simulation scene	✓	✓	X
URDF + XACRO for UR10e + Robotiq gripper	X	✓	✓
Launch files for UR10e integration	X	✓	✓

Table 7.1: Overview of the components reused, modified, or created during this thesis. Symbols indicate: ✓for present, X for not applicable.

This breakdown highlights the breadth of engineering work carried out during the thesis—from hardware-level integration to high-level tool interfacing—demonstrating the system's portability and extensibility beyond its original scope.

Chapter 8

Conclusion

This work demonstrated that integrating a language-based agent with a simulated robotic platform can be successful. Thanks to RAI's modular architecture and the support for the Grounding DINO and Grounded SAM visual perception modules, the system was able to complete a variety of user-defined tasks using only natural language commands.

By using Qwen as the reasoning engine and implementing the necessary interfaces and configuration files, the original demonstration, designed for a different robot (Franka Panda), was extended to a new configuration using the UR10e and the Robotiq 2F-140 gripper. The fact that this was achieved without modifying the core tools confirms the generalizability of the RAI agent structure.

Even if the system was only tested in simulation, the results were promising and suggest that function-calling language models, when integrated with structured robotics frameworks, can enable flexible, task-based control with minimal manual programming.

Future Work

There are many ways this project can be improved in the future:

Support for more LLMs: Adding different language models, maybe with fine-tuning or custom adapters, could make the system understand more types of instructions, work in different languages, and improve how it splits tasks into steps.

Multimodal agents: It could be useful to use models that understand both images and text directly, without needing an external perception pipeline. This would make the system architecture simpler and easier to manage.

Better error checking and feedback: Adding ways for the system to check if something went wrong—like rechecking the vision result or confirming if the action succeeded—could make the robot more stable and accurate.

Multiple agents: In the future, the system could control more than one robot at the same time. Each robot could do a different task, and they could work together. For example, one arm could pick the objects, and another could place them. RAI's modular design makes this possible, but it would need some changes to support roles, shared memory, and communication between agents.

These ideas would help make the system more intelligent, flexible, and ready to work in real-world situations where robots need to understand high-level instructions and work together.

Bibliography

[1] Open Robotics. *ROS2 Documentation — Jazzy*. https://docs.ros.org/en/jazzy/index.html#. Accessed: 26 May 2025. 2025 (cit. on pp. 5, 19).

[2] MoveIt AI. *MoveIt Concepts Documentation*. https://moveit.ai/documentation/concepts/. Accessed: 26 May 2025. 2025 (cit. on pp. 6, 20, 21).

[3] Open Robotics. *RViz — ROS2 Tutorial (Intermediate)*. https://docs.ros.org/en/jazzy/Tutorials/Intermediate/RViz/RViz-Main.html#. 2025 (cit. on pp. 7, 20, 22).

[4] Open Robotics. *ROS 2 Documentation — Jazzy*. https://docs.ros.org/en/jazzy/index.html. 2025 (cit. on p. 7).

[5] Kajetan Rachwał, Maciej Majek, Bartłomiej Boczek, Kacper Dąbrowski, Paweł Liberadzki, Adam Dąbrowski, and Maria Ganzha. *RAI: Flexible Agent Framework for Embodied AI*. 12 pages, 8 figures; submitted to PAAMS'25. 2025. DOI: 10.48550/arXiv.2505.07532. arXiv: 2505.07532 [cs.MA]. URL: https://arxiv.org/abs/2505.07532 (cit. on pp. 7, 8, 14, 19, 28, 51).

[6] Robotec.AI. *RAI - Real-time Adaptive Infrastructure*. https://github.com/RobotecAI/rai. 2024 (cit. on pp. 8, 14).

[7] Universal Robots. *UR10e e-Series Datasheet*. https://www.universal-robots.com/media/1807466/ur10e_e-series_datasheets_web.pdf. 2025 (cit. on p. 12).

[8] Universal Robots. *UR10e Collaborative Robot Arm*. https://www.universal-robots.com/products/ur10e/. 2025 (cit. on p. 12).

[9] Robotiq. *Pinze adattive (Two-Finger Gripper)*. https://robotiq.com/it/prodotti/pinze-adattive#Two-Finger-Gripper. 2025 (cit. on p. 13).

[10] Open 3D Engine Contributors. *ROS2 Gem – Open3DEngine*. https://docs.o3de.org/docs/user-guide/gems/reference/robotics/ros2/. 2025 (cit. on pp. 14, 20).

[11] Robotec.ai. *How to start your first robotics simulation projects in Open3DEngine (O3DE)*. https://www.robotec.ai/how-to-start-robotics-simulation-projects-in-open-3d-engine-o3de. 2025 (cit. on p. 14).

[12] The Linux Foundation O3DE Project. *Open 3D Engine Overview*. https://o3de.org/overview/. 2024 (cit. on p. 15).

[13] Ollama Inc. *Ollama — Get up and running with large language models*. https://ollama.com/. 2025 (cit. on pp. 16, 19).

[14] Ollama Inc. *ollama*. https://github.com/ollama/ollama. 2025 (cit. on p. 16).

[15] Roy Ben Yosef. *How to Run LLMs Locally with Ollama*. https://medium.com/cyberark-engineering/how-to-run-llms-locally-with-ollama-cb00fa55d5de. 2024 (cit. on p. 16).

[16] Docker, Inc. *Docker Overview – Get Started with Docker*. https://docs.docker.com/get-started/docker-overview/. Accessed July 9, 2025. 2025 (cit. on p. 17).

[17] Docker, Inc. *What is a Container? – Docker Documentation*. https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/. Accessed July 9, 2025. 2025 (cit. on p. 17).

[18] Docker, Inc. *What is an Image? – Docker Documentation*. https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. Accessed July 9, 2025. 2025 (cit. on p. 17).

[19] Docker, Inc. *Dockerfile Concepts – Docker Documentation*. https://docs.docker.com/build/concepts/dockerfile/. Accessed July 9, 2025. 2025 (cit. on p. 17).

[20] Docker, Inc. *Volumes – Docker Documentation*. https://docs.docker.com/engine/storage/volumes/. Accessed July 9, 2025. 2025 (cit. on p. 17).

[21] NVIDIA Corporation. *NVIDIA Container Toolkit*. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html. 2023 (cit. on p. 18).

[22] Robotec.AI. *Create robots - whoami*. `https://github.com/RobotecAI/rai/blob/1.0.0/docs/create_robots_whoami.md`. 2024 (cit. on p. 29).