Politecnico di Torino

Artificial Intelligence and Data Analytics

A.a. 2024/2025

Graduation Session July 2025

Robotic Planning in Realistic Virtual Scenarios with AI Support

Supervisors:
Guido Albertengo

Candidates:
Massimo Porcheddu

Acknowledgements

Table of Contents

List of Figures

List of Abbreviations

Acronym	**Definition**
AI	Artificial Intelligence
DDS	Data Distribution Service
DOF	Degrees of Freedom
GPU	Graphics Processing Unit
IK	Inverse Kinematics
OMPL	Open Motion Planning Library
O3DE	Open 3D Engine
PCG	Procedural Content Generation
QoS	Quality of Service
RL	Reinforcement Learning
ROS	Robot Operating System
RViz	ROS Visualization Tool
SRDF	Semantic Robot Description Format
TF	Transform Frame
URDF	Unified Robot Description Format
USB	Universal Serial Bus
VAE	Variational Autoencoder
GAN	Generative Adversarial Network

Chapter 1

Introduzione

The main contributions of this thesis are several. First of all, it shows the design and implementation of a modular simulation and control system based on ROS 2, with the integration of MoveIt for motion planning and Open 3D Engine (O3DE) for realistic and high-quality 3D simulation. This architecture allows flexible development and supports realistic visualization of robotic behaviour in virtual scenarios.

Second, the framework uses generative AI models for automatic generation of 3D elements in the scenes, increasing the diversity and complexity of the simulated environments. This method tries to better reflect the unpredictability and variation typical of real-world situations, improving the robustness of the developed robotic strategies.

Third, the system is validated by executing the same motion plans also on a real UR10e robotic manipulator with a Robotiq gripper. This comparison with the physical results gives a general evaluation of the accuracy and realism of the simulation.

Finally, everything is included inside a containerized environment to guarantee the reproducibility on different development platforms. This also helps easier deployment, consistent testing, and better collaboration in research and development teams.

1.1 Goal

The main goal of this thesis is to create a simulation-based framework for testing robotic manipulation tasks in a virtual environment that is close to the real execution. By using ROS 2, MoveIt, and Open 3D Engine (O3DE), the framework wants to support precise motion planning, real-time visualization, and interaction with the environment. Also, the integration of AI-generated 3D objects helps to improve the realism and variability of the scenes.

The final target is to check the consistency between simulation and real-world by executing the same planned trajectories on a real UR10e robotic arm with Robotiq gripper. A summary diagram of the general structure is shown in Figure 1.1

This kind of approach wants to reduce development time, lower the cost of prototyping, and prepare the base for future extensions like coordination of multiple robots and autonomous task planning.

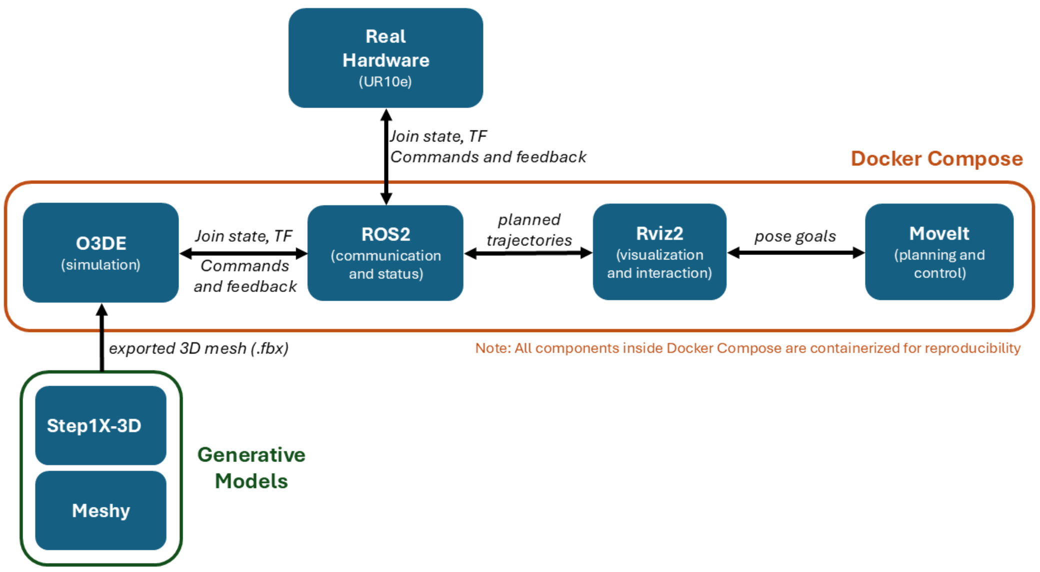


Figure 1.1: System architecture showing the interaction between simulation, planning, visualization, hardware and AI-based environment generation. All components inside the Docker Compose boundary are containerized for reproducibility.

1.2 Thesis structure

The thesis is structured as follows.

- Chapter 2 provides an overview of the theoretical and technological background, focusing on ROS 2, MoveIt, O3DE, and related tools.
- Chapter 3 presents the architecture of the implemented system, including its main components and their interactions.
- Chapter 4 describes the simulation environment and the generation of 3D virtual scenes.

- Chapter 5 discusses the task execution and trajectory generation strategies adopted.
- Chapter 6 illustrates the integration of the system with real robotic hardware.
- Chapter 7 reports on the results of the experiments conducted in both simulated and physical contexts.
- Finally, Chapter 8 concludes the work and outlines possible directions for future development.

Chapter 2

State of the art

The development of intelligent robotic systems depends on many technologies, like motion planning libraries, simulation tools, communication middleware, and AI methods for scene generation. This chapter gives an overview of the main frameworks and tools that are the technological base of the system developed in this thesis. The discussion focuses on Robot Operating System 2 (ROS 2), the MoveIt planning framework, the RViz tool for visualization, Docker strategies for containerization, the Open 3D Engine (O3DE) for 3D simulation, and also recent progress in AI-based environment modeling. All these technologies together support the modular, realistic, and testable architecture proposed in this work.

2.1 ROS 2

The Robot Operating System 2 (ROS 2) is an open-source middleware framework for robotics, made to support the development of scalable, distributed, and real-time robotic applications. It is the evolution of the first ROS 1 version, and it solves many architectural and functional problems that were limiting its use in production-level, safety-critical, and embedded systems.

Differently from ROS 1, which was mainly created for academic and research applications, ROS 2 introduces a modular and platform-independent architecture that can run on many types of hardware—from small embedded systems with limited resources to big and complex robotic platforms [1]. The main reasons behind the development of ROS 2 are the need for real-time capabilities, better multi-robot communication, improved security, and stronger support for industrial environments. Table 2.1 gives a summary of the main architectural improvements of ROS 2 compared to ROS 1, especially in communication, real-time features, and security [2].

Table 2.1: Comparison between ROS 1 and ROS 2

Feature	**ROS 1**	**ROS 2**
Communication	Custom protocol based on TCPROS/UDPROS	DDS (Data Distribution Service), standard middleware
Real-Time Support	Limited or external	Native support through real-time DDS and executors
Multi-Robot Support	Weak; complex workarounds needed	Robust via namespaces, remapping, and DDS discovery
Security	None built-in	Security plugins via DDS (authentication, encryption, access control)
Operating Systems	Primarily Linux (Ubuntu)	Linux, Windows, macOS, RTOS (real-time systems)
Launch System	XML-based (roslaunch)	Python/YAML-based (launch system 2.0)
Modularity	Monolithic nodes; poor composability	Component-based architecture; lifecycle nodes supported
Active Development	No longer actively developed	Actively maintained and extended (latest: ROS 2 Jazzy)

2.1.1 Architectural Overview

At its base, ROS 2 uses a distributed and peer-to-peer architecture where the software parts are divided into nodes. Each node includes a specific function, like perception, planning, or actuation, and it communicates with the others through a clear interface using topics, services, and actions [2]:

- **Topics** enable asynchronous, many-to-many communication using a publish-subscribe model.
- **Services** provide synchronous, request-reply communication for client-server interactions.
- **Actions** allow for long-duration tasks that require feedback and intermediate status updates.

The ROS 2 architecture uses a decentralized communication model with nodes, topics, and services, as you can see in Figure 2.1. ROS 2 also adds the idea of executors, which take care of running callbacks, timers, and other components that work with events. With this model, the developer can choose between single-threaded or multi-threaded execution depending on the real-time needs and hardware limits of the system.

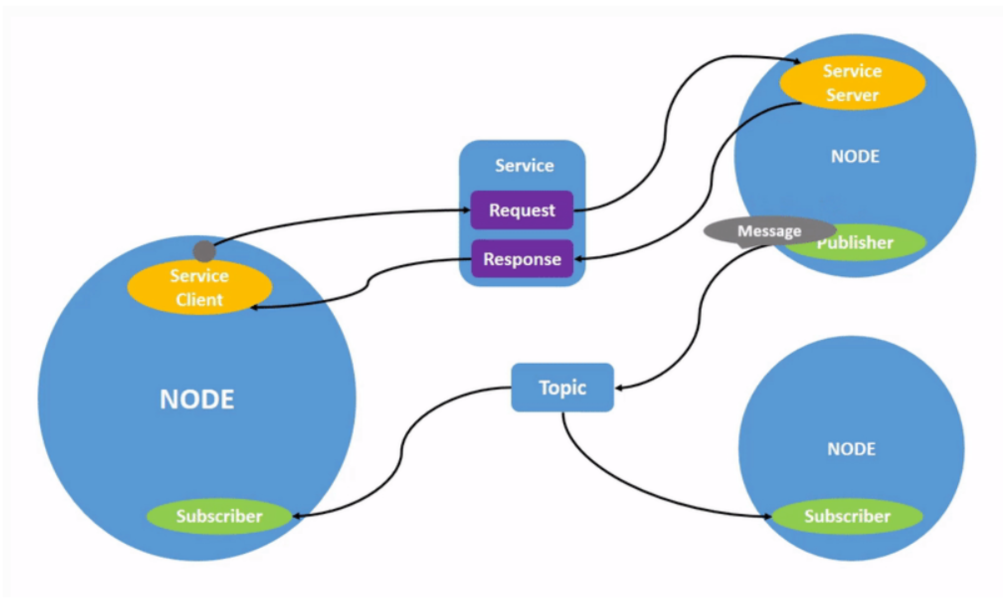


Figure 2.1: ROS 2 node-level communication model, illustrating publishers, subscribers, and client-server interactions using topics and services.

2.1.2 DDS-Based Communication Layer

One of the most significant changes introduced in ROS 2 is the transition from the custom communication infrastructure used in ROS 1 to a standardized middleware based on the Data Distribution Service (DDS) [3]. DDS offers a robust, real-time, and decentralized communication framework that brings several key advantages.

First, it enables fine-grained control over Quality of Service (QoS) parameters, including aspects such as reliability, durability, and latency constraints, allowing systems to be tuned for specific performance requirements. Additionally, DDS supports decoupled, peer-to-peer communication, removing the dependency on a centralized master node and thereby improving overall system flexibility.

This architecture also enhances scalability and fault tolerance, making it particularly well-suited for complex, distributed, or multi-agent robotic systems. Moreover, ROS 2 provides compatibility with a variety of DDS implementations, such as Fast DDS, Cyclone DDS, and RTI Connext, giving developers the flexibility to choose the middleware that best fits their needs in terms of performance, licensing conditions, and deployment scenarios.

2.1.3 The ROS 2 Jazzy Distribution

This work is based on ROS 2 Jazzy distribution, officially released in May 2024. Designed to work with Ubuntu 24.04 LTS, Jazzy brings many important improvements that increase usability and performance. Some of the main updates include native support for C++20 and Python 3.10 or newer, giving developers the possibility to use modern programming features and benefit from better runtime efficiency. The distribution also comes with improved debugging and profiling tools, which help to analyse and optimize the system in a more effective way [4]. Security has also been improved by adding secure communication between nodes, making the system more protected in distributed robotic applications. In addition, Jazzy introduces a more modular launch system that supports both YAML and Python configuration files, allowing more flexibility when setting up the system. The integration with simulation tools and hardware drivers has also been enhanced, making the interaction with both virtual and physical elements smoother. To complete all these improvements, the developer experience has been made better thanks to a more advanced command-line interface (ros2), a more stable Colcon-based build process, and better support for real-time features on Linux kernels that allow it.

2.1.4 ROS 2 for Multi-Robot Systems and Simulation

ROS 2 is now widely used in the area of multi-robot research and development, mainly because of its strong support for distributed computing and efficient message exchange between multiple agents [2, 1]. It provides powerful features like namespaces and remapping, which make possible to run the same node instances on different robots at the same time without any conflict. Its compatibility with various simulation platforms (especially Gazebo and Ignition) has been recently extended with the integration of Open 3D Engine (O3DE), allowing simulations that are photorealistic and physically more accurate. Moreover, ROS 2 supports real-time control of actuators and easy integration of sensors, using both standard and custom hardware drivers. In this thesis, ROS 2 is used as the main infrastructure to manage task orchestration, enable communication between agents, execute motion plans, and connect the virtual simulation environment with the real robotic system.

2.2 MoveIt

MoveIt is one of the most widely used open-source motion planning frameworks for robotic manipulation. Initially developed as part of the ROS ecosystem, MoveIt provides a comprehensive suite of tools and libraries that support key aspects of robotic motion, including kinematic modeling, collision avoidance, trajectory planning, and sensor integration. Its modular architecture, extensibility, and active community support have made it a de facto standard in both academic and industrial robotics applications [5].

2.2.1 Core Architecture and Capabilities

MoveIt integrates several key subsystems to support advanced robotic motion planning:

- **Kinematics:** MoveIt supports both forward and inverse kinematics using plugin-based solvers. While the default kinematic plugin is based on the Kinematics and Dynamics Library (KDL), more efficient solvers such as IKFast and Trac-IK can be used to improve the computational performance [6].
- **Motion Planning:** The planning module is built on top of the Open Motion Planning Library (OMPL), which offers a set of sampling-based motion planning algorithms like Rapidly-exploring Random Trees (RRT), Probabilistic Roadmaps (PRM), and their improved variants [7].
- **Collision Detection:** MoveIt makes use of FCL (Flexible Collision Library) and OctoMap for real-time collision checking and for the volumetric representation of the robot's environment. These tools allow the system to update the planning scene dynamically and to avoid both static and moving obstacles [8].
- **Trajectory Execution:** Once a valid trajectory is found, MoveIt communicates with the ROS control stack or with custom hardware drivers to send joint-level commands to the actuators. The trajectories are time-parameterized and optimized to satisfy constraints on velocity, acceleration, and jerk.
- **Planning Scene:** At the center of MoveIt's planning strategy is the Planning Scene, which keeps a current model of the robot and its surrounding environment. This data structure contains all the necessary information for real-time, constraint-aware motion planning.
- **Sensor Integration:** MoveIt supports the integration of external sensors (e.g., RGB-D cameras, LiDAR), which can be used to update the planning scene with real-time data and support reactive planning.

Figure 2.2 provides a high-level overview of MoveIt 2's planning pipeline, illustrating how user interfaces, planning plugins, and control modules interact through the `move_group` node.

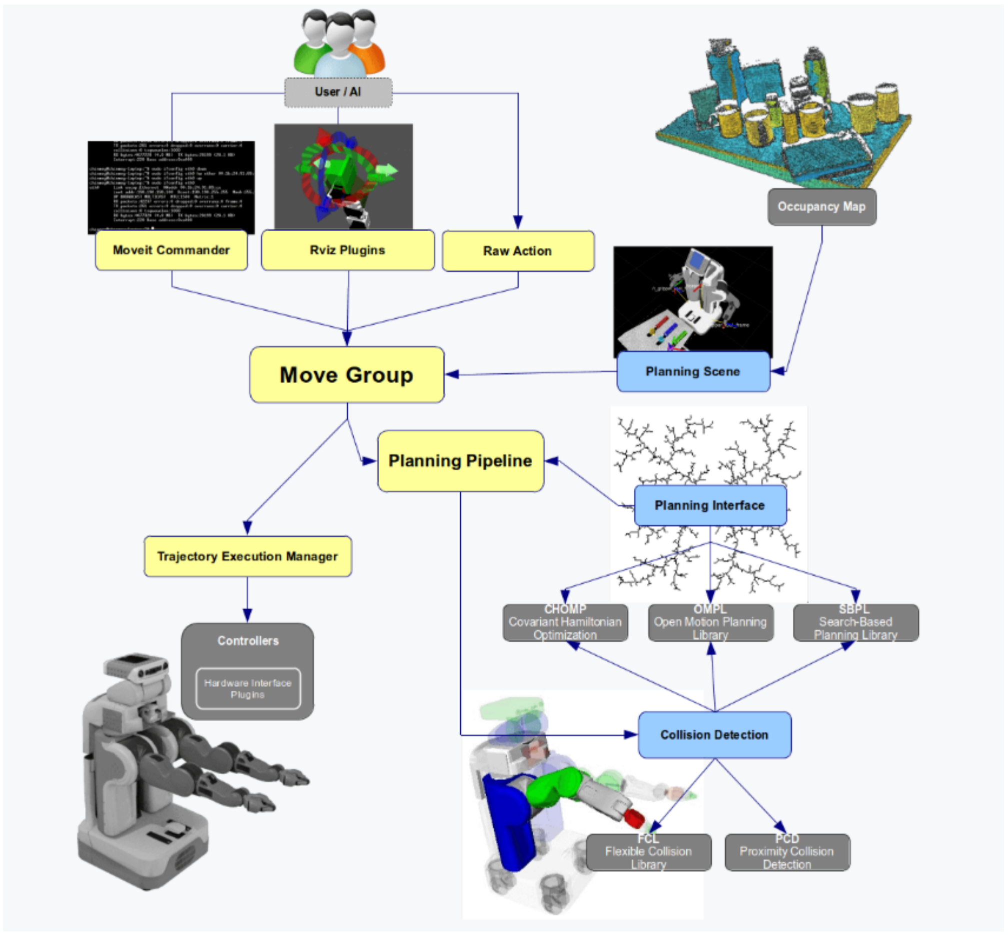

User / AI
Moveit Commander
Rviz Plugins
Raw Action
Occupancy Map
Planning Scene
Move Group
Planning Pipeline
Planning Interface
Trajectory Execution Manager
Controllers
Hardware Interface Plugins
CHOMP
Covariant Hamiltonian Optimization
OMPL
Open Motion Planning Library
SBPL
Search-Based Planning Library
Collision Detection
FCL
Flexible Collision Library
PCD
Proximity Collision Detection

Figure 2.2: Functional architecture of the MoveIt 2 framework. The system consists of user interface plugins (RViz, Commander), a planning pipeline integrating various planners (OMPL, CHOMP, SBPL), and modules for robot execution and control. Image sourced from the official MoveIt documentation: `https://moveit.ai/documentation/concepts/`

2.2.2 Setup and Configuration

MoveIt offers a semi-automatic setup process using its Setup Assistant, a tool that helps developers to generate the needed configuration packages adapted to a specific robotic platform. In this setup phase, the user defines important components like the robot description—usually written in URDF or Xacro—and also the semantic information through SRDF files. The configuration includes the definition of kinematic chains, planning groups, and also the controller interfaces and execution pipelines. In addition, the Setup Assistant allows to configure end-effectors and grasping strategies, making possible more advanced manipulation behaviors. This modular design makes it easier to adapt MoveIt to different robotic systems, and it supports both single-arm and multi-arm setups, increasing its flexibility for many use cases.

MoveIt Setup Assistant: System Configuration

To configure MoveIt 2 correctly for the UR10e arm and the Robotiq 2F-140 gripper, the MoveIt Setup Assistant was used. This graphical tool makes it easier to generate the configuration package needed for motion planning.

The setup followed a structured sequence of steps, explained below:

- **Importing the Robot Model (URDF/XACRO)**
  The file `ur10e_adapter_robotiq_2f_140_.urdf` was used to load the complete robot description, including the UR10e manipulator, a mechanical adapter, and the Robotiq 2F-140 gripper. The adapter was modeled as an intermediate link connected to the `wrist_3_link`, representing the real physical interface between the robot arm and the gripper.
- **Automatic SRDF Generation**
  The SRDF (Semantic Robot Description Format) was automatically generated by the Setup Assistant and included important semantic elements like planning groups, end-effectors, and some predefined poses.
- **Definition of Planning Groups**
  Two main planning groups were created:
  - `arm`: including the six joints of the UR10e
  - `gripper`: defined using the joints of the Robotiq 2F-140
- **End-Effector Configuration with Adapter**
  The end-effector was not attached directly to the `wrist_3_link`, but to the final link of the adapter. This solution reflects the real hardware configuration and improves the accuracy of collision checking and motion planning. Even if the adapter has only small influence on kinematics, it was included to keep the geometry coherent between simulation and real robot, especially for tasks where reach or object size is important.
- **Joint Limits Configuration**
  Joint limits for position and velocity were directly imported from the URDF file, without manual changes.
- **Virtual Joint Creation**
  A fixed virtual joint was defined between the `world` frame and the `base_link` of the robot.

- **Predefined Poses Setup**
  Several predefined poses were created using the Setup Assistant GUI, including: `home`, `pregrasp`, `grasp`, and `drop`. These poses were then used during simulations and physical executions.

- **Collision Matrix Generation**
  The Setup Assistant also generated the default collision matrix, disabling collision checking between adjacent or non-relevant links. This step helps reduce unnecessary computations during motion planning while keeping the system safe.

- **Controller Configuration**
  A `joint_trajectory_controller` was configured for the `arm` group and made compatible with `ros2_control`. The gripper was handled separately using a dedicated action server.

- **ROS 2 Controller YAML Files**
  The controller parameters were saved into YAML files inside the `config` folder. These included:

  - `ros2_controllers.yaml`, specifying the joint trajectory controller for the arm and the gripper
  - `ros2_control.xacro`, which loads the hardware interface and the controller manager

- **MoveIt Controller Configuration**
  The `moveit_controller_manager` was set up using the Setup Assistant. The YAML file `controllers.yaml` maps each planning group to the right controller. This allows MoveIt to communicate correctly with the `ros2_control` system.

- **Planner Settings**
  `RRTConnect` was selected as the default planner. All the other MoveIt settings were left as default.

- **Configuration Package Generation**
  The generated `ur10e_moveit_config` package includes:

  - Updated URDF and SRDF files
  - YAML files for planning groups, controllers, and planner settings
  - Launch files for `move_group` and RViz

The whole configuration was done using the Setup Assistant graphical wizard. The adapter was considered from the beginning to ensure both geometrical accuracy and correct functioning of the system, both in simulation and in the real robot.

2.2.3 Transition to ROS 2: MoveIt 2

MoveIt has been progressively moved to ROS 2 through the MoveIt 2 project, which focuses especially on improving performance, modularity, and support for real-time systems. MoveIt 2 works fully with the ROS 2 Jazzy distribution and uses DDS-based communication to handle planning requests, get feedback from controllers, and observe the execution process [9]. Its architecture has been updated to allow asynchronous planning and execution, which makes motion workflows more reactive and less dependent from each other. MoveIt 2 also includes tools for real-time monitoring of trajectories, better multi-threading support, and more optimized callback management. The system is made to integrate well with ROS 2 executors and lifecycle nodes, following the component-based design of ROS 2 and helping to create robotic applications that are easier to maintain and more predictable in their behavior.

2.2.4 Application in Simulation and Physical Systems

MoveIt is widely used in both simulation and real robotic systems. In simulation, it is usually combined with RViz for visualization and debugging, and with simulators like Gazebo, Ignition, or, in this work, Open 3D Engine (O3DE). In real setups, MoveIt communicates with the low-level control through standard hardware abstraction layers. The framework supports interactive planning with GUI tools, but also allows programmatic control through the MoveGroupInterface (C++/Python APIs), making it good for both research projects and production-level applications.

2.2.5 Role in This Work

In this thesis, MoveIt 2 is used as the main engine for trajectory planning and execution. It takes care of computing motion paths that avoid collisions for robotic arms, both in simulation and in the real world, using the current planning scene and task conditions. The flexibility of the framework—its capacity to work in different environments, use sensor data, and run the same plans in both simulation (with O3DE) and real robots—is a key part of the global system architecture.

2.3 RViz

RViz (ROS Visualization) is a 3D visualization tool that plays a critical role in the development, debugging, and demonstration of robotic systems within the ROS ecosystem. It provides a real-time graphical interface for visualizing a wide range of sensor data, robot states, motion plans, and environmental models, thereby enabling intuitive understanding and rapid development of complex robotic applications [10].

Originally developed for ROS 1, RViz has undergone significant updates to support the ROS 2 middleware stack, maintaining its position as a core development tool for roboticists and engineers.

2.3.1 Capabilities and Core Features

RViz provides a modular plugin-based architecture that allows users to customize the visualization according to the specific data streams and components in use. Among its most important features we can find:

- **Robot Model Visualization:** RViz shows 3D models of robots based on URDF/Xacro descriptions, letting users monitor the current configuration of joints, links, and end-effectors in real time.

- **Sensor Data Rendering:** Sensor outputs like point clouds (e.g., from LiDAR or RGB-D cameras), laser scans, and camera images can be visualized directly in RViz. These tools are very helpful for debugging perception systems and for mapping the environment.

- **TF Frames Visualization:** RViz works closely with the TF (Transform) library, making it possible to inspect in real time the transformations and relationships between coordinate frames of the robot and the world.

- **Interactive Markers and Planning Tools:** RViz supports interactive markers to give user input during manipulation tasks, for example to define target poses or path constraints for robotic arms. This is especially useful when working together with motion planning frameworks like MoveIt [11].

- **Manual Joint Control and Reverse Planning:** RViz allows users to manually modify individual joint values, which is useful for debugging or testing specific configurations. Additionally, when used with MoveIt, it supports reverse planning from a desired end-effector pose, letting users generate joint trajectories interactively by moving the goal markers in the 3D view.

- **Path and Trajectory Visualization:** Planned trajectories, feedback from controllers, and goal targets can be shown as 3D paths and markers, allowing users to verify and adjust motion plans and control strategies before executing them on the real robot.

2.3.2 Integration with ROS 2

RViz has been successfully ported to ROS 2 under the name RViz2, preserving backward compatibility while incorporating a range of enhancements made possible by the architectural advancements of ROS 2. Among these improvements is native support for DDS-based communication, which facilitates more robust and scalable data exchange across distributed nodes. RViz2 is fully compatible with ROS 2 interfaces, including topics, services, and actions, and it extends support to the new message types introduced in ROS 2. The visualization framework also offers improved capabilities for plugin development, enabling the creation of custom visualization tools tailored to specific robotic platforms and applications. In alignment with ROS 2's multi-threaded execution model, RViz2 delivers better real-time performance and more efficient event handling [12]. Additionally, it integrates with ROS 2 lifecycle nodes and offers enhanced utilities for logging, diagnostics, and debugging—features that are critical for deploying and maintaining complex, multi-node robotic systems in both simulated and real-world environments.

2.3.3 Role in Robotics Development

RViz is not only a development tool but also an important interface for operators, researchers, and other people involved in robotics, to interact with and better understand the internal state of robotic systems. In simulation processes, it offers a stable front-end to visualize dynamic environments, planning results, and diagnostic data. In real-world applications, RViz can be used for system monitoring, safety control, and even for manual override of autonomous behaviors. Its high level of customization makes it adaptable to many types of robots, like industrial arms, mobile manipulators, or aerial drones.

2.3.4 Relevance to this Work

In this thesis, RViz2 had a central role in both the simulation and the testing phase on real hardware. It was especially useful for showing the robot configurations and motion trajectories created by MoveIt, and also for observing the planning scene, including obstacles and target poses for the robot's end-effector. RViz2 also allowed direct interaction with the simulated environment, which was important to verify the system behavior before testing on the physical robot. Thanks to its real-time visualization, it was possible to get immediate feedback on task execution, motion planning, and dynamic scene generation. For this reason, RViz2 has been a key tool during the full development and debugging cycle.

2.4 Containerization in Robotics: Docker

Modern robotic systems are becoming more and more complex, with many software dependencies, hardware interfaces, and middleware components. Because of this, reproducibility and portability are now big challenges in robotics development. In this situation, Docker is becoming a powerful solution to package robotic applications together with their execution environments into isolated and lightweight containers. Even if Docker was originally made for cloud and web-scale applications, today it is more and more used in robotics to make deployment, testing, and collaboration easier [13].

2.4.1 Fundamentals of Docker

Docker is a platform that helps to automate the deployment of applications inside containers. A container is a self-contained environment that includes everything the application needs: code, libraries, and system tools. In contrast to virtual machines, containers share the same OS kernel as the host system, which gives faster startup times, less overhead, and better performance [14].

Key Docker concepts relevant to robotics include:

- **Docker Images**: Immutable blueprints for containers, defining the software stack and configuration.
- **Docker Containers**: Running instances of images, isolated from the host system.
- **Dockerfiles**: Scripts used to automate the creation of Docker images by specifying system packages, dependencies, and configuration steps.
- **Volumes and Networking**: Mechanisms for sharing data between containers or with the host system, and for enabling inter-container communication.

2.4.2 Benefits of Docker for Robotic Applications

The use of Docker in robotics brings several clear advantages:

- **Dependency Management**: Robotic systems often depend on many software layers (e.g., ROS, drivers, libraries). Docker containers help to keep versions and configurations consistent across development, testing, and production stages.
- **Reproducibility**: Since Docker encapsulates the full execution environment, it ensures that tests and behaviors can be reproduced reliably on different machines or by different users [15].

- **Modularity**: Robotic systems are usually modular (for example, perception, planning, control), and containers match well with this structure. Each module can be developed, tested, and deployed independently.
- **Platform Independence**: Containers can run on any system that supports Docker, including local computers, cloud platforms, and embedded devices, as long as the hardware requirements (like GPU access) are satisfied.
- **Isolation and Stability**: Containers isolate processes from each other, reducing the risk of conflicts between dependencies or problems caused by updates and parallel development.

2.4.3 GPU and Hardware Integration

Robotics applications that use simulation, machine learning or need high performance computing often need GPU acceleration. Docker can work with NVIDIA GPUs using the NVIDIA Container Toolkit, which allows the container to use hardware acceleration like CUDA and cuDNN [16].

Also, Docker makes possible to mount device interfaces (for example USB, serial ports, CAN bus) from the host to the container, so the container can talk directly with sensors and actuators in the real world. This feature makes Docker not only good for simulations but even for real robots used in practical environments.

2.4.4 Use in ROS and Robotics Frameworks

The ROS community officially supports Docker and provides prebuilt images and tools for container orchestration that help in development process. For instance, ROS Docker images can be found on Docker Hub and give ready-to-use environments for all ROS versions, also ROS 2, making easier to setup and deploy robotic systems. Many times, Docker Compose is used to manage systems with multiple containers, like when simulation, perception and control are separated in different containers. This allows to create more modular and scalable robotic architectures. Also, Docker is now often part of CI/CD pipelines, where it helps in automatic building and testing of robotics software. Modern robotics platforms such as Autoware and Apollo use container-based architectures to allow scalable deployment and modular development. This shows how containerization is becoming more and more popular for increasing flexibility and efficiency in advanced robotic systems.

2.4.5 Relevance to this Work

In this thesis, Docker was used to containerize the full robotic development environment, including ROS 2 Jazzy, O3DE, MoveIt 2, some custom control nodes, and different support libraries. Using containerization in this way gave many important benefits: it made easier the installation on different machines, keeping the setup the same on all systems; it improved system stability during many simulation and real robot tests; and it allowed to use GPU-accelerated simulation and NVIDIA drivers without breaking the host system's stability. Moreover, Docker containers were very useful to run repeatable simulations, so the same environment could be always reproduced exactly. This also helped a lot in moving from virtual robots to real ones, making development and testing process more efficient and smooth.

2.5 Simulation Framework: Open 3D Engine (O3DE)

2.5.1 Overview of O3DE in Robotics

The Open 3D Engine (O3DE) is a high-performance and open-source real-time 3D platform, which comes originally from Amazon Lumberyard and now is managed by the Linux Foundation. It was made not only for the game industry but also for applications based on simulation. O3DE has a modern and modular architecture that is able to support complex systems, like robotics and autonomous agents. Its strong rendering features and the fact that it is very flexible make it a good option for simulating environments that are detailed and physically realistic, where both the visual quality and the computational performance are important [17].

O3DE supports advanced rendering technologies through APIs such as Vulkan and DirectX 12 and includes a native physics engine based on NVIDIA PhysX [18]. This foundation allows for the simulation of dynamic, articulated systems under realistic physical constraints, which is essential in applications involving robotic manipulators and mobile platforms. Unlike traditional robotics simulators that emphasize kinematic accuracy at the expense of graphical fidelity, O3DE offers an integrated environment where physical realism and photorealistic rendering coexist [19].

2.5.2 Physics and Control Integration

The physics subsystem in O3DE offers real-time simulation of rigid bodies, collision detection, and support for articulated joints—features that are very important to correctly simulate robotic arms, grippers, and interactions with the environment. To simulate manipulators, it is necessary to define in detail the kinematic chains, the constraints, and the movement limits. Putting all these elements together makes possible to test complex motion plans, like the ones created by planners such as MoveIt, inside a realistic virtual space.

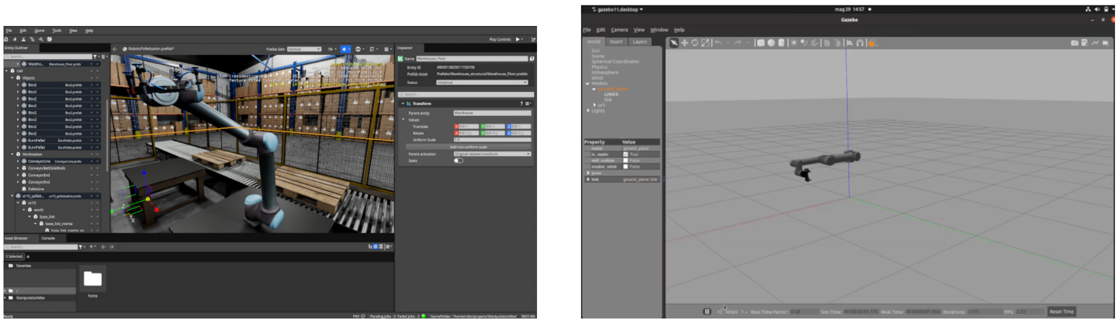
To connect O3DE with the robotic middleware, a communication interface was created for exchanging control and sensor data with ROS 2. This included publishing joint commands to the simulated robot and receiving state feedback inside ROS 2 nodes, to keep the planning part and the simulation in sync. Thanks to this integration, it was possible to verify correctly the execution of trajectories, the interaction with the environment, and the collision detection, before applying the same behavior to the real robot.

2.5.3 Scenario Realism and Visual Fidelity

More than just physical precision, O3DE also plays an important role in increasing visual realism, which becomes more and more relevant in robotics research, especially in fields like perception, scene understanding, and machine learning. The engine supports dynamic lights, shadows, material rendering and also post-processing effects that help to reproduce real-world conditions more closely. This kind of visual quality is very useful when developing vision-based control systems or testing AI models under different environmental situations [20, 21, 22].

Figure 2.3 presents a side-by-side comparison between O3DE and Gazebo renderings. The difference in realism explains the choice of O3DE for simulation tasks where high-fidelity perception is necessary.

In this thesis, O3DE has been used to build and test 3D scenes that look similar to the real environments where the physical robot would work later. These virtual scenarios had obstacles, different light conditions, and space arrangements that tested the robot's planning and control algorithms. The possibility to simulate both structured and unstructured environments helped to improve the robustness of the global system design.



(a) O3DE simulation (b) Gazebo simulation

Figure 2.3: Comparison between simulated environments rendered in O3DE and Gazebo for the same robotic manipulation task. The O3DE environment (a) provides higher visual fidelity, including dynamic lighting and detailed textures, while the Gazebo setup (b) is more limited in rendering quality. This visual difference can impact perception-driven tasks and the realism of AI-generated scenarios.

2.5.4 Advantages Over Traditional Simulators

Even if simulators like Gazebo and Ignition are often used with ROS, they can have some limitations when advanced visual effects or detailed rendering control are needed. On the other hand, O3DE solves these problems thanks to its native support for ROS and its powerful rendering engine, which makes possible to create real-time scenes with cinematic-level quality. This is very important in workflows where we need generative environments, high-fidelity perception, or reinforcement learning [20, 21, 22].

Also, O3DE's modular design allows to build custom tools and extensions that go further than what traditional, fixed simulators can do [17].

The flexibility of the engine, together with its support for GPU acceleration, lets it scale well for big environments and fast update rates. This performance is especially important when testing AI-based systems that must constantly check the environment and adapt their behavior [19].

2.5.5 Application in This Work

In this thesis, O3DE was an important part in the validation loop of the robot control system. By simulating the robotic arm movements and checking its behavior in a realistic and high-quality environment, it was possible to find planning problems, adjust trajectories, and improve the robustness of the whole system before trying anything on the real robot.

In addition, thanks to the integration with ROS 2 and MoveIt, it was possible to transfer control policies and motion plans easily between the simulation and the real robot. This consistency between simulation and reality helped to reduce the chances of failure during deployment and allowed faster development cycles. The results showed that the behaviors tested in O3DE were very similar to the ones seen on the real robot, proving that the simulator is effective as a tool for pre-deployment validation [19].

2.6 AI-Driven Environment Generation

In the last years, artificial intelligence (AI) has made big progress in automating the creation and modification of virtual environments. In the field of robotic simulation, generating environments is a very important step for testing and validating autonomous systems in many different and complex situations. Before, these environments were usually made by hand or using predefined templates. But now, AI techniques—especially those from generative models and procedural content generation—made possible to create and change simulation environments in a more dynamic way, starting from some initial conditions or specific task requirements [23].

2.6.1 Generative Models for Environment Creation

Generative models are a class of machine learning algorithms capable of creating new data instances that resemble a given dataset. In the context of 3D environments, AI models such as Generative Adversarial Networks (GANs) [24], variational autoencoders (VAEs) [25], and neural networks have been employed to generate realistic and diverse virtual environments that mimic real-world conditions. These environments can range from simple object arrangements to highly complex scenes that simulate physical constraints, lighting conditions, and various sensor inputs [26].

One of the key advantages of AI-driven environment generation is the ability to create environments with varying degrees of complexity and realism, which can be tailored for specific testing scenarios. For example, AI models can generate randomized obstacle courses or environments with varying lighting and textures, which are essential for training robotic systems to handle diverse conditions in the real world [27]. This ability to automatically generate such scenarios is particularly valuable in the context of domain randomization, a technique often used to improve the robustness of deep reinforcement learning models by exposing them to a wide range of variations during training [27].

In the case of robotic manipulation tasks, AI can also be used to generate task-specific environments. For example, a generative model could be trained on a dataset of kitchen environments and then be tasked with creating new, unseen configurations of objects on kitchen countertops, enabling testing of robot arms in scenarios that may not have been explicitly pre-designed.

2.6.2 Procedural Content Generation (PCG)

Procedural Content Generation (PCG) means creating content using algorithms instead of designing it manually. In the field of robotics, PCG has been very helpful for generating environments in a dynamic way during simulation. Compared to the traditional static environments, PCG allows to create infinite and changing worlds that can test the robot's abilities in many different tasks.

PCG can be applied to generate both 3D objects and environmental elements like terrains, buildings, paths, and obstacles. This method allows robots to interact with a large variety of scenarios without needing to manually build each one, which strongly increases the amount and diversity of data for training and testing. Many of the modern game engines, including O3DE as mentioned before, support PCG techniques, making it easy to combine AI-based content generation with realistic and high-quality simulations [28].

2.6.3 Reinforcement Learning for Environment Adaptation

Even if reinforcement learning (RL) is more and more used to create environments that change depending on how the agent behaves—allowing curriculum learning and increasing task difficulty—this thesis does not follow that direction. Instead, it focuses on static environment generation using AI models based on image and text input. However, adaptive scene generation is still a very interesting path for future work, especially for creating robotic behaviors that are more robust and able to generalize better in simulation [29].

These AI-based adaptive environments could, for example, gradually add obstacles, change object characteristics, or modify sensor conditions, making the robot face a wider variety of tasks and situations.

2.6.4 Integration with ROS 2 and Simulation Frameworks

The integration of AI-generated environments with simulation tools like O3DE and ROS 2 allows for very flexible and scalable testing workflows in robotics. In this thesis, trained generative models were used to create different robotic arm scenarios in a dynamic way, with random object positions, various lighting setups, and multiple robot configurations. Compared to environments designed manually, this method gave a much wider range and better realism to the test cases, helping to evaluate the robot's behavior in a more complete way.

The simulated scenes could also adapt in real time, thanks to ROS 2's Action and Service interfaces, which managed updates in the environment using feedback from the robot or its controllers. Moreover, the system was compatible with machine learning libraries like TensorFlow and PyTorch, making easier to include generative models focused on specific tasks. This helped even more to increase the variation and reactivity of the virtual environments.

Chapter 3

System Architecture

The system architecture developed in this thesis is made to offer a flexible and scalable framework for robot simulation and motion planning, using ROS 2, MoveIt, RViz, O3DE, and Docker. The current configuration is based on semi-manual interaction for planning robot poses and visualizing them, with real-time feedback and connection between the simulation environment and the motion planning module. This section explains the different components of the system, how they work together, and how the whole setup is deployed.

3.1 Main System Components

The system architecture consists of several core components, each with specific roles and interactions:

- **ROS 2 Core:** ROS 2 serves as the communication backbone for the entire system. It facilitates message passing between the components and ensures synchronization across different parts of the architecture, enabling efficient data flow between the simulation environment, motion planning, and control systems.

- **MoveIt:** MoveIt is responsible for the motion planning and trajectory generation. It is configured with a robot model (URDF), including predefined poses and constraints. MoveIt plans motion trajectories based on target poses sent from RViz and computes the necessary actions to move the robot to those positions. The planned trajectories are subsequently sent to the robot's controller for execution.

- **RViz:** RViz acts as the primary visualization tool for robot status and motion planning. It is used for manually setting target poses, visualizing the robot's state, and monitoring the motion planning process. RViz interacts with MoveIt by sending target poses and receiving feedback on planned trajectories.

- **O3DE (Simulation Environment):** O3DE serves as the 3D simulation environment, providing a realistic visualization of the robot's movements in a virtual world. It is synchronized with ROS 2 through the ROS 2 Frame Component, which allows O3DE to display real-time updates based on robot movements. O3DE reflects the planned motions generated by MoveIt and provides visual feedback through camera and environmental sensors.
- **Camera (Simulated):** A simulated camera is integrated into O3DE to provide visual feedback of the robot's surroundings. The images captured by the camera are transmitted through ROS 2 (e.g., `/camera/image_raw`) and can be used for debugging or further analysis.
- **Joint State Broadcaster:** This component broadcasts joint states from the robot or simulator. It sends the joint positions to the `/joint_states` topic, where they can be visualized and controlled through RViz.

These components create a semi-manual system that is used for planning, visualizing, and executing robot poses. At the moment, the system allows to configure pose targets using RViz, perform motion planning with MoveIt, and show the results in real time inside O3DE. The architecture is modular and has been designed to be easily extended in the future, for automating tasks and adding higher-level coordination.

3.2 Component Interactions

The interactions between the different parts of the system are shown in the table below:

Component	**Input**	**Output**
O3DE (Simulation)	/robot_description, movement commands	Robot state feedback, synchronization updates
ROS 2 Core	–	Transfers messages between components via topics
MoveIt	Pose target (from RViz), URDF model	Planned trajectories (sent to controller)
RViz	/robot_description, TF, Pose Goals	Pose target (sent to MoveIt), visual feedback
Camera (Simulated)	Scene data from O3DE	Images (e.g., /camera/image_raw)
Joint State Broadcaster	Robot/simulator data	/joint_states topic for visualization and control

Table 3.1: Overview of system components, their input data, and output interactions

The system follows a semi-manual flow, where pose planning is carried out through RViz and visualized in O3DE, allowing real-time evaluation of the robot's movements in the simulated environment.

3.3 Integration between RViz, MoveIt, and O3DE

The integration between RViz, MoveIt, and O3DE forms the core of the system. Each component has a distinct but interconnected role:

- **RViz** serves as the interface where users interact with the robot's state and manually set target poses for the motion planning system.

- **MoveIt** handles the motion planning process, generating feasible trajectories to move the robot from its current position to the desired pose. These trajectories are computed based on the robot's configuration and constraints, and the results are sent to the robot's controllers.

- **O3DE** acts as the 3D simulator that provides a realistic visual representation of the robot's environment. It is synchronized with ROS 2 through the ROS 2 Frame Component and reflects the robot's movements as dictated by the trajectory planning in MoveIt.

The integration relies on consistent naming conventions for the robot's frames and joints across ROS 2, MoveIt, and O3DE. Specifically, O3DE uses the ROS 2 Frame Component to:

- Manage frame and joint names associated with each robot entity.

- Dynamically update and publish transformations to the /tf topic when entities with components like JointComponent or ArticulatedComponent are present.

This makes sure that the motion generated by MoveIt is correctly shown inside O3DE. In practice, when a pose is planned using RViz and executed through MoveIt, O3DE receives the related transformations via ROS 2, updating the robot's joint positions and giving real-time visual feedback.

Even if O3DE does not directly control the robot (for example, it does not send control commands through something like /joint_trajectory_controller), it works as a passive but dynamic simulator. The robot's movement is continuously updated using the transformations published from ROS 2, so the visual feedback stays accurate without needing an active controller.

This method allows to visually check the planned motion in detail, in a very immersive environment. Because of that, O3DE becomes a strong tool for simulating and testing robot behavior before using it on the real hardware.

3.4 Handling Mimic Joints for Gripper Simulation in O3DE

While integrating the Robotiq 2F-140 gripper into the simulated environment, an important compatibility issue was found. The original robot URDF uses a mimic joint configuration to keep the left and right fingers moving in sync. However, the Open 3D Engine (O3DE), which uses NVIDIA PhysX, does not support mimic joints natively. According to the official documentation, O3DE currently supports only four types of PhysX joints: Ball, Fixed, Hinge, and Prismatic. None of them can reproduce the mimic behavior between joints [30].

To address this problem, two possible solutions were considered. The first one was to contribute to the O3DE engine by adding a custom mimic joint component. Even if this was technically possible, it would have required a lot of time and complex modifications to the engine, which were outside the goals of this thesis. The second, and more practical option, was to handle the mimic behavior externally through a control system based on ROS 2.

The chosen solution was to implement a ROS 2 node that listens to gripper commands and programmatically reproduces the mimic joint behavior. The architectural limitation caused by the lack of mimic joint support in O3DE is illustrated in Figure 3.1. Here, even though the gripper command is generated by MoveIt2, the engine itself cannot reproduce the synchronized finger movement. As a result, the simulation fails to show the correct motion, despite the control logic being implemented externally through ROS 2.

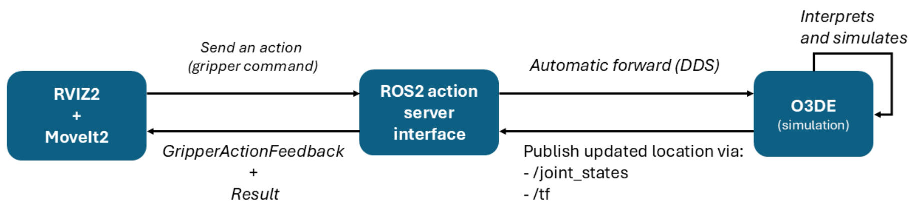


Figure 3.1: System architecture before the implementation of the mimic joint handler. The gripper command is sent, but O3DE cannot simulate the effect due to lack of mimic joint support.

Initially, the system was tested by publishing direct joint values to the simulation, verifying that O3DE correctly interpreted individual joint movements. From there, a translation layer was implemented to listen to standard ROS 2 gripper commands—typically published by MoveIt—and convert them into synchronized joint trajectories for the simulated gripper.

However, additional difficulties came up because MoveIt expects a gripper to be controlled through an action server interface. To solve this, a fake action server was created. This mock server received the goals sent by MoveIt and replied immediately with a success result, even if the real execution was actually managed separately by the control node. This trick made sure that MoveIt's internal state machine worked correctly and didn't cancel the motion plans due to missing feedback.

The final version of the implementation combined all parts into a single and unified ROS 2 node. This node managed the reception of commands, the replication of the mimic behavior, the publishing of joint values, and the feedback to MoveIt—all inside a small and flexible framework. Thanks to this, the simulated gripper could perform stable and responsive pick-and-place actions, very similar to the ones of the real hardware. The improved design of the system is shown in Figure 3.2.

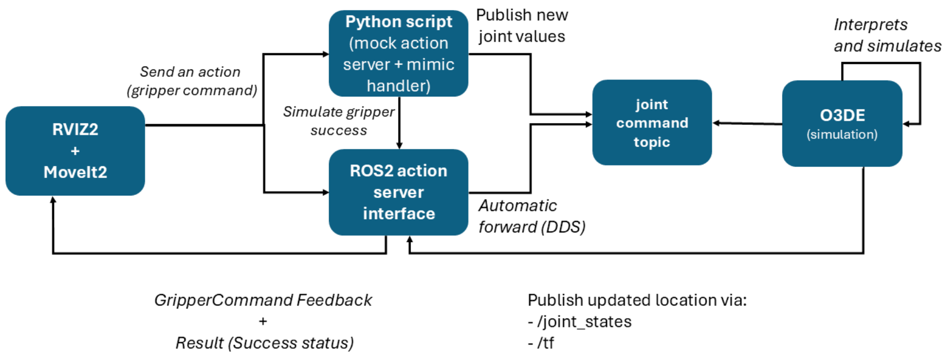

Python script (mock action server + mimic handler)
Publish new joint values
Interprets and simulates
Send an action (gripper command)
Simulate gripper success
RVIZ2 + MoveIt2
joint command topic
O3DE (simulation)
ROS2 action server interface
Automatic forward (DDS)
GripperCommand Feedback + Result (Success status)
Publish updated location via: - /joint_states - /tf

Figure 3.2: Updated architecture with a Python script acting as a mimic joint handler and mock action server. This component simulates gripper success and publishes synchronized joint values, enabling O3DE to replicate the expected behavior.

This workaround shows the flexibility of ROS 2 and underlines how important it is to customize the middleware layer when working with simulation engines that don't fully support the robot model specifications typically used in real-world setups.

Limitations of the Implemented Solution

Even if the workaround was successful in reproducing the expected gripper behavior inside the simulation, there are still some limitations. One main issue is that the simulated execution does not include physically realistic feedback, like contact forces or tactile sensing, which are very important when dealing with soft or reactive objects. The mimic behavior is created in a deterministic way, using predefined relationships between joints, which makes it hard to react or adapt to changes in the environment. Another limitation is related to the synchronization between MoveIt's action interface and the node that publishes joint values. It requires precise timing to avoid race conditions or inconsistent feedback, especially during execution. These factors can affect the reliability of the system if not handled carefully.

3.5 AI-Driven Environment Generation

To make the simulated workspace more realistic and variable, this project includes a dedicated pipeline for generating 3D objects with the help of AI. Instead of using only pre-made assets or manual 3D modeling, some elements of the environment were created through artificial intelligence, using both image-based and diffusion-based techniques. This method allows to reconstruct real-world objects quickly and flexibly from photos, which is very useful for simulating manipulation tasks in a scene that looks and feels coherent both visually and spatially. For this purpose, two complementary tools were used: Meshy (a cloud-based AI service), and StepFun (an open-source toolchain that runs locally).

3.5.1 Meshy: Text and Image-Based 3D Model Generation

Meshy supports different input types, including single or multi-image to 3D, text-to-3D, and combinations of image and text. In this thesis, the workflow was mostly based on generating 3D models from one or more 2D images, sometimes improved by adding text descriptions to better define structural or semantic details.

The Meshy platform uses vision-language models to guess the geometry, texture, and proportions from images, and then exports the object as a textured .fbx file. The .fbx format worked very well with the simulation engine, allowing direct import of both the mesh and its texture. Some of the objects created with Meshy included a joystick, spray can, adhesive tape, gripper, box, chair, and desk—each one inspired by real objects used in the physical setup [31].

Even if Meshy produced meshes with good quality and realistic textures, one frequent problem was the inconsistent scale of the models. Because the dimensions were calculated only from visual input, without any real measurement reference, it was necessary to adjust the size after importing them into the simulation. These changes were made directly inside O3DE, to make sure the simulated objects matched visually with the ones from the real environment.

3.5.2 StepFun: High-Fidelity Mesh and Texture Synthesis

A second AI pipeline was used with StepFun, an open-source tool that runs locally. In contrast to Meshy, StepFun uses a combination of VAE-DiT models for generating geometry and Stable Diffusion XL (SD-XL) for creating textures. The input is a single 2D image, which is converted into a TSDF-based format that captures surface shape and boundaries with high accuracy [32]. The final result is a textured .fbx mesh.

StepFun gave good geometric quality, but the textures sometimes needed corrections. In particular, there were issues with texture consistency and UV mapping, which had to be fixed using Blender before importing into O3DE. Even if StepFun accepts only image input (no descriptive text), it was still able to produce visually accurate models that preserved the main shape of the original object very well.

3.5.3 Comparison Between Real and AI-Generated Objects

To evaluate the realism and visual quality of the AI-generated models, a direct comparison was made between real-world objects and their versions generated using Meshy and StepFun. As shown in Figures 3.3 to 3.5, three representative objects were selected: a wire cutter, a game controller, and a green chair. The objective was to check the geometry, the scale accuracy, and how well the textures were reproduced.

Both tools were able to capture the general shape and visual appearance of the objects, but some limitations were noticed. Meshy usually gave better results for texture mapping, although sometimes the scale was not completely accurate. On the other hand, StepFun generated sharper geometry, but in some cases it showed less reliable UV mapping and texture details that were not very consistent.

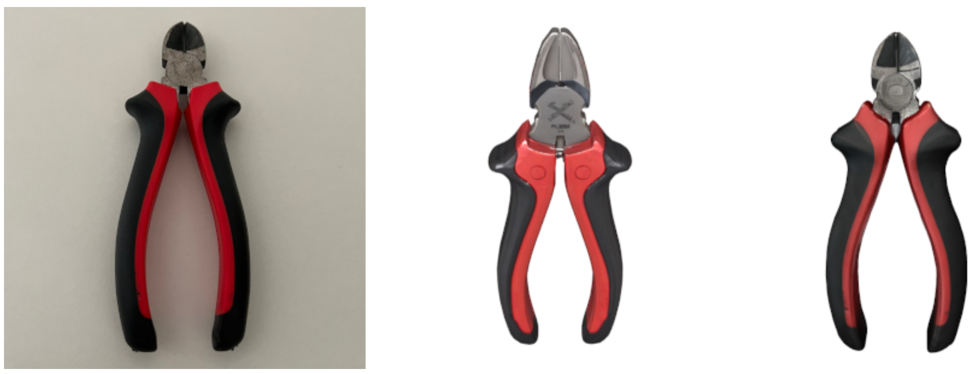


Figure 3.3: Comparison between the real wire cutter (left), the model generated with Meshy (center), and the model generated with StepFun (right).

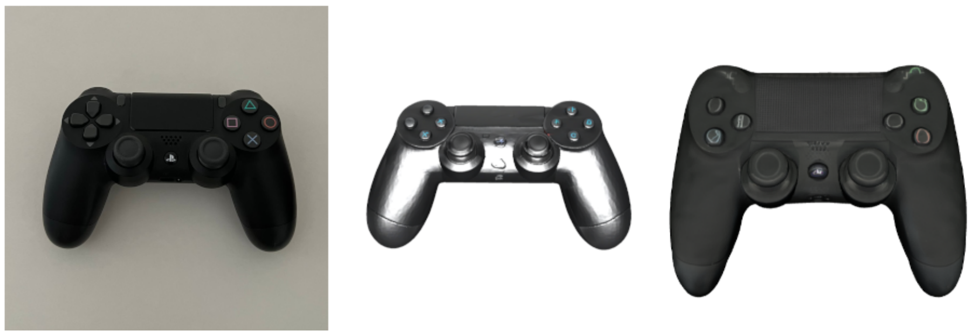


Figure 3.4: Comparison between the real joystick (left), the model generated with Meshy (center), and the model generated with StepFun (right).

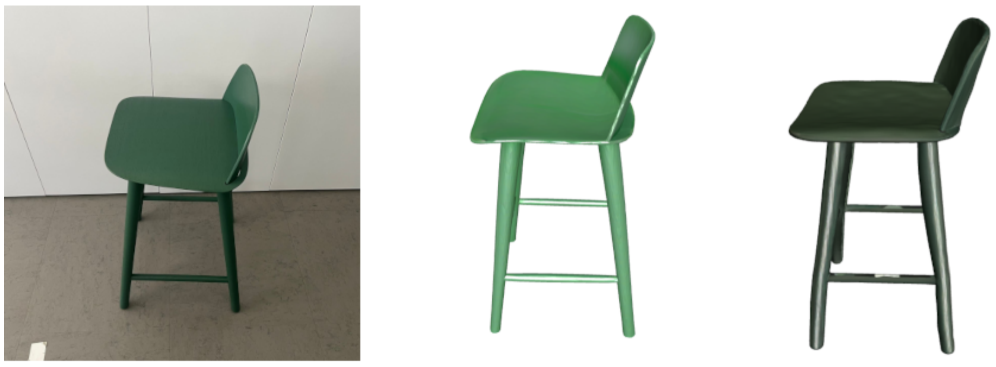


Figure 3.5: Comparison between the real chair (left), the model generated with Meshy (center), and the model generated with StepFun (right).

3.5.4 Integration into O3DE

All the generated models were imported into the Open 3D Engine (O3DE) and added to the simulation scene. After importing, each object was converted into a .prefab, which made it easier to reuse and organize them in different scenes. Then, physical components were assigned to each object, including:

- Static or dynamic colliders,
- Mass and inertia parameters,
- Friction and surface material properties.

This setup allowed the AI-generated objects to be directly used in simulated robotic tasks. For example, items like the joystick, spray can, and tape roll were used in pick-and-place experiments, while the AI-generated desk and chair helped to define the spatial layout of the environment.

3.5.5 Role in System Behavior

The AI-generated objects had an important role in improving both realism and functionality of the simulation. Without these elements, the environment would have remained too abstract or too simple, which would have reduced the ability to test and validate the robot's behavior in a meaningful way. The objects were interactable, physically believable, and visually similar to their real versions, giving a solid base for motion testing and system evaluation. Even if there are still some limitations, like scaling issues or lack of real physical modeling—mainly because there is no metric calibration—the use of AI for generating scene content helped a lot to save time on manual modeling and allowed quick prototyping of realistic scenarios. This part of the system was especially useful during testing and demonstration phases.

3.6 Containerization and Deployment

The system was containerized using Docker to provide a replicable and portable development environment. Unlike more complex setups that may require individual containers for each node, this system uses a single container that includes all necessary modules and dependencies:

- ROS 2 with MoveIt, RViz, and simulation packages.
- O3DE, including environment configuration and integration with ROS 2.

NVIDIA drivers were included to enable GPU support in the simulation environment.

The choice to use a single container was made to simplify the deployment process and avoid the complexity of managing multiple containers. This container includes all the needed dependencies and ensures that the environment stays consistent on different machines.

The Docker container was built using a Dockerfile, which defines the installation of all required components and system configurations. This approach creates a repeatable environment that can be used for development, testing, and deployment on various setups.

The container is stateless, meaning that it doesn't store any data permanently. However, for future versions, it's possible to add persistent storage if needed—for example, to save logs, configuration files, or sensor data.

Chapter 4

Simulation and Virtual Environments

Simulation has an important role in the development and testing of robotic systems, because it makes possible to run safe, repeatable, and cost-saving experiments before applying algorithms and control logic to the real robot. In this chapter, the simulation architecture developed for this thesis is presented, focusing on how O3DE, ROS 2, and MoveIt have been integrated together. It also describes how a realistic virtual environment was created, inspired by the real workspace, and explains the components, testing process, and synchronization strategies that were used to ensure that the simulated behavior is as close as possible to the planned one.

4.1 Robotic Simulation with O3DE

The simulation framework is based on Open 3D Engine (O3DE), a modular, open-source and real-time 3D engine that is able to support high-fidelity rendering and physics simulation. The project started from an O3DE template already configured for robotics, which included native support for ROS 2 through the ROS 2 Gem. This integration removed the need to use external bridges and allowed a direct communication between ROS 2 and the simulation engine.

The robot used in the simulation is a UR10e from Universal Robots, equipped with a Robotiq 2F-140 gripper. The robot model was imported using the URDF import tool available in O3DE, which helps to convert the robot description files into simulation-ready entities. However, some manual adjustments were necessary to fix small issues and to correctly import some mesh components that were not converted in the right way.

Several O3DE-specific components were then added to allow interaction between the robot and the simulation environment [33, 34]:

- **ROS 2 Frame Component:** Synchronizes joint and link frames with ROS 2, publishing real-time transforms via the `/tf` topic.
- **Camera Sensor Component:** Simulates a vision sensor, providing image data over ROS topics such as `/camera/image_raw`.
- **ROS 2 Root Control Component:** Facilitates the global placement and orientation of the robot in the virtual environment.
- **Joint Manipulation Editor Component:** Provides tools for interactively modifying joint configurations during simulation.
- **Joint Trajectory Component:** Receives joint trajectory messages and updates the robot's pose accordingly.
- **Joint Positions Editor Component:** Allows static configuration and visualization of joint states.
- **Finger Gripper Component and Gripper Action Server Component:** Emulate the behavior and control interfaces of the gripper, allowing realistic interaction with objects.

As shown in Figure 4.1, each component plays a specific role in enabling interaction between the O3DE simulation environment and the ROS 2 ecosystem, supporting visualization, control, and physical realism within the robotic workflow.

These components collectively enable full visualization, interaction, and basic control of the robot and its end-effector in a 3D simulated world, while maintaining compatibility with ROS 2 motion planning and visualization tools.

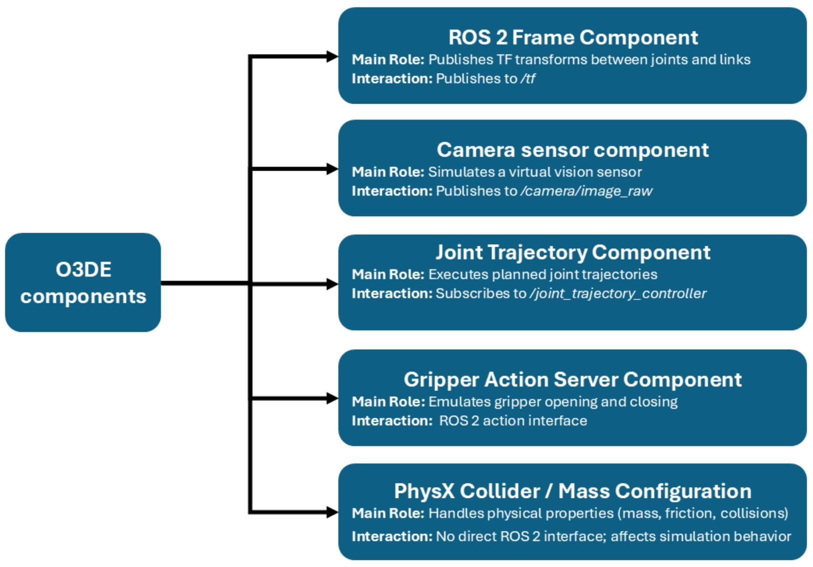

O3DE components
ROS 2 Frame Component
Main Role: Publishes TF transforms between joints and links
Interaction: Publishes to /tf
Camera sensor component
Main Role: Simulates a virtual vision sensor
Interaction: Publishes to /camera/image_raw
Joint Trajectory Component
Main Role: Executes planned joint trajectories
Interaction: Subscribes to /joint_trajectory_controller
Gripper Action Server Component
Main Role: Emulates gripper opening and closing
Interaction: ROS 2 action interface
PhysX Collider / Mass Configuration
Main Role: Handles physical properties (mass, friction, collisions)
Interaction: No direct ROS 2 interface; affects simulation behavior

Figure 4.1: Diagram showing the main O3DE components and their interaction with ROS 2. The ROS 2 Frame Component serves as the central communication hub, interfacing with the camera sensor, joint trajectory executor, gripper action server, and the PhysX physics engine.

4.2 Virtual Environment Modeling

Together with the robotic system, a typical office workspace was recreated inside O3DE. This environment included a desk, various objects, and interactive elements made to simulate the complexity of the real world. The scene was organized using .prefab assets, which are a practical way to manage and reuse groups of components and meshes. The single 3D models were first imported in .fbx format, then modified and completed with physical and ROS-related properties, and finally saved as reusable prefabs.

Even if the whole scene was not created fully by procedural generation or AI tools, some decorative and interactive parts were designed using AI-generated geometry. These elements were added on purpose to increase the realism and diversity of the environment, without the need to model each object manually.

Figure 4.2 illustrates the simulated workspace created in O3DE, including the UR10e robotic arm and various AI-generated objects arranged on a virtual desk.

The virtual environment was not static; it included physical attributes that allowed for meaningful interaction. O3DE's built-in physics engine (based on NVIDIA PhysX) enabled the assignment of physical properties such as mass, friction, restitution, and collision geometry to all simulated elements. This allowed the system to simulate physically plausible interactions such as grasping, contact dynamics, and object displacements.

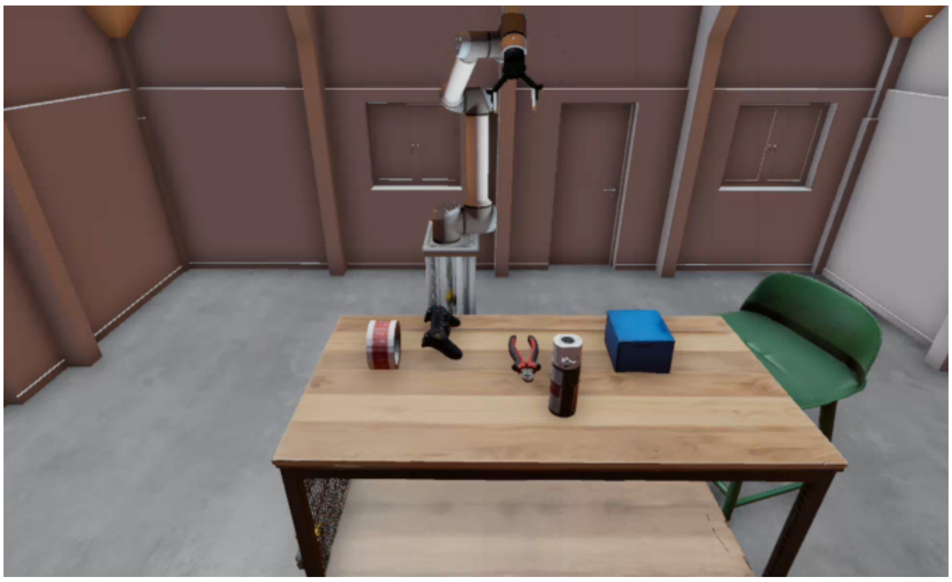


Figure 4.2: Simulated workspace in O3DE, showing the UR10e robot positioned on a desk and surrounded by AI-generated objects such as a joystick, spray can, and tape roll.

4.3 Testing in Simulation

Several test scenarios were carried out inside the simulated environment to check the correctness of motion planning and the integration of the robotic system. The main focus was on pick-and-place tasks, where the robot had to reach and manipulate virtual objects placed in different positions on the desk.

The testing followed a semi-manual process. Target poses were set using RViz and then sent to MoveIt for motion planning. The resulting trajectory was visualized both in RViz and in O3DE, allowing to verify the movement from both a planning and visual perspective. This dual-feedback approach was very useful to detect any mismatch between the planned motion and its physical realism or graphical correctness inside the virtual scene.

O3DE acted as a passive real-time simulator, reacting dynamically to joint transformations published by ROS 2 without directly executing control commands. The synchronization between RViz, MoveIt, and O3DE was effective, with no perceptible latency in the simulation thanks to the high-performance hardware used during development.

The presence of a realistic, real-time, physically active simulation allowed the validation not only of spatial and kinematic correctness but also of environmental integration and task-level feasibility. The enhanced visual fidelity offered by O3DE, compared to traditional tools such as RViz, proved especially useful for evaluating how the robot would operate from a human-centered perspective. Figure 4.4 illustrates the simulation pipeline used during pick-and-place experiments, clarifying the system's modular flow from planning to execution.

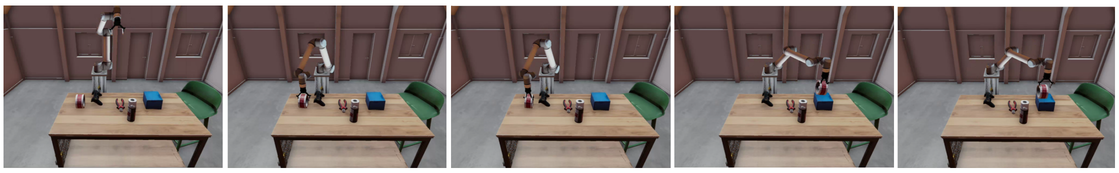


Figure 4.3: Sequence of images showing the phases of a *pick-and-place* task performed in the O3DE simulator. From left to right: (1) initial position, (2) approach to the object, (3) pick, (4) transport, (5) place.

The enhanced visual fidelity offered by O3DE, compared to traditional tools such as RViz, proved especially useful for evaluating how the robot would operate from a human-centered perspective (see Figure 4.3).

This simulation infrastructure creates a solid foundation for future developments, such as dynamic scenario generation, integration with reinforcement learning, and full autonomy pipelines. The combination of O3DE with ROS 2 and MoveIt gives a strong and flexible base where more intelligent and automated features can be added in the next stages of the project.

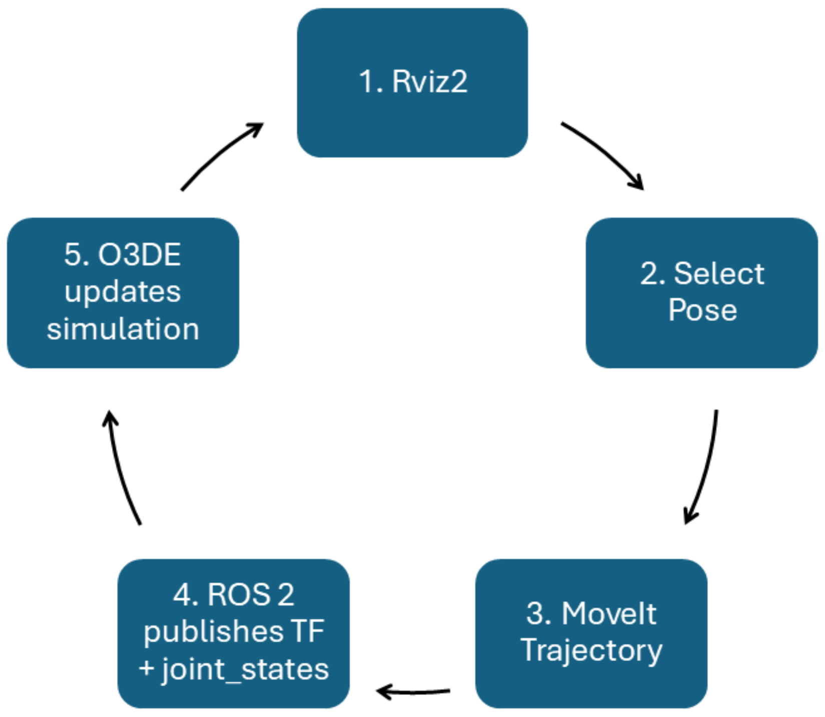

1. Rviz2
2. Select Pose
3. MoveIt Trajectory
4. ROS 2 publishes TF + joint_states
5. O3DE updates simulation

Figure 4.4: Execution pipeline for simulation-based validation. The process begins with pose selection in RViz, followed by trajectory generation in MoveIt, ROS 2 message publishing, and real-time update of the robot's motion within the O3DE environment.

Chapter 5

Task Optimization and Trajectory Generation

The ability to create feasible, collision-free, and realistic motion trajectories is one of the main functions in any robotic manipulation system. In this thesis, trajectory planning was done using the MoveIt 2 framework, integrated into the ROS 2 environment and visualized with RViz. The planned trajectories were also shown and validated inside a high-fidelity simulation using O3DE. Even if the system right now supports only semi-manual execution—without full automation or advanced optimization—the architecture built here allows for a flexible and expandable planning pipeline.

5.1 Motion Planning Configuration

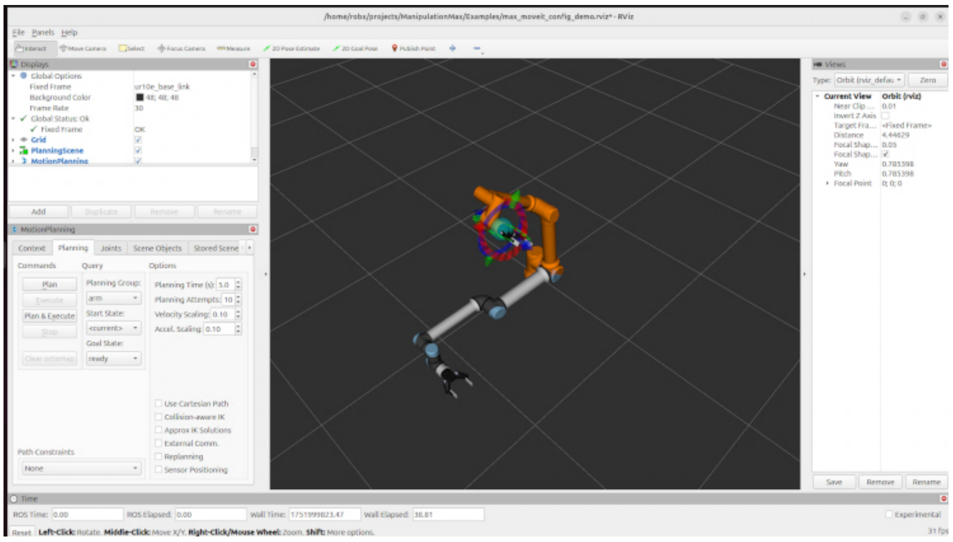
Trajectory generation in this system is mainly managed by MoveIt 2, which has been set up to work with both OMPL (Open Motion Planning Library) [11] and CHOMP (Covariant Hamiltonian Optimization for Motion Planning) as backends. Both options were tested, but most of the planning was done using OMPL, because of its good stability and its ability to deal with complex joint configurations.

The planning pipeline works using predefined goal poses, which are set through the MoveIt Setup Assistant. These poses are selected interactively inside RViz, where the user can define the desired configuration of the robot arm and the gripper. Once the goal pose is chosen, MoveIt calculates a trajectory to reach it by using sampling-based planning algorithms, such as RRTConnect—the default planner available in OMPL for MoveIt.

Even though the motion planner parameters—like planning time, level of path simplification, or sampling resolution, were not manually fine-tuned, attention was given to synchronize the timing between RViz, MoveIt, and the O3DE simulator.

This synchronization was set up in the system's launch files and helps to keep the trajectory execution and the simulation aligned in time, which allows real-time feedback and makes debugging more accurate.

The Motion Planning panel in MoveIt (see Figure 5.1) offers an intuitive interface inside RViz for setting up and executing motion plans. Users can choose the planning group—like the robotic arm or the gripper—separately, which gives detailed control over specific robot parts. It is also possible to select predefined start and goal poses from a list configured using the MoveIt Setup Assistant, making it easier to create repeatable motion tasks. In addition, users can change several planning parameters, such as the allowed planning time, number of planning attempts, and velocity or acceleration scaling factors. This graphical interface is very useful for quickly testing different planning strategies without needing to write scripts, and it supports real-time trajectory execution both in simulation and on the physical robot.


/home/robs/projects/ManipulationMas/Examples/max_moveit_config_demo.rviz* - RViz

Displays
Global Options
Fixed Frame: ur10e_base_link
Background Color: 48; 48; 48
Frame Rate: 30
Global Status: Ok
Fixed Frame: OK
Grid ✓
PlanningScene ✓
MotionPlanning ✓

MotionPlanning – Planning
Planning Group: arm
Start State: <current>
Goal State: ready
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 0.10
Accel. Scaling: 0.10
Path Constraints: None

Views
Type: Orbit (rviz_defau
Current View: Orbit (rviz)
Near Clip ...: 0.01
Target Fra...: <Fixed Frame>
Distance: 4.44629
Focal Shap...: 0.05
Focal Shap...: ✓
Yaw: 0.785398
Pitch: 0.785398
Focal Point: 0; 0; 0

Time
ROS Time: 0.00
ROS Elapsed: 0.00
Wall Time: 1751999823.47
Wall Elapsed: 38.81
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.
31 fps

Figure 5.1: Screenshot from RViz showing the UR10e robotic arm and the trajectory generated by MoveIt. The planned path is highlighted in orange. No obstacles are visible in the RViz planning scene, but they are present in the corresponding simulated scenario within O3DE.

5.2 Trajectory Optimization Approach

In this work, no custom trajectory optimization methods were developed. The system uses only the default optimization features provided by MoveIt. When the CHOMP planner is selected, it improves the trajectory using cost gradients that take into account both obstacle avoidance and smoothness of the path [35].

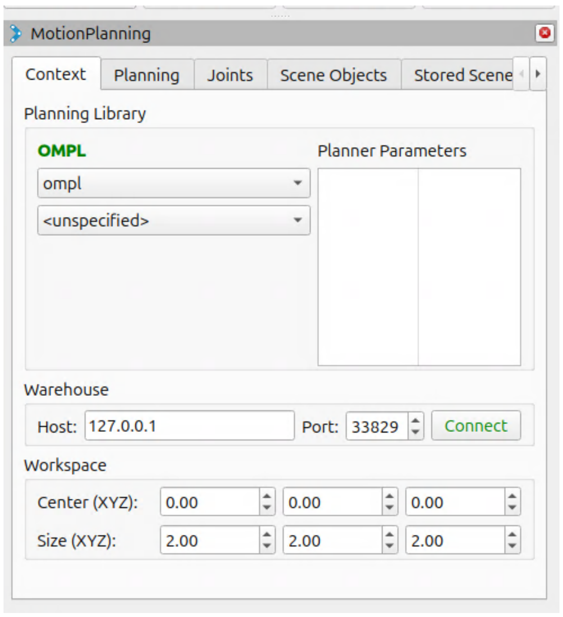
Other trajectory optimization techniques, like STOMP (Stochastic Trajectory Optimization for Motion Planning) and TrajOpt (Trajectory Optimization for Motion Planning) [36, 37], were also reviewed during the early stages of the project. These planners are especially suitable for complex planning problems in high-dimensional spaces and dynamic environments, where soft constraints and stochastic approaches may give better results in terms of convergence or constraint handling. A summary of the main features of these planners is shown in Table 5.1. However, they were not included in this thesis, since the main goal was to validate basic motion planning strategies in static and semi-structured settings.

Table 5.1: Comparison of motion planning and trajectory optimization methods.

Planner	**Technique**	**Advantages**	**Limitations**
OMPL	Sampling-based	Fast, flexible, easy to integrate	Discrete paths, less smooth
CHOMP	Gradient-based	Smooth trajectories, obstacle avoidance	Slower, requires good initial guess
STOMP	Stochastic optimization	Handles noise, good for constraints	High computational cost
TrajOpt	Sequential convex optimization	Supports complex constraints	Requires accurate collision checking

In contrast, OMPL planners generate trajectories using random sampling methods and typically include basic path simplification as a post-processing step.

As shown in Figure 5.2, MoveIt provides a dedicated interface to configure OMPL planners, including workspace boundaries and optional database connections. However, in this work, the planner was used with its default settings to maintain a minimal and replicable configuration.


MotionPlanning

Context | Planning | Joints | Scene Objects | Stored Scene

Planning Library

OMPL

ompl

<unspecified>

Planner Parameters

Warehouse

Host: 127.0.0.1 Port: 33829 Connect

Workspace

Center (XYZ): 0.00 0.00 0.00

Size (XYZ): 2.00 2.00 2.00

Figure 5.2: Screenshot of the OMPL configuration tab in MoveIt's Motion Planning interface. This panel allows users to select the planning library (e.g., OMPL), connect to a planning database, and optionally define workspace boundaries or planner-specific parameters. In this work, default parameters were used without custom tuning.

While CHOMP also allows the use of custom constraints—such as keeping the end-effector orientation, penalizing joint limit violations, or enforcing workspace boundaries—these advanced options were not used or extended in this work. This choice was made to keep the system simpler and to focus on building a minimal but solid pipeline that could work reliably both in simulation and in real-world scenarios.

Still, the system architecture is modular and can be extended in future developments to support constraint-based planning, especially for high-precision operations or in environments with limited space.

5.3 Task-Level Planning and Workflow

The strategy used for task execution in this system is semi-manual and based on interactive usage of RViz and MoveIt. Goal poses are defined during the setup phase using the "Robot Poses" interface from the MoveIt Setup Assistant, which lets users configure and save specific positions for the robot arm and the gripper that can be reused later.

During operation, a target pose is selected in RViz, and MoveIt is used to generate a trajectory to reach it. The planned trajectory is then visualized in RViz and mirrored in real time in O3DE, providing a dual-feedback channel to evaluate correctness and feasibility. Notably, thanks to proper frame and topic configuration, the system supports reverse planning, allowing the user to select an end-effector pose and automatically compute the joint configuration required to reach it.

Although no scripts or automatic task sequences have been implemented at this stage, the current architecture supports such extensions in the future, as the planning and execution loop is fully accessible through the MoveIt API and ROS action servers.

5.4 Simulation-Based Validation

A set of motion planning trials was performed in simulation to check the correctness and realism of the generated trajectories. The robot, simulated as a UR10e manipulator with a Robotiq 2F-140 gripper, was placed inside a virtual office environment built in O3DE. This environment included a desk, several objects, and physical constraints that tried to reproduce real-world interactions.

Different goal poses were tested in multiple scene variations, including changes in object positions and small adjustments in the geometry of the workspace. These tests were done to verify that the robot's motions were compatible with its real capabilities, and that the poses planned were actually reachable considering the scene constraints.

Even though no quantitative metrics (like trajectory length, execution time, or energy usage) were calculated, the visual feedback from RViz and O3DE made it possible to evaluate the quality of the motions. It was possible to observe how smooth the trajectories were, how well obstacles were avoided, and if the overall behavior looked physically correct.

Chapter 6

Integration with real hardware

One of the main goals of this thesis was to validate the consistency and realism of the simulation infrastructure by executing motion plans on a real robotic platform. Moving from simulation to physical execution is a fundamental step of verification, as it confirms that the control and planning strategies developed in the virtual environment can also be applied reliably in real conditions. This chapter explains how the integration between the ROS 2-based software stack and the real robotic hardware was done, and it discusses the results, limitations, and insights observed during the execution of trajectories on the actual robot.

6.1 Robotic Platform and Setup

The physical system used in this thesis includes a UR10e robotic arm by Universal Robots, together with a Robotiq 2F-140 parallel gripper. This hardware setup is exactly the same as the one used in the simulation phase. The robot was connected to the development workstation through a direct USB connection, which allowed low-latency communication for sending commands and monitoring feedback. The physical workspace was organized to be as similar as possible to the virtual one created in O3DE, with comparable furniture layout and object positions.

Hardware Specifications of the Development Environment

The development and validation procedures were executed on a high-performance workstation configured to support real-time robotic simulation, GPU-accelerated rendering, and low-latency communication with physical hardware. The system setup is summarized in Table 6.1.

Table 6.1: Hardware and software configuration of the development environment

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)

This configuration provided sufficient computational resources for real-time physics simulation, AI-based 3D asset generation, and ROS 2-based motion planning. The NVIDIA RTX 5080 GPU ensured smooth rendering and physics performance within O3DE. Docker was used to containerize the entire stack, guaranteeing reproducibility and cross-platform consistency during development and testing.

6.2 ROS 2 Integration and Control

The integration with ROS 2 was accomplished using the official driver packages for both the UR10e arm (`ur_robot_driver`) and the Robotiq gripper. These packages provide native ROS 2 support and expose standard interfaces for motion execution, joint state publishing, and gripper control. No additional bridging layers or wrappers were required, thanks to the drivers' compatibility with the DDS-based communication infrastructure of ROS 2.

The robot received planned trajectories via standard `/joint_trajectory_controller` topics, managed by the `ros2_control` interface. The gripper control was handled through a dedicated action server, consistent with the interfaces used during simulation. Importantly, the exact same MoveIt 2 configuration used in simulation—comprising the robot model, controller settings, planning parameters, and predefined goal poses—was reused for the real hardware, demonstrating the simulation's ability to serve as a reliable precursor to real-world deployment.

A minor adaptation was required to replicate the gripper behavior in the physical system, as the simulation logic was implemented through a script that also had to be ported for hardware execution.

6.3 Observed Performance and Correspondence

The execution of the planned trajectories on the real robot confirmed a strong level of correspondence between the simulation and the physical setup. The robot was able to follow the same poses and motion paths that were defined and tested previously in the O3DE environment. However, some small differences were noticed during execution, especially regarding latency, timing, and physical interaction.

Slight delays were observed between sending a command and the actual movement of the robot. These delays are probably caused by the hardware feedback mechanisms and communication overhead, which are not present in the simulation. In addition, physical effects such as friction and mechanical tolerances introduced minor variations in how the robot moved compared to the ideal behavior shown in simulation. These differences show how important it is to carefully adjust the physical properties in the simulated environment, especially when a high level of accuracy is needed between virtual and real execution.

In addition, the mesh models used in the simulation—although visually accurate—did not perfectly match the real geometry and material properties of the physical objects. Because of this, small differences were noticed during grasping and object clearance, but they did not affect the overall success of the tasks.

6.4 Safety Considerations and Experimental Environment

No extra safety systems were added beyond those already available in the UR10e's built-in firmware, such as the emergency stop, joint limits, and force thresholds. All physical experiments were carried out in a controlled environment designed to reproduce the simulated scene as closely as possible. This included accurate positioning of the furniture and objects, which made it possible to compare simulation and real execution in a consistent and meaningful way.

Chapter 7

Results and validation

A key phase of this thesis was the evaluation of the system in both simulation and real-world scenarios. The goal of this validation was to check the accuracy and consistency of motion planning and execution between the two environments, and also to verify how robust the system architecture was under realistic working conditions. This chapter describes the testing process, the qualitative results that were collected, and the differences observed between the simulation and the physical execution.

7.1 Safety Considerations and Experimental Environment

The tests have been done using both simulation tools and the real robot. For the simulation part, RViz, MoveIt, and Open 3D Engine (O3DE) were used together to provide full visualization, motion planning, and also realistic 3D rendering of the robot movements. Many screenshots and some video recordings were taken during the simulation phase to verify the correctness of the movements and how the robot was interacting with the environment.

For the real-world validation, the same MoveIt configuration and planning pipeline were used on a physical UR10e robotic arm with a Robotiq 2F-140 gripper. The robot was positioned in a workspace that was built to be very close to the one in simulation.

The evaluation was focused on pick-and-place tasks with common tabletop objects like a joystick, adhesive tape, an aerosol can, a mechanical gripper, and a cardboard box. These objects were selected because they have different shapes, masses, and difficulty to be grasped. The tests were repeated with different object positions, orientations, and table layouts, in order to give the system a variety of motion planning and execution challenges.

Additional scenario variations were introduced during simulation, such as:

- Slight shifts in object placement to test the adaptability of the planner.
- Inclusion of virtual obstacles (e.g., a coffee mug or a cable) to assess collision avoidance.
- Varying end-effector orientations to challenge inverse kinematics resolution.

7.2 Validation Objectives and Criteria

The first goal of the validation was to be sure that the simulation stack—including planning, visualization, and rendering—was able to reproduce the real robot behavior in a faithful way. This included testing the accuracy of the kinematics, the correct alignment of the coordinate frames, and the joint behavior under realistic physical conditions.

The second objective was to verify that the trajectories planned in simulation could be executed on the real robot directly, without the need of tuning parameters, model adjustments, or reconfigurations. This kind of "simulation-to-reality consistency" was considered very important to prove that the system can be used as a reliable tool for fast prototyping and early testing.

Tests were considered successful if:

- The robot reached the target pose without collision or interruption.
- The motion appeared smooth and continuous, both visually and kinematically.
- The object was successfully grasped and placed within a reasonable spatial margin.
- The result matched the simulation both in path shape and temporal structure (within expected tolerances).

7.3 Observations and Qualitative Results

The simulation phase produced stable and reliable trajectories in all the tested scenarios. The robot motions were visually smooth in RViz and were also correctly reflected inside O3DE. The high graphical quality of O3DE helped to better evaluate the spatial relationship between the robot and the environment, giving a more realistic idea of how feasible the tasks were and how they could be perceived by a human operator. The visual fidelity of AI-generated objects was qualitatively validated through direct comparison with real-world counterparts (see Figures 3.3–3.5), confirming their suitability for perception-based testing in simulated environments.

A summary of the grasp success rate for five different objects tested on the real setup is shown in Table 7.1. These values show the consistency of the system even with different object shapes and physical conditions. To better understand how the execution worked, a detailed timing analysis was done for two representative objects: the adhesive tape and the spray can. Each pick-and-place task was divided into four phases: positioning, pick, transport, and place. Table 7.2 shows the timing results for both simulation and real execution. Even if the simulation gave a good approximation, some delays were noticed during real execution, especially in the gripper activation and the placement steps. These delays are mainly due to mechanical inertia and actuation latency.

In the real-world tests, the robot performed the same planned trajectories with a high level of matching. However, some small differences were noticed:

- Slight latencies and timing mismatches between simulation and reality.
- Positional deviations in grasping tasks, likely caused by friction and inertial effects not fully modeled in simulation.
- Variability in grasp success due to object properties not represented in the simulator (e.g., material compliance, non-uniform mass distribution).
- Minor geometric inconsistencies between simulated and physical objects, attributed to the use of AI-generated models based on 2D images, which introduced estimation errors in depth and scale.

Despite these differences, the robot was in general able to complete the pick-and-place tasks in a successful way. The qualitative behaviour of the hardware confirmed that the simulation and planning pipeline was valid, with only small performance differences that can be explained by known physical limitations.

Table 7.1: Grasp success rate across physical test objects

Object	**Attempts**	**Successes**	**Success Rate**
Joystick	10	8	80%
Tape	10	10	100%
Spray Can	10	10	100%
Cutter	10	4	40%
Box	1	0	0%

Table 7.2: Execution Time Breakdown

Object	Phase	Sim Time (s)	Real Time (s)	Δ Time (s)	Δ %
Tape	Spot	1.2	1.5	0.3	+25%
Tape	Pick	0.8	1.1	0.3	+37.5%
Tape	Transport	1.6	1.9	0.3	+18.8%
Tape	Place	1.0	1.3	0.3	+30%
Spray Can	Spot	1.1	1.4	0.3	+27.3%
Spray Can	Pick	0.9	1.2	0.3	+33.3%
Spray Can	Transport	1.5	1.8	0.3	+20.0%
Spray Can	Place	1.0	1.2	0.3	+20.0%

7.4 Conclusions on System Fidelity

The experiments show that the system developed gives a valid and efficient simulation environment for designing robotic manipulation tasks. The strong consistency of configuration between simulation and real robot, thanks to ROS 2 and MoveIt, made possible to transfer the trajectories from the virtual plan to the real execution without big problems.

Anyway, the results show also that it is important to improve some parts of the simulation fidelity, especially:

- The accuracy of 3D models, which could benefit from integration with depth-sensing technologies or structured-light scanning.
- The physical properties of simulated objects, such as mass, center of gravity, and surface friction, which currently rely on idealized parameters.
- Fine-tuning of the temporal dynamics, especially in scenarios where real-world latencies and compliance could affect performance.

While these refinements fall outside the original scope of this thesis, they represent natural directions for enhancing the realism and applicability of simulation-driven development pipelines in future work.

Chapter 8

Conclusions and future works

The work showed in this thesis had the goal to investigate how to simulate intelligent robotic behaviours in realistic environments, especially focusing on the generation of 3D scenes, motion planning with ROS 2 and MoveIt, and the validation of these parts on real robotic hardware. The main purpose was to build a virtual environment that can be very close to the real working space, in order to allow safe testing and development before deploying on the real system. This target has been mostly reached thanks to the good integration between simulation, planning and execution levels.

The system we proposed is based on ROS 2 as communication middleware, MoveIt for the motion planning, and O3DE as simulation engine. Artificial intelligence was also applied to create some elements of the environment, making the virtual scenes more rich and useful for testing in different situations. During the implementation, some adjustments were necessary, especially to make the Robotiq gripper and the UR10e manipulator work correctly together. In the beginning, different simulators and operating systems were taken into account; finally, the choice of O3DE running on Ubuntu was made because of its good compatibility with ROS 2 and the better visualization quality.

8.1 Contributions

This thesis has developed a complete and working simulation framework for robotics that makes possible to quickly define, plan and test robot motions inside complex environments. One of the most important contributions is the possibility to test robot actions — including object manipulation — in a simulated world that is very similar to the real one, and then perform the same actions on the real robot with only few adjustments. This fast workflow from simulation to real application can help to reduce design costs and make the development process shorter, especially in industrial or research scenarios.

Other important contributions are:

- The integration of MoveIt, RViz, and O3DE into a coherent ROS 2-based architecture.
- The use of AI-generated 3D assets to populate the environment with realistic scene elements.
- The validation of planned trajectories across both simulated and physical platforms.
- The containerization of the environment using Docker, ensuring portability and reproducibility.

As summarized in Table 8.1, the system developed in this thesis brings together modular architecture, AI-driven simulation, and robust real-world validation.

Contribution	**Description**
Integration of ROS 2, MoveIt, and O3DE	A unified and replicable architecture combining robotic planning, simulation, and visualization, deployed through a Docker container.
AI-generated environment elements	Utilization of tools such as Meshy and StepFun to create realistic and diverse 3D assets for simulation.
Sim-to-real validation with UR10e and Robotiq	Execution of identical planned trajectories in both simulated and physical environments to assess behavioral consistency.
Visual and functional validation tools	Effective use of RViz and O3DE to support planning visualization, trajectory debugging, and interactive testing.

Table 8.1: Summary of the main technical contributions achieved in this thesis.

8.2 Limitations

Even if the developed system is solid and complete from a functional point of view, there are still some limitations that have been observed. First of all, the geometric and physical precision of the 3D objects generated by AI is not yet very high. In particular, the lack of proper information about depth, volume and mass distribution from 2D inputs creates differences in grasp accuracy and contact simulation. This problem can affect especially pick-and-place operations, where small differences in object shape or friction can cause unexpected behaviours when moved to the real robot.

One possible improvement to overcome these limits could be the use of depth-sensing technologies like RGB-D cameras, LiDAR, or structured-light scanners, in order to obtain 3D models with metric accuracy. These devices can give better geometrical information, reduce the need of manual corrections, and increase the realism of the simulation. Furthermore, combining the AI-generated meshes with data coming from depth sensors might be a good compromise between high-quality visual appearance and precise shape, making the simulation more useful and closer to real validation needs.

In the future, the pipeline could be improved by automating several post-processing steps that for now are still done manually. For example, the problems with object scaling—caused by inconsistent sizes from AI-generated models—could be solved by using Blender scripts (with the bpy module) to automatically normalize the scale, fix object origins, and make sure that the physical representation in the simulation is correct. Also, physical parameters like mass, center of gravity, and friction could be estimated in automatic way using mesh analysis tools like Assimp or some custom tool for inertia estimation. A system based on templates could also be added to apply the same material and dynamic properties to all new objects, making the integration of assets faster and more uniform.

For what concerns UV mapping and problems with textures, using automatic UV unwrapping together with texture baking and adjustments through Blender scripting could help a lot to improve how 3D models look and behave in the simulation. These improvements would make assets more reliable and help to reduce errors or differences when moving from simulation to the real-world manipulation.

Moreover, the current system does not support dynamic planning, automatic task assignment, or coordination between multiple agents. All the robot actions are configured and launched manually, which makes it hard to scale the system to more complex or fully autonomous applications. In addition, the 3D models created with AI-based tools like Meshy and StepFun still needed manual corrections during post-processing, especially for scale and physical parameters. This shows that future improvements should focus on adding automatic calibration techniques to make the physical behaviour of simulated environments more realistic and trustworthy.

8.3 O3DE limitations and practical challenges

Even if the Open 3D Engine (O3DE) has been a powerful tool for realistic simulation and for working together with ROS 2, several technical and practical issues were faced during its usage in this project. First of all, the Linux version of O3DE—compiled and installed manually during development—showed some problems specific to the platform. These problems included graphical rendering bugs, strange behaviour in the Asset Browser, and missing or incomplete features in some parts of the editor. Some of these issues were fixed with the help of the online community, workarounds or small custom patches, but others were still not solved at the moment of writing. Even if these problems were not blocking for the main simulation functionalities, they caused some delays and made the debugging and testing process more difficult.

O3DE has native ROS 2 integration through the ROS 2 Gem, which is an advantage because it does not require extra bridges. However, the support for some features specific to robotics is still under active development. Some physical elements, especially articulated joints used for robot grippers, were not fully supported or needed extra steps to make them work properly. For example, when integrating the Robotiq 2F-140 gripper, some problems appeared related to the way O3DE handles joint constraints and actuation. These limitations affected the correct functioning of the gripper during both simulation and real-world validation phases.

Another technical constraint involved the lack of deterministic physics and robust sensor emulation tools comparable to those available in more established robotics simulators. As the system evolves, future improvements could include custom O3DE components that replicate advanced ROS-compatible sensor outputs and enable consistent physical interaction behavior across test runs. Integrating configuration presets or scripting templates for physical entities—similar to those used in Gazebo—would also help reduce scene preparation time and improve simulation reproducibility.

Finally, the tools and plugins available around O3DE are still not as complete or stable as the ones in more traditional simulators used in robotics. Functions like real-time sensor simulation, deterministic physics, or standard ROS 2 robot controllers needed to be implemented by hand or were not working directly with the usual ROS 2 setup. These points should be carefully evaluated before selecting O3DE for robotics research, especially when the project has strict time limits or needs to run in production environments.

Another important point is the long-term stability and support of O3DE, especially if we compare it with more established simulators like Gazebo or Ignition. O3DE offers very good visual quality and uses a modern rendering engine, but its usage in the robotics field is still not so common. Also, the documentation and official resources about ROS 2 integration are still not so complete or detailed. Because of this, there is a possible risk of technological lock-in, where in the future development or maintenance might need custom solutions that are not easy to reuse or move on other platforms.

In contrast, tools like Gazebo (and its newer version Ignition Gazebo) have the advantage of a big open-source community, regular updates, and strong support from organizations like Open Robotics. For research or industrial applications where stability, reproducibility or native ROS integration are more important than high-quality graphics, these traditional simulators might be a safer and more reliable option for the long term.

So, even if O3DE was chosen for its high-level visual realism, its use should be carefully considered in future improvements of this work, especially when scalability, community support, or compatibility across different platforms becomes a priority.

8.4 Future Work

Finally, improving the physical simulation parameters in O3DE—like object mass, material settings, and friction values—would help the virtual environment to better reflect how things behave in the real world, reducing even more the difference between simulation and reality.

At the same time, a specific architectural improvement could be the native support of mimic joints directly inside the O3DE physics engine. This could be done by developing a dedicated component or by adding a custom extension at the source level of the PhysX backend. With this upgrade, it would be possible to simulate kinematic dependencies—especially in multi-fingered grippers—in a more realistic and direct way, without using external control scripts. This would improve the physical consistency during manipulation tasks and make the simulation setup more clean and robust.

Beyond simulation and perception, future work could also look into the automation of task-level logic and decision making. Even if the current framework allows pose-based motion planning using RViz and MoveIt, the task execution is still semi-manual and limited to predefined goal poses. A possible improvement could be the integration of task-level planning systems, like symbolic planners (for example, PDDL) or hierarchical control structures such as Behavior Trees (BTs), to let the system break down high-level goals into ordered motion sequences. This kind of extension would make the system able to manage more complex tasks and dynamic situations by itself, without the need of manual control from the user.

Bibliography

[1] Yutaka Maruyama, Kojiro Makino, and Naoya Ando. «Exploring the Performance of ROS 2». In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*. Accessed: July 7, 2025. Pittsburgh, PA, USA, Oct. 2016, pp. 1–10. DOI: `10.1145/2968478.2968502`. URL: `https://web.ics.purdue.edu/~rvoyles/Classes/ROS_MFET642/Maruyama.ExploringROS2.2016.pdf` (cit. on pp. 4, 7).

[2] Open Source Robotics Foundation. *ROS 2 Design Documentation*. `https://design.ros2.org/`. [Online; accessed July 5, 2025]. 2025 (cit. on pp. 4, 5, 7).

[3] Object Management Group. *Data Distribution Service (DDS) Specification Version 1.4*. `https://www.omg.org/spec/DDS/`. Formal Specification, accessed: July 5, 2025. 2015 (cit. on p. 6).

[4] Open Robotics. *ROS 2 Documentation: Jazzy Jalisco Release*. `https://docs.ros.org/en/jazzy/Releases.html`. Accessed: July 7, 2025. 2025 (cit. on p. 7).

[5] Michael Görner, Robert Haschke, Helge Ritter, and Jianwei Zhang. «MoveIt! Task Constructor for Task-Level Motion Planning». In: *Proceedings of the 2019 IEEE International Conference on Robotics and Automation (ICRA)*. Montreal, Canada: IEEE, 2019, pp. 7095–7101. DOI: `10.1109/ICRA.2019.8793898`. URL: `https://doi.org/10.1109/ICRA.2019.8793898` (cit. on p. 8).

[6] Rosen Diankov. «Automated Construction of Robotic Manipulation Programs». Accessed via doi.org. PhD thesis. Carnegie Mellon University, 2018. URL: `https://doi.org/10.1184/R1/6714905.v1` (cit. on p. 8).

[7] Ioan Sucan and Lydia Kavraki. «A Sampling-Based Tree Planner for Systems With Complex Dynamics». In: *IEEE Transactions on Robotics* 28.1 (Feb. 2012), pp. 116–131. DOI: `10.1109/TRO.2011.2160466` (cit. on p. 8).

[8] Radu B. Rusu and Steve Cousins. «3D Is Here: Point Cloud Library (PCL)». In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011, pp. 1–4. DOI: 10.1109/icra.2011.5980567. URL: http://dx.doi.org/10.1109/icra.2011.5980567 (cit. on p. 8).

[9] MoveIt! Development Team. *MoveIt! Documentation*. https://moveit.picknik.ai/main/index.html. Accessed: 2025-04-23. 2025 (cit. on p. 12).

[10] ROS Development Team. *rviz: 3D Visualization Tool for ROS*. http://wiki.ros.org/rviz. Accessed: 2025-04-23. 2018 (cit. on p. 13).

[11] Ioan Sucan, Mark Moll, and E.E. Kavraki. «The Open Motion Planning Library». In: *Robotics & Automation Magazine, IEEE* 19.4 (2012), pp. 72–82. DOI: 10.1109/MRA.2012.2205651 (cit. on pp. 13, 39).

[12] Jacob Perron, Michael Jeronimo, et al. *RViz2: ROS 2 3D Visualization Tool*. Accessed: 2025-04-23. 2025. URL: https://github.com/ros2/rviz (cit. on p. 14).

[13] Francesco Lumpp, Marco Panato, Nicola Bombieri, and Franco Fummi. «A Design Flow Based on Docker and Kubernetes for ROS-based Robotic Software Applications». In: *ACM Transactions on Embedded Computing Systems* 23.5 (2024). Published: 14 August 2024, 74:1–74:24. DOI: 10.1145/3594539. URL: https://doi.org/10.1145/3594539 (cit. on p. 15).

[14] Dirk Merkel. «Docker: Lightweight Linux Containers for Consistent Development and Deployment». In: *Linux Journal* 2014.239 (2014) (cit. on p. 15).

[15] Ben Kehoe, Sachin Patil, Pieter Abbeel, and Ken Goldberg. «A Survey of Research on Cloud Robotics and Automation». In: *IEEE Transactions on Automation Science and Engineering* 12.2 (2015), pp. 398–409 (cit. on p. 15).

[16] NVIDIA Corporation. *NVIDIA Container Toolkit Documentation*. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit. Accessed: 2025-04-26. 2023 (cit. on p. 16).

[17] Linux Foundation and O3DE Community. *Open 3D Engine (O3DE) Documentation*. https://www.docs.o3de.org/docs/. [Online; accessed May 5, 2025]. 2025 (cit. on pp. 17, 19).

[18] NVIDIA Corporation. *NVIDIA PhysX SDK*. https://developer.nvidia.com/physx-sdk. [Online; accessed May 5, 2025]. 2025 (cit. on p. 17).

[19] Naomi Chukwurah, Abiodun Adebayo, Olanrewaju Ajayi, and Anfo Pub. «Sim-to-Real Transfer in Robotics: Addressing the Gap between Simulation and Real-World Performance». In: *Journal of Frontiers in Multidisciplinary Research* 5 (2024). Published: March 23, 2024, pp. 33–39. DOI: 10.54660/.IJFMR.2024.5.1.33-39 (cit. on pp. 17, 19, 20).

[20] Guntur Wijaya, Wahyu Caesarendra, Iskandar Petra, Grzegorz Krolczyk, and Adam Glowacz. «Comparative Study of Gazebo and Unity 3D in Performing a Virtual Pick and Place of Universal Robot UR3 for Assembly Process in Manufacturing». In: *Simulation Modelling Practice and Theory* 132 (2024). Published: April 1, 2024, p. 102895. DOI: `10.1016/j.simpat.2024.102895` (cit. on pp. 18, 19).

[21] M. Santos Pessoa de Melo, J. Gomes da Silva Neto, P. Jorge Lima da Silva, J. M. X. Natario Teixeira, and V. Teichrieb. «Analysis and Comparison of Robotics 3D Simulators». In: *2019 21st Symposium on Virtual and Augmented Reality (SVR)*. Rio de Janeiro, Brazil, 2019, pp. 242–251. DOI: `10.1109/SVR.2019.00049` (cit. on pp. 18, 19).

[22] Marian Körber, Johann Lange, Stephan Rediske, Simon Steinmann, and Roland Glöck. «Comparing Popular Simulation Environments in the Scope of Robotics and Reinforcement Learning». In: *arXiv preprint arXiv:2103.04616v1 [cs.RO]* (2021). Accessed: May 5, 2025. URL: `https://arxiv.org/abs/2103.04616` (cit. on pp. 18, 19).

[23] Kun Zhang et al. «Generative Artificial Intelligence in Robotic Manipulation: A Survey». In: *arXiv preprint arXiv:2503.03464v2 [cs.RO]* (2025). Submitted on 5 Mar 2025, last revised 11 Mar 2025, accessed: May 5, 2025. URL: `https://doi.org/10.48550/arXiv.2503.03464` (cit. on p. 20).

[24] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. «Generative Adversarial Networks». In: *arXiv preprint arXiv:1406.2661* (2014). Submitted on June 10, 2014, accessed: July 4, 2025. URL: `https://doi.org/10.48550/arXiv.1406.2661` (cit. on p. 20).

[25] Diederik P. Kingma and Max Welling. «Auto-Encoding Variational Bayes». In: *International Conference on Learning Representations (ICLR)*. arXiv:1312.6114 [stat.ML], accessed: July 4, 2025. 2014. URL: `https://arxiv.org/abs/1312.6114` (cit. on p. 20).

[26] Zimeng Qiu. «A Review of the State of the Art 3D Generative Models and Their Applications». In: *Applied and Computational Engineering* 112 (2024). Published: November 29, 2024, pp. 123–129. DOI: `10.54254/2755-2721/2024.17919` (cit. on p. 20).

[27] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. «Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World». In: *arXiv preprint arXiv:1703.06907v1 [cs.RO]* (2017). Submitted to 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2017), accessed: May 5, 2025. URL: `https://doi.org/10.48550/arXiv.1703.06907` (cit. on p. 20).

[28] Mark Hendrikx, Sebastiaan Meijer, Joeri Velden, and Alexandru Iosup. «Procedural Content Generation for Games: A Survey». In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)* 9 (2013). DOI: 10.1145/2422956.2422957 (cit. on p. 21).

[29] Pascal Klink, Carlo D'Eramo, Jan Peters, and Joni Pajarinen. «Self-Paced Deep Reinforcement Learning». In: *arXiv preprint arXiv:2004.11812v5 [cs.LG]* (2020). Submitted on 24 Apr 2020, last revised 23 Oct 2020, accessed: May 5, 2025. URL: https://doi.org/10.48550/arXiv.2004.11812 (cit. on p. 21).

[30] Open 3D Engine Contributors. *O3DE Documentation: PhysX Joint Types.* https://www.docs.o3de.org/docs/user-guide/interactivity/physics/nvidia-physx/joint-intro/. Accessed: July 7, 2025. 2025 (cit. on p. 26).

[31] Meshy AI Development Team. *Meshy AI – Discover Generate 3D Models with AI.* https://www.meshy.ai/discover. [Online; accessed July 4, 2025]. 2025 (cit. on p. 29).

[32] Weiyu Li et al. «Step1X-3D: Towards High-Fidelity and Controllable Generation of Textured 3D Assets». In: *arXiv preprint arXiv:2505.07747* (2025). Accessed: July 4, 2025. URL: https://arxiv.org/abs/2505.07747 (cit. on p. 29).

[33] Linux Foundation and O3DE Community. *Open 3D Engine: Gem Reference.* https://docs.o3de.org/docs/user-guide/gems/reference/. [Online; accessed July 4, 2025]. 2025 (cit. on p. 34).

[34] Linux Foundation and O3DE Community. *Open 3D Engine: Component Reference.* https://docs.o3de.org/docs/user-guide/components/reference/. [Online; accessed July 4, 2025]. 2025 (cit. on p. 34).

[35] Michael Zucker, Nathan Ratliff, Anca Dragan, Mihail Pivtoraiko, Matthew Klingensmith, Christopher M. Dellin, J. Andrew Bagnell, and Siddhartha S. Srinivasa. «CHOMP: Covariant Hamiltonian Optimization for Motion Planning». In: *The International Journal of Robotics Research* 32.9–10 (2013), pp. 1164–1193. DOI: 10.1177/0278364913488805. URL: https://doi.org/10.1177/0278364913488805 (cit. on p. 40).

[36] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. «STOMP: Stochastic Trajectory Optimization for Motion Planning». In: *2011 IEEE International Conference on Robotics and Automation (ICRA).* Accessed: July 7, 2025. 2011, pp. 4569–4574. DOI: 10.1109/ICRA.2011.5980280. URL: https://doi.org/10.1109/ICRA.2011.5980280 (cit. on p. 41).

[37] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel. «Motion Planning with Sequential Convex Optimization and Convex Collision Checking». In: *The International Journal of Robotics Research* 33.9 (2014). Accessed: July 7, 2025, pp. 1251–1270. DOI: `10.1177/0278364914528132`. URL: `https://doi.org/10.1177/0278364914528132` (cit. on p. 41).