Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

Automation & Intelligent Cyber-Physical Systems

A.y. 2025/2026

Graduation Session March 2026

Vision-Language-Action models for industrial robotics

Manipulation task application on UR10e

Supervisor:

prof. Alessandro Rizzo

Candidate:

Novia Francesco Antonio

Abstract

Recent developments about Vision-Language-Action (VLA) models are representing a remarkably innovative approach in the field of robotics.
This class of models promises to play a noticeable role in robotics research, analogously to the deep innovation brought by foundation models for a large part of modern AI technologies, in several contexts. Leveraging multi-modal input understanding, LLMs generative capabilities, and an effective translation layer to perform real world actions, VLAs aim to achieve the concept of a unified physical intelligence, which can then easily apply to different unseen embodiments and can increase modern autonomous robotic systems' versatility and robustness.

Being able to adapt and fine-tune these models for various types of tasks and environments can potentially enhance the capabilities of common robotic systems, such as industrial manipulators, as well as facilitate users to control them with a more natural approach.
The aim of this thesis is to explore and integrate existing SoTA VLA models for industrial applications, specifically involving context-aware manipulation and pick-and-place use cases.

In particular, the Isaac-GR00T (v1.5) model from NVIDIA is chosen, analyzed, and integrated on the UR10e manipulator platform, first validating its performance in a simulation environment (IsaacSIM) and then proceeding with the actual deployment on the real robot.
A complete model fine-tuning and optimization pipeline is developed for this specific embodiment, from dataset collection to elaboration and smoothing of output actions, targeting an effective and reliable global performance level.

Table of Contents

List of Tables

List of Figures

Glossary

VLA Vision-Language-Action Model for robotic control

TCP Tool Center Point of a robotic manipulator

ROS 2 Robot Operating System 2

UR10e Universal Robots e-series industrial manipulator

Sim-to-Real Simulation to Reality transfer

Sim-to-Sim Simulation to Simulation transfer

VLM Large Vision-Language Model

LLM Large Language Model

DoF Degrees of Freedom

DDS Data Distribution Service

URDF Unified Robot Description File

USD Univresal Scene Description

Chapter 1

Introduction

Industrial robotic manipulators have become a cornerstone of modern manufacturing, logistics, and automation systems. Such robots are widely adopted because of their precision, repeatability, and compliance with industrial safety standards. However, despite their reliability, they are typically equipped with control systems that remain largely constrained by rigid paradigms, limited autonomy, and a strong dependence on pre-programmed behaviors. These limitations significantly reduce their adaptability in unstructured or dynamic environments, where the variability in objects, tasks, and surroundings cannot be exhaustively modeled in advance.

While this standardized approach is effective in predefined static settings, it struggles to generalize across different tasks and often requires substantial manual engineering effort to reconfigure. Moreover, interaction modalities are usually limited to low-level interfaces, such as joint-space or cartesian commands, making human–robot interaction unintuitive and inaccessible to non-expert users. As a result, industrial robots are often underutilized in scenarios that demand flexibility, semantic understanding, and high-level reasoning.

Recent advances in artificial intelligence, particularly multimodal foundation models, have opened new opportunities to address these limitations. **Vision-Language-Action (VLA)** models represent a promising paradigm shift by jointly

leveraging visual perception, natural language understanding, and action generation within a unified framework. By grounding high-level semantic instructions in sensory observations and translating them into executable control policies, VLA models have the potential to enable robots to reason about tasks, interpret human intent, and adapt their behavior in real time.

Specifically, this thesis focuses on the application of **Isaac-GR00T** (n1.5), released by NVIDIA, among the most recent state-of-the-art VLA models, and its integration with **UR10e**, an industrial manipulator by Universal Robots.
After a brief analysis of the model architecture, a complete sim-to-real fine-tuning and deployment pipeline is presented, starting from integration inside the **Isaac Sim** simulation framework, until the actual refinement and adaptation for the real robot using **Robot Operating System 2**.
Eventually, in order to perform a satisfactory evaluation of such trained policy, the actual setup for *lift* and *pick-and-place* tasks experiments is described, and the relative results are discussed and analyzed.

This thesis work has been developed in collaboration with Reply S.p.a., a leading company in the field of digital solutions, ICT consulting and system integration, inside their Area 42 Lab in Turin.

1.1 Goal

This work aims to provide the following contributions:

- a general overview about current VLAs research scenario, reporting an overall classification of most used architectures and methods for this class of models;
- an integration pipeline for adapting and fine-tuning one specific VLA (NVIDIA Isaac-GR00T) to operate with the Universal Robots UR10e industrial manipulator
  - first, inside a simulation environment (Isaac Sim)
  - then, deploying the model on the real robot through ROS 2

- evaluation of performance, sim-to-real gap, possible limitations and enhancements of this approach compared to the traditional one, both in simulation and reality.

1.2 Thesis outline

The thesis is structured as follows:

- **Chapter 2:** General overview and classification of VLAs, brief report with recent benchmark and comparison between the most popular models; focus on Isaac-GR00T architecture and functionality
- **Chapter 3:** Presentation of the overall components and configuration of the system, both in real and simulated setup through Isaac Sim framework
- **Chapter 4:** Specification of Isaac-GR00T repository structure, inference API and detailed integration within Isaac Sim
- **Chapter 5:** Explanation of dataset collection and training pipeline to optimize the Isaac-GR00T policy for UR10e embodiment
- **Chapter 6:** Adaptation and setup for the actual deployment on the real manipulator: experimental evaluation of the fine-tuned model performances in simulation and reality
- **Chapter 7:** Summary about thesis contributions and main results; discussion about areas of improvement and future work

Chapter 2

VLA Models: background concepts and State of the Art

Vision-Language-Action models represent an emerging paradigm which targets embodied intelligence capable of unifying visual perception, natural language understanding, and action generation within a single multimodal foundation model. Unlike traditional robotic control systems, which decouple perception, planning, and control, VLAs aim to directly map sensory observations and language instructions to robot actions through learned representations. This integration potentially enables robots to perform complex manipulation tasks in unstructured environments with minimal task-specific programming or even without handcrafted controllers [1].

At the core of VLA research is the extension of **Large Vision-Language Models** (VLMs) and **Large Language Models** (LLMs) — such as those originating from advances in transformer architectures — into systems capable of acting in the physical world. Traditional VLMs process images and text to generate semantic outputs, but VLAs extend these capabilities to produce action interpretable commands suitable for robotic controllers. This design typically involves a pretrained vision-language backbone, fine-tuned on several datasets of robot trajectories paired with language instructions and visual context.

2.1 Theoretical Foundations of VLAs

A VLA can be viewed as a multimodal conditional policy that learns a distribution $\pi(a_t \mid o_{1:t}, l, s_{1:t})$, where o are visual observations, l is a language instruction, and s is state/proprioception. The goal is to build shared representations in which language can compositionally constrain motor decisions, moving beyond the classic "perception–planning–control" pipeline with a single end-to-end model that integrates vision, language, and action [1, 2, 3].

From a theoretical standpoint, VLAs address three core challenges:

1. **Multimodal grounding:** aligning linguistic and perceptual concepts with affordances and physical dynamics, linking language entities and relations to observable states and executable actions [1, 3].
2. **Action representation:** choosing the right output space (continuous, discrete, or tokenized) and temporal encoding, which governs motor precision, long-horizon planning, and compatibility with transformer backbones [4, 5, 6, 7].
3. **Generalization and transfer:** learning policies that reuse knowledge from heterogeneous data (robots, contexts, tasks) while remaining robust to domain shift and embodiment changes [8, 9, 10].

2.1.1 Data, pretraining, and transfer

Behavioral quality depends strongly on data coverage. Recent models combine web-scale vision-language pretraining with fine-tuning on robot demonstrations, aiming to transfer rich semantic knowledge into actionable control. For instance, RT-2 [4] formalizes this idea by co-finetuning a VLM on robot data and web vision-language tasks, representing actions as textual tokens so the model can reuse semantic reasoning learned in pretraining for manipulation. OpenVLA [11] applies the same principle at larger scale: a 7B-parameter model pre-trained and then finetuned on 970k real demonstrations, with gains in generalization and cross-embodiment transfer.

Standardized multi-robot datasets are crucial to enable policies working with

several kinds of robotic platforms. The Open X-Embodiment (OXE) project [9] defines common formats and RT-X models to make multi-robot generalist policy training more systematic. In parallel, real-world large-scale datasets such as BridgeData V2 provide tens of thousands of multi-task trajectories collected on low-cost robots, enabling models that generalize across environments and tasks [10].

2.1.2 Action representation and decoding

Three main representation families are common:

- **Tokenized actions:** actions are discretized or encoded as textual tokens, making autoregressive decoding natural. RT-2 shows that this strategy unifies actions and language in a single token space, encouraging semantic transfer [4]. VIMA [5] similarly adopts an autoregressive multimodal prompt setup where actions are generated as sequences.
- **Continuous actions:** policies directly output continuous vectors (positions, velocities, gripper), preserving fine motor granularity and control stability. This is common in foundation models for real manipulation and often combined with embodiment-specific heads [11, 12].
- **Generative policies (diffusion/flow):** actions are sampled from a generative process (diffusion or flow-matching), useful for modeling multimodal distributions and robust trajectories [7, 6].

2.1.3 Training objectives and generalization

Most VLAs are trained via imitation (behavior cloning) on human or teleoperated demonstrations, sometimes augmented with auxiliary objectives (prediction, grounding, or world-modeling). RT-1 [8] introduces a class of transformer policies trained on large real-world datasets and highlights how data diversity and model capacity affect generalization. Generalization is evaluated not only on new objects or environments but also on language instructions unseen during training, as shown in RT-2 evaluations [4]. Modern benchmarks also emphasize

long-horizon composition: CALVIN [13], for example, measures the ability to chain skills in long sequences conditioned on language.

2.1.4 Temporal abstraction and compositionality

A central theoretical issue is how to represent temporal abstraction: a robot must execute low-level motor commands while also respecting higher-level subgoals implied by language. Some approaches encode actions as short-horizon tokens (easy to decode but requiring frequent re-planning), while others generate longer trajectories or use generative processes to capture multi-step intent [4, 7]. Benchmark suites such as CALVIN explicitly test compositionality by requiring skill chaining under language constraints, exposing error accumulation and the need for robust temporal credit assignment [13]. Prompt-based approaches (e.g., VIMA) highlight how language and visual context can act as a high-level program that structures the action sequence, which is especially useful for zero-shot task composition [5].

2.2 Common architectures taxonomy

Most VLA models share a dual-stage architecture:

1. a **VLM** for perception and contextual reasoning;
2. an **Action Decoder** that converts latent representations into robotic actions or control sequences.

The VLM stage often originates from pretrained multimodal models (e.g., vision-language transformer backbones), while the action stage adapts or extends the representation to continuous or discrete action spaces depending on the task and embodiment [1].

Autoregressive and Token-Based Models: Some VLAs discretize actions into token sequences that the model predicts in an autoregressive manner, enabling compatibility with existing transformer infrastructures. This approach appears in models like π_0 and its variants [6].
Diffusion-Based Methods: Models such as *dVLA* employ diffusion processes to

generate action sequences, theoretically providing smoother and more robust multimodal chain-of-thought integration across vision, language, and action [7].
Reinforcement Learning Hybrids: Some architectures incorporate reinforcement learning or world models to enhance decision making under uncertainty, which can be critical for long-horizon tasks [2].

In practice, recent architectures combine frozen VLM/LLM backbones with lightweight adapters or specialized heads, preserving large-scale semantic knowledge while enabling efficient finetuning on robot demonstrations [11, 12]. In many cases, separating a multimodal encoder from an action decoder allows reusing generalist encoders and swapping only the control component when the embodiment or action target changes [9, 3].

Another recurring design pattern is the explicit handling of multimodal context: multi-view images are tokenized and concatenated with language tokens, while proprioception is encoded through dedicated MLPs and optionally augmented with embodiment identifiers. This choice enables a single policy to support multiple robot morphologies without retraining the perception stack from scratch, and is frequently paired with an embodiment-specific action head for consistent control output [11, 9, 12].

2.2.1 Representative Models

Among all the VLAs representing the State of the Art of this emerging field, the most relevant are reported below:

- **RT-1** (*Google*, 2022–2023) [8]: a transformer policy trained on large-scale real robot data, designed to study scaling with data diversity and model size.
- **RT-2** (*DeepMind, 2023*) [4]: among the pioneering VLAs that transferred internet-scale vision-language knowledge into robotic manipulation without manual segmentation of perception vs. control.
- **RT-X / Open X-Embodiment** (*OXE Collaboration, 2023*) [9]: standardized datasets and "X-robot" models to make multi-robot training and transfer more systematic.

- **OpenVLA** (*Stanford/UCB et al., 2024*) [11]: combines pretrained visual encoders with a language backbone and extensive robot trajectory datasets, showing strong multi-task manipulation performance.
- **VIMA** (*Stanford/NVIDIA*, 2022) [5]: an autoregressive model that uses multimodal prompts (text+images) to unify language instructions, visual goals, and one-shot demonstrations.
- π_0 and π_0-FAST (*Physical Intelligence, 2024-2025*) [6]: large generalist VLAs trained across multiple robot embodiments that leverage innovative flow-matching action tokenization.
- **SmolVLA** (*2024*) [14]: a compact open-source VLA demonstrating competitive success rates despite significantly fewer parameters.
- **Isaac-GR00T** (*NVIDIA, 2025*) [15]: a family of VLA models (e.g., GR00T N1, N1.5, N1.6) trained on mixed real, simulation, and synthetic data, achieving high success rates in bimanual manipulation benchmarks and demonstrating effective sim-to-real transfer.

2.2.2 Qualitative comparison of recent models

Table 2.1 summarizes key differences among leading models. The comparison highlights three design axes: (i) action representation (token vs. continuous vs. generative), (ii) pretraining strategy and data scale, and (iii) openness and ease of reuse/finetuning.

2.2.3 State-of-the-art model comparison

RT-1 established a scalable baseline for real-world robot control with transformer policies trained on large multi-task datasets, showing that data diversity and model capacity are key levers for generalization [8]. **RT-2** extends this idea by explicitly injecting web-scale VLM knowledge and by representing actions as text tokens, enabling semantic transfer from internet vision-language data to robotic manipulation [4]. The shift from RT-1 to RT-2 illustrates a broader trend: endowing policies with language-enabled reasoning by aligning action spaces

Model	Action representation	Data and scale	Notes
RT-1	continuous	large real-world multi-task dataset	scaling and generalization with diverse data [8]
RT-2	text tokens	fine-tuning on web + robot data	actions as tokens, emergent semantic reasoning [4]
OpenVLA	continuous	970k real demos, 7B params	open-source, Llama 2 + DINOv2 + SigLIP backbone [11]
VIMA	autoregressive tokens	600k+ simulated demos	multimodal prompts and strong zero-shot generalization [5]
π_0	flow-matching	multi-embodiment	generative policy for continuous actions [6]
GR00T N1.5	diffusion/flow	real+sim+synthetic	frozen VLM backbone, embodiment-specific policy head [12]

Table 2.1: Qualitative comparison of recent VLA models.

with the token space of pretrained LLM/VLM backbones.

RT-X / Open X-Embodiment takes a different angle by standardizing datasets and model interfaces across robots, enabling systematic cross-embodiment training and evaluation [9]. This benchmark-driven approach has influenced how newer models report results, emphasizing transfer rather than single-robot performance alone.

OpenVLA represents one of the most complete open-source stacks: it combines a 7B-parameter language backbone (Llama 2) with strong vision encoders (DINOv2 and SigLIP), and is finetuned on 970k real demonstrations. Its evaluations show improvements over prior baselines on multi-task robot benchmarks, making it a widely adopted reference for open VLA research [11].

VIMA frames manipulation as multimodal prompting, where text and images define the task and actions are generated autoregressively. This design facilitates zero-shot task composition and provides a clean experimental testbed for prompt-based generalization [5].

π_0 and **dVLA** illustrate the growing interest in generative policies: diffusion-based or flow-matching action generation captures multimodal behaviors and can model multiple valid trajectories for a task, a feature that is hard to achieve with deterministic regression heads [6, 7]. **SmolVLA** addresses another practical axis: parameter efficiency and accessibility, showing that smaller open models can still be competitive on manipulation tasks [14].

GR00T N1.5 focuses on cross-embodiment generalist control for humanoids, combining a frozen VLM backbone with a diffusion-style policy head and mixed real/sim/synthetic training data [12, 15]. This design choice prioritizes robustness and transfer, particularly in long-horizon, language-conditioned manipulation.

2.3 Benchmarks and Comparisons

VLA models are typically evaluated on robot manipulation benchmarks that assess success rates on tasks like tabletop manipulation, pick-and-place, articulated object handling, and multi-step language conditioned tasks. Comparisons across architectures highlight both relative strengths and limitations. The most widely used benchmarks now serve as common references to evaluate **generalization**, **long-horizon** performance, and **language grounding**.

2.3.1 Most widely used benchmark suites

The following suites are among the most commonly used:

- **RLBench** (2019): 100 diverse tasks with visual observations (RGB, depth, masks) and unlimited demonstrations generated by a motion planner, useful for imitation learning and cross-task generalization [16].

- **CALVIN** (2021): open-source benchmark for language-conditioned long-horizon manipulation tasks, with zero-shot evaluation on new instructions and environments [13].
- **LIBERO** (2023): lifelong-learning benchmark with four suites (130 tasks), focusing on declarative/procedural knowledge transfer and robustness to task ordering [17].
- **VIMA-Bench** (2022): simulated tabletop environment with thousands of tasks and 600k+ trajectories, designed to measure generalization via multi-modal prompts [5].
- **Meta-World** (2019): 50 simulated manipulation tasks for multi-task and meta-RL evaluation, often used as a rapid generalization baseline [18].
- **ManiSkill** (2021): physics-based simulation benchmark with diverse 3D objects and large-scale demonstrations (tens of thousands of trajectories), focused on generalization across shapes and goals [19].
- **Open X-Embodiment (OXE)**: multi-robot collection with standardized formats and RT-X models for cross-embodiment policies [9].
- **BridgeData V2** (2023): real-world multi-task dataset with 60,096 trajectories across 24 environments, useful to evaluate scaling and transfer in real settings [10].

These benchmarks cover complementary axes: simulated suites such as Meta-World, RLBench, ManiSkill, and VIMA-Bench enable controlled ablations and rapid iteration, while real-world datasets such as BridgeData V2 and OXE emphasize transfer across embodiments and real sensor noise. As a result, performance rankings can vary substantially across benchmarks, and architectural choices that help in one regime (e.g., tokenized actions for language grounding) may not translate directly to another (e.g., precision control in real-world manipulation) [1, 3, 20].

2.3.2 Evaluation metrics and protocols

Modern benchmarks typically measure:

- **Task-level success rates** on single tasks and skill compositions (for instance, CALVIN for long-horizon) [13].
- **Zero-shot generalization** to unseen objects, environments, or instructions (e.g., RT-2 and VIMA) [4, 5].
- **Cross-embodiment transfer** and robustness to robot changes (e.g., OXE/RT-X) [9].
- **Scaling behavior** with data and model size (e.g., RT-1, OpenVLA) [8, 11].

Similar benchmarking efforts (e.g., MultiNet) systematically evaluate models like OpenVLA, π_0, and other state-of-the-art VLAs across diverse procedural environments, highlighting how architectural choices — such as action representation and model scale — influence generalization to unseen tasks [20].

2.4 Isaac-GR00T

Throughout all this thesis work, **NVIDIA Isaac GR00T (N1.5)** [15, 21] is the VLA foundation model that will be focused on. It is designed to serve as a generalist policy for robot control across diverse manipulation tasks. At its core, the GR00T architecture integrates a VLM to encode multimodal observations — such as egocentric camera images and natural language instructions — into a shared embedding space. These embeddings are then processed by a policy head that predicts continuous robot actions conditioned on the current state and task instruction. In the N1.5 release, the Eagle 2 VLM backbone [22] is deliberately frozen during both pretraining and fine-tuning to preserve its learned ability to ground language and perception into meaningful task semantics, which improves generalization to unseen tasks and environments. Compared to previous versions, N1.5 also streamlines adapter layers and incorporates additional training objectives (e.g., FLARE, DreamGen trajectories) to improve grounding and behavior diversity.

2.4.1 Model architecture

Isaac GR00T N1.5 uses vision and text transformers to encode the robot's image observations and text instructions. The architecture handles a varying number of views per embodiment by concatenating image token embeddings from all frames into a sequence, followed by language token embeddings.

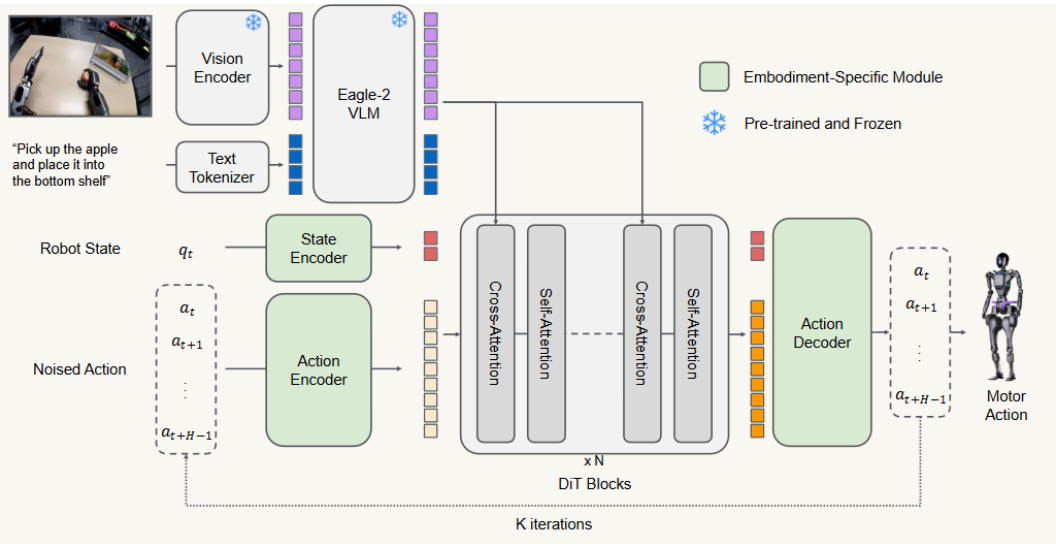


Figure 2.1: NVIDIA Isaac-GR00T general architecture (from [21])

To model proprioception and a sequence of actions conditioned on observations, Isaac GR00T N1.5-3B uses a flow matching transformer, which interleaves self-attention over proprioception and actions with cross-attention to the vision and language embeddings. During training, the input actions are corrupted by randomly interpolating between the clean action vector and a gaussian noise vector. At inference time, the policy first samples a gaussian noise vector and iteratively reconstructs a continuous-value action using its velocity prediction. In GR00T-N1.5, the MLP connector between the vision-language features and the diffusion-transformer (DiT) was trained jointly with flow matching and world-modeling objectives.

The schematic diagram is shown in figure 2.1. Red, Green, Blue (RGB) camera frames are processed through a pre-trained vision transformer (SigLip2). Text

is encoded by a pre-trained transformer (T5), and robot proprioception is encoded using a multi-layer perceptron (MLP) indexed by the embodiment ID. In order to handle variable-dimension state observations, inputs are padded to a configurable max length before feeding into the MLP. Actions are encoded and positions/velocity predictions decoded by an MLP, one per unique embodiment. The flow matching transformer is implemented as a diffusion transformer (DiT), in which the diffusion step conditioning is implemented using adaptive layernorm (AdaLN) [12].

Input Component	**Type**	**Format & Parameters**
Vision	Image Frames	Variable number of RGB images from robot cameras [224×224×3 `uint8`]
State	Robot Proprioception	1D floating-point vector
Language Instruction	Text	1D string

Output Component	**Type**	**Format & Parameters**
Actions	Continuous-value vectors	2D vectors corresponding to motor controls; dimensionality depends on robot embodiment DoF

Table 2.2: GR00T N1.5 Input-Output Specifications

2.4.2 Performance

Public evaluations of GR00T N1.5 demonstrate significant improvements over its predecessor (N1) on language-conditioned manipulation benchmarks [21]. For instance, in simulated manipulation tasks involving language instruction following on the Fourier GR-1 humanoid robot, N1.5 achieved a 93.2 % success rate compared to 52.8 % for the original N1 model. In data-limited post-training regimes, N1.5 also achieved higher success rates on benchmarks such as RoboCasa and Sim GR-1, indicating superior generalization and few-shot learning

capability. Real-world tests on placing objects based on natural language directives showed N1.5 success rates of 38.3 % on novel tasks versus 13.1 % for N1, underlining both robustness and adaptability to real embodiments with previously unseen objects.

2.4.3 Embodiments

In the context of GR00T projects, **embodiment** refers to the specific physical structure and control space of a robot that the learned policy must operate on. Rather than being tied to a single robot platform, the GR00T architecture is designed to be cross-embodiment, meaning a single model can be adapted to multiple robot morphologies (e.g., bimanual humanoids, single-arm manipulators) through designated embodiment tags and fine-tuning routines. This concept is crucial for generalist robotics: it allows the foundational VLA policy to be transferred and specialized to new embodiments with limited additional data, enabling broader applicability of learned behaviors without retraining from scratch.

The open-source implementation in the NVIDIA/Isaac-GR00T GitHub repository (n1.5-release) [23] supports training and inference code, model checkpoints, and utilities for fine-tuning on custom datasets with different robot embodiments. Embodiment tags in the codebase help map the generalized policy to concrete control spaces, such as single-arm or humanoid configurations, and define how actions are interpreted relative to the robot's physical morphology.
The `new_embodiment` tag, suggested for generic adapting the model to unseen embodiments through ad-hoc fine-tuning, was used for the UR10e robotic arm.

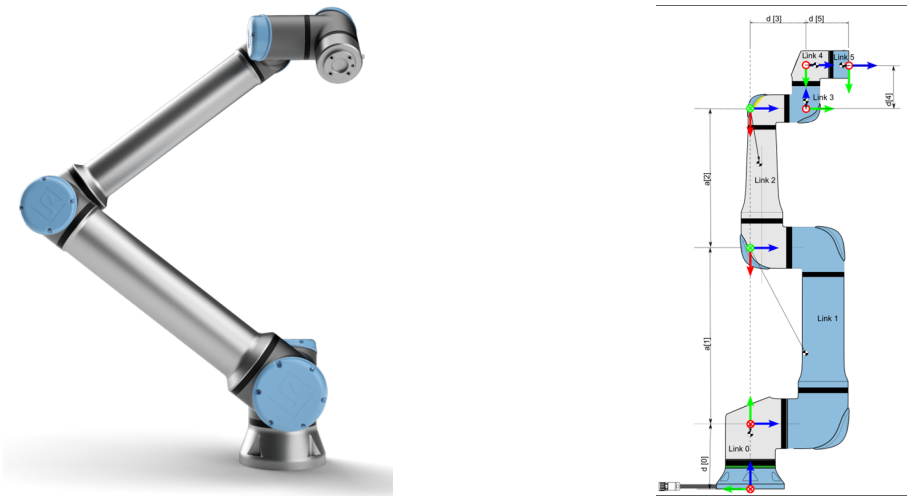
Chapter 3

System architecture and setup

In this section all the components of the system to integrate are presented. A description of the physical environment and its configuration is provided as well, with particular focus on the target tasks to be performed, both in simulated and real setup.

3.1 UR10e robotic manipulator

The **UR10e** [24], developed by Universal Robots, is a 6-DoF collaborative industrial manipulator designed for flexible automation tasks. It features a nominal payload of 12.5 kg and a reach of approximately 1300 mm, making it suitable for medium-scale manipulation, assembly, and pick-and-place operations. As part of the e-Series, the UR10e integrates a built-in force/torque sensor at the wrist, enabling compliant control strategies and safe human-robot collaboration. The robot supports position, velocity, and force control modes and can be interfaced externally through real-time communication protocols, including Ethernet-based industrial communication and ROS 2 drivers. Its widespread adoption in research and industry makes it a reliable platform for evaluating advanced VLA models in real-world manipulation scenarios.



(a) **(b)**

Figure 3.1: UR10e manipulator. **(a):** picture from user manual [25]; **(b)**: schematics for Denavit-Hartenberg parameters [26]

Joint	a [m]	d [m]	α [rad]	**Link**	**Mass [kg]**	**Center of Mass [m]**	**Inertia Matrix [kg m^2]**
Joint 1	0	0.1807	$\pi/2$	Link 0	7.369	$\begin{bmatrix} 0.021 \\ 0.000 \\ 0.027 \end{bmatrix}$	$\begin{bmatrix} 0.0341 & 0.0000 & -0.0043 \\ 0.0000 & 0.0353 & 0.0001 \\ -0.0043 & 0.0001 & 0.0216 \end{bmatrix}$
Joint 2	-0.6127	0	0	Link 1	13.051	$\begin{bmatrix} 0.38 \\ 0.000 \\ 0.158 \end{bmatrix}$	$\begin{bmatrix} 0.0281 & 0.0001 & -0.0156 \\ 0.0001 & 0.7707 & 0.0000 \\ -0.0156 & 0.0000 & 0.7694 \end{bmatrix}$
Joint 3	-0.57155	0	0	Link 2	3.989	$\begin{bmatrix} 0.24 \\ 0.000 \\ 0.068 \end{bmatrix}$	$\begin{bmatrix} 0.0101 & 0.0001 & 0.0092 \\ 0.0001 & 0.3093 & 0.0000 \\ 0.0092 & 0.0000 & 0.3065 \end{bmatrix}$
Joint 4	0	0.17415	$\pi/2$	Link 3	2.1	$\begin{bmatrix} 0.000 \\ 0.007 \\ 0.018 \end{bmatrix}$	$\begin{bmatrix} 0.0030 & -0.0000 & -0.0000 \\ -0.0000 & 0.0022 & -0.0002 \\ -0.0000 & -0.0002 & 0.0026 \end{bmatrix}$
Joint 5	0	0.11985	$-\pi/2$	Link 4	1.98	$\begin{bmatrix} 0.000 \\ 0.007 \\ 0.018 \end{bmatrix}$	$\begin{bmatrix} 0.0030 & -0.0000 & -0.0000 \\ -0.0000 & 0.0022 & -0.0002 \\ -0.0000 & -0.0002 & 0.0026 \end{bmatrix}$
Joint 6	0	0.11655	0	Link 5	0.615	$\begin{bmatrix} 0 \\ 0 \\ -0.026 \end{bmatrix}$	$\begin{bmatrix} 0.0000 & 0.0000 & -0.0000 \\ 0.0000 & 0.0004 & 0.0000 \\ -0.0000 & 0.0000 & 0.0003 \end{bmatrix}$

Table 3.1: UR10e Kinematic and Dynamic Parameters for DH convention

3.1.1 Robotiq gripper

The **Robotiq 2F-140** [27] is a two-finger adaptive parallel gripper designed for industrial manipulation tasks. It supports adjustable stroke up to 140 mm and is capable of grasping objects with varying geometries thanks to its underactuated finger mechanism. The gripper provides configurable force and speed control, enabling robust grasp execution across different object categories. Integrated with the UR10e manipulator, it is necessary to extend the robot's capabilities in order to perform reliable pick-and-place and lifting tasks in unstructured environments. Its ROS-compatible driver allows seamless integration into the control pipeline, enabling command-based actuation and state feedback monitoring.

Figure 3.2: Robotiq 2F-140 gripper (taken from [27])

3.2 Cameras

A key component for a successful integration of the VLA is the vision system. The **Intel Realsense D435i** [28] is an RGB-D camera integrating a global shutter stereo depth module, an RGB sensor, and an inertial measurement unit (IMU). It provides synchronized color and depth streams, enabling 3D perception and spatial reasoning in robotic manipulation tasks. The depth estimation is based on active stereo vision with an infrared projector, allowing reliable operation in

indoor environments with varying lighting conditions.
Within the system configuration adopted, two D435i cameras are used to acquire visual observations both from an eye-in-hand configuration and from an external viewpoint. The system exploits only one RGB channel (mono) for each camera, and it is configured to work at a resolution of 640x480 pixels, and frequency of 30Hz.

Figure 3.3: Intel Realsense D435i Stereo Camera

3.3 ROS 2

ROS 2 (Robot Operating System 2) [29, 30] is a distributed middleware framework for robotic applications. It provides communication abstractions based on a publish-subscribe paradigm, service calls, and action interfaces, enabling modular and scalable system design. Compared to its predecessor, ROS 2 introduces improved real-time support, enhanced security mechanisms, and better multi-platform compatibility. The middleware relies on DDS (Data Distribution Service) for network communication, ensuring deterministic and reliable message exchange. In this system architecture, ROS 2 acts as the integration layer between the VLA model, the robot driver, and external components such as cameras, enabling structured and synchronized data flow for sensor observations, action commands, and state feedback.

3.3.1 URDF

The **URDF** (Unified Robot Description Format) [31] is an XML-based format used in ROS ecosystems to describe the kinematic and dynamic structure of robotic

systems. A URDF model specifies links, joints, inertial properties, collision geometries, and visual representations. It enables consistent robot representation across simulation, visualization (e.g. RViz and Isaac Sim), and control frameworks. In this work, the URDF model of the UR10e provides a standardized description of the manipulator's kinematic chain, and its connection with the two-fingers gripper, facilitating motion planning through inverse kinematics, state proprioception, and compatibility between simulation and real hardware deployments.

3.4 Docker

Docker [32, 33] is a containerization platform that enables the packaging of applications and their dependencies into isolated, reproducible environments called containers. Containers encapsulate runtime libraries, system tools, and software frameworks, ensuring consistent behavior across different host machines. In robotics and machine learning workflows, Docker is particularly valuable for managing complex dependency stacks involving GPU drivers, deep learning libraries, and ROS 2 distributions, and simulation software. Within this project, Docker is used to guarantee reproducibility of the VLA integration pipeline, simplify deployment, and isolate experimental configurations across development in simulation and testing on the actual environments.

3.5 Isaac Sim

Isaac Sim [34] is a robotics simulation environment developed by NVIDIA, built on top of the Omniverse platform. It provides physically accurate simulation based on GPU-accelerated physics and photorealistic rendering, enabling high-fidelity synthetic data generation and robotic experimentation. Isaac Sim supports articulated robots, sensor simulation (RGB, depth, LiDAR), and domain randomization, making it particularly suitable for training and validating learning-based robotic policies. In the context of this work, Isaac Sim is used for data collection, dataset augmentation, and preliminary validation of policies before deployment on the physical UR10e manipulator. Its tight integration with

machine learning workflows and support for Python scripting facilitate rapid prototyping and evaluation.

3.5.1 USD

Universal Scene Description (USD) [35], originally developed by Pixar, is a scalable and extensible framework for describing, composing, and exchanging complex 3D scenes. USD enables hierarchical scene representation, layering, instancing, and non-destructive editing, making it particularly suitable for simulation and collaborative workflows. In robotics simulation contexts such as Isaac Sim, USD serves as the underlying scene graph representation, encoding robot models, environment assets, physics properties, sensors, and annotations. Its modular structure facilitates domain randomization, scene manipulation, and reproducible experimental configurations. Within this project, USD provides the structural backbone for defining the simulated environment setup and ensuring consistency between visual assets, physical properties, and robot configuration.

3.6 Tasks configuration and setup

In order to evaluate the actual performance of Isaac-GR00T VLA, the choice was to setup an operating environment that could resemble a realistic application context, such as a factory station. Two tasks were properly selected for training and testing the model in this configuration:

- ***Lift:*** consists of picking up a specified object. It involves identifying and locating the item referred inside the instruction, then approaching it with the correct grasp pose, and finally making the actual grip and keeping it while the object is lifted from the work surface. Possible disturbing elements are the presence of other confounding objects, physical obstacles interfering with the trajectory, and visual obstructions for the cameras.
- ***Pick and place:*** it is an extension of the *lift* task, with the addition of the transport phase and the eventual release of the object in a specified location. Along with the previously described issues, other problems may arise due

to the interaction with a wider area inside the environment and the obstacle represented by the placing container (such as a box).

The complete experimental setup is shown in figure 3.4. It was used for conducting evaluation and test trials, both in simulation and reality framework.
As previously mentioned, two cameras were adopted in this configuration: the external camera (from which fig. 3.4a, 3.4b were captured) is placed in order to have a comprehensive overview of the most relevant elements: the wrist camera (see fig. 3.5a, 3.5b) was mounted on the end-effector of the robotic arm, in order to exploit a closer view in the critical steps concerning the approach, grasp and release phase. The UR10e is positioned and fixed on a stable base, while a table used for picking objects is in front of it; the placing area consists just of a simple box, raised enough to be about level with the first table surface.

3.6.1 Scene reconstruction in Isaac Sim

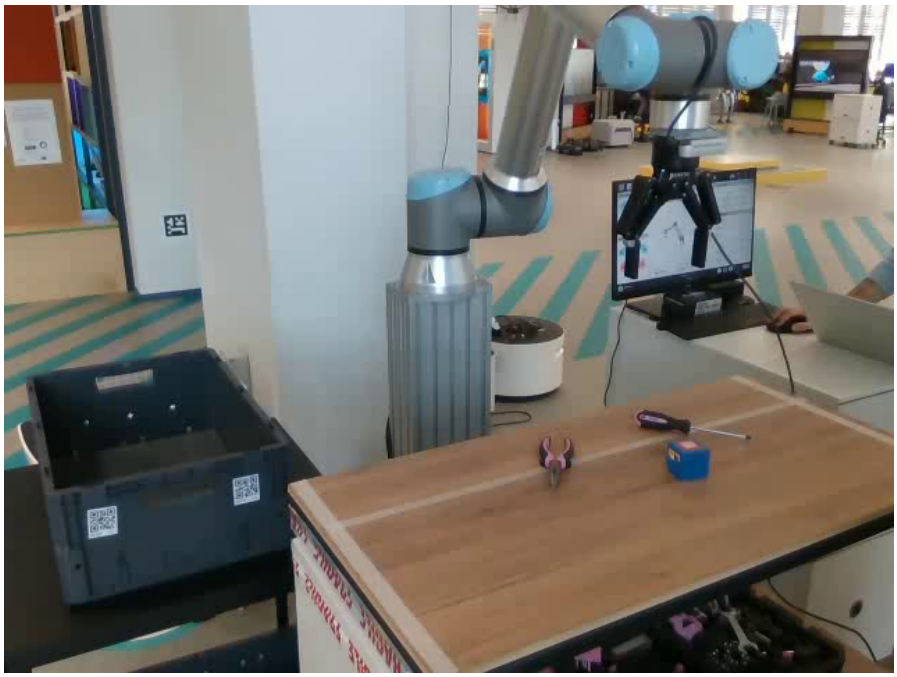
The fundamental approach followed in this thesis work was to develop and test all the components of the system in simulation environment first, and then proceed with the real deployment. For this reason, in order to better target and reduce the Sim-to-Real gap, a brand new custom USD scene was prepared.
The main assets and components were so reproduced:

- *UR10e + Robotiq gripper*: a built-in asset from Omniverse robotics library was taken with some adjustments, such as mesh and material properties refinement for the gripper to achieve a more realistic behavior of its fingers when interacting and colliding with the objects. All the predefined physical parameters were checked against the official ones (see fig. 3.1), and the URDF from the publicly available ROS 2 Universal Robots driver library [36] was employed to ensure the correctness of proprioception sensing and kinematics computation.

- *cameras*: two pre-built camera assets (ZED X and ZED X-mini models) were used for emulating the external and wrist view of the scene; their optical parameters (e.g. focal length, aperture, resolution) were aligned with the actual ones provided by Realsense datasheet [37].

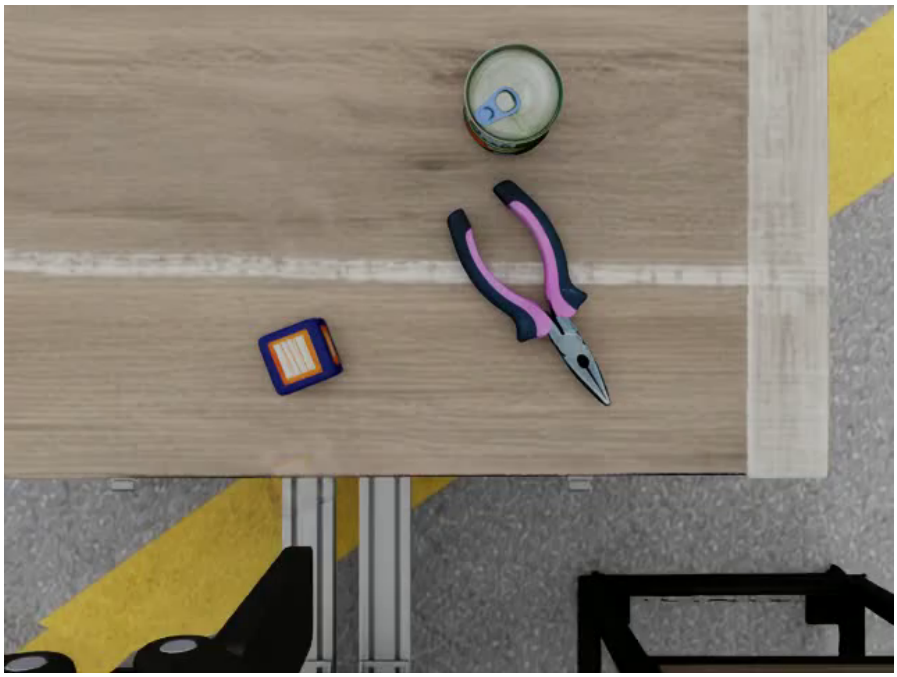
- *tables, box, UR10e base, picking objects*: these assets were completely 3D reconstructed starting from images such as real photos or video frames, exploiting a generative pipeline built on top of Trellis 2 AI engine, developed by Microsoft [38]; such generated structured files (`.glb` or `.obj` format) were then directly imported inside Isaac Sim, which automatically detects and creates meshes, material appearance and correct geometries. A manual refinement was eventually necessary to configure the main physical properties such as the weight, the shape of colliders, and the static and dynamic friction for surfaces.
- *background*: all the surrounding elements on the scene background were captured in a single image and this was placed behind the robot and all the aforementioned assets; since the of the scene background was to enhance fidelity to reality uniquely for the visual part, it was simply disposed in order to be just correctly visible and aligned for the external camera view.

(a) Isaac Sim

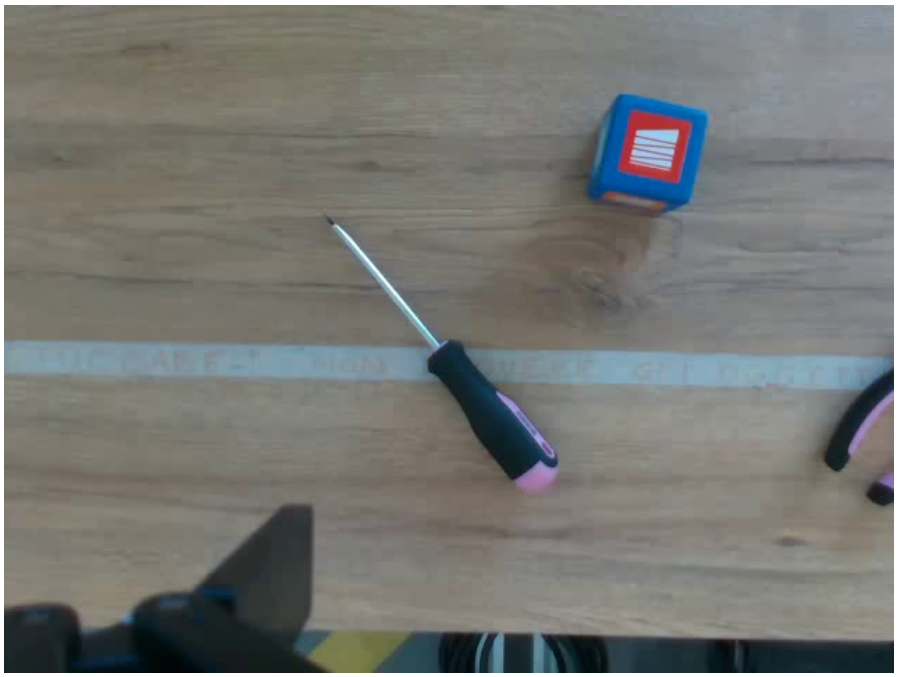


(b) Real

Figure 3.4: Experimental setup: external view [sim. vs real]



(a) Isaac Sim



(b) Real

Figure 3.5: Experimental setup: wrist view [sim. vs real]

Chapter 4

Isaac-GR00T integration in Isaac Sim

The first integration step of Isaac-Gr00t VLA was to connect it with the Isaac Sim environment. In this chapter, the whole process of integration and communication between the model and the simulation is described in detail, starting from the PyTorch implementation directly provided by NVIDIA [23].

4.1 GR00T inference with PyTorch

The open sourced model is available on Hugging Face hub (version N1.5 [12]). As many others VLAs, Isaac-GR00T is a generalist foundation model, so it aims to embed a global environment-agnostic physical intelligence. This means that, in its basic version, it cannot be directly usable out-of-the-box without a properly tailored fine-tuning process: this is what will be done indeed for making the model work with the UR10e + gripper system. Nevertheless, it also offers some pre-configured embodiment adapters:

- `EmbodimentTag.GR1`: Designed for humanoid robots with dexterous hands using absolute joint space control
- `EmbodimentTag.OXE_DROID`: Optimized for single arm robots using delta

end-effector (EEF) control, it was post-trained using Open X Embodiment and DROID datasets [9, 39]

- `EmbodimentTag.AGIBOT_GENIE1`: Built for humanoid robots with grippers using absolute joint space control

When running inference, the `EmbodimentTag` has to be specified in order to select the most suitable action head for the robot embodiment in use. Also, the `EmbodimentTag.NEW_EMBODIMENT` is provided for post-training the policy to work with new robot embodiments.

The GitHub repository [23] includes the complete implementation of the VLA structure, using Python with PyTorch library [40, 41]. As described before (see section 2.4.1), the architecture is mainly composed of three distinct blocks:

- the Eagle2 VLM Backbone, responsible for vision and language understanding, maps RGB frames and text inputs into a comprehensive latent space
- the Diffusion Transformer (DiT), consisting of several concatenated self-attention and cross-attention modules, performs the actual generative task, exploiting the VLM output as well as the specific action and state encoding
- the Action Head, responsible for interconnecting the external layers with the backbone, uses information coming from current proprioception observations and actions to generate properly encoded tensors, suitable for the subsequent DiT layers; it has to be specifically tuned for each unseen embodiment

4.1.1 `data_config` interface for UR10e

Another relevant aspect for the integration process refers to the `data_config` modality interface: it identifies the set of configurations to take into account for input and output data elaboration during the model training and inference workflow. Its purpose is to standardize visual observations, robot states, actions, and language annotations before they are processed by the model, and to ensure that model outputs remain compatible with the robot control space. A remarkable feature of this model implementation is that is possible to customize new

`data_config` modalities based on those that are most suitable for the reference embodiment.
For instance, when interacting with humanoid robots, it may be useful to state observations and actions in terms of absolute joint positions, while for a single robotic arm it may be preferable to deal with delta end-effector pose in cartesian space and gripper open/close command. This flexibility potentially enables a better interoperability with different fine-tuning datasets, each with its own typology of data representation. By isolating embodiment-specific details within a dedicated configuration layer, it enables integration of the foundation model without requiring architectural modifications.

A specific `data_config` is set for UR10e + Robotiq gripper embodiment: it is accomplished by creating a simple Python class and configuring some attributes like the functions to use for data transform, the expected format of the input modalities, and the length of the actions array to be predicted by the policy. The `UR10eDataConfig` class defines the embodiment-specific data interface used to adapt the sensory inputs and control signals of the manipulator to the multi-modal input/output format required by Isaac GR00T.

The configuration specifies four primary modalities:

Video inputs These correspond to an external camera and a wrist-mounted camera, offering complementary perspectives of the manipulation scene.

- `video.exterior_view`
- `video.wrist_view`

State variables The state representation includes joint positions of the manipulator and the Robotiq gripper aperture.

- `state.arm_joint_position`
- `state.gripper_qpos`

Action variables Actions consist of commanded joint positions and a binary gripper closing signal ($0 \rightarrow$ `open`, $1 \rightarrow$ `close`).

- `action.arm_joint_position`
- `action.gripper_close`

Language annotations Enables grounding of robot behavior in natural language

instructions.

- `annotation.human.action.task_description`

Moreover, the following custom options are set:

Temporal configuration Supports single-step state conditioning and multi-step action prediction.

- Observation horizon: current timestep only
- Action horizon: 16 future steps

Data transformation pipeline: [1]

Video Processing Ensures spatial consistency and improves robustness through controlled visual variability.

- Conversion to tensor format
- Spatial cropping (scale 0.95)
- Resizing to 224×224
- Photometric augmentation (brightness, contrast, saturation, hue)
- Conversion back to `numpy` format

State Processing Bounds numerical values and improves stability and regularization during training and inference.

- Tensor conversion
- Min–max normalization

Action Processing Preserves the physical semantics of continuous and discrete control signals.

- Tensor conversion
- Min–max normalization for joint commands
- Binary encoding for gripper command

Multimodal Concatenation Produces a unified multimodal representation and ensures compatibility with the transformer-based backbone of the model.

- Ordered merging of video, state, and action tensors

[1] all data are saved and elaborated as `numpy.ndarray` structure

4.1.2 Inference service API

The repository provides some standalone scripts to run training, evaluation, and inference of the VLA, allowing to specify several parameters such as the policy identifier, the `EmbodimentTag`, the `data_config` modality, and the dataset(s) path to consider.
The inference module is deployed as a standalone service, started via a dedicated initialization script. Upon execution, the service opens an HTTP (or ZMQ) socket and exposes two RESTful API endpoints:

- `/health [GET]`: checks the status of the server returning the corresponding code (i.e. `200 OK` when policy is correctly loaded and running)
- `/act [POST]`: triggers the actual inference using observations, instruction text, and video frames properly formatted from the request payload, and returns the computed actions in a JSON structure

If not differently specified in `data_config`, the policy outputs a chunk of 16 actions, meaning that the prediction is valid for 16 subsequent timesteps. This design decouples the model execution layer from the robotic control stack, allowing modular integration with the UR10e control pipeline.

4.2 Integration in Isaac Sim

Before delving into the fine-tuning process, it was first necessary to create a proper control flow interface, communicating with GR00T inference service, in order to be synchronized with the simulation environment. For this purpose, a standalone script was prepared leveraging the Isaac Sim Python API. The script content is structured as follows:

- The simulation is initialized and the custom **USD stag**e (described in 3.6.1) is loaded
- The **simulation step** accuracy is properly configured: in Isaac Sim, the simulation loop is controlled by two separate time-step parameters: the physics time step ($\Delta t_{\text{physics}}$) and the rendering time step ($\Delta t_{\text{rendering}}$). The former defines the integration interval used by the physics engine to numerically

solve rigid-body dynamics, contact interactions, and joint constraints, so it directly affects the stability and accuracy of the simulation. Reducing this step length increases numerical precision and improves contact stability, at the cost of higher computational load. Conversely, increasing it may introduce instability or physically inaccurate behavior. The rendering time step, instead, determines how frequently frames are generated for visualization and sensor simulation (e.g., RGB cameras). Modifying $\Delta t_{\text{rendering}}$ primarily impacts visual smoothness and the update rate of perception data, without altering the underlying physical evolution of the system.
Since the control loop has to be synchronized with the render frames, because it needs up-to-date camera observations, it was reasonable to set: $\Delta t_{\text{physics}} = \frac{1}{120}$ s, $\Delta t_{\text{rendering}} = \frac{1}{40}$ s. These values result in a good tradeoff between physical accuracy, smoothness, and well-balanced computational workload.

- 3 synchronized **RGB cameras** are configured:
  1. Scene camera (800×600 resolution)
  2. Exterior workspace camera (640×480 resolution)
  3. Wrist-mounted camera (640×480 resolution)

- The **action rate** is set to $a_{\text{rate}} = \frac{1}{20}$ s (20 Hz control frequency): it enables the actual synchronization between the simulation loop and the inner control loop iterations, every n frames, where $n = a_{\text{rate}}/\Delta t_{\text{rendering}}$. This action rate strictly depends on the optimal operative conditions of the VLA policy, which relate to the actual observations/actions capturing frequency employed inside the diverse training datasets: for Isaac-GR00T, it is documented that an appropriate value for control frequency should to be in a range between 10 and 30 Hz.

- The instruction of the task to execute ($\boldsymbol{\tau}_{\text{task}}$) is captured from user input: it is expressed as a natural language string and it is memorized and used for the entire duration of the simulated episode.

- The simulation starts and the complete **control loop** is triggered according to the previously defined a_{rate}.

A single control loop iteration involves several steps:

1. Collect observations for current frame[2]:

   $\boldsymbol{\tau}_{\textbf{task}}$**:** the natural language task description

   UR10e + gripper state: expressed in joint positions (angles in radiants) and gripper position:

   $$\mathbf{s} = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, q_{\text{grip}}]^T, \texttt{ shape=[7×1], dtype=float64}$$

   following the standard convention:

   θ_1: shoulder pan joint
   θ_2: shoulder lift joint
   θ_3: elbow joint
   θ_4: wrist 1 joint
   θ_5: wrist 2 joint
   θ_6: wrist 3 joint
   q_{grip}: finger joint (gripper position)

   RGB camera frames: one for each view (external and wrist-mounted):

   $$\mathbf{I}_{\text{ext}}, \mathbf{I}_{\text{wrist}}, \texttt{ shape=[480×640×3], dtype=uint8}$$

   The whole observation payload, created as a Python dictionary, is then:

   $$\textbf{obs} = \{\mathbf{I}_{\text{ext}}, \mathbf{I}_{\text{wrist}}, \mathbf{s}_{\text{arm}}, q_{\text{gripper}}, \boldsymbol{\tau}_{\text{task}}\}$$

2. Send GR00T inference request with the observation dictionary as payload, and wait for the server to produce the predicted action sequence:

   $$\mathbf{a}_{\text{horizon}} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_H]$$

   where H is prediction horizon (default: 16 steps) and each action

   $$\mathbf{a}_i = \{\boldsymbol{\theta}_{\text{target}}, g_{\text{cmd}}\}$$

   incorporates the target arm joint positions to reach ($\boldsymbol{\theta}_{\text{target}}$ in radiants), and

[2] all data are saved and elaborated as `numpy.ndarray` structure

a binary gripper command $g_{\text{cmd}} \in \{0.0, 1.0\}$

$$g_{\text{action}} = \begin{cases} \text{close} & \text{if } g_{\text{cmd}} = 1.0 \\ \text{open} & \text{otherwise} (= 0.0) \end{cases}$$

3. Interpolate actions linearly to match physics rate:

$$r_{\text{interp}} = \frac{a_{\text{rate}}}{\Delta t_{\text{physics}}} \rightarrow \mathbf{a}_{\text{interp}} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_{H \cdot r_{\text{interp}}}]$$

4. Reprocess $\mathbf{a}_{\text{interp}}$ passing it through a denoising filter (e.g. Gaussian filter) in order to mitigate high frequency components that might introduce some jerky oscillation and degrade the performance

5. Apply such computed actions via `ArticulationAction` API as Isaac Sim controller: send action commands for the equivalent of $H \cdot r_{\text{interp}}$ step frames, until the end of $\mathbf{a}_{\text{interp}}$ array is reached; then retrieve new actions commands restarting from point 1.

4.2.1 ROS 2 bridge

The custom USD stage is configured to seamlessly integrate a ROS 2 perception and control layer, exploiting the built-in ROS 2 Bridge optimized for Omniverse framework. So, it is possible to run the simulation script with a related `--ros2` flag, which enables:

- Joint states publishing directly from simulation (on `/joint_states` topic)
- Camera images streaming to RGB `/rgb_exterior` and `/rgb_wrist` topics
- Command action reception via `/joint_command` topic subscription

For this purpose, two custom ROS 2 nodes are initialized and run using `rclpy` Python library:

- `JointController` subscribes to `/joint_states` and publishes actions on `/joint_command`
- `CameraReader` subscribes to RBG topics and periodically reads and stores the camera frames ready to be provided for to main control loop perception layer

4.2.2 Zero-shot test

After setting up all the components described so far, a zero-shot test of the policy (i.e. without any fine-tuning procedure) was conducted using the ready-to-use `EmbodimentTag.OXE_DROID`, which is specifically thought for single robotic arm embodiments performing manipulation task. As mentioned before, this action head was pre-trained on large and rich datasets such as Open X Embodiment and DROID, but unfortunately, it did not perform well within this environment: given the simple instruction *"lift the cube"*, the policy output showed a really noisy and unstable movement, always failing the task assigned. The main reason might be related to the scarcity and lack of variance of training episodes specifically involving the UR10e manipulator: although these huge datasets include demonstration with several kinds of robotic arms, there is evidence that in this context, such largely pre-trained policy could not achieve a valuable performance.

Chapter 5

Fine-tuning pipeline

In order to unlock and explore the full capability of Isaac-GR00T VLA, it was then necessary to perform ad-hoc fine-tuning for the almost unseen UR10e + gripper embodiment. First, the possibility to exploit open and available datasets, that were already suitable for this purpose, was investigated, but eventually this idea was discarded due to the lack of existing good quality and generalist data involving the specific robotic platform of interest in this workflow.
The choice was then to collect a brand-new dataset, totally manageable and customizable for the specific scene and setup prepared. It is worth reminding that the target here is specifically to enhance and optimize the model, aiming to improve its behavior inside a single, fixed environment, which is what a generic foundation model is supposed to be able of.
Given these necessities, a dedicated data collection pipeline was developed and integrated inside the Isaac Sim framework.

5.1 LeRobot Dataset format

The LeRobot V2.0 dataset format defines a standardized structure for storing multimodal robotic manipulation data, supporting imitation learning, offline reinforcement learning, and Vision-Language-Action training pipelines. The

format used by Isaac-GR00T is analogous to LeRobot standard, with some customization that will be discussed further on.

A dataset compliant with the LeRobot specification is organized into three main components, each with a distinct semantic and functional role:

```
./
├── meta/
├── data/
└── videos/
```

Data storage (`data/`)

The `data/` directory contains one Parquet file per episode, organized in chunks:

```
data/
└── chunk-000/
    ├── episode_000000.parquet
    ├── episode_000001.parquet
    └── ...
```

Each `.parquet` file represents a single episode and stores time-indexed robot transitions. The required columns are:

- `observation.state`: a one-dimensional float array containing the full robot state at time t. It is defined as the concatenation of all state modalities (e.g., joint positions, velocities, gripper state).
- `action`: a one-dimensional float array containing the action applied at time t, also stored as a concatenated vector.
- `timestamp`: floating-point scalar representing time in seconds from episode start.
- `episode_index`: integer identifying the episode.
- `index`: global observation episode index across the entire dataset.
- `task_index`: index referencing a specific task instruction expressed in natural language.
- `next.reward`: reward associated with the transition.
- `next.done`: boolean indicating episode termination.

Additional annotation fields follow the convention:

`annotation.<source>.<type>(.<name>)`

Each possible annotation column, the task description included, stores an integer index pointing to an entry in `meta/tasks.jsonl`. In standard LeRobot datasets, language supervision is typically provided via the `task_index` column.

Video Observations (`videos/`)

The `videos/` directory contains RGB recordings aligned with each episode:

```
videos/
└── chunk-000/
    └── observation.images.<camera_name>/
        ├── episode_000000.mp4
        ├── episode_000001.mp4
        └── ...
```

Video data are stored separately from structured state data and aligned implicitly through the episode index. At inference time, each video frame is coupled with the corresponding entry in the `.parquet` file of the episode based on the `timestamp` value. The directory name encodes the modality: for instance, in UR10e configuration `<camera_name>` values are `wrist_view` and `exterior_view`.

Metadata (`meta/`)

The `meta/` directory contains dataset-level metadata.

```
meta/
├── episodes.jsonl
├── info.json
└── tasks.jsonl
```

The `tasks.jsonl` file stores all language instructions, one per line:

```
{"task_index": 0, "task": "pick up the cube"}
```

The `.parquet` files then reference task instructions via integer indices rather than storing raw strings.

The `episodes.jsonl` file stores episode-level information:

```
{"episode_index": 0, "tasks": [...], "length": 416}
```

Each entry specifies the episode identifier, associated tasks, and episode length.

The `info.json` file contains general dataset information such as version, provenance, statistics, and modality configurations. For instance, the `info.json` version for UR10e specifies:

- resolution, frame-rate, and codec used for video modalities (`wrist_view` and `exterior_view`): `[480x640x3], 20 fps, h264`
- data type and names for state/actions modalities:

```
{
"dtype": "float64",
"shape": [7],
"names": [
    "shoulder_pan_joint",
    "shoulder_lift_joint",
    "elbow_joint",
    "wrist_1_joint",
    "wrist_2_joint",
    "wrist_3_joint",
    "finger_joint"
    ]
}
```

LeRobot-like format for Isaac GR00T

The GR00T LeRobot format is an extension of the standard LeRobot V2.0 specification. It remains backward-compatible while introducing additional structure and constraints required by large-scale Vision-Language-Action models.

Unlike standard LeRobot, GR00T requires an additional file:

```
meta/
└── modality.json
```

This file provides a structured decomposition of:

- `observation.state`
- `action`
- `video`
- `annotation`

```
"state": {
    "arm_joint_position": {
        "start": 0,
        "end": 6
    },
        "gripper_qpos": {
        "start": 6,
        "end": 7
    }
},
"action": {
    "arm_joint_position": {
        "start": 0,
        "end": 6
    },
        "gripper_close": {
        "start": 6,
        "end": 7
    }
},
"video": {
    "exterior_view": {
        "original_key": "observation.images.exterior_view"
    },
    "wrist_view": {
        "original_key": "observation.images.wrist_view"
    }
},
"annotation": {
    "human.action.task_description": {},
    "human.validity": {},
    "human.coarse_action": {
        "original_key": "annotation.language.language_instruction"
    }
}
```

Figure 5.1: Content of modified `modality.json` file for custom data configuration

Since state and action are stored as flat arrays, `modality.json` defines:

- Start and end indices for each modality
- Rotation representation (e.g., quaternion, axis-angle, Euler)
- Absolute vs. relative action encoding
- Optional data type and normalization range
- Camera key remapping
- Explicit annotation channels

This enables fine-grained supervision, structured normalization, and rotation-aware transformations during training. For the environment configured and used in this thesis work, a few customized modalities are set among those: the actual configuration is reported in fig. 5.1.

Moreover, GR00T supports multiple independent annotation channels following the format: `annotation.<source>.<type>`.

Each annotation stores an integer index pointing to `meta/tasks.jsonl` file and must be explicitly declared in `modality.json`. This allows:

- Coarse- and fine-grained language supervision
- Human and synthetic annotations
- Validation and quality flags
- Multi-task conditioning

GR00T introduces explicit support for rotation encodings within `modality.json`, including:

- axis-angle
- quaternion
- 6D rotation representation
- rotation matrix
- multiple Euler angle conventions

This avoids geometric ambiguity and ensures consistency across embodiments.

Finally, the standard format may include the `meta/stats.json` file, which is it is not required by the GR00T data loader and can be omitted. It stores several

statistical metrics for each field collected inside the `.parquet` files, such as mean, standard deviation, q01, q99.

The additional structure introduced in the GR00T flavor is motivated by the requirements of large-scale transformer-based Vision-Language-Action models, ensuring consistent tokenization of modalities, structured supervision, and cross-embodiment compatibility.

5.2 Dataset acquisition pipeline

Once the dataset structure format has been fixed, another custom standalone Python script was developed to perform the actual registration of teleoperated demonstration episodes within Isaac Sim. The script workflow is quite similar to the one described in previous section (4.2) to run inference: the base USD scene, the initialization flow, the physics configuration, and the synchronization loop act substantially the same way.
The overall pipeline follows an episode-based acquisition paradigm, where each episode corresponds to one complete execution of a task described in natural language.

The crucial steps and solutions adopted are described below.

Dataset integration and metadata handling: preliminary steps

The script perfectly integrates with an existing dataset directory containing the metadata files (`/meta` folder, see 5.1). From `info.json`, the script retrieves:

- The total number of episodes and chunks,
- The global frame counter,
- The recording frame rate (FPS),
- Path templates for trajectory and video storage,
- Feature definitions.

This mechanism guarantees that newly recorded episodes are appended consistently without overwriting existing data.

After importing the USD stage, setting up the physics for UR10e + Robotiq gripper, both external and wrist-mounted cameras are activated and associated with a `cv2.VideoWriter` object to record synchronized video streams during each episode.

At the beginning of each episode, the operator provides a natural language description of the task. The script:

- Matches the description against existing tasks,
- Assigns a task index,
- Creates a new task entry if no match is found,
- Updates `tasks.jsonl` accordingly.

Manual teleoperation via End-Effector control

During episode recording, the robot is controlled manually through keyboard input. In this manner it is possible to perform arbitrary trajectories in real time: for this purpose a third camera with complete view of the scene was used as reference for ensuring better controllability by the user.

Incremental cartesian commands are applied to the end-effector:

- Arrow keys (←↑→↓): planar motion along the x–y axes.
- Numpad keys (`Numpad.0`,`Numpad.5`): motion along the z axis.
- Additional numpad keys (`Numpad.4`,`Numpad.8`,`Numpad.6`,`Numpad.2`): small rotational increments on *pitch* and *yaw* axis, relative to the TCP reference frame.

At each frame, such keyboard input are captured and cartesian deltas are converted into joint-space commands using inverse kinematics via a custom `KinematicsSolver`, that exploits the standardized URDF for UR10e equipped with the Robotiq 2F-140 gripper. The target end-effector pose is thus updated incrementally at each rendering step.

For gripper control, the open/close binary actions are associated to `Numpad.ADD` and `Numpad.ENTER` keys respectively. The actions are mapped to joint position commands and applied through the Isaac Sim articulation controller API.

Scene randomization

When enabled, domain randomization is applied before episode execution, during the initialization phase:

- Objects are randomly scattered on the desk surface.
- Objects' orientations are randomized around the vertical axis.
- The height of both tables is randomized (by about $\pm 10\%$ of their initial height).
- The position of the camera mounted on the end-effector is slightly randomized along the z-axis of the wrist.
- The initial end-effector pose is sampled within a bounded workspace.

This aims to increase variability in demonstrations and improve generalization during fine-tuning. It is a crucial element for effectively addressing the possible issues deriving from Sim-to-Real gap.

Episode recording

An episode begins after simulation reset and task specification by the operator. At a fixed acquisition frequency, equivalent to the `fps` value stored in dataset metadata (`meta/info.json`):

1. The current joint states (positions) are recorded.
2. The executed action vector is reconstructed by applying the proper teleoperation command to the end-effector current pose, and then performing inverse kinematics to retrieve new target joint positions; if a gripper command is triggered as well, it is registered as a `finger_joint` position in radiants.
3. A timestamp relative to the episode start is stored.
4. RGB frames from each camera are captured and written to `.mp4` video files.

For each timestep, the following data fields are thus collected:

- `observation.state` (joint + gripper positions),
- `action` (joint + gripper new target positions),
- `timestamp`,
- `task_index` (index of the task identified by the description provided, referring to `meta/tasks.jsonl`),
- `episode_index`

Episode finalization and storage

When the simulation stops, the operator decides whether to save or discard the episode. If discarded, video files are deleted and all data structures used for recording are emptied. Otherwise:

- A `.parquet` file containing the trajectory data is written to disk.
- Video files are post-processed using `ffmpeg` for encoding compatibility.
- Metadata counters are updated.
- The episode entry is appended to `episodes.jsonl`.

Dataset-level statistics such as total episodes, total frames, and total tasks are updated consistently in `info.json`.

Restarting the simulation, it is possible to reset the scene and restart the loop for a new episode recording.

5.3 Fine-tuning process

With the such collected dataset(s), the actual fine-tuning procedure of Isaac-GR00T is conducted by using one of the provided Python scripts in the repository, with some adjustments. This script, as well as the whole repo, exploits PyTorch library for the entire pipeline, from data loading to checkpoint saving.

The fine-tuning script handles several arguments, useful for better customizing the training process parameters in order to fit its computational workload for the target platform on which it is supposed to run. Some of them are here reported,

with their actual values used in the training sessions carried out during this thesis work (formatted as `[:= <value>]`):

- **dataset_path**: List of dataset directories used for training; it is possible to specify either a single dataset or a number of these.
- **output_dir**: Directory where checkpoints and training artifacts are stored.
- **data_config**: The data configuration identifier for defining modalities, preprocessing transforms, action dimensionality and everything related (see 4.1.1).
- **batch_size**: Batch size per GPU `[:= 32]`.
- **max_steps**: Maximum number of optimization steps `[:= 10000]`.
- **save_steps**: Interval (in steps) between checkpoint saves `[:= 2000]`.
- **base_model_path**: Path or HuggingFace identifier of `GR00T-N1.5` model base policy; it is useful also for resuming training from a previous checkpoint, maybe considering a new dataset.
- **tune_llm**: Enables fine-tuning of the language backbone `[:= False]`.
- **tune_visual**: Enables fine-tuning of the vision encoder `[:= False]`.
- **tune_projector**: Enables fine-tuning of the action head projector `[:= True]`; this represents the model section that is actually responsible for learning and adapting the policy to new embodiments, keeping unaltered the base knowledge incorporated by the backbone.
- **tune_diffusion_model**: Enables fine-tuning of the diffusion-based action model `[:= False]`.
- **learning_rate**: Learning rate for the AdamW optimizer `[:= 1e-4]`.
- **weight_decay**: Weight decay coefficient `[:= 1e-5]`.
- **warmup_ratio**: Fraction of total training steps that are used for learning rate warmup `[:= 0.05]`.
- **gradient_accumulation_steps**: Number of gradient accumulation steps before optimizer update `[:= 1]`.
- **lora_rank**: Rank of LoRA adaptation (if > 0, enables parameter-efficient fine-tuning) `[:= 0]`.
- **video_backend**: Backend used for video decoding `[:= decord]`.
- **balance_dataset_weights**: Balances sampling frequency across datasets in mixture mode.

- **balance_trajectory_weights**: Samples trajectories proportionally to their length in mixture mode.

After parsing all the arguments, the script performs the following high-level steps:

1. Load and preprocess the dataset (single or mixture).
2. Load the pre-trained GR00T model.
3. Adapt the action head to match the dataset's action dimensionality and horizon.
4. Optionally apply LoRA (Low Rank Adaptation) for parameter-efficient fine-tuning.
5. Configure HuggingFace `TrainingArguments`, among which the learning rate scheduling (cosine scheduler) and AdamW optimizer.
6. Launch `torch` training using a custom `TrainRunner`.

The whole training process was performed on a high-performance workstation equipped with a NVIDIA RTX 5090 GPU (Blackwell architecture). A complete fine-tuning run on a quite rich dataset, equivalent to 10000 steps, took for these experiments less than 2 hours, using the parameters specified above.

Chapter 6

Real-world deployment and Sim-to-Real transfer

After completing Isaac-GR00T integration into simulation and defining the fine-tuning process, the last and most important step was to actually perform the policy deployment inside a real environment.

Of course, deploying complex models like VLAs on physical robotic platforms introduces a set of challenges that extend well beyond simulation-based validation. The transition to a real industrial manipulator inevitably exposes discrepancies in perception, dynamics, latency, and safety constraints. All these mismatches, commonly referred to as the Sim-to-Real gap, can significantly degrade policy performance if not properly addressed.

6.1 System architecture for deployment

All the system components (described in chapter 3) are connected to a single high-performance workstation. This machine is responsible for:

- Running the GR00T inference service inside a Docker container configured with a Conda environment for better isolation and ease of replicability, exposing the aforementioned API endpoints (see 4.1.2);

- Hosting the ROS 2-based communication stack;
- Interfacing with cameras and the robot controller.

The UR10e operates in remote control mode, continuously streaming state feedback, and thus enabling fully closed-loop operation. A dedicated Ethernet link is present between the workstation and the manipulator control box, in order to minimize communication latency and ensure deterministic command delivery through a direct connection.

6.1.1 ROS 2 configuration

The software architecture is built upon ROS 2 Jazzy, using a custom-developed package derived from NVIDIA Isaac ROS ([42], [43], [44]). The package integrates multiple subsystems into a unified communication layer:

Universal Robots ROS2 driver Provides access to joint states, robot status, and low-level command interfaces. As for perception layer, the `/joint_states` topic is used to retrieve the actual pose of the manipulator via joint positions, while `/tf` provides the end-effector up-to-date configuration. Regarding control layer, the driver enables different kinds of `ros2-controller`: for this configuration the `/scaled_joint_trajectory_controller` is exploited. It enables smooth position-based control, which is necessary for the kind of actions retrieved by the policy, and allows to specify the desired trajectory duration; it also adapts the computed joint velocities to the overall speed scaling factor which can be optionally edited from the UR10e teach pendant by the operator for safety reasons.

Custom gripper adapter Implements a dedicated interface for controlling the Robotiq 2F-140 gripper, exposing open/close and position commands consistent with the binary action representation expected by the policy. Thanks to the URCap [Universal Robots Capability] plugin interface, the state of the gripper is directly read from `/joint_states` topic message, corresponding to `/finger_joint` denomination; instead, the open/close command is encapsulated inside an action goal and sent to the ROS 2 action `/robotiq_gripper_controller/gripper_cmd`.

RealSense camera drivers Handle synchronized RGB acquisition from both the external and wrist-mounted sensors. Cameras are set-up with a standard resolution of 640x480, and frequency of 30 Hz. For this use case, only one RGB channel (left sensor) is considered: the frames are read from the `camera_<n>/color/image_raw` topic, where `n` is `1` for the end-effector camera, and `2` for the external one.

All ROS 2 nodes run on a separated Docker container, on the same workstation where inference is executed. This co-location eliminates inter-machine serialization overhead and reduces end-to-end control latency.

6.2 Orchestration and control loop

The overall execution is orchestrated by a Python control script that acts as a high-level runtime manager. The script performs the following operations:

- First, two ROS 2 nodes are initialized through `rclpy` library, similarly to what was done in Isaac Sim framework (section 4.2.1):
  - `JointController` node deals with the manipulator interaction: it collects and stores joint states proprioception and sends target position commands leveraging the `/scaled_joint_trajectory_controller` and gripper ROS 2 action interface.
  - `CameraReader` node is responsible for intercepting raw RGB frames publication from both cameras, and making them available in the proper format when the observation dictionary is built up.
- The system receives a natural language instruction describing the task to execute. This instruction is preprocessed and then passed to the GR00T inference API request payload.
- Before starting with the actual inference loop, the robot is optionally commanded to reach a predefined *home* pose, fixed in order to permit the wrist-camera to have a comprehensive view of the working area, and the gripper is opened. This behavior is manageable through a simple flag passed as script argument.

- The continuous control loop is initiated, structured in a very similar way to that in simulation (see 4.2):

  Perception phase: Collect observations and organize them in a structured dictionary, involving robot joint and gripper states, and RGB frames from both external and wrist-mounted cameras[1].

  Inference phase: Forward pass through the deployed Isaac-GR00T (v1.5) model via `POST /act` API. Generate the action vector conditioned on current instruction and observation, and reprocess it passing through a linear interpolation followed by the application of a noise-mitigating Gaussian filter.

  Control phase: Iterate over the such predicted actions array, and map it to the effective commands to send using `JointController` node: the model outputs actions as absolute joint positions, so for each control timestep a new `JointTrajectory` message is created and populated with a single `JointTrajectoryPoint` corresponding to the position to reach; the trajectory duration is calibrated in order to complete right before the next action command, so

  $$d_t = \frac{1}{f_{\text{ctrl}} \cdot s_{\text{speed}}}$$

  where d_t is the commanded trajectory duration, f_{ctrl} is the actual control frequency of this high-level script (i.e. the rate at which the ROS 2 commands are sent to the robot), and s_{speed} [from 1 to 100%] is the speed scaling factor set directly on the teach pendant connected to the manipulator control box.

  The binary gripper command ($0 \rightarrow$ `open`, $1 \rightarrow$ `close`) is computed at the same control frequency as well, but it is actually triggered only if a state change is detected from the previous loop iteration: this check is needed to avoid sending unnecessary repeated commands. The gripper action goal is sent synchronously, in the sense that the script waits for

[1] all data are handled as `numpy.ndarray` structures

the `open/close` execution to complete (i.e. a successful response for ROS 2 action) before resuming execution.

- When the predicted action horizon is elapsed (by default, after 16 time-steps), a new perception–inference–control cycle iteration is performed, until the task is completed and the operator interrupts the script execution.

As it happened in simulation, this script is programmed to compute and send new action commands roughly at 20 Hz frequency: to ensure the best possible sim-to-real transfer, it corresponds to the action rate at which the training episodes were collected. During actual control execution, indeed, the command rate is always bounded by the slowest component in the loop structure, typically dominated by inference latency (about 0.7-0.9 seconds on the employed platform) and camera acquisition time, which as previously mentioned is handled by a separate ROS 2 node and in principle should not affect the main execution flow. However, the longer is the prediction horizon considered for a single action chunk retrieved, the lower this inevitable slowdown effect ends up to alter the overall system performance.

6.3 Experimental evaluation

Having specified the implementation and all technical details, in this section we focus on the performance evaluation of the fine-tuned Isaac-GR00T policy.

As previously said (see 3.6), throughout all this thesis work the target application of the model is to execute basically two kinds of tasks:

Lift task: consists of reaching and grasping a specified object from the operating surface and maintaining an effective grip while lifting;

Pick-and-place task: consists of lifting a specified object, then moving it towards the target placing area (a box), and finally performing the release phase inside it. It is fundamental to avoid collisions during the entire task execution.

The experimental setup is described in section 3.6, too. The main items subject to interaction with the manipulator are here summarized:

- the operating desk, in front of the UR10e, on which the objects to manipulate are placed: a little cube with multicolor texture, a tin can, a screwdriver, some pliers and a wrench;
- a secondary table, placed on the right side of the robot, on top of which there is a plastic black box corresponding to the placing area when *pick-and-place* is performed;

6.3.1 Dataset specifications

Several trial datasets were collected for the fine-tuning phase (see 5.2), and since the pipeline was totally based on hand-recording operations inside Isaac Sim, each one of them had quite different features, involving surrounding context variability, distribution of task episodes, duration and complexity of the trajectories and domain randomization dimensionality.
Among all these, the GR00T policy eventually employed for this evaluation was post-trained with just a single comprehensive dataset:

- It is composed of **75 episodes**, corresponding to about **35k frames** in total;
- It involves episodes concerning both *lift* and *pick-and-place* tasks, with 38 and 27 episodes respectively;
- All episodes have been recorded at a rate of **20 frame-per-second**, with duration ranges of about 10-15 seconds for *lift* and 22-45 seconds for *pick-and-place*; despite it might appear unequally distributed, indeed the *pick-and-place* episodes are less but last longer so they contribute with more frames;
- For simplicity, the cube was selected as the target object for both manipulation tasks throughout all the recorded episodes: the other objects on the desk act as confounding elements or obstacles;
- All episodes related to the same task share a unique grammatical formulation for the natural language instruction:
  - `"pick up the cube"`, in case of *lift*
  - `"pick up the cube and put it inside the box"`, for *pick-and-place*

- Domain randomization (see 5.2) and data augmentation techniques were exploited during dataset collection in order to guarantee a better generalizability of the trained policy:
  - All objects on the operating desk are randomly scattered on the x-y plane.
  - Objects' initial orientations are randomized around the vertical axis.
  - The height of both tables is randomized.
  - The position of the camera mounted on the end-effector is slightly randomized along the z-axis of the wrist.
  - The initial end-effector pose is sampled within a bounded workspace.
- For each episode, it was crucial to record a clean and smooth trajectory, ensuring the best possible data quality, but it was also useful to include some partial failures and defects along the movement execution, for instance a wrong or sub-optimal grasp pose, a little crash or an object falling, followed by corrective actions in order to make the policy more robust in case of real failures at inference time.

Although it might be unrealistic to train the policy to deal with only one object to manipulate, the choice of operating always with the cube came from the fact that the amount of good quality episode data that was possible to collect manually is really limited and time-consuming. As a fully experimental context, the goal here was to reduce as much as possible environmental secondary factors that could complicate and slow down the training effectiveness: since the language and vision perception capabilities of the model are freezed inside the VLM backbone, the main scope for improvement consists in matching the proprioception to the trajectory generation for the new UR10e embodiment, which in general should not be much influenced by the specificity of the manipulated object, at least for the approaching and transport phase. Of course, it may become more relevant when dealing with different grasp poses, but here the target was to investigate whether decoupling the actual physical and perception part could in fact reduce the amount of data required for a successful policy adaptation.

6.3.2 Tests configuration

A total of **34** experimental tests were articulated in order to evaluate the policy according to four complementary categories:

1. **Grasping accuracy**
2. **Robustness to object and TCP configuration**
3. **Robustness to language instructions**
4. **Task consistency with different target object**

Each test consists of issuing a natural language instruction to the model while varying the spatial configuration of the objects and the robot's TCP (referred to as *Config.* index) at start time. The robot's execution was then evaluated as successful [**Y**], failed [**N**], or partially successful [**Y***] depending on whether the intended manipulation was completed. Also, some notes are provided to add some information about the failing issues.
If not differently specified, the default instruction used during evaluation are the same as in the training dataset:

- `"pick up the cube"`, in case of *lift*
- `"pick up the cube and put it inside the box"`, for *pick-and-place*

Complete task trajectories are represented in figures 6.1, 6.2: the frames are taken from real test execution for both tasks.

6.4 Results and discussion

The detailed test configurations and outcomes for both tasks are reported in the following tables:

- Tables 6.1, 6.3 report results in Isaac Sim simulation environment, thus addressing the Sim-to-Sim transfer
- Tables 6.2, 6.4 report results in reality framework, using the experimental setup previously described

Overall results metrics, considering both simulation and real-world environment, are reported in table 6.5.

Lift: Sim-to-Sim evaluation

Test #	Instruction	Config.	Success	Notes
1	*default**	#1	**Y**	-
2	*default*	#2	**Y**	-
3	*default*	#3	**Y**	-
4	*default*	#4	**Y**	-
5	*default*	#5	**Y**	-
6	*default*	#6	**Y**	-
7	*default*	#7	**Y**	-
8	*default*	#8	**Y**	-
9	*default*	#9	**Y**	-
10	*default*	#10	**N**	Missed grasp (it retries but fails)
21	"lift the cube"	#1	**Y**	-
22	"grasp the cube"	#1	**Y**	-
23	"can you pick up the cube?"	#1	**Y**	-
24	"lift the cube"	#2	**Y**	-
25	"grasp the cube"	#2	**Y**	-
26	"can you pick up the cube?"	#2	**Y**	-
34	"pick up the tin can"	#1	**Y**	-

Table 6.1: *Lift* task: Results of evaluation tests in Isaac Sim

Lift: Sim-to-Real evaluation

Test #	Instruction	Config.	Success	Notes
1	*default**	#1	**Y**	-
2	*default*	#2	**N**	Gripper hits the table when closing
3	*default*	#3	**Y**	-
4	*default*	#4	**N**	-
5	*default*	#5	**N**	Missed grasp (view obstructed by the screwdriver)
6	*default*	#6	**Y**	-
7	*default*	#7	**N**	Obj. mismatch: it grasps the tin can
8	*default*	#8	**Y**	-
9	*default*	#9	**Y**	-
10	*default*	#10	**N**	Missed grasp (it retries but fails)
21	"lift the cube"	#1	**Y**	-
22	"grasp the cube"	#1	**Y**	-
23	"can you pick up the cube?"	#1	**Y***	-
24	"lift the cube"	#2	**Y**	-
25	"grasp the cube"	#2	**Y**	-
26	"can you pick up the cube?"	#2	**Y**	-
34	"pick up the tin can"	#1	**Y**	-

Table 6.2: *Lift* task: Results of evaluation tests in real-world

Lift: Complete task demonstration

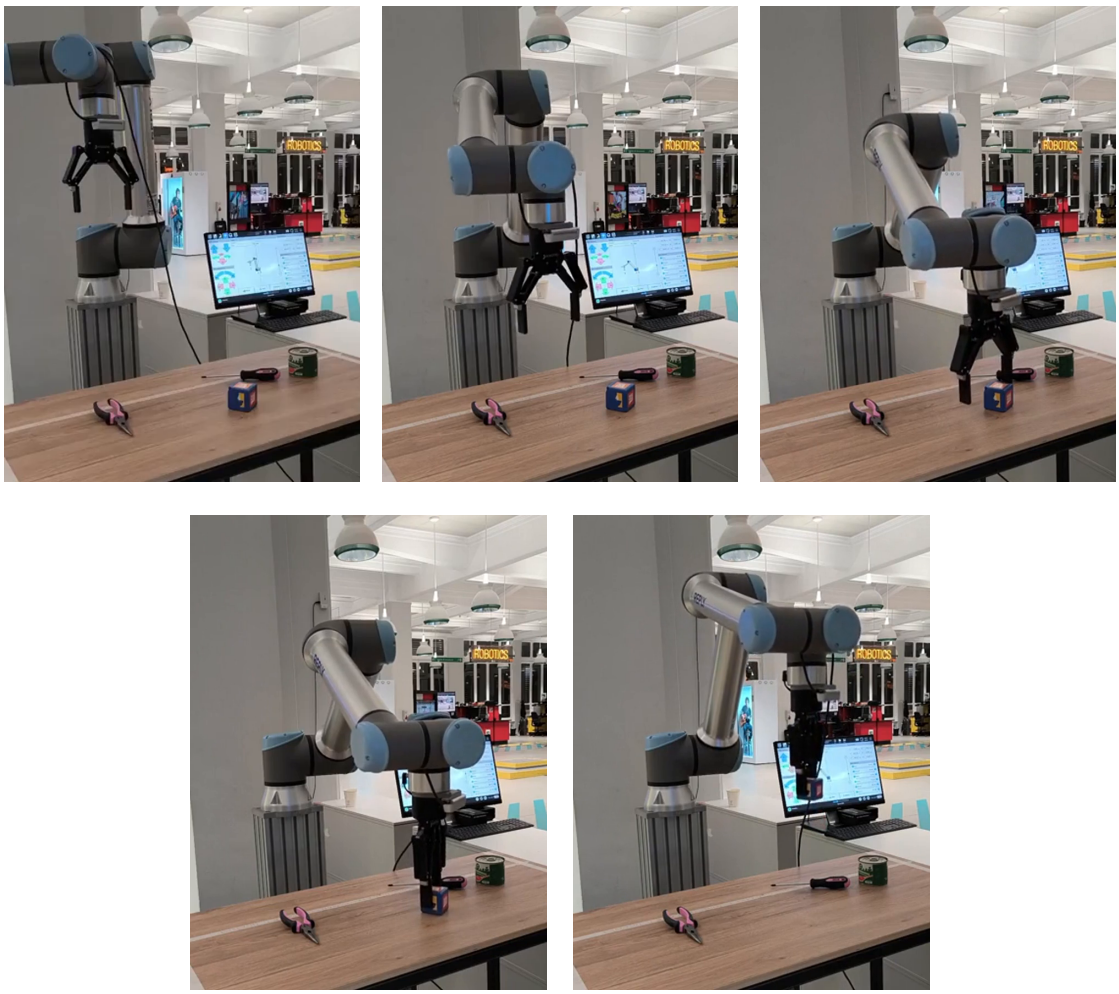


Figure 6.1: *Lift*: Key frames from full real-world task demonstration

Pick and Place: Sim-to-Sim evaluation

Test #	Instruction	Config.	Success	Notes
11	*default**	#1	**Y**	-
12	*default*	#2	**Y***	Lift OK but hits the bottom of the box
13	*default*	#3	**Y**	-
14	*default*	#4	**N**	Obj. mismatch: it grasps the tin can
15	*default*	#5	**N**	Missed grasp
16	*default*	#6	**Y**	-
17	*default*	#7	**Y**	-
18	*default*	#8	**Y**	-
19	*default*	#9	**Y**	-
20	*default*	#10	**N**	-
27	"take the cube and put it inside the tray"	#1	**Y**	-
28	"could you place the cube in the container?"	#1	**Y**	-
29	"move the cube into the box"	#1	**Y**	-
30	"take the cube and put it inside the tray"	#2	**Y**	-
31	"could you place the cube in the container?"	#2	**Y**	-
32	"move the cube into the box"	#2	**N**	-
33	"place the cube in the box"	#11	**Y**	-

Table 6.3: *Pick-and-place* task: Results of evaluation tests in Isaac Sim

Pick and Place: Sim-to-Real evaluation

Test #	Instruction	Config.	Success	Notes
11	*default**	#1	**Y***	Lift OK but bumps into the box walls
12	*default*	#2	**Y***	Lift OK but hits the bottom of the box
13	*default*	#3	**Y**	-
14	*default*	#4	**N**	Obj. mismatch: it grasps the tin can
15	*default*	#5	**N**	Missed grasp (wrong approaching pose)
16	*default*	#6	**Y**	-
17	*default*	#7	**Y**	-
18	*default*	#8	**Y**	-
19	*default*	#9	**Y**	-
20	*default*	#10	**N**	Uncertain movement, almost stationary
27	"take the cube and put it inside the tray"	#1	**Y**	-
28	"could you place the cube in the container?"	#1	**Y**	-
29	"move the cube into the box"	#1	**Y**	-
30	"take the cube and put it inside the tray"	#2	**Y**	-
31	"could you place the cube in the container?"	#2	**Y**	-
32	"move the cube into the box"	#2	**Y**	-
33	"place the cube in the box"	#11	**Y**	-

Table 6.4: *Pick-and-place* task: Results of evaluation tests in real-world

Pick and Place: Complete task demonstration

Figure 6.2: *Pick-and-place*: Key frames from full real-world task demonstration

Metric	Value [Sim]	Value [Real]
Total number of tests	34	34
Successful executions	28	19
Failed executions	5	12
Partially successful executions	1	3
Overall success rate (partial = 0.5)	**83.82 %**	**60.29 %**

Table 6.5: Summary of evaluation results [sim vs real]

In general, the experiments demonstrate that the fine-tuned model is capable of transferring manipulation behaviors learned from the dataset to a real robotic platform. The robot frequently succeeds in approaching the correct object and performing the grasp motion, indicating that the visual-motor policy learned during training captures meaningful manipulation patterns.

As expected, performance in Sim-to-Sim evaluation are better and more robust with respect to the real-world experiments, with a higher overall success rate (83 vs 60 %), but the gap seems not to be much marked. The most relevant observations and outcomes are reported hereinafter.

Grasping accuracy

A first group of experiments focuses on the robot's ability to correctly localize and grasp the target object. These tests evaluate whether the policy can estimate a suitable grasp pose from the visual input and execute the approach trajectory correctly. The majority of failure cases observed in the experiments are related to grasp misalignment. In some cases, the robot approaches the object but performs the grasp at an incorrect height or position. For example, some trials show the gripper closing too low relative to the object, resulting in a failed pickup. In other cases, the end-effector gets close to the object but misses the grasp point.

These behaviors suggest that while the policy can often guide the manipulator near the target object, the final grasp pose estimation remains sensitive to small perception or pose estimation errors.

Robustness to object and TCP configuration

Another set of tests investigates the robustness of the policy with respect to different spatial configurations of the object and the robot's TCP starting position. In these experiments, the target object (the cube) was placed at varying positions on the workspace and the robot was initialized from different configurations. The objective was to verify whether the learned policy generalizes beyond the exact configurations observed during training.
The results indicate that the policy is generally capable of approaching the target object even when the configuration differs from the training demonstrations. However, some executions show hesitation or incomplete trajectories, where the robot appears to stall or behave indecisively before completing the manipulation. This behavior suggests possible uncertainty in the policy when encountering less familiar spatial arrangements.

Robustness to language instructions

Some experiments were then focused to the language understanding part, verifying that the semantic interpretation of the task instruction is not affected by different formulations, grammar expressions, synonyms and so on. As previously described, the pure language interpreting capability should be not strictly related to the custom fine-tuning process for the new embodiment, since it resides inside the VLM backbone of the model, maintained freezed and unaltered. In facts, these expectations about language robustness are quite confirmed: some tests were run replacing the *default* instructions* with alternative forms, using synonyms for names and verbs, and also changing the sentence structure from imperative to interrogative. The model proved to be capable of associating these new instructions to the correct task and behavior, demonstrating that this modality is well managed and hardly affects at all the overall policy performance.

Task consistency with different target object

Some tests also reveal issues related to object selection consistency. In certain trials the robot successfully grasps an object, but it is not the intended one.
This behavior highlights a limitation in the perception-language alignment of the

model. Although the system correctly executes the grasp motion, the semantic grounding between the instruction and the visual scene is occasionally ambiguous. Such cases indicate that improving the dataset diversity or strengthening the visual-language association could improve task reliability.

On the other hand, one representative test reveals the ability of the model to generalize the task execution when the target object indicated in the instruction is different from the cube: for instance, the modified task instruction "pick up the tin can" is successfully completed in both simulated and real environment. Although the tin can is in this case a well known object within the fine-tuning dataset, this capability of the model is for sure related to its embedded vision intelligence as well: once again, the VLM backbone enables a considerable flexibility in this respect.

Pick-and-place

While the robot is often able to complete the pick phase, presenting a similar behavior to what seen with the *lift*, some failures occur during the placement. For example, in some trials the robot lowers the object too deeply inside the container, causing contact with the box boundaries. In other cases the robot reaches the container but performs an imprecise release.

These results suggest that placement precision is more challenging than the initial grasping phase, possibly due to accumulated errors in the trajectory execution or limited representation of the placement task together with the staging area in the training dataset.

6.4.1 Sim-to-Real gap

From a methodological perspective, the dataset acquisition strategy already included several techniques explicitly designed to reduce the sim-to-real mismatch. Domain randomization was applied at episode level (object scattering and orientation, table height perturbation, wrist-camera offset, and randomized initial TCP pose), with the objective of preventing overfitting to a single deterministic geometry.

As previously mentioned, another relevant choice was to prioritize trajectory quality during teleoperation while intentionally preserving a limited amount of imperfect executions and corrective maneuvers. This introduced recovery-like patterns in the training distribution, making the policy less brittle when facing small disturbances in real execution. In addition, maintaining the same acquisition frequency (20 Hz) between demonstration recording and deployment contributed to reducing temporal distribution shift in action generation.

Additional fine-tuning attempts were also carried out using a small real-world dataset, collected with a teleoperation workflow analogous to the one used in simulation. In this case, however, no consistent improvement was observed in the final evaluation. A plausible reason is the limited quality of several recorded trajectories: keyboard-based teleoperation through ROS 2 proved to be relatively inefficient in this setting, often producing less smooth and less precise demonstrations than required for effective policy refinement.

6.4.2 Failure patterns and common shortcomings

In summary, some limitations emerge from this sim-to-real evaluation. The most frequent failure modes include imprecise grasp positioning, incorrect object selection, and inaccurate placement inside containers (for *pick-and-place* task). These issues suggest that the policy is still sensitive to small perception errors or misalignments and that the dataset employed may not sufficiently cover the variability encountered in the real environment.

Despite these issues, achieving a success rate close to 60% in real-world trials, training on less than a 100 demonstrations, proves that the model has learned an actual functional manipulation strategy. Future improvements could be inspired by this same approach, extending randomization factors and increasing the amount of high-quality demonstrations in edge-case configurations.

Chapter 7

Conclusions

This thesis investigated whether a recent Vision-Language-Action model can be effectively adapted to an industrial manipulation scenario through a complete sim-to-real pipeline. The core objective was to integrate Isaac-GR00T (v1.5) on a UR10e manipulator equipped with a Robotiq gripper, and to assess its practical reliability on language-conditioned *lift* and *pick-and-place* tasks.

7.1 Summary of contributions

Throughout this thesis work a summary of contributions can be identified:

1. **A structured technical framing of the VLA landscape.** A background analysis of architectures, training paradigms, benchmarks, and transfer mechanisms was provided, with specific focus on cross-embodiment adaptation constraints. This contextualization motivated the choice of GR00T as a suitable foundation model for industrial-oriented experiments.

2. **The definition of an end-to-end integration workflow in Isaac Sim.** A custom embodiment interface (`data_config`) was designed for UR10e + gripper, aligning multimodal observations (dual RGB cameras, joint state, language instruction) and control outputs with the GR00T action space. In addition, an inference service based on API endpoints was connected to a

simulation control loop with interpolation and denoising, providing stable policy execution at appropriate control frequency.

3. **The development of a dedicated data collection and fine-tuning pipeline.** A LeRobot-compatible dataset structure, extended with GR00T-specific modality configuration, was used to record teleoperated demonstrations in simulation. The acquisition process included metadata management, synchronized multimodal recording, and scene randomization strategies to increase variability. Fine-tuning was then performed by adapting the embodiment-specific policy components while preserving the pretrained language-vision backbone.

4. **Real deployment architecture.** A ROS 2-based communication stack was implemented to connect perception, control, and inference modules on the real platform, ensuring closed-loop interaction with the UR10e and both cameras. This enabled a coherent transition from simulation validation to real execution in the same task setting.

5. A comprehensive **evaluation campaign** was carried out in both simulation and reality, using the same testing protocol over 34 trials per domain. Results show an overall success rate of **83.82%** in simulation and **60.29%** in real-world tests. These outcomes confirm that the policy learns meaningful manipulation behaviors and that transfer to the physical setup is feasible, while also highlighting a non-negligible sim-to-real degradation.

Qualitative analysis of failures reveals that the most recurring issues are grasp pose inaccuracies, occasional object-instruction mismatch, and limited precision in the placement phase. At the same time, robustness to alternative linguistic formulations remained generally strong, suggesting that language understanding is not the main bottleneck in this configuration. Overall, the work demonstrates that adapting a foundation VLA to an industrial arm is technically viable with moderate data, but reliable deployment still depends on improving perceptual grounding, trajectory precision, and dataset coverage.

7.2 Future work

Future developments should first target data scale and diversity. Increasing the number of demonstrations, balancing task distributions, and extending object categories would improve generalization and reduce ambiguity in object selection. In particular, collecting additional episodes with varied clutter conditions, difficult grasp geometries, and richer placement constraints would likely address the most frequent failure patterns observed in this thesis.

Moreover, action execution can be refined by introducing more precise control strategies, for instance in the placement phase. Hybrid control approaches combining policy outputs with local visual servoing, collision-aware corrective behaviors, or force-informed release checks may improve final positioning accuracy. This is particularly relevant for industrial scenarios where tolerances, repeatability, and safety margins are strict.

Sim-to-real transfer can be further improved by broadening randomization dimensions and calibration procedures. Beyond object pose variation, future datasets should include wider perturbations in lighting, camera noise, material properties, contact parameters, and timing/latency effects. Coupling this with small amounts of targeted real-world post-training data could significantly reduce residual transfer gap.

In addition, exploiting modern synthetic generation pipelines, using powerful instruments such as NVIDIA Cosmos foundation world model, can effectively address the existing limitation about the scarcity of embodiment-specific data episodes, potentially unlocking a larger and faster applicability of world-acting policies such as those promoted by VLAs.

Another important direction is reliability-aware deployment. Estimating policy uncertainty, adding confidence-based safety gates, and integrating fallback reactive routines would make the system more robust in edge cases. Such mechanisms are essential to move from proof-of-concept demonstrations toward production-ready industrial applications, such as assembling and kitting workflows.

Finally, a broader benchmarking framework should be considered. Future evaluations could include long-horizon multi-step tasks, comparison against classical modular pipelines, and additional metrics such as completion time, collision rate and recovery capability.

In conclusion, this thesis provides concrete insights for bringing modern VLA models into industrial manipulation contexts. The proposed pipeline demonstrates both the potential and the current limitations of sim-to-real adaptation, and it opens interesting research directions toward more reliable, scalable, and semantically grounded robotic autonomy.

Bibliography

[1] Kento Kawaharazuka, Jihoon Oh, Jun Yamada, Ingmar Posner, and Yuke Zhu. «Vision-Language-Action Models for Robotics: A Review Towards Real-World Applications». In: *IEEE Access* 13 (2025), pp. 162467–162504. ISSN: 2169-3536. DOI: 10.1109/access.2025.3609980. URL: http://dx.doi.org/10.1109/ACCESS.2025.3609980 (cit. on pp. 4, 5, 7, 12).

[2] Dapeng Zhang, Jing Sun, Chenghui Hu, Xiaoyan Wu, Zhenlong Yuan, Rui Zhou, Fei Shen, and Qingguo Zhou. *Pure Vision Language Action (VLA) Models: A Comprehensive Survey*. 2025. arXiv: 2509.19012 [cs.RO]. URL: https://arxiv.org/abs/2509.19012 (cit. on pp. 5, 8).

[3] Rui Shao, Wei Li, Lingsen Zhang, Renshan Zhang, Zhiyang Liu, Ran Chen, and Liqiang Nie. «Large VLM-based Vision-Language-Action Models for Robotic Manipulation: A Survey». In: (2025). arXiv: 2508.13073 [cs.RO]. URL: https://arxiv.org/abs/2508.13073 (cit. on pp. 5, 8, 12).

[4] Anthony Brohan et al. «RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control». In: *arXiv preprint arXiv:2307.15818*. 2023 (cit. on pp. 5–10, 13).

[5] Yunfan Jiang et al. «VIMA: General Robot Manipulation with Multimodal Prompts». In: (2022). arXiv: 2210.03094 [cs.RO]. URL: https://arxiv.org/abs/2210.03094 (cit. on pp. 5–7, 9–13).

[6] Physical Intelligence. *π_0: A Vision-Language-Action Flow Model for General Robot Control*. White paper / technical report. Accessed: 2026-02-19. 2024. URL: https://www.pi.website/download/pi0.pdf (cit. on pp. 5–7, 9–11).

[7] Junjie Wen, Minjie Zhu, Jiaming Liu, Zhiyuan Liu, Yicun Yang, Linfeng Zhang, Shanghang Zhang, Yichen Zhu, and Yi Xu. *dVLA: Diffusion Vision-Language-Action Model with Multimodal Chain-of-Thought*. 2025. arXiv: 2509.25681 [cs.RO]. URL: https://arxiv.org/abs/2509.25681 (cit. on pp. 5–8, 11).

[8] Anthony Brohan et al. «RT-1: Robotics Transformer for Real-World Control at Scale». In: (2022). arXiv: 2212.06817 [cs.RO]. URL: https://arxiv.org/abs/2212.06817 (cit. on pp. 5, 6, 8–10, 13).

[9] Open X-Embodiment Collaboration, Abby O'Neill, Karl Pertsch, et al. «Open X-Embodiment: Robotic Learning Datasets and RT-X Models». In: (2023). arXiv: 2310.08864 [cs.RO]. URL: https://arxiv.org/abs/2310.08864 (cit. on pp. 5, 6, 8, 10, 12, 13, 28).

[10] Homer Walke et al. «BridgeData V2: A Dataset for Robot Learning at Scale». In: (2023). arXiv: 2308.12952 [cs.RO]. URL: https://arxiv.org/abs/2308.12952 (cit. on pp. 5, 6, 12).

[11] Moo Jin Kim et al. «OpenVLA: An Open-Source Vision-Language-Action Model». In: (2024). arXiv: 2406.09246 [cs.RO]. URL: https://arxiv.org/abs/2406.09246 (cit. on pp. 5, 6, 8–10, 13).

[12] HuggingFace. *nvidia/GR00T-N1.5-3B*. Sept. 17, 2025. URL: https://huggingface.co/nvidia/GR00T-N1.5-3B (visited on 02/27/2026) (cit. on pp. 6, 8, 10, 11, 15, 27).

[13] Oier Mees, Lukas Hermann, Erick Rosete-Beas, and Wolfram Burgard. «CALVIN: A Benchmark for Language-Conditioned Policy Learning for Long-Horizon Robot Manipulation Tasks». In: (2021). arXiv: 2112.03227 [cs.RO]. URL: https://arxiv.org/abs/2112.03227 (cit. on pp. 7, 12, 13).

[14] Cheng-Hao Tu et al. *SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics*. 2024. arXiv: 2405.19726 [cs.RO] (cit. on pp. 9, 11).

[15] NVIDIA et al. «GR00T N1: An Open Foundation Model for Generalist Humanoid Robots». In: *ArXiv Preprint*. Mar. 2025. arXiv: 2503.14734 (cit. on pp. 9, 11, 13).

[16] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. «RLBench: The Robot Learning Benchmark & Learning Environment». In: (2019). arXiv: 1909.12271 [cs.RO]. URL: https://arxiv.org/abs/1909.12271 (cit. on p. 11).

[17] Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. «LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning». In: (2023). arXiv: 2306.03310 [cs.AI]. URL: https://arxiv.org/abs/2306.03310 (cit. on p. 12).

[18] Tianhe Yu et al. «Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning». In: (2019). arXiv: 1910.10897 [cs.LG]. URL: https://arxiv.org/abs/1910.10897 (cit. on p. 12).

[19] Tongzhou Mu, Zhan Ling, Fanbo Xiang, Derek Yang, Xuanlin Li, Stone Tao, Zhiao Huang, Zhiwei Jia, and Hao Su. «ManiSkill: Generalizable Manipulation Skill Benchmark with Large-Scale Demonstrations». In: (2021). arXiv: 2107.14483 [cs.LG]. URL: https://arxiv.org/abs/2107.14483 (cit. on p. 12).

[20] Pranav Guruprasad, Yangyue Wang, Sudipta Chowdhury, Harshvardhan Sikka, and Paul Pu Liang. «Benchmarking Vision, Language, & Action Models in Procedurally Generated, Open Ended Action Environments». In: (2025). arXiv: 2505.05540 [cs.CV]. URL: https://arxiv.org/abs/2505.05540 (cit. on pp. 12, 13).

[21] NVIDIA Research. *GR00T N1.5*. June 11, 2025. URL: https://research.nvidia.com/labs/gear/gr00t-n1_5 (visited on 02/23/2026) (cit. on pp. 13–15).

[22] Zhiqi Li et al. «Eagle 2: Building Post-Training Data Strategies from Scratch for Frontier Vision-Language Models». In: *arXiv:2501.14818* (2025) (cit. on p. 13).

[23] NVIDIA Research. *GR00T N1.5*. Dec. 15, 2025. URL: https://github.com/NVIDIA/Isaac-GR00T/releases/tag/n1.5-release (visited on 02/23/2026) (cit. on pp. 16, 27, 28).

[24] Universal Robots. *UR10e Collaborative Robot – Technical Specifications*. 2024. URL: https://www.universal-robots.com/media/1807466/ur10e_e-series_datasheets_web.pdf (visited on 02/23/2026) (cit. on p. 17).

[25] Universal Robots. *UR10e User Manual en Global.pdf*. 2023. URL: https://myurhelpresources.blob.core.windows.net/resources/PDF/SW_5_13/UR10e_User_Manual_en_Global.pdf (visited on 02/24/2026) (cit. on p. 18).

[26] Universal Robots. *DH Parameters for calculations of kinematics and dynamics*. Jan. 11, 2026. URL: https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics (visited on 02/24/2026) (cit. on p. 18).

[27] Robotiq. *2F-140 Adaptive Gripper Documentation*. 2024. URL: https://robotiq.com/products/adaptive-grippers#Two-Finger-Gripper (visited on 02/23/2026) (cit. on p. 19).

[28] Realsense. *Depth Camera D435i*. 2024. URL: https://www.realsenseai.com/products/depth-camera-d435i/ (visited on 02/23/2026) (cit. on p. 19).

[29] Steve Macenski, Tully Foote, Brian Gerkey, Charles Lalancette, and William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022) (cit. on p. 20).

[30] Open Robotics. *ROS 2 Documentation*. 2024. URL: https://docs.ros.org/en/jazzy/index.html (visited on 02/23/2026) (cit. on p. 20).

[31] Open Robotics. *URDF: Unified Robot Description Format*. 2024. URL: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/URDF/URDF-Main.html (visited on 02/23/2026) (cit. on p. 20).

[32] Docker Inc. *Docker Documentation*. 2024. URL: https://docs.docker.com (visited on 02/23/2026) (cit. on p. 21).

[33] Dirk Merkel. «Docker: Lightweight Linux Containers for Consistent Development and Deployment». In: *Linux Journal* 239 (2014) (cit. on p. 21).

[34] NVIDIA. *NVIDIA Isaac Sim Documentation*. 2024. URL: https://docs.isaacsim.omniverse.nvidia.com/5.0.0/index.html (visited on 02/23/2026) (cit. on p. 21).

[35] NVIDIA Developer. *OpenUSD for Developers*. 2024. URL: https://developer.nvidia.com/openusd (visited on 02/23/2026) (cit. on p. 22).

[36] Universal Robots. *Universal Robots ROS 2 Driver*. 2024. URL: https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver (visited on 02/23/2026) (cit. on p. 23).

[37] Realsense. *Intel RealSense™ Camera 400 Series Product Family Datasheet*. Dec. 2025. URL: https://realsenseai.com/wp-content/uploads/2025/12/RealSense-D400-Series-Datasheet-Dec-2025.pdf (visited on 02/26/2026) (cit. on p. 23).

[38] Jianfeng Xiang et al. «Native and Compact Structured Latents for 3D Generation». In: *Tech report* (2025) (cit. on p. 24).

[39] Alexander Khazatsky et al. «DROID: A Large-Scale In-The-Wild Robot Manipulation Dataset». In: (2024) (cit. on p. 28).

[40] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703 (cit. on p. 28).

[41] PyTorch. *PyTorch 2.5 documentation*. URL: https://docs.pytorch.org/docs/2.5/index.html (visited on 02/27/2026) (cit. on p. 28).

[42] NVIDIA-ISAAC-ROS. *Universal Robots ROS2 Driver*. 2025. URL: https://github.com/NVIDIA-ISAAC-ROS/Universal_Robots_ROS2_Driver (visited on 03/05/2026) (cit. on p. 49).

[43] NVIDIA-ISAAC-ROS. *ROS 2 Robotiq gripper*. 2025. URL: https://github.com/NVIDIA-ISAAC-ROS/ros2_robotiq_gripper (visited on 03/05/2026) (cit. on p. 49).

[44] NVIDIA-ISAAC-ROS. *ROS2 Wrapper for Intel® RealSense™ Devices*. 2025. URL: https://github.com/NVIDIA-ISAAC-ROS/realsense-ros.git (visited on 03/05/2026) (cit. on p. 49).

Appendix A

Workstation specifications

A.1 Workload Allocation

This appendix reports the technical specifications of the workstations used for the three main activities in this thesis:

- **Simulation and dataset generation:** Isaac Sim scene execution, teleoperated episode collection, and data export in LeRobot/GR00T format were run on a machine equipped with an high-performance GPU oriented specifically to graphics and rendering tasks (table A.1).
- **Model fine-tuning:** Isaac-GR00T post-training on the collected dataset, using the training settings discussed in Chapter 5 was executed instead on a different machine, more appropriate for typical AI workloads.
- **Inference and control runtime:** real-time policy inference service, ROS 2 nodes, camera acquisition, and command orchestration for UR10e deployment were also run on the same workstation used for fine-tuning (table A.2).

A.2 Software Environment

The software stack used for both simulation, training, and deployment is summarized below:

Operating system:	Ubuntu 24.04 LTS
NVIDIA driver:	570.172.08
CUDA / cuDNN:	13.0
Isaac Sim:	5.0.0
Docker:	29.3.0
ROS 2:	Jazzy
PyTorch:	3.10
GR00T policy:	Isaac-GR00T v1.5 (fine-tuned policy)

A.3 Hardware Configuration

Component	Specification
CPU	AMD Ryzen 9 9950X (16 cores / 32 threads; 5756 MHz max clock frequency)
GPU	NVIDIA RTX A6000 (Ada Lovelace architecture; 48 GiB VRAM)
System memory (RAM)	64 GiB
Storage	1.8 TB NVMe SSD; 3.6 TB SATA SSD

Table A.1: Hardware specifications of the workstation used for running Isaac Sim environment.

Component	Specification
CPU	Intel Core Ultra 9 285K (24 cores / 32 threads; 5756 MHz max clock frequency)
GPU	NVIDIA RTX 5090 (Blackwell architecture; 32 GiB VRAM)
System memory (RAM)	64 GiB
Storage	3.6 TB NVMe SSD
Network interface	WiFi 7 + Ethernet 10 Gbit/s

Table A.2: Hardware specifications of the workstation used for running policy fine-tuning and inference in real-world deployment.