UniROS: ROS-Based Reinforcement Learning Across Simulated and Real-World Robotics

Jayasekara Kapukotuwa , Brian Lee , Declan Devine , Yuansong Qiao *

Posted Date: 1 July 2025

doi: 10.20944/preprints202507.0012.v1

Keywords: Reinforcement learning (RL); Robot Operating System (ROS); Real-time Robotics; Sim-to-Real Transfer; Concurrent RL Environments

 doi:10.20944/preprints202507.0012.v1

Article

UniROS: ROS-Based Reinforcement Learning Across Simulated and Real-World Robotics

Jayasekara Kapukotuwa [1,*], Brian Lee [1], Declan Devine [2] and Yuansong Qiao [1,*]

[1] Software Research Institute, Technological University of the Shannon: Midlands Midwest, Athlone, Co Westmeath, Ireland

[2] PRISM Research Institute, Technological University of the Shannon: Midlands Midwest, Athlone, Co Westmeath, Ireland

* Correspondence: j.kapukotuwa@research.ait.ie (J.K.); yuansong.qiao@tus.ie (Y.Q)

Highlights

What are the main findings?

- Developed UniROS, a unified ROS-based reinforcement learning framework supporting real-time learning across both simulated and physical robots.
- Demonstrated asynchronous, concurrent control of multiple real and simulated robots using UniROS, effectively reducing latency and improving scalability.

What is the implication of the main finding?

- Enables more efficient and realistic training of multiple robotic agents in the real world, addressing the limitations of existing sequential, single-robot frameworks.
- Facilitates learning across different robots and network conditions, accelerating the deployment of real-world reinforcement learning systems.

Abstract

Reinforcement Learning (RL) enables robots to learn and improve from data without being explicitly programmed. It is well-suited for tackling complex and diverse robotic tasks, offering adaptive solutions without relying on traditional, hand-designed approaches. However, RL solutions in robotics have often been confined to simulations, with challenges in transferring the learned knowledge or learning directly in the real world due to latency issues, lack of standardized structure, and the complexity of integrating with real robot platforms. Furthermore, existing robotic RL frameworks typically support sequential, turn-based agent-environment interactions, which fail to represent the continuous, dynamic nature of real-time robotics. This paper addresses this gap by proposing UniROS, a novel Robot Operating System (ROS)-based RL framework explicitly designed for real-time multi-robot/task applications. UniROS introduces a ROS-centric implementation strategy for creating RL environments that support asynchronous, concurrent processing, which is pivotal in reducing the latency between agent-environment interactions. This study validates UniROS through practical robotic scenarios, including direct real-world learning, sim-to-real policy transfer, and concurrent multi-robot/task learning. The proposed framework, including all the examples and supporting packages developed in this study, is publicly available on GitHub, inviting wider use and exploration in the field. https://github.com/sri-tus-ie/UniROS

Keywords: reinforcement learning (RL); Robot Operating System (ROS); real-time robotics; sim-to-real transfer; concurrent RL environments

1. Introduction

Robot-based reinforcement learning [1,2] usually depends on simulation models for learning robotic applications and transferring the learned knowledge to real-world robots. This is a critical

 doi:10.20944/preprints202507.0012.v1

stage where most simulation frameworks face challenges in effectively showcasing how to transfer the learned behaviors from simulation models to real robots. One of the main challenges is that the currently available robotics simulators cannot fully capture the exact varying dynamics and intrinsic parameters of the real world. Therefore, the agents trained in the simulation models cannot typically be directly generalized to the real world due to the domain gap (reality gap) [2] introduced by the simulators' discrepancies and inaccuracies. To overcome this issue, the experimenters must perform additional steps to the learning task, which require applying Sim-to-real [3] or domain adaptation [4] techniques to transfer the learned policies from simulation to the real world.

Even after addressing these concerns, one key challenge in real-world robotic learning is managing sensorimotor data in the context of RL real-time scenarios [5]. In robotic RL, *'real-time'* refers to the environment's ability to operate at a pace where the robot's decision-making and execution of actions must occur within a specific time frame. This rapid pace is essential for the robot to interact with its environment effectively, ensuring that the processing of sensory data and the execution of actuator responses are both timely and accurate. This aspect is particularly critical when creating simulation-based learning tasks to transfer the learning to real-world robots. Currently, in most simulation-based learning tasks, computations related to environment-agent interaction are typically performed sequentially. Therefore, to comply with the Markov Decision Process (MDP) architecture [6], which assumes no delay between observing and acting, most simulation frameworks pause the simulation to construct the observations, reward, and other computations. In contrast, time advances continuously between agent and environment-related interactions in the real world. Hence, the learning is typically done with delayed sensorimotor information, potentially impacting the synchronization and effectiveness of the agent's learning process in real-world settings [7].

Therefore, these turn-based systems do not mirror the continuous and dynamic nature of real-world interactions and can lead to a mismatch in the timing of sensorimotor events compared to real-world situations. These issues stem from the agent receiving outdated information about the state of the environment and the robot not receiving the proper actuation commands to execute the task. While several solutions exist in the literature, they predominantly focus on single-robot scenarios or systems comprising robots from the same manufacturer, limiting their applicability in heterogeneous multi-robot settings [8]. Furthermore, they often overlook the computational challenges inherent in scaling to multiple robots, particularly the CPU bottlenecks that can arise from processing data from various sensors, such as vision systems, which may require CPU-intensive preprocessing operations [9,10].

Another challenge in this process is the difference in the programming languages of simulation frameworks from those of real robots. Most current simulation frameworks used in RL are commonly implemented in languages like Python, C#, or C++. However, real robots typically have proprietary programming languages such as RAPID, Karel, and URScript or may utilize the Robot Operating System (ROS) for communication and control. Therefore, it is not possible to transfer the learned knowledge directly without recreating the RL environment in the recommended robot programming language to communicate with the physical hardware [7]. Furthermore, this challenge also applies when learning needs to happen directly in the real world without relying on knowledge transferred from a digital model. These include cases such as dealing with liquids, soft fabrics, or granular materials, where the physical properties are challenging to model precisely in simulations [11]. In these scenarios, the experimenters must establish a communication interface with the physical robots to enable the agent to interact with the real world directly.

Fortunately, utilizing the Robot Operating System (ROS) presents a promising solution to some of the earlier challenges. This is due to ROS being widely acknowledged as the standard for programming real robots, as well as the massive support it receives from manufacturers and the robotics community. This makes it an ideal platform for constructing learning tasks applicable in simulations and real-world settings. Currently, numerous simulation frameworks are available for creating RL environments with ROS, with most prioritizing simulation over real-world applications.

 doi:10.20944/preprints202507.0012.v1

A fundamental limitation of these simulation frameworks, such as OpenAI_ROS[1] , gym-gazebo [12], ros-gazebo-gym[2] , and FRobs_RL [13], lies in their inability to support creating real-time RL simulation environments due to their use of turn-based learning approaches. Therefore, the full potential of ROS for setting up learning tasks that can easily transfer the learning to the real world is not utilized correctly. Furthermore, with the current offerings, ROS lacks Python bindings for some crucial system-level features needed to create RL environments, such as launching multiple roscores, nodes, and launch files, which are currently confined to manual configurations (Command Line Interface – CLI approaches). Moreover, the full potential of ROS in creating real-time RL environments that achieve precise time synchronization, which is essential for aligning the sequence and timing of sensor data acquisition, decision-making processes, and actuator responses, thereby reducing latency in agent-environment interactions, has not been thoroughly studied yet. Addressing these gaps in ROS could further streamline the development of effective and efficient RL environments for robotics.

Therefore, this study addresses the question of "*how to set up ROS-based learning tasks to learn across simulation and real robots*". This approach proposes a comprehensive framework designed for creating RL environments that cater to both simulation and real-world applications. This includes adding support for ROS-based concurrent environment creation, which is a requirement for multi-robot/task learning techniques such as multi-task [14] and meta-learning [15], enabling it to simultaneously handle learning across multiple simulated and/or real RL environments. Furthermore, the study explores how this framework can be utilized to create real-time RL environments, leveraging a ROS-centric environment implementation strategy that can bridge the gap between transferring the learning from simulation to the real world. This aspect is vital for ensuring reduced latency in agent-environment interactions, which is crucial for the success of real-time tasks.

Furthermore, the study introduces benchmark learning tasks to evaluate and demonstrate some of the use cases of the proposed approach. These learning tasks are built around the ReactorX200 (Rx200) robot by Trossen Robotics and the NED2 robot by Niryo and are used to explain the design choices. This study also lays the groundwork for multi-robot/task learning techniques, allowing for the sampling of experiences from multiple concurrent environments, whether they are simulations, real, or a combination of both.

Summary of Contributions:

- **Unified RL Framework:** Development of a comprehensive, ROS-based framework (UniROS) for creating reinforcement learning environments that work seamlessly across simulation and real-world settings.
- **Concurrent Env Learning Support:** Enhancement of the framework to support vectorized [16], multi-robot/task learning techniques, enabling efficient learning across multiple environments.
- **Real-Time Capabilities:** Introduction of a ROS-centric implementation strategy for real-time RL environments, ensuring reduced latency and synchronized agent-environment interactions.
- **Benchmarking and Evaluation:** Empirical demonstration through benchmark learning tasks, addressing these challenges using the proposed framework, using three distinct scenarios.

2. Background

2.1. Formulation of Reinforcement Learning Tasks for Robotics

A reinforcement learning task comprises two main essential components: an "*Agent*" and an "*Environment*", where they interact with each other as modeled by the *Markov decision process* (MDP), as illustrated in **Figure 1**. In an MDP, the role of the agent is to interact with the environment at discrete time steps $t = 1, 2, 3, \ldots$, where at each time step t, the environment receives an action A_t

[1] http://wiki.ros.org/openai_ros

[2] https://github.com/rickstaa/ros-gazebo-gym

 doi:10.20944/preprints202507.0012.v1

and returns the state of the environment $S_t \in \mathcal{S}$ and a scalar feedback reward $R_t \in R$ back to the agent. The agent follows a stochastic policy π characterized by a probability distribution $\pi(a|s) = \mathcal{P}[A_t = a|S_t = s]$ to choose action $A_t \in \mathcal{A}$. The execution of the action pushes the environment into a new state S_{t+1} and produces a new scalar reward R_{t+1} at next time step $t+1$ according to a state transition probability $\mathcal{P}^a_{ss'} = p(s', r|s, a) = \mathcal{P}[S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a]$. The primary goal of the agent is typically to find the optimal policy that maximizes the total discounted reward, also defined as the expected return $G_t = \sum_{k=t}^{\infty} \gamma^{(k-t)} R_{k+1}$, where $\gamma \in [0,1]$ is the discount factor.

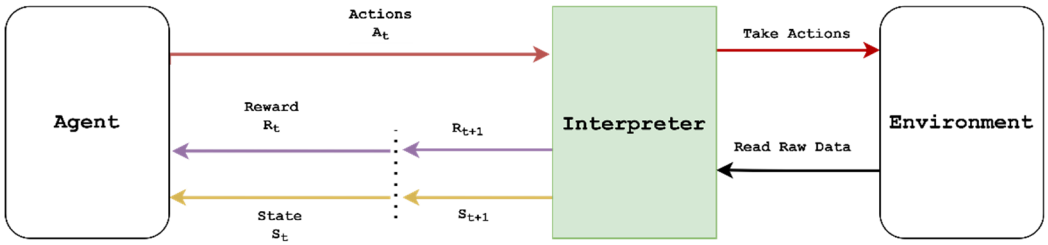


Figure 1. Basic reinforcement learning diagram depicting the critical components, including the agent, environment, and interpreter. It visualizes the flow of information from each component, providing observations (state) and rewards for the agent's decision-making process.

However, the agent typically cannot directly interact with the environment and requires the assistance of an interpreter to mediate between them. The interpreter's role is to transform raw environmental states into a format compatible with the agent and to process the actions received from the agent into commands that can affect the elements in the environment. Therefore, from the agent's perspective, the combination of interpreter and environment constitutes the effective "RL environment," replacing the abstract "environment" in the classical MDP formulation. In robotics, this process involves mapping observations and actions to their real-world or simulated counterparts, ensuring the RL agent sends actuator commands to control the robots and receives accurate sensor readings as observations from the real world or simulation.

In most robot-based learning tasks, the state of the environment is not fully observable, and the agent relies on real-time sensor data to partially observe the environment using the observation vector o_t. This observation space generally contains continuous values as it usually holds perception and sensory data. Therefore, most robot-based learning tasks incorporate deep learning techniques as function approximators in conjunction with reinforcement learning (also known as deep reinforcement learning, or DRL) to effectively handle continuous state spaces. In traditional RL, these techniques and methods typically center around learning the optimal policy through evaluating action values, as in the Deep Q-Network (DQN) [17], or directly parameterizing the policy with neural networks and optimizing them, as in the Twin Delayed Deep Deterministic Policy Gradient (TD3) [18] algorithms. Currently, various third-party frameworks provide libraries for state-of-the-art DRL algorithms. These frameworks provide clean, robust code with simple and user-friendly APIs (Application Programming Interfaces), enabling users to experiment with and monitor the learning of various DRL algorithms. Therefore, if users are not benchmarking their custom algorithms, they can employ the functionality of these packages, such as Stable Baselines3 (SB3) [19], Tianshou [20], Tensorforce [21], CleanRL [22], or RLlib [23], to effectively train robotic agents.

2.2. Applying Reinforcement Learning to Real-World Robots

Training real robots directly using reinforcement learning from scratch can be challenging, and most of the time, it is not actively considered due to the random nature of the exploration of the RL algorithms [24]. Initially, the agent must explore the environment thoroughly to collect data, which often requires a certain level of randomness in the agent's actions. However, as the learning progresses, it is desirable for the agent's actions to become less random and more consistent with each iteration. [6] This is vital to ensure that the agent does not exhibit unexpected or unpredictable

 doi:10.20944/preprints202507.0012.v1

behavior when deployed in the real world. Nevertheless, with careful safety considerations, handpicked learning parameters, and a well-defined observation space, action space, and reward structure for the task, it is possible to train a robot in the real world directly [25,26]. Furthermore, incorporating techniques such as Curriculum Learning [27], where the agent maneuvers through complex tasks by breaking down tasks into simpler subtasks and gradually increasing the task's difficulty during learning, can help train robots directly without a simulation environment [28].

While it is possible to learn directly with real robots, there are several constraints in applying RL to real-world learning tasks. One of the significant hurdles is ensuring the safety of the robot, its platform, and the surrounding environment. Due to the arbitrary nature of the initial learning stage, where the agent attempts to learn more about the environment through random actions, the robot can potentially cause damage to itself and nearby expensive equipment. Another challenge is sample inefficiency in real-world RL environments. Most RL-based algorithms require a large number of samples to find the optimal policy. However, unlike in a digital simulation model, it is difficult for real-world RL environments to provide a fast and nearly unlimited number of samples for the agent to learn. It becomes especially apparent in episodic tasks that require environment resetting at the end of each episode. This process is relatively straightforward in a digital model where the environment can be reset via the physics simulator's API and a function that programmatically sets the initial conditions of the environment. However, in the real world, it is a tedious task where an operator may often have to constantly monitor and physically rearrange the environment at the end of each training episode. Similarly, even if it is possible to speed up the learning using an environment vectorizing approach for sampling experience with multiple concurrent environment instances, the constant monitoring and associated costs may render it unfeasible for many robotics tasks.

2.3. Use Of Simulation Models For Robotic Reinforcement Learning

Simulation-based robot learning begins by establishing a programmatic interface with a physics simulator [29] to interact with sensors and actuators to read sensory data and execute actuator commands. These types of interfaces enable the creation of RL environments that follow the conventional reinforcement learning architecture illustrated in **Figure 1**. One of the most effective structures for creating RL environments was introduced with the OpenAI Gym [30]. It became the widely adopted standard for RL-based environment creation, and a significant portion of research in the field follows a variation of this standard or builds upon the Gym package for different robotic and physics simulators. Currently, a plethora of RL simulation frameworks [31] are available for robotic task learning and typically provide prebuilt environments or the tools to create custom environments. The prebuilt learning tasks are primarily used for benchmarking new learning algorithms and are extensively employed by researchers to demonstrate the RL approaches in robotics. The popularity of these learning tasks is due to relieving users from many task setup details, such as defining the observation space, action space, and reward architecture. However, custom environment creation is generally more challenging, as it requires users to become familiar with the API of the RL simulation framework and task setup details, which also includes providing a detailed description of the robot and its surrounding environment in a format [32] compatible with the chosen simulator. Once the environment is created, users can utilize third-party RL library packages such as SB3 or custom learning algorithms to find the optimal policy for the custom learning task.

3. Related Work

Most RL-based simulation frameworks for robots built on simulators such as MuJoCo [33], PyBullet [34], and Gazebo [35] prioritize offering accelerated simulations for developing complex robotic behaviors, often with less emphasis on the seamless transition of policies to real-world robots. A recent advancement in this field is the Orbit [36], a framework built upon Nvidia's Isaac Gym [37] to provide a comprehensive modular environment for robot learning with photorealistic scenes. It stands out for its extensive library of benchmarking tasks and the capabilities that potentially ease the policy transfer to physical robots with ROS integration. However, at the current stage, its focus

 doi:10.20944/preprints202507.0012.v1

remains mainly on simulation rather than direct real-world learning. As such, while it provides tools for simulated training and real-world applications, it may not yet serve as a complete solution for real-world robotics learning without additional customization and system integration efforts. Furthermore, the high hardware requirements[3] of Isaac Sim may restrict accessibility for many researchers and roboticists, limiting its widespread adoption at present.

SenseAct [7] is a notable contribution that highlighted the challenges of real-time interactions with the physical world and the importance of sensor-actuator cycles in realistic settings. They have proposed a computational model that utilizes multiprocessing and threading to perform asynchronous computations between the agent and the real environment, aiming to minimize the delay between observing and acting. However, this design is primarily tailored for single-task environments and shows limitations when extended to multi-robot/task research, including learning together with simulation frameworks or concurrently with multiple environments. This limitation partly stems from its architecture, which allocates a single process with separate threads for the agent and the environment. The scalability of this approach, particularly for concurrent learning with multiple RL environments, is hindered by Python's Global Interpreter Lock (GIL)[4], which restricts parallel execution of CPU-intensive tasks. Hence, incorporating multiple RL environment instances within a single process is not computationally efficient, especially when real-time interactions are critical. Furthermore, the difficulty of synchronizing the different processes and establishing communication layers with various robots and sensors from different manufacturers may limit the potential of their proposed approach.

Table 1 provides a comprehensive comparison between UniROS and existing RL frameworks with a focus on ROS integration, real-time capabilities, and multi-robot support. Unlike most prior tools, which are either simulation-centric or designed for single-robot real-world use, UniROS is uniquely positioned to support scalable, low-latency training across both simulation and physical robots concurrently.

Table 1. Comparison of UniROS with related reinforcement learning frameworks for robotic control.

Frameworks	UniROS	OpenAI_ROS	gym-gazebo2	FRobs_RL	ros-gazebo-gym	SenseAct	Orbit
Real-Time Capability	✓	×	Limited	×	×	✓	✓
Supports Real Robots	✓	✓	✓	✓	✓	✓	Partial
Simulation Support	✓	✓	✓	✓	✓	×	✓
Sim-to-Real Support	✓	Partial	Partial	Partial	Partial	×	✓
ROS Integration	✓	✓	✓	✓	✓	×	✓

[3] https://docs.omniverse.nvidia.com/isaacsim/latest/installation/requirements.html#isaac-sim-requirements-isaac-sim-system

[4] https://wiki.python.org/moin/GlobalInterpreterLock

 doi:10.20944/preprints202507.0012.v1

Concurrent Multi-Robot Support	✓	×	×	×	×	×	Partial
Latency Handling / Modeling	✓	×	×	×	×	✓	Partial
Python API for Env Management	✓	×	Manual	×	Partial	×	✓
Actively Maintained	✓	×	×	✓	✓	×	✓
Open Source	✓	✓	✓	✓	✓	✓	Partial

Beyond comprehensive frameworks, several research efforts have tackled specific aspects of bridging simulation and real-world robot learning. There are many domain randomization approaches [38–40] that either dynamically adjust simulation parameters based on real-world data or vary simulation parameters to improve sim-to-real transfer. However, their methods often require extensive manual tuning of randomization ranges and do not address the fundamental timing mismatches between simulation and real-world execution. While other domain adaption approaches [41,42] leverage demonstrations in both simulation and real-world settings to accelerate robot learning, their approach requires separate implementations for each domain. Since these approaches do not provide a unified interface for concurrent learning across simulation and real environments, they highlight the need for more efficient frameworks that can leverage both simulation and real-world data concurrently.

4. Learning Across Simulated and Real-World Robotics Using UniROS

This section provides a high-level overview of the proposed UniROS framework formulation, which facilitates learning across both simulated and real-world domains. The aim is to present a comprehensive overview of the framework's architecture and functionalities, setting the stage for a more detailed examination of its components in subsequent sections.

4.1. Unified Framework Formulation

As illustrated in **Figure 2**, a ROS-based unified framework is proposed, containing two distinct yet interoperable packages to bridge the learning across simulated and real-world RL environments. Our prior work, MultiROS [43], provides simulation environments using ROS and Gazebo as its core. In contrast, the newly introduced RealROS package, detailed in Section **5**, is designed explicitly for real-world learning applications. The intuition behind dividing the framework into two packages is to offer users flexibility. Depending on specific requirements, users can utilize each package independently, focusing solely on either simulated or real-world scenarios or leverage them collectively for comprehensive simulation-to-reality learning tasks. Furthermore, Section **7** presents a ROS-centric RL environment implementation strategy to bridge the learning gap between the two domains. It aligns the conditions and dynamics of the Gazebo simulation more closely with those of the real world to allow for a smoother deployment of policies from simulation to the real world. This environment implementation strategy can also be employed with the RealROS package to develop and deploy robust policies by sampling directly in real-world environments without relying on a simulated approach.

 doi:10.20944/preprints202507.0012.v1

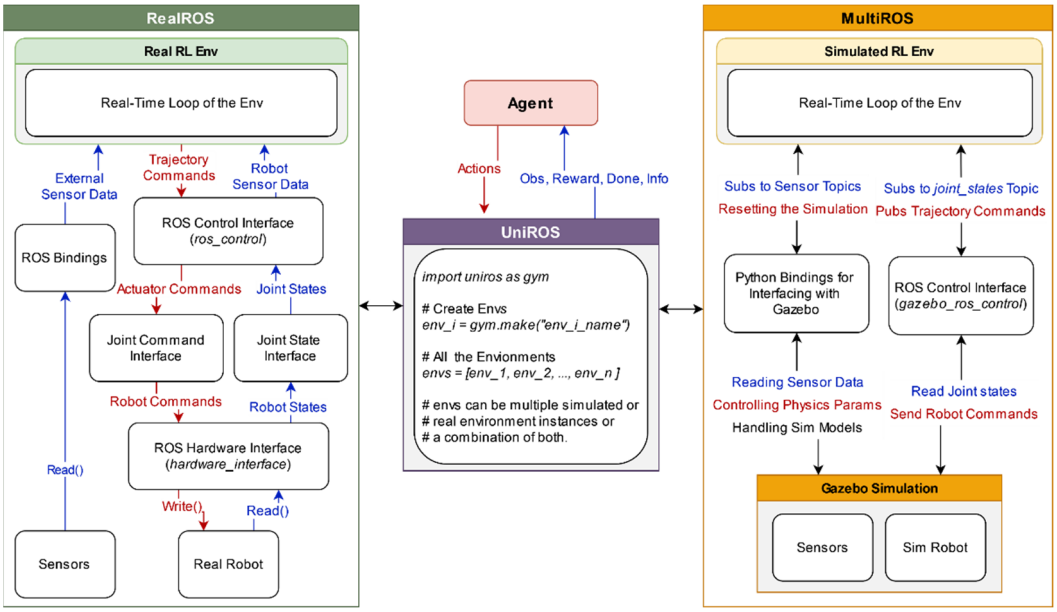


Figure 2. This diagram illustrates the interconnected components of the proposed system, highlighting the dual-package approach with RealROS for real-world robotic applications and MultiROS for simulated environments. The UniROS layer acts as an abstraction to standardize the communication process, facilitating the creation and management of multiple environment instances. The diagram encapsulates the ROS-based data flow and the primary interfaces for robot control, which handles real-time interactions.

4.2. Modularity of the Framework

The architecture of both MultiROS and RealROS leverages a modular design by segmenting the RL environment creation into three distinct classes. The primary focus of this segmentation is to enable flexibility and encourage efficient code reuse during the design of RL environments. These segmented environment classes (*Env*) provide a structure for users to format their code easily and minimize the system integration efforts when transferring policies from the simulation interface to the real world. Therefore, this framework provides an architecture comprising three major components to deliver these services. They are (A) the *Base Env*, the foundational layer of the RL environment, which facilitates the main interface of the standard RL structure for the agent. It inherits its core functionality from the OpenAI Gym and includes static code essential for the basic functioning of any RL task. (B) The *Robot Env*, which is built upon the *Base Env*, outlines the sensors and robots used in the learning task. It encapsulates all the hardware components involved in the task and serves as the bridge that connects the RL environment with the ROS middleware suite. (C) The *Task Env* extends from the *Robot Env* and specifies the structure of the task that the agent needs to learn, which includes learning objectives and task-specific parameters. These modular *Envs* provide a significant degree of flexibility, allowing the users to create multiple *Task Envs* with a single *Robot Env*, as with the Fetch environments[5] (*FetchReach*, *FetchPush*, *FetchSlide*, and *FetchPickAndPlace*) of OpenAI Gym robotics. It is important to note that while each *Task Env* is compatible with its respective *Robot Env* (which may include multiple robots and different sensors), they are not universally interchangeable across different *Robot Env*. Therefore, modularity is for diverse task development within the basis of a specific *Robot Env*. The main reason for this composition is due to the *Base Env* inheriting from OpenAI Gym to retain the compatibility of third-party reinforcement learning frameworks such as Stable Baselines3, RLlib, CleanRL, and others.

[5] https://robotics.farama.org/envs/fetch/index.html

 doi:10.20944/preprints202507.0012.v1

4.3. Role of Concurrent Environments

In modern RL research, there is a growing interest in leveraging knowledge from multiple RL environments instead of training standalone models. One of the advantages of this approach is that it can improve the agent's learning by generalizing knowledge across different scenarios [44]. Furthermore, combining concurrent environments with diverse sampling strategies can effectively speed up the agent's learning process [45]. This leveraging process can expose the agent to learning multiple tasks simultaneously rather than learning each task individually (multi-task learning). It is also similar in meta-learning-based RL applications, where the agent can quickly adapt and acquire new skills in new environments by leveraging prior knowledge and experiences through learning-to-learn approaches. Another advantage of the concurrent environments is scalability, which allows for the simultaneous training of multiple robots in parallel, whether in a vectorizing fashion or for different tasks or domain learning applications [46]. Therefore, creating concurrent environments is crucial for efficiently utilizing computing resources to accelerate learning in real-world applications where multiple robots need to be trained and deployed efficiently. Hence, this study investigated how to create ROS-based concurrent RL environments (sim and real) with seamless simultaneous communication between each environment, as described in Section **6.**

4.4. Python Bindings For ROS

One of the drawbacks of ROS is that some crucial system-level components required for creating RL environments do not have Python or C++ bindings. For example, executing and terminating ROS nodes or launch files requires terminal commands (CLIs). Similarly, running multiple roscore instances concurrently using Python or C++ interfaces and managing communication with each other is currently not natively supported in ROS. These functions also require command-line interfaces, making them undesirable for seamless RL environment creation. Therefore, the UniROS framework contains comprehensive Python-based bindings for ROS, enabling users to utilize the full potential of ROS for RL environment creation. Key features include the ability to launch multiple roscores on distinct or random ports without overlap, manage simultaneous communication between concurrent roscores, run roslaunch and ROS nodes with specific arguments, terminate specific ROS masters, nodes, or roslaunch processes within an environment, retrieve and load YAML files from a package to the ROS Parameter Server, and upload a URDF to the parameter server or process URDF data as a string.

4.5. Additional Supporting Utilities

Apart from the stated ROS bindings, the framework also provides utilities based on Python for users to quickly start creating environments without wasting too much time on ROS implementations. It is also beneficial for users unfamiliar with ROS to create environments without expert knowledge of programming with ROS. These utilities include the **ROS Controllers** module, which allows comprehensive control over ROS controllers to load, unload, start, stop, reset, switch, spawn, and unspawn controllers. The **ROS Markers** module provides methods for initializing and publishing Markers or Marker arrays to visualize vital components of the task, such as the current goal, the pose of the robot's end-effector, and the robot's trajectory. This makes it easy to monitor the status of the environment using Rviz[6], a 3D visualization tool for ROS, to visualize the Markers. The **ROS kinematics** module provides forward (FK) and inverse kinematics (IK) functionality for robot manipulators. This class uses the KDL[7] library to perform kinematics calculations. The **MoveIt** Module offers essential functionalities for managing ROS MoveIt [47] in manipulation tasks, including collision checking, planning, and executing trajectories. Additionally, common **wrappers**

[6] http://wiki.ros.org/rviz

[7] https://www.orocos.org/kdl.html

 doi:10.20944/preprints202507.0012.v1

are included to limit the number of steps in the environment and to normalize actions and observation spaces.

The complete source code of MultiROS[8] and RealROS[9], along with the necessary supporting materials[10], is currently available online to the robotics community as public repositories. They accompany well-documented templates, guides, and examples to provide clear instructions on their installation, configuration, and usage.

5. An In-Depth Look into ROS-Based Reinforcement Learning Package for Real Robots (RealROS)

This section describes the overall system architecture of the RealROS package. It is designed to be compatible with the MultiROS simulation package and is structured to minimize the typical extensive learning curve associated with switching from simulation to real environments. The three core components (*Base Env*, *Robot Env*, and *Task Env*) provide the main API for the experimenters to encapsulate the task, as shown in **Figure 3**. The integration of stated attributes and the structure of the three main components of the framework are described in more detail in the following sections.

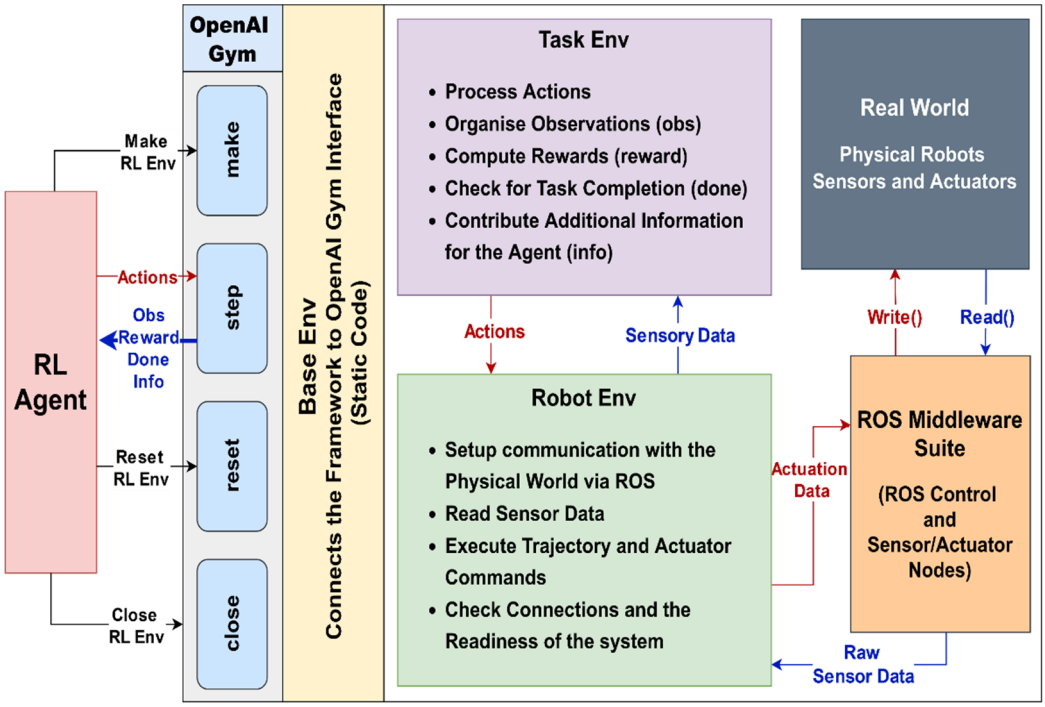


Figure 3. The architecture of the proposed RealROS provides a modular structure that inherits from OpenAI Gym. The *Task Env* inherits from the *Robot Env*, which, in turn, inherits from *Base Env*, and *Base Env* inherits from the OpenAI Gym. An environment instance created with this architecture only exposes the standard RL structure to the agent, allowing it to accept actions, pass them through to the *Robot Env* from *Task Env*, and execute them in the real world. Then, the *Task Env* obtains observations from the methods defined in the *Robot Env*, calculates the Reward and Done flag, and returns them to the agent.

5.1. Base Env

The *Base Env* serves as the superclass that provides the foundation and main interface that specifies the standard RL structure for the RealROS. It provides the necessary infrastructure and other essential components for RL agents to interact with the environment. Here, RealROS offers two

[8] https://github.com/ncbdrck/multiros

[9] https://github.com/ncbdrck/realros

[10] https://github.com/ncbdrck/uniros_support_materials

 doi:10.20944/preprints202507.0012.v1

options for users to create environments. The default "standard" (*RealBaseEnv*) is based on the inheritance of $gym.Env$, and the other "goal-conditioned" (*RealGoalEnv*) is from the $gym.GoalEnv$ of the OpenAI Gym package. This *Base Env* defines the $step$, $reset$, and $close$ methods, which are the main standardized interface of OpenAI gym-based environments. When initializing the *Base Env*, experimenters can pass arguments from the *Robot Env* to perform several optional functions based on user preferences. These include starting communication with real robots, changing the current ROS environment variables, loading ROS controllers for robot control, and setting a seed for the random number generator.

5.2. Robot Env

The *Robot Env* is a crucial component for describing the robots, sensors, and actuators used in the real-world environment using the ROS middleware suite. It inherits from the *Base Env*, allowing it to initialize and access methods defined in the superclass. However, it adds additional functionalities specific to robots and sensors used in the real world. This *Robot Env* encapsulates the following.

Robot Description: One of the primary tasks of the *Robot Env* is to define the physical robot's characteristics, such as its kinematics, dynamics, and available sensors, using a format that ROS can understand. For that purpose, the ROS requires the robot description to be loaded into the ROS parameter server[11] (a shared, multi-variate dictionary that is accessible via ROS network APIs). In ROS, the robot description is in the Universal Robot Description Format (URDF[12]) and contains all the relevant information about the composition of the robot. These include the joint types, joint limits, link lengths, and other intrinsic parameters of the robot. By loading the robot description, ROS packages can utilize it to perform collision detection, inverse kinematics (IK), and forward kinematics (FK) calculations of the robot arm. Suppose multiple robots are needed for the learning task (inside the same RL environment). In that case, each robot's description can be loaded with a unique ROS namespace identifier to distinguish it from the others.

Set up communications with robots: Currently, most robot manufacturers or the ROS community typically provide essential ROS packages specifically designed for commercial robots to establish a communication channel with their robots. One of the prime examples is the ROS-Industrial[13] project, which extends the advanced capabilities of ROS for industrial robots from manufacturers[14] such as ABB, Fanuc, and Universal Robots. These packages include the robot's controller software, which is responsible for managing the robot's motion and maintaining communication with the robot's motors and sensors, as well as with external systems. As for custom robots, the official ROS tutorials[15] provide details on creating a custom URDF file and ROS controllers[16] for interfacing with physical hardware. Once these packages are properly configured, the RL environment can send commands to control the robot's motors and access its sensor readings (including motor encoders and others) through ROS.

Set up communication with the Sensors: Similarly, connecting external sensors using ROS enables the acquisition of data from various sensors such as cameras, lidars, proximity sensors, and force/torque sensors. This can help portray a vivid picture of the real world, enabling the RL agent to perceive the current state of the environment (observations). Currently, most vision-based sensors[17]

---

[11] http://wiki.ros.org/Parameter%20Server

[12] http://wiki.ros.org/urdf

[13] https://rosindustrial.org/

[14] http://wiki.ros.org/Industrial/supported_hardware

[15] http://wiki.ros.org/ROS/Tutorials

[16] http://wiki.ros.org/ros_control/Tutorials

[17] https://rosindustrial.org/3d-camera-survey

 doi:10.20944/preprints202507.0012.v1

and others[18] have ROS packages provided by the manufacturers or the ROS community, allowing users to easily plug and play the devices.

Robot Env specific methods: These methods are for the RL agent to interface with robots and the other equipment in the environment. They can be in the form of planning trajectories, calculating IK, and controlling the joint positions, joint forces, and speed/acceleration of the robot. These methods are then used in the *Task Env* to execute the agents' actions in the real world. Furthermore, ROS's inbuilt utilities and other packages, such as MoveIt, provide functionality for obtaining the transformations of each joint (tf[19]) and the robot end-effector's current pose (3D position and orientation) for portraying the current state of the robot. Combining them with the data acquired from custom methods for interfacing the external sensors enables the *Task Env* to construct observations of the environment.

5.3. Task Env

The *Task Env* serves as the module that outlines the structure and requirements of the task that the RL agent needs to learn. It inherits from the *Robot Env* and builds upon its functionalities to create a real-time loop (Section **7**) that executes actions, constructs observations, calculates the reward, and checks for the termination of the task (optional). Therefore, *Task Env* defines the observation space, action space, goal or objectives, rewards architecture, termination conditions, and other task-specific parameters. These components help with the main *Step* function of the environment to take a step in the real world and send feedback (observations, reward, done, and info) to the agent. Furthermore, knowledge acquired in simulation (MultiROS) can be efficiently transferred to real-world environments by reusing the same code to create the Task Env, thanks to the modular design of the UniROS framework and the compatibility between the MultiROS and RealROS packages, with the caveat that the new Robot Env having the same helper functions.

6. ROS-Based Concurrent Environment Management

This section discusses how the UniROS framework sets up ROS-based concurrent environments to maintain seamless communication with each other. It first deliberates the challenges and provides solutions for initiating multiple ROS-based simulated and real-world environment instances inside the same script. Then, it describes steps for launching multiple Gazebo simulation instances and connecting real-world robots over local and remote connections while ensuring that each environment operates independently and interacts effectively without interference.

6.1. Launching ROS-Based Concurrent Environments

Launching OpenAI Gym-based concurrent environments presents several unique challenges when working with ROS. One challenge is the functionality of the OpenAI Gym interface, which executes all the environment instances within the same process when launched using the standard $gym.make$ function. In contrast, ROS requires each environment instance (sim or real) or process to initialize a unique ROS node[20] to utilize ROS's built-in functions and to communicate with corresponding sensors and robots via the ROS middleware suite. However, ROS typically does not support initializing multiple nodes inside the same Python process due to its fundamental design architecture, which is centered around process isolation for enhanced reliability, modularity, and robustness. Therefore, launching multiple ROS-based environments within the same script is challenging, as it typically requires initializing separate ROS nodes for each instance. While there is

[18] http://wiki.ros.org/Sensors

[19] http://wiki.ros.org/tf

[20] http://wiki.ros.org/Nodes

 doi:10.20944/preprints202507.0012.v1

a *roscpp*[21] (C++ client of ROS) feature for launching multiple nodes inside the same script called *nodelet*[22], the Python client *rospy*[23] does not currently support this function.

One workaround is employing Python multi-threading to launch each RL environment instance, allowing the execution of multiple ROS nodes within the same process. However, this can lead to unexpected behaviors, especially for efficiently utilizing computational resources and parallel processing between each instance. One limiting factor with multi-threading is that Python's Global Interpreter Lock (GIL) prevents multiple threads from executing Python bytecodes simultaneously. This is unfavorable for working with concurrent environments, as each environment instance requires performing CPU-intensive real-time agent-environment interactions, as highlighted in the following section **7.**

Therefore, the UniROS framework utilizes a Python multiprocessing wrapper on top of the OpenAI Gym interface to launch each environment instance as a separate process to address these limitations. Initializing separate processes for each environment instance also launches a separate Python interpreter for each process, allowing them to overcome the bottleneck placed by the Python GIL. This process isolation can significantly contribute to optimized resource allocation for efficiently performing parallel computations with each environment instance. For users, this transition is virtually effortless as they need to utilize UniROS, as in **Figure 2**, instead of the standard *gym.make* function to launch environments. This adaption can effortlessly leverage optimized parallel processing capabilities that are crucial for real-time, CPU-intensive computation tasks in ROS-based concurrent environments.

6.1. Maintaining Communication with Concurrent Environments

In managing concurrent environments, it is essential to have a robust communication infrastructure to ensure seamless interaction between multiple real robots and Gazebo simulation instances. **Figure 4** outlines the methodologies employed in the UniROS framework to facilitate such communication across sim and real environments. In MultiROS-based simulation environments, this was achieved by launching an individual roscore[24] for each environment instance. In ROS, roscore provides necessary services such as naming and registration to a collection (graph) of ROS nodes to communicate with each other. Without a roscore, nodes would be unable to locate each other, exchange messages, or establish parameter settings, making it a critical component in ensuring coordinated operations and data exchange within the ROS ecosystem. Therefore, by utilizing separate roscores, each roscore acts as the server and creates a dedicated communication domain for each node (clients) within the environment instance to exchange data without cross-interference from other concurrent instances. Therefore, for simulations, MultiROS utilizes separate roscores and dynamically assigns the environment variables mentioned in **Figure 4** inside each simulated environment, allowing for precise management of interactions and data flow across concurrent environments. Here, the ROS Python bindings of the UniROS framework were employed for launching roscores with non-overlapping ports and Gazebo simulations, as well as functions for handling ROS-related environment variables to streamline the process without relying on CLI configurations.

As for real-world learning tasks, the RealROS package facilitates robot connections via both local and remote methods. In local setups, the default configuration typically involves using the standard roscore, which uses port *'11311'* as the ROS master[25] by default. In such arrangements, each robot and its corresponding sensors can be assigned a distinct namespace to distinguish themselves from

[21] http://wiki.ros.org/roscpp

[22] http://wiki.ros.org/nodelet

[23] http://wiki.ros.org/rospy

[24] http://wiki.ros.org/roscore

[25] http://wiki.ros.org/rosmaster

 doi:10.20944/preprints202507.0012.v1

the other robots within the same shared roscore. This unique namespace can be used in the respective environment to ensure orderly and isolated interactions despite the shared communication space with other robots. However, for scenarios requiring enhanced communication isolation, each robot can be configured to connect to different roscores, each operating on unique, non-overlapping ROS master ports. Then, setting the appropriate *ROS_MASTER_URI* environment variable in each environment instance adds an extra layer of process isolation to effectively eliminate the potential for cross-interference typically inherited with namespace-based setups.

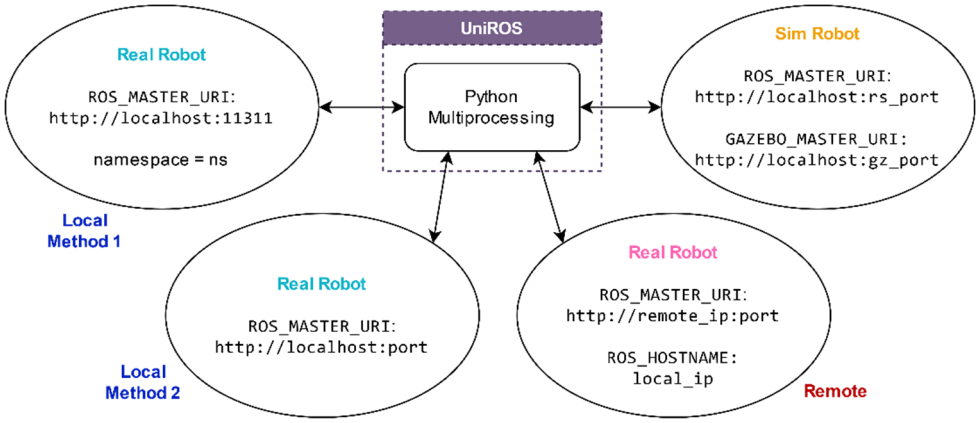


Figure 4. Communication Schemes of the UniROS Framework for Concurrent ROS-Based Environments. The diagram presents (1) two methods for local connections, utilizing either namespaces or separate roscores for enhanced process isolation; (2) a configuration employing ROS multi-device mode for robust remote communication; and (3) a configuration for Gazebo-based simulations that enables concurrent environments.

In remote configurations where robots are connected to the same network, robots can be connected in ROS multi-device mode, where the local PC functions as a client capable of reading and writing data to the remote server (robot) that runs the master roscore. This setup eliminates the need for Secure Shell (SSH) approaches and allows all the processing required for learning to be conducted on the local PC. This approach is particularly beneficial for robots created for research where the robot's ROS controller is based on less powerful edge devices such as the Raspberry Pi, Intel NUC, and Nvidia Jetson. This ROS multi-device mode can be configured by setting the environment variables on both the robot and the local PC, as depicted in **Figure 4**. This setup allows for multiple concurrent environments, each connecting to a remote robot by setting the ROS environment variables utilizing the Python ROS binding of the UniROS framework. This approach ensures a clear distinction between each remote robot, facilitating organized communication across concurrent environments.

With these methodologies, the UniROS framework can efficiently manage multiple concurrent environments, whether they are purely simulated robots, real robots, or a combination of both. Furthermore, this study provides ready-to-use templates within the respective MultiROS or RealROS packages to create and manage their RL environments, eliminating the need to handle these ROS Python bindings directly. These templates, including the respective *Base Env* of the packages, relieve experimenters from explicitly handling ROS system-level configurations, allowing them to focus more on implementing their learning tasks.

7. Setting Up Real-Time RL Environments with the Proposed Framework

This section outlines the essential components necessary for establishing real-time learning tasks using ROS. It explores various options for each component and details the specific recommendations for implementing real-time RL environments using the UniROS framework. The ROS-based environment implementation strategy described here ensures that simulated environments closely mirror real-world scenarios for seamless policy transfer and that the real-world environments are safe and effective in facing the challenges of real-world conditions and dynamics.

 doi:10.20944/preprints202507.0012.v1

7.1. Overview of The Real-Time Environment Implementation Strategy

One key challenge when working on real-time learning tasks is that sensory data and control commands are typically processed in a sequence that inherently introduces latency. This latency results from the time it takes the environment to construct observations and rewards and then for the agent to decide on the subsequent action. Since all calculations in the typical MDP architecture are performed sequentially, this can lead to a misalignment between the environment's state and the agent's perception of it. Therefore, it makes the learning problem more complex and may push the agent to develop suboptimal strategies simply because it reacts to outdated information. From the agent's side, augmenting the state space with a history of actions may help the agent anticipate and adapt to delays. However, these methods do not reduce the actual latency of the system but merely help the agent cope with it [48].

Therefore, this study proposes an asynchronous scheduling approach for agent-environment interactions, allowing concurrent processing to minimize overall system latency. This design partitions the reinforcement learning agent and the environment into two distinct processes, as depicted in **Figure 5**. Inside the environment process, multi-threading is employed to concurrently read sensor data and send the robot's actuator commands via the sensorimotor interface to avoid unnecessary system delays. Similarly, an additional dedicated environment loop thread is used to periodically oversee the construction of observations, the calculation of rewards, and the verification of task completion. It also iteratively performs safety checks and updates actuation commands in response to the agent's actions in real time. The intuition behind employing an environmental loop is to allow the agent to make decisions and send actions without waiting for sensor readings or actuator commands to be processed, which helps minimize the agent-environment latency.

On the other hand, the RL agent process updates the actions using the observations and performs learning updates to refine its policy. It also defines the task-specific duration between successive actions (action cycle/ step size), a hyperparameter to set the rate of agent-environment interactions. This computational model of the framework is scalable for learning scenarios that require multiple concurrent environments by utilizing one process per environment, as described in Section **6.1,** enabling the agent to use multi-robot/task learning approaches. Here, the proposed approach does not use multi-threading packages of Python and, instead, utilizes the offerings of ROS to implement the proposed real-time RL environment implementation strategy. The primary reason for this selection is the specialized capabilities of ROS, which offer optimized low-latency communication and a flexible, scalable architecture better suited for complex robotic systems.

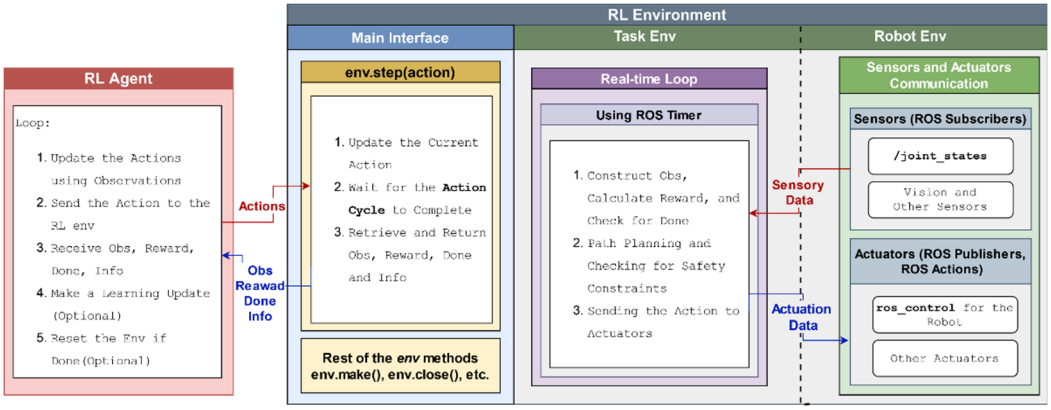


Figure 5. Diagram of a real-time environment implementation strategy for RL with ROS integration, detailing the interaction between the RL Agent process and the RL Environment process. The agent process updates actions based on observations and refines its policy, while the environment-loop thread of the Environment process handles observation creation, reward calculation, and sending actuator commands, all in real-time. This design minimizes latency, allowing for dynamic interaction between the RL agent and the environment in both simulation and real-world learning tasks.

 doi:10.20944/preprints202507.0012.v1

7.2. Reading Sensor Data with ROS

In the ROS framework, sensor packages typically employ ROS Publishers to publish data across various ROS topics[26]. For instance, the *joint_state_controller*[27] of the *ros_control*[28] package launches a ROS node, essentially a separate process dedicated to continuously monitoring the robot's joint positions, velocities, and efforts based on sensor (joint encoders) feedback. This node then periodically publishes captured data to the *joint_states* topic, allowing other packages to monitor the robot's status by subscribing to it. The ROS Subscriber is a built-in ROS component that listens to messages on a specific topic, which triggers a callback function to handle the incoming data each time a message is published into the topic. Since the handling of callbacks of subscribers in *rospy* (Python client library for ROS) is inherently multi-threaded, each sensor callback is handled in a separate thread, allowing for concurrent processing of sensor data from different topics. This approach is adaptable and can be extended to integrate additional sensors, such as a Kinect camera (RGBD), into ROS using respective sensor packages. These packages are responsible for interfacing with the sensor hardware, processing the data, and publishing them to relevant ROS topics, which can be used as part of the observations to provide a comprehensive overview of the environment. However, it should be noted that while Python GIL does not hamper the I/O-bound operations such as ROS subscribers, it can cause a bottleneck if the callback function contains CPU-intensive operations (e.g., preprocessing camera data for object detection). In such scenarios, using a separate ROS node, which creates a separate process, may be beneficial for handling CPU-intensive computations.

7.3. Sending Actuator Commands with Ros

In real-time learning tasks, controlling the robot by directly accessing the low-level control interface of ROS is needed instead of relying on high-level motion planning packages such as MoveIt. This choice was motivated by the need for real-time control of trajectories, a key feature better managed by the robot's low-level controllers. This approach provides the flexibility necessary for dynamic behavior, such as smoothly and rapidly switching between trajectories. While MoveIt is excellent for planning and executing trajectories, its higher level of abstraction limits its capability for instant preemption of trajectories and swift adjustments while maintaining a continuous motion. Therefore, this implementation strategy bypasses MoveIt and directly interacts with the controllers by publishing to the respective *ros_control* topic since real-time learning requires rapid trajectory updates.

7.4. Environment Loop

The environment loop is the cornerstone of the proposed RL environment implementation strategy that facilitates interaction between the RL agent and the robotic environment. It manages the timing and synchronization to ensure seamless integration between the agent's decision-making process and the robot's trajectory executions. This loop is implemented using ROS timers, a built-in feature of ROS that enables the scheduling of periodic tasks. One significant advantage of this method is that the callback function for a ROS timer is executed in a separate thread, similar to the callbacks of the ROS Subscribers. Therefore, this approach ensures that the computational loads of other threads (mostly I/O-bound) do not interfere with the performance of the environment loop.

Hence, the proposed implementation strategy initializes a ROS timer for the environment loop to trigger at a fixed frequency (environment loop rate) throughout the entire operation runtime to execute a sequence of operations, as shown in **Figure 5**. At each trigger, it captures the current state of the environment through data acquired from sensors and formulates them into observation. Then, it calculates the reward and assesses whether the conditions for a done flag are met to indicate the

[26] http://wiki.ros.org/Topics

[27] http://wiki.ros.org/joint_state_controller

[28] http://wiki.ros.org/ros_control

 doi:10.20944/preprints202507.0012.v1

end of an episode. Finally, it checks for safety constraints and executes the last action received from the RL agent. When executing actions, it repeats the previous action if no new action is received during the next timer trigger. This approach, also known as action repeats [17], reduces the agent's computation load and allows for smoother and more stable manipulations as the robot maintains its course of action for a consistent duration. Typically, the environment loop rate is multiple times higher than the action cycle time (step size) of the robot. Therefore, every time the agent process sends an action, the environment process does not need to wait to process the relevant returns (observations, reward, done, and info) after the step size passes, as it can retrieve the latest returns already constructed in the environment loop and pass it to the agent.

Here, the environment loop rate is one of the hyperparameters used to configure real-time environment implementation. However, it is essential to mention that each robot has its own hardware control loop frequency, which determines how quickly it can process information, execute relevant control commands, and read sensor (joint encoder) readings. Therefore, setting the environment loop rate higher than the hardware control loop frequency of the robot is counterintuitive, as the robot cannot physically execute control commands or retrieve the robot's status at a higher frequency than its hardware control loop allows. This hardware control loop frequency depends on the robot's capabilities and is typically set by the manufacturer in the *hardware_interface*[29] of the robot's ROS controller.

The ROS Hardware Interface of a robot is responsible for reading sensor data, updating the robot's state via the *joint_states* topic, and issuing commands to the actuators. This loop runs continuously during the entire robot's operation and is tightly synchronized with the robot's control system. Hence, operating at the hardware control loop frequency ensures that the sensor data readings and actuator command executions are closely aligned with the robot's operational capabilities. This loop rate is usually included in the robot's documentation or inside a configuration file (YAML file) of the robot's ROS controller repository. However, it is also possible to select an environment loop rate lower than the robot's hardware control loop frequency, as this does not hinder the robot's normal operation. The effects of choosing a lower or higher frequency for the environment loop rate, along with the impact of action cycle time on learning, are further discussed in Section **9**, using benchmark learning tasks introduced in Section **8**.

8. Benchmark Tasks Creation

This section discusses the development of benchmark tasks using both the MultiROS and the RealROS packages. These simulated and real environments are then used in the subsequent sections to explain and evaluate the proposed real-time environment implementation strategy and to demonstrate some of the use cases of the UniROS framework. These tasks are modeled closely after the Reach task of the OpenAI Gym Fetch robotics environments [49], where an agent learns to reach random target positions in 3D space. In each Reach task, the robot's initial pose is the default *"home"* position of the robot (typically set zero for all the joint angles of the robot), and the agent's goal is to move the end-effector to a target position (x_g, y_g, z_g), to complete the task. Therefore, each task generates a random 3D point as the target at each environment reset, and the task is completed when the end-effector reaches the goal within ε_r Euclidean distance where ε_r (reach tolerance) is set to 0.02 m. However, unlike the Fetch environments, where the action space represents the Cartesian displacement of the end effector, these tasks use the joint positions of the robot arm as actions. This selection was motivated because the joint position control typically aligns better with realistic robot manipulations, offering enhanced precision and simpler action spaces. The following describes the details of the ReactorX 200 and NED 2 robots and the Reacher tasks (Rx200 Reacher and Ned2 Reacher) creation.

The ReactorX 200 robot arm (Rx200) by Trossen Robotics is a five-degree-of-freedom (5-DOF) arm with a 550 mm reach. It moderately operates at a hardware control loop frequency of 10 Hz. This

[29] http://wiki.ros.org/hardware_interface

 doi:10.20944/preprints202507.0012.v1

compact robotic manipulator is most suitable for research work and natively supports ROS Noetic[30] without requiring additional configuration or setup. It connects directly to a PC via a USB cable for communication and control, providing a reliable and straightforward method of connectivity (uses the default ROS master port). All the necessary packages for controlling the Rx200 using ROS are currently available from the manufacturer as a public repository on GitHub[31]. Similarly, the NED2 robot by Niryo is also designed for research work and features six degrees of freedom (6 DOF) with a 490mm reach. It has a slightly higher control loop frequency of 25 Hz and natively runs ROS Melodic[32] on an enclosed Raspberry Pi. Niryo offers three communication options for connecting the NED2, including a Wi-Fi hotspot, direct Ethernet, or connecting both devices to the same local network. As SSH-based access was not desirable, this study opted for a direct Ethernet connection and utilized the ROS multi-device mode, as described in Section **6.2**, to ensure a robust communication setup. Furthermore, Niryo also provides the necessary ROS packages[33] to be installed on the local system, enabling custom messaging and service interfaces to access and control the remote robot through ROS.

Since two variants of the *Base Env* (standard and goal-conditioned) are available in this framework, two types of RL environments were created for both simulation and the real world. These simulated and real environments bear continuous actions and observations and support sparse and dense reward architectures. In the created goal-conditioned environments, the agent receives observations as a Python dictionary containing the typical observation, achieved goal, and desired goal. The achieved goal is the current 3D position of the end-effector (x_a, y_a, z_a), which is obtained using FK calculations, and the desired goal is the randomly generated 3D target (x_g, y_g, z_g). One of the decisions made during task creation is to include the previous action as part of the observation vector, as this can minimize the adverse effects of delays on the learning process [50]. Additionally, the observation vector also includes the position of the end effector (EE) with respect to the base of the robot, the current joint angles of the robot, the Cartesian displacement, and the Euclidean distance between the EE position and the goal. Additional experiment information, including details on actions, observations, and the reward architecture, is explained further in **Appendix A**.

Furthermore, specific constraints were implemented on the operational range of both types of environments to ensure the safe operation of the robot and prevent any harm to itself or its surroundings. One of the steps taken here is to limit the goal space of the robot so it cannot sample negative values in the z-direction in the 3D space. This is vital since the robot is mounted on a flat surface, making it impossible to reach locations below it. Additionally, before the agent executes the actions with the robot, the environments check for potential self-collision and verify whether the action would cause the robot to move towards a position in the negative z-direction. Therefore, forward kinematics are calculated using the received actions before executing them to avoid unfavorable trajectories, allowing the robot to operate within a safe 3D space. Hence, considering the complexity of the tasks and compensating for the gripper link lengths, the goal space was meticulously refined to have maximum 3D coordinates of $x: 0.40, y: 0.40, z: 0.40$ and minimum of $x: 0.15, y: -0.40, z: 0.15$ (in meters) for both robots.

As for the learning agents of the experiments in this study, the vanilla TD3 was used for standard-type environments and TD3+HER for goal-conditioned environments. TD3 is an off-policy RL algorithm and can only be used for environments with continuous action spaces. It was introduced to curb the overestimation bias and the other shortcomings of the Deep Deterministic Policy Gradient (DDPG) algorithm [51]. Here, TD3 was extended by combining it with Hindsight Experience Replay (HER) [52], which encourages better exploration for goal-conditioned environments with sparse rewards. By incorporating HER, TD3+HER improves sample efficiency

[30] http://wiki.ros.org/noetic

[31] https://github.com/Interbotix/interbotix_ros_manipulators

[32] https://wiki.ros.org/melodic

[33] https://github.com/NiryoRobotics/ned_ros

 doi:10.20944/preprints202507.0012.v1

due to HER utilizing unsuccessful trajectories and adapting them into learning experiences. This study implemented these algorithms using the custom TD3+HER implementations and the Stable Baselines3 (SB3) library, adding ROS support to facilitate their integration into the UniROS framework. The source code and supporting utilities are available on GitHub[34], allowing other researchers and developers to leverage and build upon this work. The detailed information on the RL hyperparameters used in the experiments is summarized in **Appendix B.** Furthermore, all compute during experiments were conducted on a PC with Nvidia 3080 GPU (10 GB VRAM) and Intel i7-12700 processor with 64 GB DDR4 RAM.

9. Evaluation And Discussion of the Real-Time Environment Implementation Strategy

This section examines the intricacies of the proposed ROS-based real-time RL environment implementation strategy, utilizing benchmark environments as the experimental setup. The primary goal here is to discuss the two main hyperparameters of the proposed implementation strategy and gain an understanding of how to select suitable values for them, as they largely depend on the hardware capabilities of the robot(s) used in the learning task. Initially, experiments were conducted to investigate different action cycle times and environment loop rates, aiming to uncover the intricate balance between control precision and learning efficiency. Then, exploration was extended to include an empirical evaluation of the asynchronous scheduling within the proposed environment implementation strategy. This process involves thoroughly analyzing the time taken for each action and actuator command cycle across numerous episodes.

9.1. Impact of Action Cycle Time on Learning

In the real-time RL environment implementation strategy, the action cycle time (step size) is a crucial hyperparameter that determines the duration between two subsequent actions from the agent. The selection of this duration impacts the learning performance due to the use of action repeats in the environment loop. Action repeats ensure the robot can perform smooth and continuous motion over a given period, especially when the action cycle time is longer. This technique helps stabilize the robot's movements and maintain consistent interaction with the environment between successive actions.

Selecting a shorter action cycle time, close to the environment loop rate, would reduce reliance on action repeats and enable faster data sampling from the environment due to more frequent agent-environment interactions. This would allow the agent to have finer control (high precision) over the environment at the cost of seeing minimal changes in the observations. Such minimal changes can adversely affect training, as the agent may not perceive significant variations in the observations necessary for effectively updating deep neural network-based policies, such as TD3. Conversely, selecting a longer action cycle time could lead to more action repeats and substantial changes in observations between successive actions, potentially easing and enhancing the learning for the agent. However, this comes at the risk of reduced control precision and potentially slower reaction times to environmental changes, which can be detrimental when working in highly dynamic environments. Furthermore, this could potentially slow down the agent's data collection rate, leading to a longer training time.

Therefore, to study the effect of action cycle time on learning, experiments were conducted with multiple durations, selecting a baseline and comparing the effects of longer action cycle times, as depicted in **Figure 6**. These experiments were conducted in the real-world Rx200 Reacher task, using the same initializing values and conditions and employing the vanilla TD3 algorithm. This figure contains three graphs that illustrate the learning curves of the training process for all the selected action cycle times. **Figure 6 (a)** shows the mean episode length, representing the mean number of interactions the agent had with the environment while trying to achieve the goal within an episode. Ideally, the episode length should shorten over time as the agent learns the optimal way to behave in

[34] https://github.com/ncbdrck/sb3_ros_support

 doi:10.20944/preprints202507.0012.v1

the environment. Similarly, **Figure 6 (b)** depicts the mean total reward obtained per episode during training. The agent's goal is to maximize this reward by improving its policy and learning to complete the task efficiently. Furthermore, in the benchmark tasks, the maximum allowed number of steps per episode is set to 100, providing a maximum of 100 agent-environment interactions to achieve the task. Exceeding this limit results in an episode reset and failure to complete the task. These failure conditions and successful task completion conditions are used to illustrate the success rate curve in **Figure 6 (c)** (refer to **Appendix A** for more information).

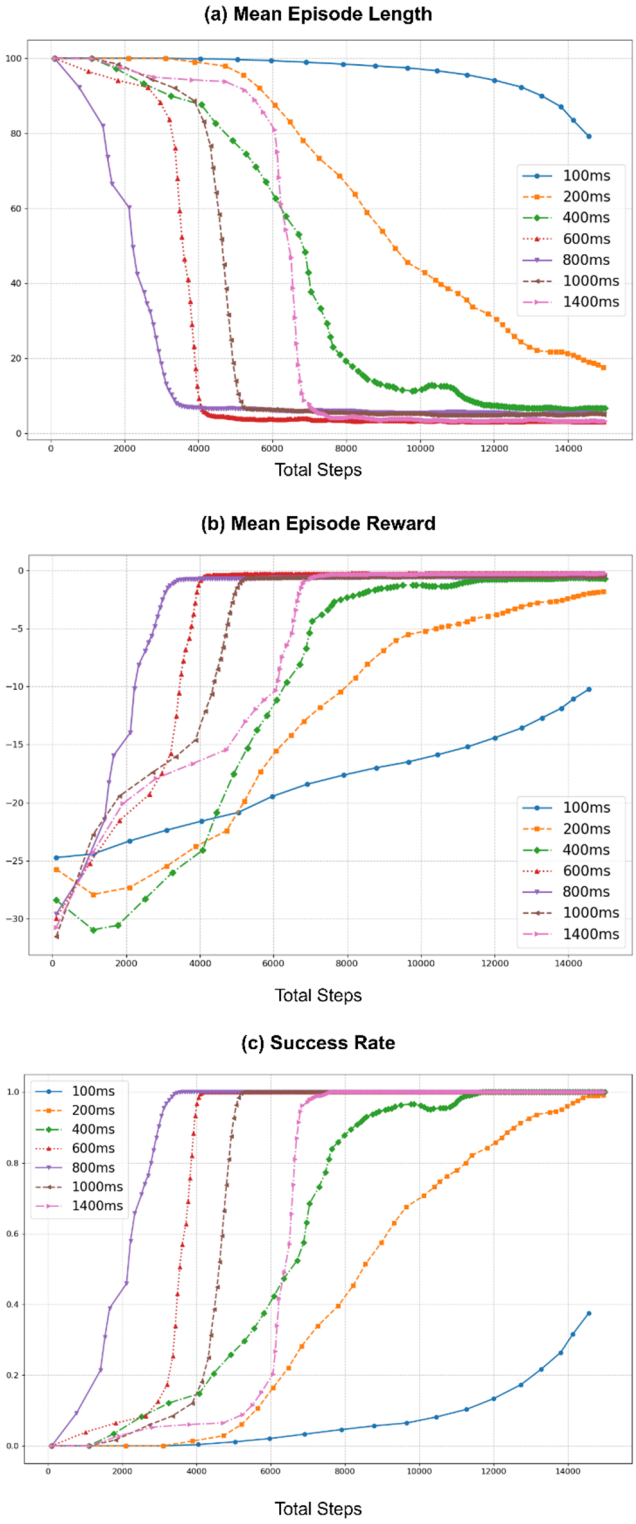


Figure 6. Impact of action cycle times on the learning performance of the Rx200 Reacher benchmark task. (a) Mean episode length, where a shorter length indicates better performance. (b) Mean total reward per episode,

 doi:10.20944/preprints202507.0012.v1

with higher rewards reflecting improved learning. (c) Success rate, indicating the proportion of episodes that were successfully completed. Optimal performance is observed at action cycle times of 600 ms to 800 ms, with performance declining at longer cycle times due to reduced control precision and slower reaction times.

In the experiments, the baseline was set at 100 ms, matching the duration of the hardware control loop frequency of the Rx200 robot (10 Hz), which is used as the default environment loop rate (10 Hz) of the Rx200 Reacher benchmark task. This selection represents the shortest action cycle time that can be used in this benchmark task, as the Rx200 robot does not function properly below this duration. Then, the training was repeated to obtain learning curves for action cycle times of 200 ms, 400 ms, 600 ms, 800 ms, 1000 ms, and 1400 ms. Furthermore, every run of the experiment was conducted for 30,000 steps, allowing sufficient time for each run to find the optimal policy. However, data points illustrated in Figure 6 were smoothed using a rolling mean with a window size of 10, plotted every 10th step, and shortened to the first 15K steps to improve readability.

As shown in **Figure 6**, increasing the action cycle time can improve performance up to a certain point compared to using the same time duration as the environment loop rate (100ms). For this benchmark task, the learning curves for action cycle times of 600ms and 800ms showed the best performance, quickly stabilizing with shorter episode lengths, higher total rewards, and higher success rates. This improvement can be attributed to the balance between sufficient observation changes and the agent's ability to interact effectively with the environment. However, as the action cycle time increased beyond 800 ms, the performance started to degrade, as the learning curves for action cycle times of 1000 ms and 1400 ms required a larger number of steps to stabilize to an optimal policy. This decline in performance is likely due to the agent receiving less frequent updates, which introduces potentially more significant errors in the policy updates, causing the agent to struggle with maintaining optimal behavior.

Overall, the experiments demonstrate that while increasing the action cycle time can initially improve learning by providing more substantial observation changes, there is a threshold beyond which further increases become detrimental. Therefore, the choice of action cycle time and the use of action repeats must be balanced based on the specific requirements of the task and the capabilities of the robot. Fine-tuning these parameters is crucial for optimizing learning performance and ensuring robust real-time agent-environment interactions.

9.2. Impact Of Environment Loop Rate On Learning

To assess the impact of various environment loop rates, the same learning process was repeated using rates of 1 Hz, 5 Hz, 10 Hz, 20 Hz, 50 Hz, and 100 Hz. Here, the action cycle time was set for each run to match the environment loop rate to simplify the task by eliminating action repeats. Furthermore, the baseline for these experiments was set at 10 Hz to align with the hardware control loop frequency of the Rx200 robot used in the benchmark task. Similar to the previous section, Error! Reference source not found. illustrates the learning curves across different environment loop rates, with the same post-processing methods employed to enhance the readability of the curves.

As shown in Error! Reference source not found., at lower environment loop frequencies (1 Hz and 5 Hz), the learning performance was better compared to the baseline of 10 Hz. This improvement can be attributed to the longer action cycle times that take larger actions (joint positions), which result in more significant variability in observations, aiding the learning process. However, performance starts to degrade as the environment loop rate increases beyond 10 Hz. The learning curves for higher loop rates (20 Hz, 50 Hz, and 100 Hz) show increased mean episode lengths and lower mean rewards, indicating less efficient learning. This decline is due to robots' inability to effectively process commands and read joint states at these higher frequencies. Although control commands are sent at a higher rate, the robot's hardware control loop operates at 10 Hz, causing instability in command execution. Furthermore, as ROS controllers typically do not buffer control commands, the hardware control loop processes only the most recent command on the relevant ros topic, leading to instability in training.

 doi:10.20944/preprints202507.0012.v1

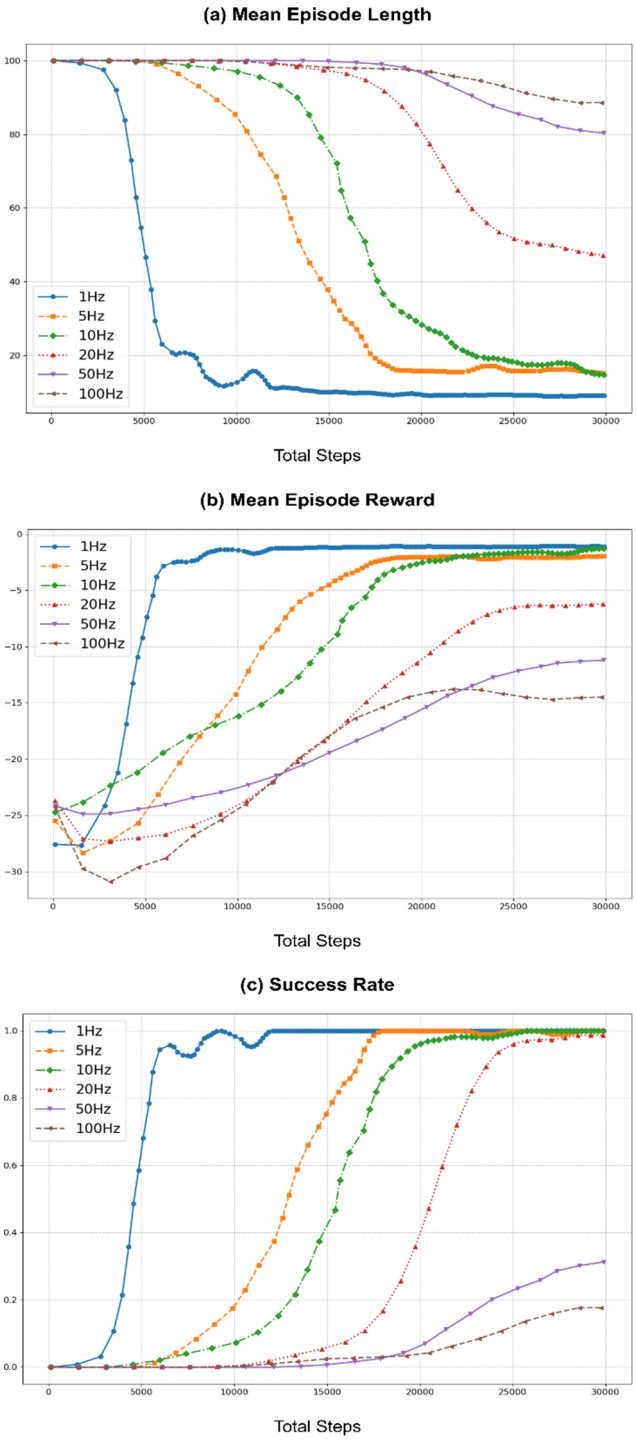


Figure 7. Learning performance of the Rx200 Reacher task at various environment loop rates. The results demonstrate that lower loop rates (1 Hz and 5 Hz) than the baseline (10 Hz) improve learning performance due to larger action cycle times, which provide greater variability in observations. Higher loop rates (20 Hz, 50 Hz, and 100 Hz) degrade performance due to instability caused by the robot's inability to process commands and joint states effectively at frequencies above its hardware control loop frequency.

9.3. Empirical Evaluation of Asynchronous Scheduling of the Real-Time Environment Implementation Strategy

To empirically validate the real-time RL environment implementation strategy, the time taken for each action cycle and each actuator command cycle across numerous episodes was logged. The experiments were conducted using both the Rx200 Reacher and the Ned2 Reacher, with environment loop rates of 10 Hz and 25 Hz, respectively. These rates correspond to a 100 ms period and 40 ms to

 doi:10.20944/preprints202507.0012.v1

send actuator commands to the robot, with an action cycle time set at 800 ms, providing ample time to execute action repeats. The goal was to observe the effectiveness of the asynchronous scheduling approach in managing agent-environment interactions and to measure the latency inherent in the system quantitatively.

The boxplot in **Figure 8 a)** depicts the distribution of cycle durations for the actuator and action cycles within the Rx200 Reacher Task during training in the real world. It shows a median action cycle time of 803.3 ms and a median actuator cycle of 100.11 ms, which closely approximates the respective preset threshold values for the benchmark task. Furthermore, despite the presence of some outliers, the compact interquartile ranges in both plots indicate that the system performs with a high degree of consistency overall with negligible variability. However, it should be noted that the variations observed in the action cycle durations are partly due to the use of the TD3 implementation in the Stable Baselines3 (SB3) package as the learning algorithm. SB3 is a robust framework for reinforcement learning. However, it is not explicitly designed for robotic applications or real-time training scenarios in mind and is more of a general-purpose RL library. Therefore, SB3 does not typically schedule policy updates immediately after sending an action or use asynchronous processing to update the policy, which can introduce delays and variations. Therefore, the absence of asynchronous scheduling in SB3 means the agent waits for the policy update to be completed before proceeding with the following action. This synchronous approach can lead to slight variations in the action cycle times, as seen in the box plot.

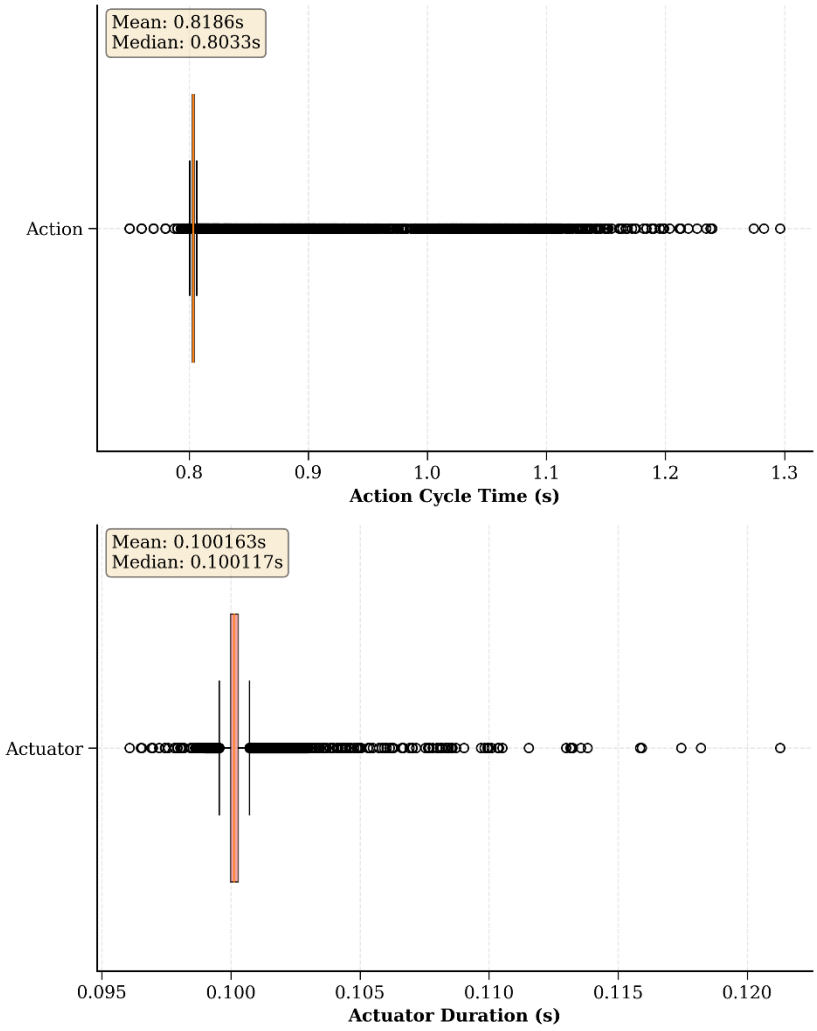


Figure 8. Distribution of cycle durations for the action and actuator loops in the Rx200 Reacher Task using TD3 implemented with Stable-Baselines3 (SB3). While actuator durations remain consistent, the wider spread in

 doi:10.20944/preprints202507.0012.v1

action cycle times reflects the limitations of SB3, which is not optimized for real-time robotic training and does not support asynchronous policy updates.

One solution to this issue is to develop custom RL implementations that incorporate asynchronous policy updates [53]. This approach allows the policy to be updated in the background while the agent continues to interact with the environment, thereby reducing latency and improving the efficiency of real-time learning. By scheduling policy updates asynchronously, these methods can ensure that agent-environment interactions are not interrupted, maintaining the consistency and precision required for effective real-time learning. To evaluate the impact of this approach, additional experiments were conducted using a custom TD3 implementation in which policy updates were explicitly scheduled asynchronously relative to data collection. As illustrated in **Figure 9**, the Rx200 Reacher demonstrated improved temporal consistency in action execution. The Rx200 Reacher task displayed a similar median action cycle time of 803.3 ms, with a narrower interquartile range compared to the standard SB3 implementation. This reduction in variability confirms that asynchronous scheduling effectively mitigates timing disruptions introduced by synchronous policy updates, leading to improved temporal consistency during task execution.

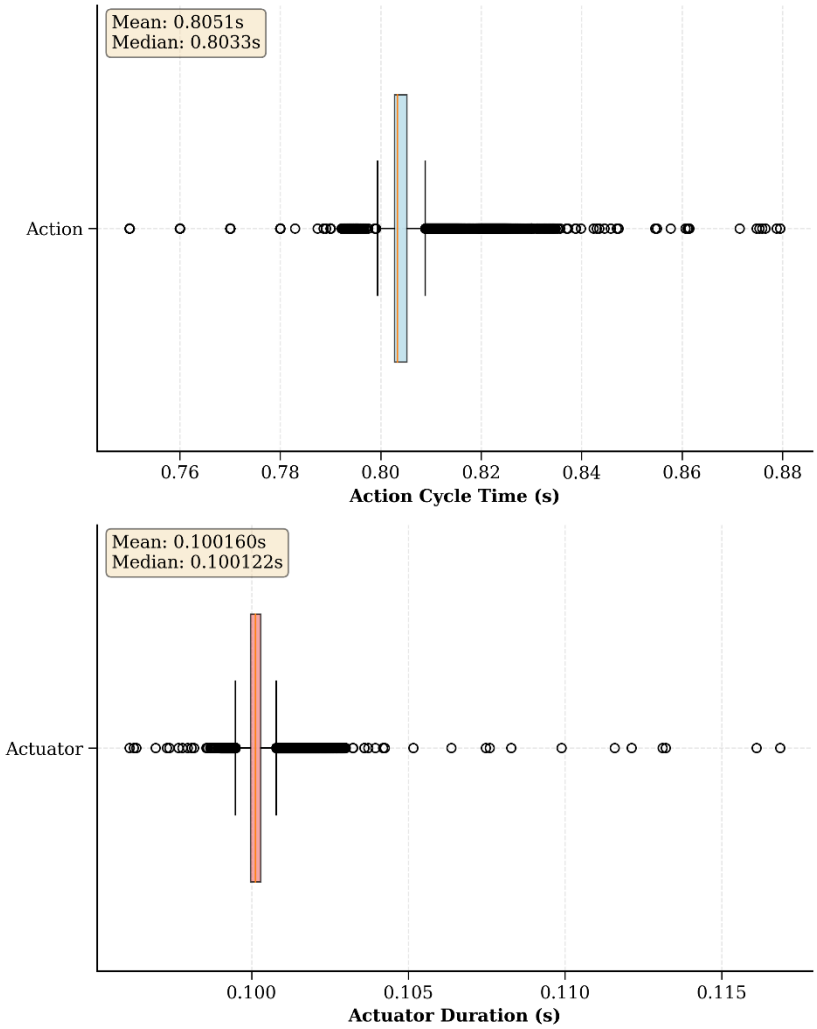


Figure 9. Distribution of cycle durations for the action and actuator loops in the Rx200 Reacher Task using a custom TD3 implementation with asynchronous policy updates. Compared to the standard SB3-based TD3, the action cycle times exhibit a narrower interquartile range, indicating improved temporal consistency. This suggests that asynchronous scheduling of policy updates effectively reduces variability in real-time training.

 doi:10.20944/preprints202507.0012.v1

Furthermore, to evaluate the proposed approach under high computational load, the experiments were extended to support concurrent learning across both simulated and real environments. In this setup, all four environments (two simulated and two real environments - Rx200 Reacher and Ned2 Reacher) were trained simultaneously using the asynchronous policy update mechanism. The results, as depicted in **Figure 10**, show that the distribution of action and actuator cycle times across all four tasks remains consistent with those observed in the single-environment experiments. Each subplot within the composite boxplot illustrates minimal variation, with median values closely aligning with the predefined cycle thresholds, indicating that asynchronous scheduling sustains reliable timing even under multi-environment execution. These results support the proposed concurrent processing methodology, which minimizes overall system latency and facilitates real-time agent-environment interactions.

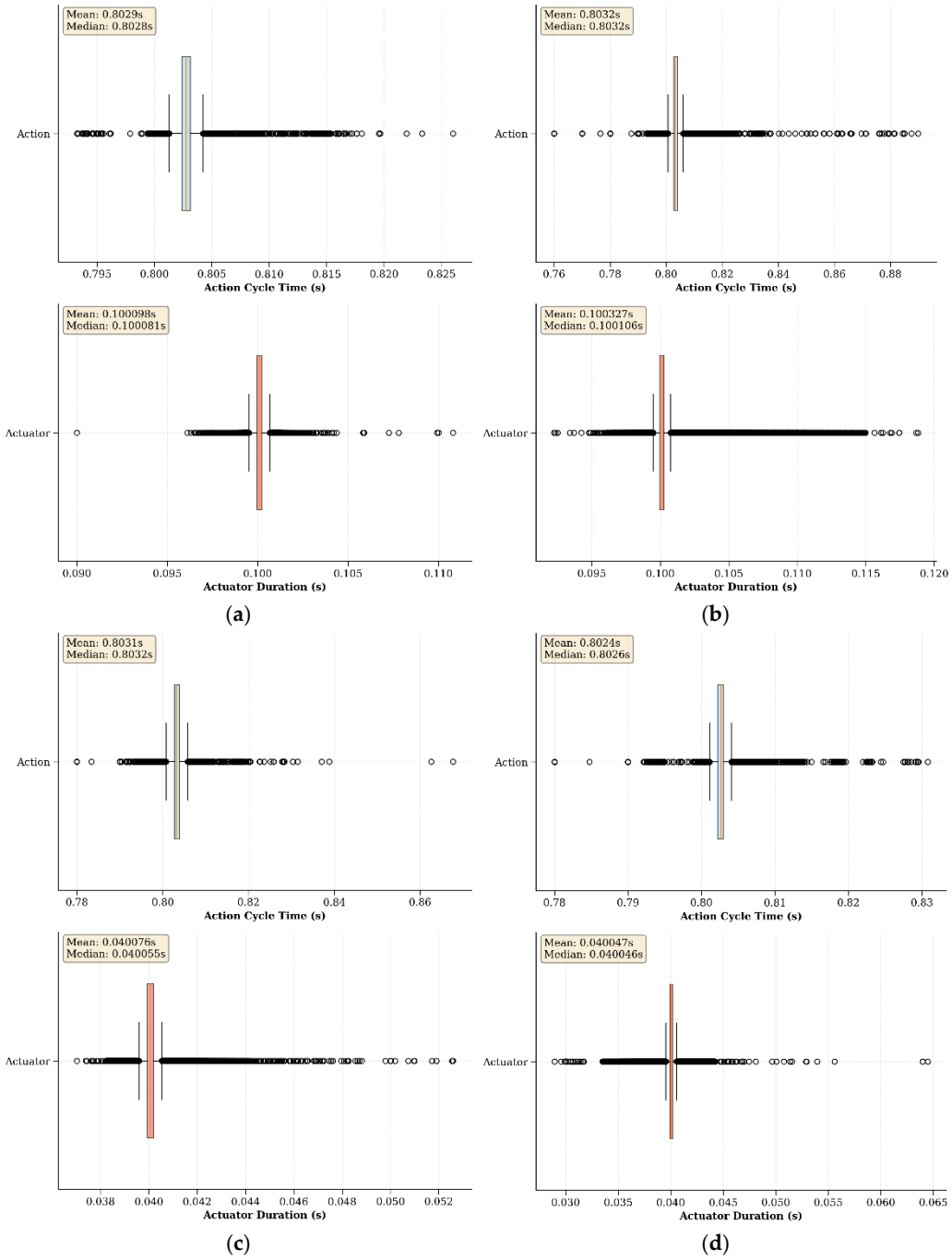


Figure 10. Distribution of action and actuator cycle durations during concurrent training in four environments using asynchronous policy updates. Subplots (a) and (b) show the results for the Rx200 Reacher task in real and simulated settings, respectively, both configured with a 100 ms actuator cycle and an 800 ms action cycle.

 doi:10.20944/preprints202507.0012.v1

Subplots (c) and (d) correspond to the Ned2 Reacher task in real and simulated environments, with a 40 ms actuator cycle and the same 800 ms action cycle. All plots exhibit narrow interquartile ranges and closely aligned medians, demonstrating consistent temporal performance and minimal variability across both platforms.

9.4. Discussion

All the experiments performed in this study were conducted with a single robot in each RL environment. However, as discussed in the previous sections, the proposed UniROS framework enables the use of multiple robots in the same RL environment, particularly when these robots need to collaborate to complete a task. If the robots used in the task are of the same make and model, the experimenters can use the hardware control loop frequency of the robots as the environment loop rate. However, this setup could introduce additional complexities, particularly when the robots have different hardware control loop frequencies. For instance, consider that the two robots used in this study are combined in a single RL environment, where the Rx200 robot has a hardware control loop frequency of 10 Hz, and the Niryo NED2 operates at 25 Hz. In such a scenario, it is desirable to use the lower hardware control loop frequency (in this scenario, 10 Hz) as the environment loop rate for the entire system to ensure synchronized operation. This approach prevents the faster robot from issuing more frequent commands that the slower robot cannot keep up with, thereby maintaining consistent interaction with both robots. Furthermore, pairing a slower robot like the Rx200 (10 Hz) with a more industrial-grade manipulator, such as the UR5e robot, which has a hardware control loop frequency of 125 Hz, may not be ideal. The disparity in control loop rates could lead to inefficiencies and instability in the learning process. The slower robot would become a bottleneck, hindering the performance of the faster robot and potentially disrupting the overall task execution.

Additionally, when initializing the *joint_state_controller* that publishes the robot's joint state information, the publishing rate can be set at any frequency. This will lead to the ROS controller publishing at the specified rate, even though it differs from the hardware control loop frequency of the said robot. While setting a higher frequency does not impact learning, as the ROS controller publishes the same joint state information multiple times, setting a lower frequency will lead to degraded performance because the agent will not receive the most up-to-date information from the robot. Therefore, the most straightforward solution is to adjust the publishing rate of the *joint_state_controller* to match the hardware control loop frequency or higher, as this ensures that each published message corresponds to an actual update from the hardware.

Similarly, suppose an external sensor in the task operates at a lower rate than the hardware control loop frequency of the robot. In that case, it is essential to account for this in the environment loop of the RL environment. This could mean using the latest available sensor data, even if it is not updated every loop iteration. In these scenarios, action repeats can be beneficial, especially when dealing with robots with higher hardware control loop frequencies. It will make learning easier for the RL agent by receiving observations that display more substantial changes at each instance rather than infrequent and minimal changes that make learning harder.

10. Use Cases

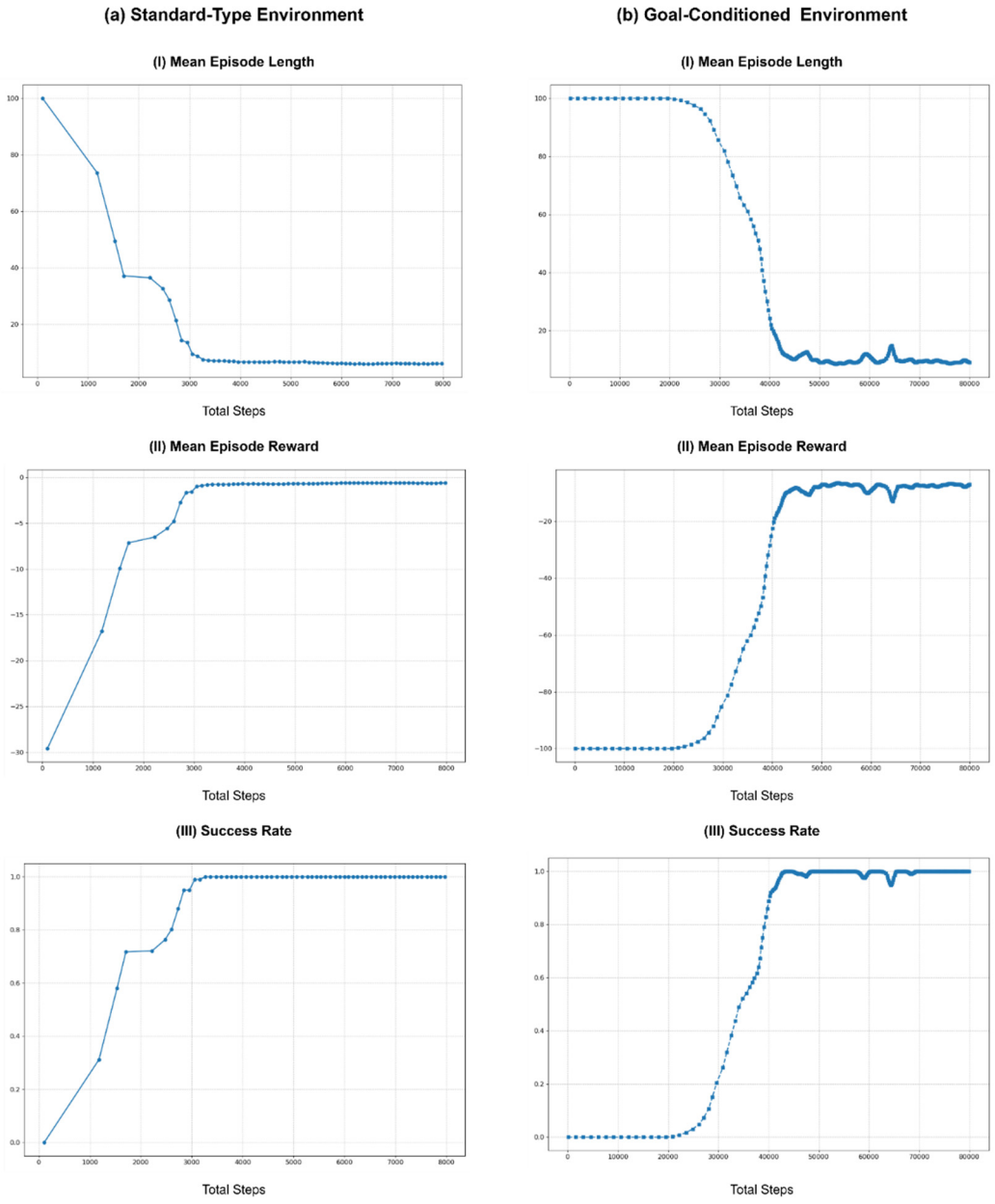
Here, three possible use cases of UniROS are exhibited, each highlighting a unique aspect of its application. The first use case demonstrates training a robot directly in the real world, showcasing how to utilize the framework for learning without relying on simulation. The second use case demonstrates zero-shot policy transfer from simulation to the real world, highlighting the capability of the proposed framework for transferring learned policies from simulation to the real world. Finally, the last use case demonstrates the framework's ability to learn policies applicable in both simulated and real-world environments. In these use cases, the environment-loop rate was set to 10 Hz and the action cycle time to 800 ms, as this configuration showed the best results for the Rx200 Reacher task in Section **9.1.** Similarly, an environment-loop rate of 25 Hz and the same action cycle time of 800 ms were used for the Ned2 Reacher due to the multi-task learning setup in one of the use cases (Section **10.3**), ensuring that both robots received actions at the same temporal frequency. This

 doi:10.20944/preprints202507.0012.v1

consistency in action dispatching across tasks facilitated stable training and improved coordination when learning shared representations in the multi-robot learning setups.

10.1. Training Robots Directly in The Real World

The first use case is demonstrated using the Rx200 Reacher task. Here, the physical robot was directly trained in the real world using TD3 (for standard-type environment) and TD3+HER (for goal-conditioned environment). Then, to evaluate the performance of learning, the success rate, mean total reward per episode, and the number of steps (agent-environment interactions) taken by the robot to reach the goal position in an episode were plotted. **Figure 11** shows the learning metrics of the trained robot in each environment. It should be mentioned that since the experiment using the standard-type environment with the stated action cycle duration of 800 ms and environment loop rate of 10 Hz has already been conducted and showcased in **Figure 6**, the curves were replotted in **Figure 11 (a)** for readability and to ensure that the results are easily interpretable without the complexity of multiple curves in a single figure. Therefore, to provide a clear presentation of the learning curves for both environments, **Figure 11** was organized into two parts. Part **(a)** presents the learning metrics for the standard-type environment and part **(b)** presents the learning metrics for the goal-conditioned environment. The learning curves of the next two use cases also follow this same convention.



 doi:10.20944/preprints202507.0012.v1

Figure 11. Learning curves from direct real-world training of the Rx200 robot. Part (a) illustrates the learning metrics for the standard-type environment using the TD3 algorithm, and Part (b) presents the learning metrics for the goal-conditioned environment using the TD3+HER algorithm. These metrics collectively demonstrate the robot's learning performance of the proposed training approach in real-world environments.

Here, it was observed that both environments performed well on the Reacher task, as their success rates and mean rewards steadily increased, while the average steps gradually decreased as learning progressed. Furthermore, the plateau of the success rate in both environments indicates that the robot had learned a near-optimal policy for the given task. These results demonstrate that the proposed study enables the direct training of robots in the real world using RL with a single stream of experience.

Furthermore, it is essential to note that during the initial stage of training, challenges arose due to some of the joints attempting to move beyond the restricted workspace. Therefore, the initial solution of simply restricting the end-effector (EE) pose to be within a workspace proved insufficient. Instead, the action (containing joint position values) was used to calculate forward kinematics (FK) for all the joints to check if any were trying to exceed the workspace limits. If any of the resulting joints were found to be out of bounds, the robot was prevented from executing that action. This additional step ensured the robot's safety and stability during training. These findings highlight the feasibility and benefits of using the UniROS framework for direct real-world training of robotic systems, laying the groundwork for more complex future applications.

Additionally, it should be noted that the learning progress differs between standard-type and goal-conditioned environments, with the former achieving a near-optimal policy before 10K steps and the latter taking around 50K to 80K steps. A detailed explanation for this disparity is provided in Appendix C, which discusses the differences in reward architectures and environment types, highlighting how dense rewards in standard-type environments facilitate quicker convergence compared to sparse rewards in goal-conditioned environments.

10.2. Simulation to Real-World

This section presents the experimental results obtained from training the Rx200 robot in simulation environments and subsequently transferring the learned policies to the physical robot. This experiment utilizes the simulated Rx200 Reacher task environments and employs the same RL algorithms to train the agents. These simulated environments were created in accordance with the proposed real-time implementation strategy, setting the same environment loop rate and action cycle time to match the previous real-world use case. Furthermore, to ensure seamless policy transfer from the simulation to the real world, the simulated environments were configured to mirror real-world conditions as closely as possible, which includes not pausing the simulation. The primary aspect of this process was mimicking the hardware control loop of the real robot within the Gazebo simulation, ensuring that the timings of the environment loop in the simulated environment closely matched those of the real-world counterpart. **Figure 12** shows the constructed real-world environment inside the Gazebo simulator.

 doi:10.20944/preprints202507.0012.v1

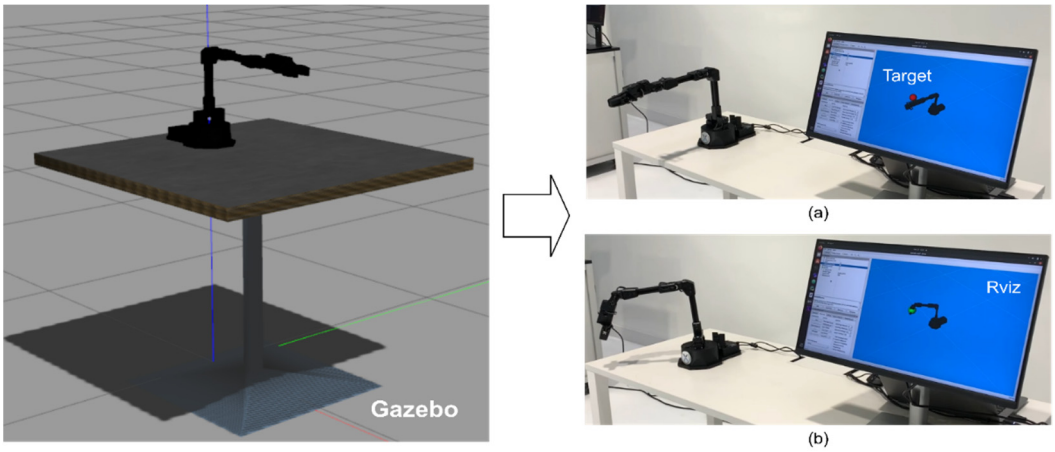


Figure 12. From Simulation to the Real World: The left image depicts the simulated environment in the Gazebo simulator, featuring the Rx200 robot and a table used to train the reaching task with the MultiROS package. The right image shows the real-world setup for evaluating the transferred policy, where Rviz is used to visualize the randomly generated target and the task's status. The red marker (a) represents the goal the robot must reach, which turns green (b) upon successful completion of the task.

Additionally, the simulated robot's URDF file (robot description) contains the *gazebo_ros_control*[35] plugin, which loads the appropriate hardware interfaces and controller manager for the simulator to update at the hardware control loop frequency of the real robot. This plugin configuration ensures that actuator commands are processed at the desired control frequency that matches the real-world RL environment. However, it was observed that manually setting this parameter in the URDF can sometimes cause the robot to exhibit unexpected behaviors in the simulation. One solution for these scenarios is to use the hardware control loop frequency of the real robot as the environment loop rate, forcing the simulated robot to receive control commands and operate at the actual hardware control loop frequency. Therefore, setting these configurations enables learning in the simulated environment to resemble real-world learning closely.

As illustrated in **Figure 13**, the learning curves were plotted to evaluate the learning performance for both types of simulated environments. Similar to the previous use case, it was observed that both agents learned nearly an optimal policy as the success rate plateaued for the Reacher task. Furthermore, the gradual increase in the success rate and mean reward, along with the decrease in the mean episode length, indicate that the learning was stable and consistently improved throughout the training. Once the trained policies were obtained from the simulation environments, a zero-shot transfer was performed directly for the physical Rx200 robot (**Figure 12**) without employing any sim-to-real techniques or domain adaptation methods to bridge the reality gap. The primary objective of zero-shot transfer was to evaluate the proposed framework's ability to generalize its learning from ROS-based simulation environments to the real world without requiring any additional training.

[35] https://classic.gazebosim.org/tutorials?tut=ros_control

 doi:10.20944/preprints202507.0012.v1

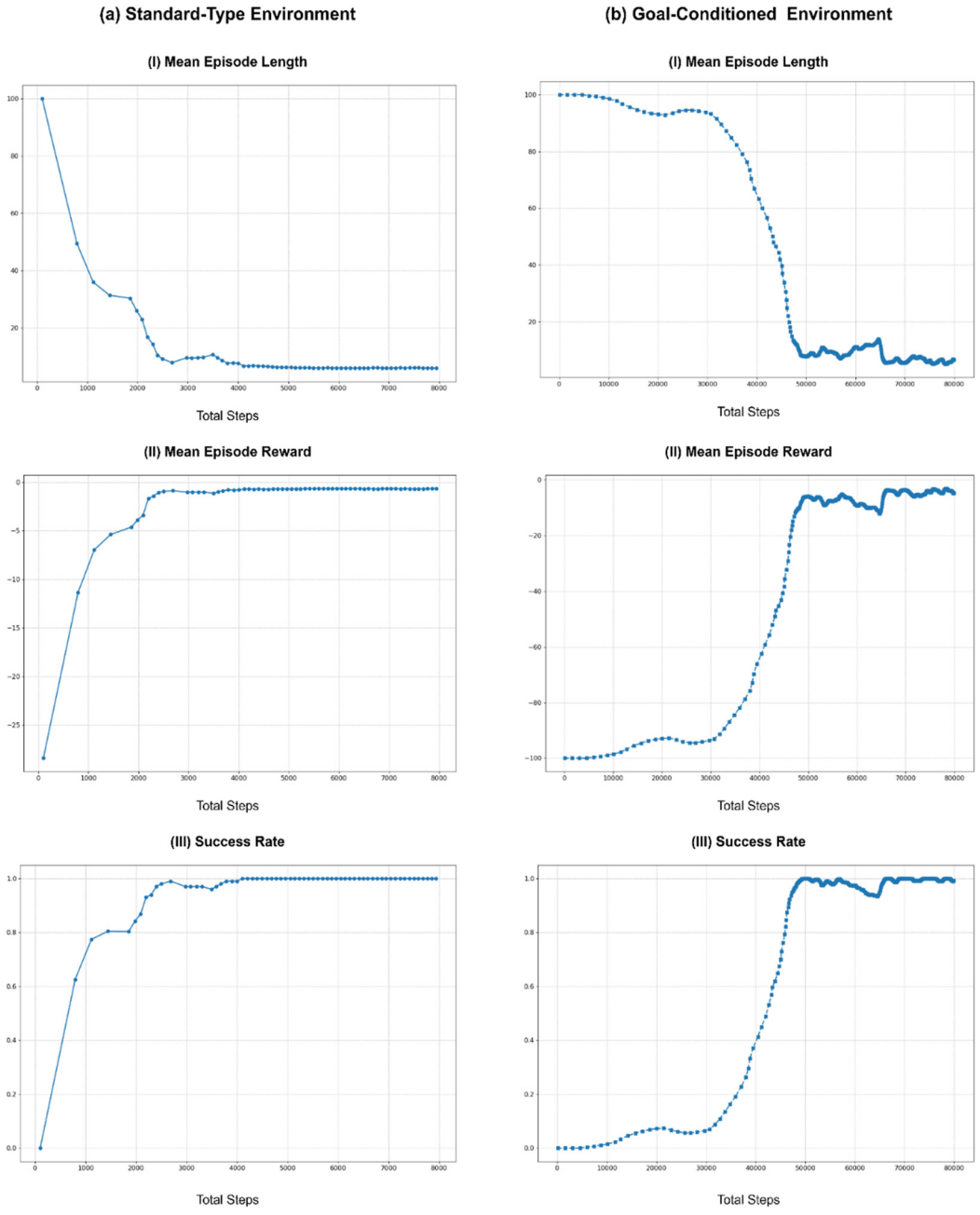


Figure 13. Learning metrics for training the Rx200 robot in simulation. Part (a) illustrates the learning metrics for the standard-type environment using the TD3 algorithm, and Part (b) presents the learning metrics for the goal-conditioned environment using the TD3+HER algorithm. These metrics demonstrate the robot's learning performance in simulation, which was then used to transfer the learned policies to real-world environments.

To evaluate the policy transfer, two hundred episodes of the Reach task were conducted to record the success or failure of each episode. This provided valuable insight into the performance of the transferred policies in the physical robot. Here, it was observed that the trained TD3 model in the standard-type environment and the TD3+HER model in the goal-conditioned environment achieved nearly a 100% success rate in the real world. This result indicates that the transferred policies could generalize and accurately guide the physical robot without requiring additional fine-tuning. However, it should be mentioned that while the relatively simple Rx200 Reacher task could achieve successful zero-shot policy transfer without any additional fine-tuning, it will not necessarily be the same for all the tasks. This is especially true for complex tasks involving additional sensors (cameras and lidars), as they may require domain randomization techniques such as sampling data from multiple environments, each with different seeds and gazebo physics parameters (which can be

 doi:10.20944/preprints202507.0012.v1

tuned using UniROS). In such cases, zero-shot transfer might not be feasible, and fine-tuning the policies in the real world could be necessary. However, the initial training in the simulated environment can provide a good starting point for further optimization in the real world.

10.3. Concurrent Training In Real and Simulation Environments

This use case is comprised of two experiments. The first experiment demonstrates how real-world dynamics and kinematics can be learned concurrently using less expensive simulation environments to expedite the learning process. The second showcases one of the multi-robot/task learning approaches using the proposed framework and the environment implementation strategy.

10.3.1. Learning a Generalized Policy

This experiment aims to exhibit the capability of the proposed framework for training a generalized policy that can perform well in both domains by leveraging knowledge from simulation and real-world data using concurrent environments. This experiment was designed using the created Rx200 Reacher real and simulation environments to learn the same reach task. Furthermore, to be consistent with the previous use cases, one generalized policy was trained for standard-type environments (sim and real) with dense rewards and another for goal-conditioned environments with sparse rewards. Here, the proposed framework's ability to execute concurrent environments was exploited to enable synchronized real-time learning in the simulation and real-world environments.

In this experiment, an iterative training approach was employed to update a single policy by sampling trajectories from both real and simulated environments, enabling the agent to integrate real-world dynamics and kinematics into the learning process. Similar to the previous use cases, a TD3-based learning strategy was applied for the standard-type environments and a TD3+HER strategy for goal-conditioned environments. The implemented learning strategy is shown in **Algorithm 1.** Then, the performance of the learning process was evaluated by plotting the learning curves, as shown in **Figure 14**. As observed in the previous use cases, the gradual decrease in the mean episode length and increase in the rewards and success rate implies that the learning was stable and consistent throughout the training. Additionally, both agents learned nearly an optimal policy, as indicated by the plateauing of the success rate. Furthermore, similar to the previous use case, deploying the trained agent in the respective domain yielded 100% accuracy with both types of environments.

 doi:10.20944/preprints202507.0012.v1

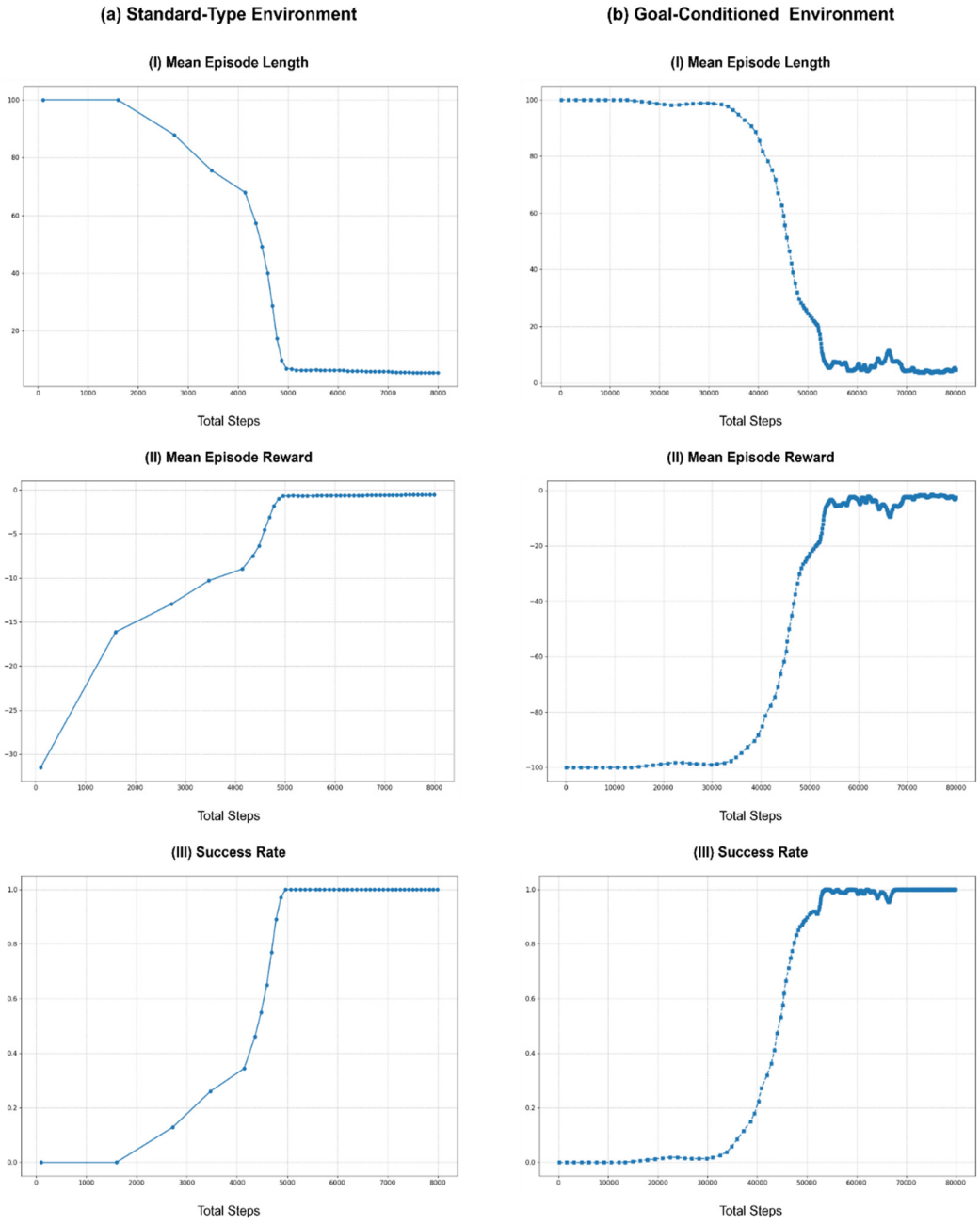


Figure 14. Learning metrics for training the Rx200 robot in both real and simulation environments simultaneously. Part (a) illustrates the learning metrics for the standard-type environment using the TD3 algorithm, and Part (b) presents the learning metrics for the goal-conditioned environment using the TD3+HER algorithm. These metrics demonstrate the robot's learning performance when trained in a combined real and simulated environment, resulting in a generalized policy applicable to both domains.

While the results illustrated in **Figure 14** do not seem very impressive compared to the previous use cases, the main advantage of the use case lies in its demonstration of the capability to train concurrently in both real and simulated domains using the same policy. Here, the relatively simple nature of the reach task does not fully reflect the potential benefits of this concurrent learning approach, as the gap between simulation and real-world performance is minimal. However, the true strength of this concurrent learning strategy becomes more evident when extended to more complex robotic tasks, such as manipulation involving deformable objects, tool use, or contact-rich interactions, where discrepancies between simulated and real-world physics become more noticeable. In such scenarios, learning from real-world data becomes crucial for bridging the reality gap, while simulation continues to provide large-scale data for rapid iteration and policy refinement.

 doi:10.20944/preprints202507.0012.v1

10.3.2. Multi-Task Learning

To demonstrate one of the multi-robot/task learning approaches, the previous experiment was extended by incorporating all the created task environments of Rx200 Reacher and Ned2 Reacher into the environments array A of **Algorithm 1**. Then, by sampling experience from multiple environments, the agent was exposed to multiple tasks across different domains (robot types and physical/simulated instances), enabling it to learn optimal behavior under varying conditions. Furthermore, to accommodate the differences in observation and action spaces between the Rx200 and Ned2 Reacher environments, the agent's architecture was configured using the largest observation space among the environments (from the Rx200 Reacher). For environments with smaller observation spaces, zero-padding was applied to match the input dimensions (for the Ned2 Reacher). Similarly, the largest action space across the environments (from the Ned2 Reacher) was used as the action dimension for the agent, and any additional unused actions were simply ignored when executing actions in environments with smaller action spaces (for Rx200 Reacher). This design choice ensured compatibility across heterogeneous environments while maintaining a shared policy architecture.

Algorithm 1: Multi-Task Training Strategy for TD3/TD3+HER

Initialize the TD3 or TD3+HER agent.
Initialize replay buffer R (HER for Goal Envs)
Initialize an array A of real and sim environments.
for episode = 1 **to** M (max episodes) **do**
  Sample an environment instance $I \in A$
  Sample a goal position g_t and an initial state s_0
  for step = 1 **to** *T-1* (max steps per episode) **do**
    Sample action $a_t \leftarrow \pi(\hat{s}_t)$ using the actor-network with exploration noise.
    Execute action a_t, receive reward r_t, new state s_{t+1} and augmented state $\hat{s}_{t+1}$ and done d_t
    Add $(\hat{s}_t, a_t, r_t, \hat{s}_t + 1, d_t)$ into R
  end for
  Augment R with pseudo-goals (Only TD3+HER)
  Update the actor and critic networks using samples. from R
end for

Although more sophisticated techniques, such as using task-conditioned policies with task embeddings [54] or adaptation layers [55], could help manage heterogeneous input/output structures more elegantly, a simpler approach was deliberately chosen to keep the training pipeline minimal. This approach of padding and unifying action and observation spaces has also been used in prior multi-task reinforcement learning research [56] and serves as a reasonable baseline when dealing with a limited number of tasks and known environment interfaces.

Similar to the previous use cases, the training curves for this experiment were plotted and are shown in **Figure 15**. As before, it was observed that the training remained stable across both types of environments, and the agents successfully learned policies that converged toward optimal behavior. The advantage of this approach lies in the agent's ability to learn a single policy that generalizes well across multiple tasks and domain configurations, ultimately saving time and computational resources by avoiding separate training for each environment.

 doi:10.20944/preprints202507.0012.v1

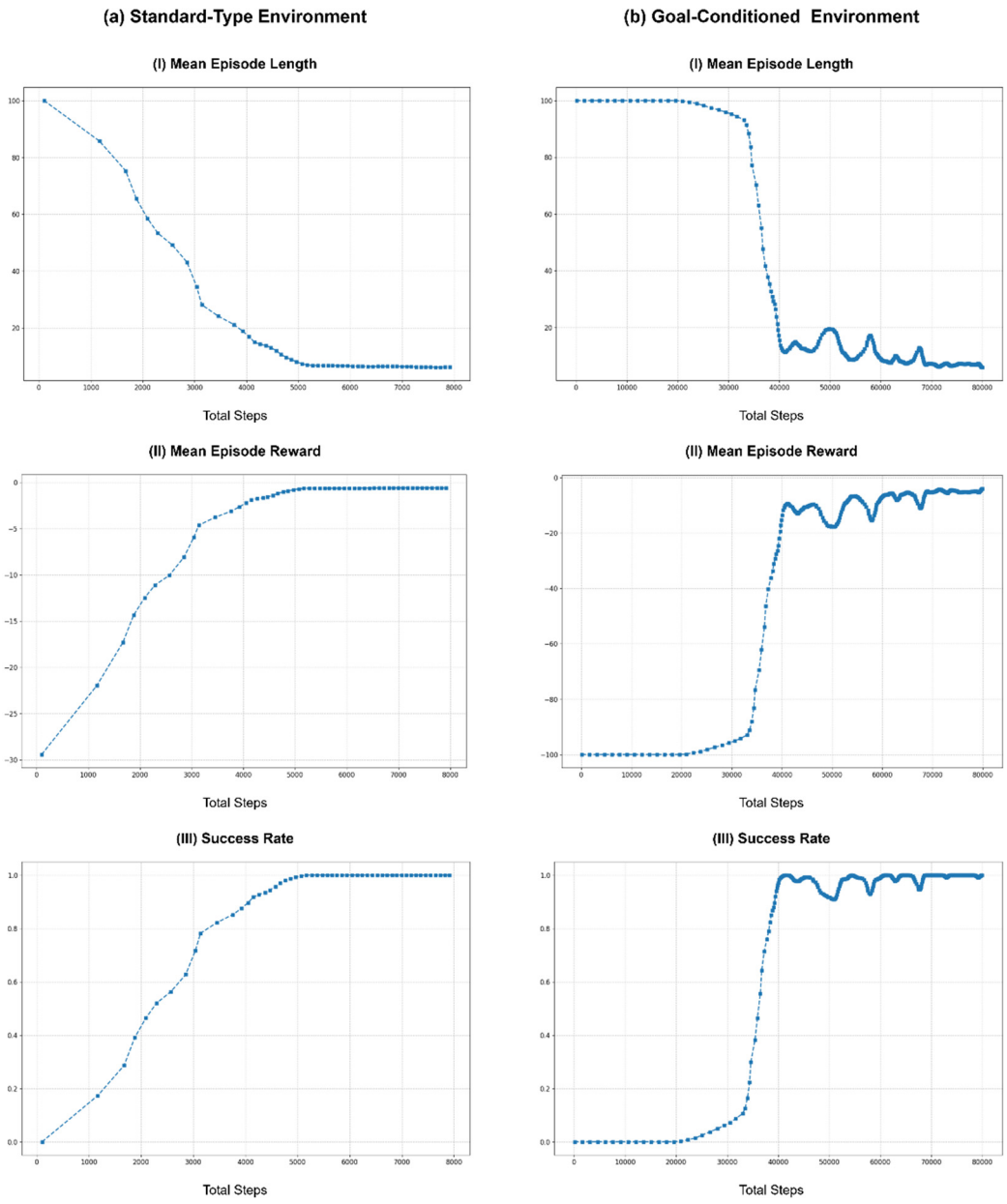


Figure 15. Learning metrics for multi-task learning during training across multiple environments comprising both the Rx200 and Ned2 Reacher tasks in simulated and real domains. (a) shows performance using standard dense-reward environments trained with the TD3 algorithm, while (b) shows performance using goal-conditioned sparse-reward environments trained with TD3+HER. These results demonstrate stable convergence and effective policy generalization across heterogeneous robot platforms and domains.

While the deployment of the trained agent in each environment showed similar 100% accuracy as with the previous use cases, it is important to note that these results are specific to simple reach tasks, which have low-dimensional state and action spaces and minimal domain discrepancy. As more complex tasks are introduced, techniques like task embeddings, modular policies [57], or adaptation layers may become necessary to manage the increased diversity in task structure and dynamics. Similarly, meta-learning algorithms such as Model Agnostic Meta-learning (MAML) [58] may enable learning a model that can quickly adapt to new tasks with minimal data when the tasks are complex.

In summary, these experiments demonstrate the capability of the UniROS framework for training policies that perform well in both simulated and real-world environments. The ability to leverage both domains during training using concurrent environments provides a robust solution for developing versatile and adaptive robotic systems. The primary goal of these demonstrations is not

 doi:10.20944/preprints202507.0012.v1

to determine the best method for setting up learning tasks in the real world. Instead, it showcases the versatility and robustness of the UniROS framework, providing a platform for researchers to extend it further and adapt it to their specific research needs. Therefore, these use cases demonstrate the framework's current capabilities and lay the groundwork for future research in more complex areas, such as robot-based multi-agent systems, multi-task learning, and meta-learning.

11. ROS1 vs ROS2 Support in the UniROS Framework

While this study is primarily based on the first major version of the Robot Operating System (ROS 1), it is nearing the end of life (EOL) in 2025, as ROS 2 is emerging as the new standard. Hence, plans to upgrade the package to support ROS 2 are currently underway, starting with ROS 2 Humble [36] with Gazebo Classic (Gazebo 11). However, it is essential not to abandon ROS1 prematurely, as many robots in the ROS industrial repository and other ROS sensor packages have not yet been upgraded to ROS2, or they remain unstable at present. Similarly, only the latest robots are currently configured to work with ROS2, while some older generations are either discontinued, not maintained by the manufacturer, or abandoned due to the companies having closed down (such as the Baxter robot). Therefore, experimenters may need to wait for the ROS community to step up and upgrade these packages to ROS 2.

Moreover, in terms of the real-time learning strategy proposed in this study, it is worth noting that the design choices made, such as ROS timers, ROS publishers and subscribers, and action repeats, are inherently compatible with both ROS 1 and ROS 2. These components form part of the core middleware tools that remain largely consistent across both distributions. Consequently, the implementation strategy outlined in this work can be readily adapted to ROS2 with minimal modification, preserving its real-time characteristics and concurrent environment support.

12. Conclusion & Future Work

This study introduced a ROS-based unified framework designed to bridge the gap between simulated and real-world environments in robot-based reinforcement learning. The dual-package approach of the framework, which contains MultiROS for simulations and RealROS for the real world, facilitates learning across simulated and real-world scenarios by utilizing a ROS-centric environment implementation strategy. This implementation strategy aids real-time decision-making by employing built-in multi-threading capabilities of ROS for subscribers and timers, supporting asynchronous and concurrent execution of operations necessary for efficient RL workflows. By using this approach and controlling robots directly via ROS's low-level control interface, this study has effectively tackled the challenges of ROS-based real-time learning. This was demonstrated using a benchmark learning task, highlighting the low latency and dynamic interaction between the agent and the environment.

Furthermore, the OpenAI Gym library has been deprecated recently and is now replaced by the Gymnasium [37] [59] library. Therefore, the UniROS framework has been upgraded to support Gymnasium, including the ROS-based support package for the Gymnasium-based Stable Baselines3 versions. For simplicity, only OpenAI Gym is referenced in the above sections.

Additionally, it is worth noting that all experiments focused on learning a simple reach task, and training with multiple complex tasks that incorporate external sensors, such as vision sensors, was not explored in this study. Therefore, one potential area for future work is to assess the framework's capabilities for complex multi-task and meta-learning scenarios across various benchmark task environments. Furthermore, it is essential to acknowledge that the robot was connected to the PC via a wired connection in the experiments, ensuring a stable and reliable communication interface. It was beneficial for conducting experiments in a controlled manner but may not accurately reflect the

[36] https://docs.ros.org/en/humble/index.html

[37] https://gymnasium.farama.org/

 doi:10.20944/preprints202507.0012.v1

challenges experimenters may encounter with wireless or less stable connections. Therefore, another area for future work could be to explore the implications of varying connection types on the performance and reliability of the framework.

Furthermore, the robot used in this experiment is configured with effort controllers that accept *Joint Trajectory* messages. However, some robots, such as the UR5e, can be configured to work with effort, position, or velocity controllers, where each controller type has its advantages and disadvantages, depending on the task requirements and the specific capabilities of the robot. Therefore, another area for future work could be to investigate the impact of using different types of controllers to learn the same benchmark task. This would provide valuable insights into studying the effects of controller differences on learning performance.

Moreover, the study primarily discussed CPU utilization during learning and interaction with the environment. However, given that many deep RL architectures leverage GPU acceleration, future experiments should explore CPU-GPU co-utilization metrics. Understanding how GPU-based training and inference affect the timing and synchronization of agent-environment loops could further improve the real-time applicability of the framework. Similarly, future work could involve exploring other reinforcement learning algorithms and off-the-shelf RL library frameworks that are better suited for real-time control, thereby further enhancing the performance and reliability of the proposed solution.

While this study focused on manipulation tasks, the proposed framework is not limited to stationary arms. The same ROS-based abstraction can be extended to mobile robots, drones, and other robotic systems that support ROS interfaces. Therefore, future use cases could include mobile navigation, exploration, and hybrid scenarios involving both mobility and manipulation. Examining the impact of such configurations on learning efficiency, task performance, and system stability would provide valuable insights for designing robust ROS-based multi-robot RL systems.

In conclusion, the proposed ROS-based RL framework addresses the challenges in bridging simulated and real-world RL environments. Its modular design, support for concurrent environments, Python bindings for ROS, and real-time RL environment implementation strategy collectively enhance robotic reinforcement learning tasks' efficiency, flexibility, reliability, and scalability. The experiments done with the benchmark task further illustrate the practical applicability of the framework in real-world robotics, showcasing its potential in advancing the field of reinforcement learning. Therefore, we encourage the community to build upon this foundation, exploring more intricate tasks and environments and pushing the boundaries of what is achievable in robotic reinforcement learning.

Funding: "This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number SFI 16/RC/3918, co-funded by SFI 22/CC/11147, the Department of Agriculture, Food and the Marine's Competitive Research Funding Programme (Grant Number 2024ICTAGRIFOOD852), and the European Regional Development Fund."

Data Availability Statement: The source code for the UniROS framework, along with all associated simulation environments, experimental scripts, and configuration files used in this study, is publicly available on GitHub at:

- https://github.com/sri-tus-ie/UniROS
- https://github.com/ncbdrck/rl_environments

Acknowledgments: The authors would like to thank the Software Research Institute (SRI) at the Technological University of Shannon (TUS) for their support throughout the development of this framework. Special thanks to the maintainers of ROS and the open-source robotics community for their invaluable tools and libraries that enabled this work. The experiments conducted with the NED2 and ReactorX200 robots were made possible by the infrastructure provided through the SRI's robotics lab.

Abbreviations

The following abbreviations are used in this manuscript:

 doi:10.20944/preprints202507.0012.v1

ROS	Robot Operating System
RL	Reinforcement Learning
MDP	Markov Decision Process
CLI	Command Line Interface
API	Application Programming Interface
DRL	Deep Reinforcement Learning
DQN	Deep Q-Network
TD3	Twin Delayed Deep Deterministic Policy Gradient
SB3	Stable Baselines3
GIL	Global Interpreter Lock
URDF	Universal Robot Description Format
IK	Inverse Kinematics
FK	Forward Kinematics
EE	End-Effector
DOF	Degree of Freedom
HER	Hindsight Experience Replay
SSH	Secure Shell
PC	Personal Computer
VRAM	Video Random Access Memory
GPU	Graphics Processing Unit
CPU	Central Processing Unit

Appendix A

In this study, four RL environments were designed for each robot using the UniROS framework. The two real-world environments were created using the proposed RealROS package, and the two simulation environments were created using the MultiROS package. **Table A1** and **Table A2** show an overview of the environments for respective robots.

Table A1. Overview Of the Rx200 Reacher Environments

	Standard Env	Goal Env	
Action Space	Joint positions 5 actions (Continuous)		
Observation	EE position	3	
	Cartesian Displacement	3	
	Euclidean distance (EE to Reach goal)	1	
	Current Joint values	8	
	Previous action	5	
Achieved Goal	N/A	EE position	3
Desired Goal	N/A	Goal position	3
Reward Architecture	Dense Sparse	Dense Sparse	

Table A2. Overview Of the Ned2 Reacher Environments

	Standard Env	Goal Env
Action Space	Joint positions 6 actions (Continuous)	
Observation	EE position	3

 doi:10.20944/preprints202507.0012.v1

	Cartesian Displacement	3	
	Euclidean distance (EE to Reach goal)	1	
	Current Joint values	6	
	Previous action	6	
Achieved Goal	N/A	EE position	3
Desired Goal	N/A	Goal position	3
Reward Architecture	Dense Sparse	Dense Sparse	

In the environments, the 3D position of the end-effector (x_a, y_a, z_a) and the current goal (x_g, y_g, z_g) are used to find the relative cartesian displacement. This is found using the following equations.

$$Displacement\ EE\ to\ Goal = Current\ Goal - Current\ EE\ Position$$

The Euclidean distance (d_e) from the current position of the end-effector to the current goal position is calculated as follows.

$$Sparse\ Reward = \begin{cases} +1 & if\ d_e \leq \varepsilon_r (reach\ tolerance) \\ -1 & otherwise \end{cases}$$

Since this study is based on ROS-based environments, the dense reward architecture differs from the others, where they only use the reward as a negative value of the Euclidean distance between the end-effector and goal positions. The dense reward architecture of the Reacher task is depicted in **Algorithm A1.**

Algorithm A1: Dense Reward Architecture

Initialize the reward. $r = 0$

Check if the end effector has reached the goal (done)

done = $d_e \leq \varepsilon_r$ (reach tolerance)

if the end-effector reached the goal, **then**

  Update the reward: r += τ_g (reached goal reward)

else

  r += -τ_d (distance reward) * d_e

  r += τ_s (constant step reward)

end if

Check if the joint positions are within the limits.

if not within limits, **do**

  r += τ_l (joint limits reward)

end if

Check if the movement execution was unsuccessful.

if unsuccessful **then**

  r += τ_e (execution failed reward)

end if

Return the reward: r

 doi:10.20944/preprints202507.0012.v1

Here, in the above dense reward architecture, the reached goal reward τ_g is set to 20.0, multiplier distance reward τ_d to 2.0, constant step reward τ_s to -0.5, not within joint limits reward τ_l to -2.0 and execution failed reward τ_e to -5.0. Furthermore, this reach task is episodic and terminates when $d_e \leq \varepsilon_r$ or the agent exceeds the maximum allowed steps in the environment. Additionally, to get the success rate graphs, the environment set $info['is\_success'] = True$ when the task is done successfully and $info['is\_success'] = False$ otherwise.

Appendix B

This section presents the hyperparameters used in experiments to evaluate the real-time environment implementation strategy and three use cases of the proposed framework. Here, all standard-type environments with dense rewards were trained using the TD3 algorithm, and goal-conditioned environments with sparse rewards were trained using the TD3+HER algorithm. Furthermore, to be consistent, the same hyperparameters in **Table B1** were used for all TD3 and those in **Table B2** for TD3+HER implementations.

Table B1: Hyperparameters for TD3 Implementation

Actor and Critic learning rate	0.0003
Size of the replay buffer	1000000
Minibatch size for each gradient update	256
Soft update coefficient	0.005
Discount factor	0.99
Entropy regularization coefficient	0.01
Maximum episode time steps	100
Success distance threshold	0.02m
hidden layers	400, 300
Optimizer	Adam

Table B2: Hyperparameters for TD3+HER Implementation

Actor and Critic learning rate	0.0003
Size of the HER replay buffer	1000000
Goal selection strategy	future
Number of future goals	4
Minibatch size for each gradient update	256
soft update coefficient	0.005
Discount factor	0.99
Entropy regularization coefficient	0.01
Maximum episode time steps	100
Success distance threshold	0.02m
Hidden layers	400, 300
Optimizer	Adam
Policy delay	2
Target policy noise (smoothing noise)	0.2
Limit for the absolute value of target policy smoothing noise	0.5

 doi:10.20944/preprints202507.0012.v1

Appendix C

In all three use cases in Section **10**, the learning was stable, and each achieved a near-optimal policy within 10K steps for standard-type environments and 60K for goal-conditioned environments. This is primarily due to the efficacy of the proposed RL environment implementation strategy and the stability of the TD3 algorithm. It should be noted that the distinction in learning progress between standard-type and goal-conditioned environments does not indicate one being superior to the other. The main reason for this disparity is due to the selected reward structure of each environment type, as it dictates the pace of learning. Therefore, the faster learning time (fewer number of steps) for the standard-type environments can be attributed to the dense rewards providing more frequent and informative feedback to the agent, which helps with quicker convergence and policy optimization compared to goal-conditioned environments with sparse rewards. The main benefit of using sparse rewards in robotics is that they encourage the development of more robust and generalizable policies due to the agent exploring more to achieve the ultimate task goal rather than optimizing for intermediate rewards [60]. This makes learning behaviors more adaptable to real-world applications. Moreover, carefully hand-coding the dense reward architecture may not be feasible when the task is complex, as with many robot applications.

References

1. Toner, T.; Saez, M.; Tilbury, D.M.; Barton, K. Opportunities and Challenges in Applying Reinforcement Learning to Robotic Manipulation: An Industrial Case Study. *Manuf. Lett.* **2023**, *35*, 1019–1030, doi:10.1016/j.mfglet.2023.08.055.
2. Ibarz, J.; Tan, J.; Finn, C.; Kalakrishnan, M.; Pastor, P.; Levine, S. How to Train Your Robot with Deep Reinforcement Learning: Lessons We Have Learned. *Int. J. Robot. Res.* **2021**, *40*, 698–721, doi:10.1177/0278364920987859.
3. Chebotar, Y.; Handa, A.; Makoviychuk, V.; Macklin, M.; Issac, J.; Ratliff, N.; Fox, D. Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience. In Proceedings of the 2019 International Conference on Robotics and Automation (ICRA); May 2019; pp. 8973–8979.
4. Bousmalis, K.; Irpan, A.; Wohlhart, P.; Bai, Y.; Kelcey, M.; Kalakrishnan, M.; Downs, L.; Ibarz, J.; Pastor, P.; Konolige, K.; et al. Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping. In Proceedings of the 2018 IEEE International Conference on Robotics and Automation (ICRA); May 2018; pp. 4243–4250.
5. Dulac-Arnold, G.; Levine, N.; Mankowitz, D.J.; Li, J.; Paduraru, C.; Gowal, S.; Hester, T. Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks and Analysis. *Mach. Learn.* **2021**, *110*, 2419–2468, doi:10.1007/s10994-021-05961-4.
6. Sutton, R.S.; Barto, A.G. *Reinforcement Learning, Second Edition: An Introduction*; MIT Press, 2018; ISBN 978-0-262-35270-3.
7. Rupam Mahmood, A.; Korenkevych, D.; Komer, B.J.; Bergstra, J. Setting up a Reinforcement Learning Task with a Real-World Robot. In Proceedings of the 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS); October 2018; pp. 4635–4640.
8. Fan, T.; Long, P.; Liu, W.; Pan, J. Distributed Multi-Robot Collision Avoidance via Deep Reinforcement Learning for Navigation in Complex Scenarios. *Int. J. Robot. Res.* **2020**, *39*, 856–892, doi:10.1177/0278364920916531.
9. Gleeson, J.; Gabel, M.; Pekhimenko, G.; de Lara, E.; Krishnan, S.; Janapa Reddi, V. RL-Scope: Cross-Stack Profiling for Deep Reinforcement Learning Workloads. *Proc. Mach. Learn. Syst.* **2021**, *3*, 783–799.
10. Wiggins, S.; Meng, Y.; Kannan, R.; Prasanna, V. Evaluating Multi-Agent Reinforcement Learning on Heterogeneous Platforms. In Proceedings of the Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications V; SPIE, June 12 2023; Vol. 12538, pp. 501–505.
11. Wu, Y.; Yan, W.; Kurutach, T.; Pinto, L.; Abbeel, P. Learning to Manipulate Deformable Objects without Demonstrations Available online: https://arxiv.org/abs/1910.13439v2 (accessed on 4 December 2023).
12. Zamora, I.; Lopez, N.G.; Vilches, V.M.; Cordero, A.H. Extending the OpenAI Gym for Robotics: A Toolkit for Reinforcement Learning Using ROS and Gazebo. *ArXiv160805742 Cs* **2017**.

13. Fajardo, J.M.; Roldan, F.G.; Realpe, S.; Hernández, J.D.; Ji, Z.; Cardenas, P.-F. FRobs_RL: A Flexible Robotics Reinforcement Learning Library. In Proceedings of the 2022 IEEE 18th International Conference on Automation Science and Engineering (CASE); August 2022; pp. 1104–1109.
14. Deisenroth, M.P.; Englert, P.; Peters, J.; Fox, D. Multi-Task Policy Search for Robotics. In Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA); May 2014; pp. 3876–3881.
15. Alet, F.; Lozano-Perez, T.; Kaelbling, L.P. Modular Meta-Learning. In Proceedings of the Proceedings of The 2nd Conference on Robot Learning; PMLR, October 23 2018; pp. 856–868.
16. Mnih, V.; Badia, A.P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous Methods for Deep Reinforcement Learning. In Proceedings of the Proceedings of The 33rd International Conference on Machine Learning; PMLR, June 11 2016; pp. 1928–1937.
17. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-Level Control through Deep Reinforcement Learning. *Nature* **2015**, *518*, 529–533, doi:10.1038/nature14236.
18. Fujimoto, S.; Hoof, H.; Meger, D. Addressing Function Approximation Error in Actor-Critic Methods. In Proceedings of the Proceedings of the 35th International Conference on Machine Learning; PMLR, July 3 2018; pp. 1587–1596.
19. Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A.; Ernestus, M.; Dormann, N. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *J. Mach. Learn. Res.* **2021**, *22*, 268:12348-268:12355.
20. Weng, J.; Chen, H.; Yan, D.; You, K.; Duburcq, A.; Zhang, M.; Su, Y.; Su, H.; Zhu, J. Tianshou: A Highly Modularized Deep Reinforcement Learning Library. *J Mach Learn Res* **2022**, *23*.
21. Schaarschmidt, M.; Kuhnle, A.; Ellis, B.; Fricke, K.; Gessert, F.; Yoneki, E. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations 2018.
22. Huang, S.; Dossa, R.F.J.; Ye, C.; Braga, J.; Chakraborty, D.; Mehta, K.; Araújo, J.G.M. CleanRL: High-Quality Single-File Implementations of Deep Reinforcement Learning Algorithms. *J. Mach. Learn. Res.* **2022**, *23*, 1–18.
23. Liang, E.; Liaw, R.; Nishihara, R.; Moritz, P.; Fox, R.; Goldberg, K.; Gonzalez, J.; Jordan, M.; Stoica, I. RLlib: Abstractions for Distributed Reinforcement Learning. In Proceedings of the Proceedings of the 35th International Conference on Machine Learning; PMLR, July 3 2018; pp. 3053–3062.
24. Kormushev, P.; Calinon, S.; Caldwell, D.G. Reinforcement Learning in Robotics: Applications and Real-World Challenges. *Robotics* **2013**, *2*, 122–148, doi:10.3390/robotics2030122.
25. Gomes, N.M.; Martins, F.N.; Lima, J.; Wörtche, H. Reinforcement Learning for Collaborative Robots Pick-and-Place Applications: A Case Study. *Automation* **2022**, *3*, 223–241, doi:10.3390/automation3010011.
26. Liu, D.; Wang, Z.; Lu, B.; Cong, M.; Yu, H.; Zou, Q. A Reinforcement Learning-Based Framework for Robot Manipulation Skill Acquisition. *IEEE Access* **2020**, *8*, 108429–108437, doi:10.1109/ACCESS.2020.3001130.
27. Bengio, Y.; Louradour, J.; Collobert, R.; Weston, J. Curriculum Learning. In Proceedings of the Proceedings of the 26th Annual International Conference on Machine Learning; Association for Computing Machinery: New York, NY, USA, June 14 2009; pp. 41–48.
28. Beltran-Hernandez, C.C.; Petit, D.; Ramirez-Alpizar, I.G.; Harada, K. Accelerating Robot Learning of Contact-Rich Manipulations: A Curriculum Learning Study 2022.
29. Santos Pessoa de Melo, M.; Gomes da Silva Neto, J.; Jorge Lima da Silva, P.; Natario Teixeira, J.M.X.; Teichrieb, V. Analysis and Comparison of Robotics 3D Simulators. In Proceedings of the 2019 21st Symposium on Virtual and Augmented Reality (SVR); October 2019; pp. 242–251.
30. Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. OpenAI Gym 2016.
31. Ferigo, D.; Traversaro, S.; Metta, G.; Pucci, D. Gym-Ignition: Reproducible Robotic Simulations for Reinforcement Learning. In Proceedings of the 2020 IEEE/SICE International Symposium on System Integration (SII); January 2020; pp. 885–890.