Ark: An Open-source Python-based Framework for Robot Learning

Magnus Dierking[1], Christopher E. Mower[2,⋆], Sarthak Das[2], Huang Helong[2], Jiacheng Qiu[1,2], Cody Reading[2], Wei Chen[2,3], Huidong Liang[2,4], Huang Guowei[2], Jan Peters[1], Quan Xingyue[2], Jun Wang[5,⋆], Haitham Bou-Ammar[2,5,⋆]

[1] Technical University of Darmstadt [2] Huawei Noah's Ark [3] Imperial College London
[4] University of Oxford [5] University College London
⋆ Corresponding authors: {christopher.mower, haitham.ammar}@huawei.com, jun.wang@cs.ucl.ac.uk

Abstract: Robotics has made remarkable hardware stridesfrom DARPA's Urban and Robotics Challenges to the first humanoid-robot kickboxing tournament-yet commercial autonomy still lags behind progress in machine learning. A major bottleneck is software: current robot stacks demand steep learning curves, low-level C/C++ expertise, fragmented tooling, and intricate hardware integration, in stark contrast to the Python-centric, well-documented ecosystems that propelled modern AI. We introduce Ark, an open-source, Python-first robotics framework designed to close that gap. Ark presents a Gym-style environment interface that allows users to collect data, preprocess it, and train policies using state-of-the-art imitation-learning algorithms (e.g., ACT, Diffusion Policy) while seamlessly toggling between high-fidelity simulation and physical robots. A lightweight clientserver architecture provides networked publisher-subscriber communication, and optional C/C++ bindings ensure real-time performance when needed. Ark ships with reusable modules for control, SLAM, motion planning, system identification, and visualization, along with native ROS interoperability. Comprehensive documentation and case studiesfrom manipulation to mobile navigationdemonstrate rapid prototyping, effortless hardware swapping, and end-to-end pipelines that rival the convenience of mainstream machine-learning workflows. By unifying robotics and AI practices under a common Python umbrella, Ark lowers entry barriers and accelerates research and commercial deployment of autonomous robots.

A long-standing goal for the robotics community has been to move robots out of controlled laboratory environments and deploy them commercially in the real world. There have been several impressive demonstrations over the years of robots operating in challenging environments, such as: the 2007 DARPA Urban Challenge, devised to improve self-driving vehicles that can navigate busy city streets and interact safely with other cars in real time; the 2015 DARPA Robotics Challenge, created to develop semi-autonomous ground robots able to handle intricate tasks in hazardous, human-built environments; and, in 2025, the world's first humanoid-robot kickboxing tournament showcased the impressive hardware capabilities of modern humanoid robots. Despite these, successful commercial applications of robotics are primarily found in car manufacturing, with little to no autonomous capabilities and relatively small domestic applications of autonomous robot vacuums and

hotel delivery robots.

Although robot hardware has advanced in recent years, the robot's ability to reason effectively in real-world environments remains a significant challenge. Machine learning solutions are generally considered by many to be the most promising avenue to resolve this issue. Therefore, many recent works focus on the integration of machine learning methods into robotic workflows. A key contributing factor to the success of machine learning is the abundance of Python-based high-quality open-source software. On the other hand, software for robotics is far more complex, requiring in-depth knowledge in several fields.

Robotics software has undergone significant evolution over the past several decades, yet it remains considerably more complex and fragmented than in fields such as machine learning. To understand the current challenges in robotics software development, it is useful to consider how robotics software has evolved. Early industrial robots in the 1960s were programmed via record-and-playback mechanisms, with no real software abstraction [1, 2]. Around the same time, academic robotics began exploring software-controlled systems: Shakey [1], for example, integrated symbolic AI planning, hierarchical control, and sensor feedback using LISP-based programs. The 1970s and 1980s saw the introduction of some of the first high-level robot programming languages (e.g., WAVE, AL, and RAPT) [2, 3], along with real-time embedded systems that enabled robots to respond to their environments. These developments improved modularity and sensing integration but were still limited by hardware specificity and lack of standardization.

In the 1990s, hybrid architectures combining deliberative planning and reactive control became more common, supported by libraries like the Robotics Toolbox [4]. This laid the groundwork for the 2000s emergence of middleware such as Player/Stage [5] and eventually ROS [6], which standardized communication and hardware abstraction across research platforms. Despite these advances, robotics software today remains complex and fragmentedoften requiring extensive C++ programming, specialized hardware drivers, and manual integration of learning components. Unlike machine learning frameworks, robotics lacks unified, Python-first tools that seamlessly support data collection, policy training, simulation, and deployment.

Over the past few decades, artificial intelligence has seen several significant advances, such as the advent of deep learning [7, 8] development of architectures such as convolutional and recurrent neural networks [9, 10], and more recently a class of models known as transformers have enabled many impressive results in a variety of fields such as image and natural language generation [11–13]. Due to the successes in other fields deep-learning has also shifted robotics research away from traditional model-based approaches [14, 15] and towards robot learning [16, 17]; see reviews by Kroemer et al. [18], Xiao et al. [19], Billard et al. [20].

High-quality, well-documented software frameworks–PyTorch [21], Scikit-learn [22], OpenAI Gym [23] and TensorFlow [24]–are one of the primary factors contributing to the success of machine learning: they are routinely used to teach university courses, drive the bulk of AI research output and power large-scale commercial products such as ChatGPT. With only a modest laptop, a newcomer can train their first neural network in under an hour by following the PyTorch quick-start tutorial, yet the same library also scales to the world's most advanced models [12, 13]. Although all of these frameworks can run on a CPU, serious work usually demands specialized GPU hardware, and transferring data between devices is

typically as simple as changing a single argument (e.g., the `device` flag in PyTorch). The seeming ease of swapping hardware, however, hides a crucial caveat: in practice the target hardware device is almost always an NVIDIA GPU running CUDA and cuDNN. Alternative back-ends exist but remain less mature, and the ubiquity of the CUDA softwarehardware stack has become so pronounced that many research projects and industrial pipelines now assume its presence by default [25]. This de-facto standardization has accelerated progress by letting researchers write portable code and cloud providers offer uniform, highly optimized GPU instances at scale [26, 27].

In contrast to the relative ease of developing and deploying machine learning applications, the development and deployment of robotics software is significantly more challenging for a variety of reasons–though this list is by no means exhaustive. (C1) Popular software frameworks for robotics (e.g., ROS and Orocos) require steep learning curves for novice users, generally due to lack of documentation [28], and are not readily integrated with tools for robot learning (e.g., data collection and processing, model training, and deployment). (C2) Unlike most machine learning packages–where you can build your application using entirely Python–robotics development still typically requires knowledge of C and C++. Most foundational robotics libraries exist only in those languages, exposing their functionality from Python requires skill with binding tools such as pybind11, Cython, or SWIG (among others). (C3) Robotics involves integrating a diverse range of hardware components–such as actuators, sensors, wheels, and onboard computers–each of which typically requires a custom driver interface or specific communication protocols, making it difficult to set up and switch between high-fidelity virtual simulations of custom robot setups. (C4) Developing robot systems require large teams with knowledge in many different fields such as control theory, kinematics and dynamics, motion planning, computer vision and perception, signal processing, machine learning, and electrical and mechanical engineering [29].

To address these challenges, we present ARK—a Python-based robotics framework that is designed to accelerate development and prototyping, and enable users to deploy new methods on both simulated and real world robots. ARK is designed to integrate effortlessly with standard machine-learning workflows: it lets you gather data from simulators or real robots, preprocess it, and train policies with state-of-the-art imitation-learning methods such as ACT and Diffusion Policy. The main user-interface is designed based on the OpenAI Gym to make it familiar for machine learning researchers, and also so it integrates with common machine learning workflows (e.g., imitation learning). Additionally, we use a client-based system where various Python nodes can communicate over a network using a publisher-subscriber architecture. Although ARK is designed as a Python-first framework, it also ships with utilities for exposing C/C++ functionality when performance matters. Comprehensive documentation and several examples show how ARK can be installed, setup, and deployed in simulated and real robot setups. For advanced users, ARK exposes native ROS bindings, allowing seamless integration with existing ROS codebases. Beyond the core environment interface, Ark provides reusable modules for low-level control, data collection, visualization, system identification, and mobile-base navigation. Several real-world and simulated case studies illustrate Ark's flexibility and ease of use. In this paper, we provide an overview of our proposed framework called Ark, outlining the design principles that shaped its development and detailing its core capabilities. We then demonstrate Ark's versatility through extensive use-cases that give details on how to setup several examples and demonstrate the ease in which one can switch between simulation and the real robot sys-

tem, various data-collection strategies for imitation learning, policy training with several established imitation-learning algorithms, map building via SLAM and motion planning for mobile robots, and finally we provide several embodied AI demonstrations.

Framework Overview

Machine learning, based on deep learning, has emerged as one of the most promising and effective approaches to enable intelligent robots to complete complex reasoning tasks. Although numerous studies report encouraging demonstrations, these systems have yet to progress beyond controlled laboratory settings into real-world deployments. A key limiting factor in this line of research is that there lacks a clear consensus on what constitutes an appropriate system architecture for embodied AI [30]. So far, recent works have largely converged on three alternative architectures. The first approach assumes a library of parameterized skills executable by the robot, with an LLM or VLM selecting the appropriate skill at each environment step [31, 32]. A second, trains a VLA model typically by fine-tuning a VLM to output directly actions that are executed on the robot [17]. A third, uses a VLA or VLM to output action tokens in some latent space which are then mapped by another (often smaller) model to robot control actions [33]. Moreover, for each architecture setup, there also exist many problems around training these models (e.g, scaling data collection, sim-to-real, generalization) and deploying the results on real hardware. In order to promote future research in embodied AI, Ark has been designed in such a way that is aligned with typical machine learning workflows and enables users to easily prototype novel architectures and deploy them on physical robots.

We have implemented Ark with three core design philosophies in mind. (D1) We design Ark's user interface to align with well-known machine learning libraries. Robotics inherently demands expertise from a wide range of disciplines. With the growing influence of machine learning in the field, many researchers and engineers specializing in machine learning are now focusing on its deployment in robotic systems. However, robotics software still lacks the maturity and standardization seen in machine learning libraries, making development and adoption more difficult. Ark aims to bridge this gap by offering a more familiar and accessible interface for those coming from a machine learning background. (D2) Developing and testing novel methods on real robots introduces a range of safety concerns; particularly when developers are in close proximity to the physical systems, as is commonly the case in robotics laboratories worldwide. While simulators provide a valuable environment for early-stage development, they often differ significantly in architecture from real-time, event-driven robotic systems. As a result, transitioning from simulation to physical deployment can be cumbersome and error-prone. Ark is designed to reduce this barrier by enabling seamless switching between simulated and real-world environments. (D3) Python is arguably the most used programming language in machine learning, and it is widely regarded as significantly more user-friendly than languages like C, C++, or Java. In addition, Python boasts a vast ecosystem of well-maintained libraries and packages, making it an ideal choice for rapid development and experimentation. We therefore have developed Ark with a Python-centric focus. We acknowledge that in some cases, robot software is required to operate at high frequencies that is challenging for Python to handle (e.g., low-level controllers for dynamic tasks such as locomotion). In these cases, we

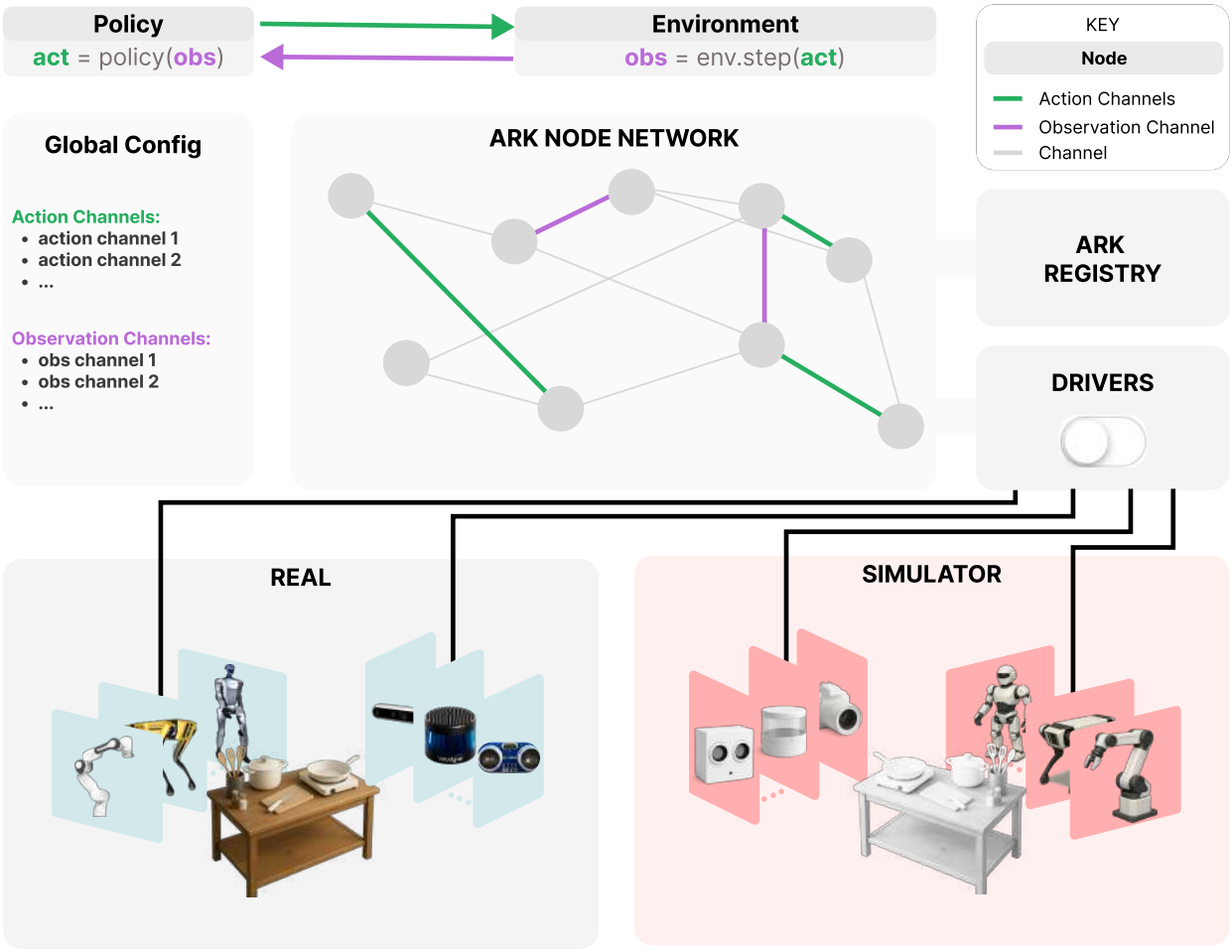

Figure 1. Ark uses a unified configuration file to define action and observation channels, which are then instantiated within a distributed node-based network. This architecture supports both real and simulated hardware through interchangeable drivers and identical communication interfaces. The Ark Registry manages active nodes, while each component (e.g., sensors, actuators, policies) operates as an independent process. As a result, pipelines developed in simulation can also be used on physical systems without code modification, ensuring a consistent sim-real interface.

offer a range of tools that help users to expose C/C++ code to Python.

The key aspects of the Ark framework are shown diagrammatically in Figure 1. The following sub-sections provide an overview of the main features of Ark.

Ark Network

A fundamental software design principle is modularity, which promotes maintainability, code re-use, and fault isolation [34]. A robot system can be divided into specialized tasks such as data acquisition, state estimation, task planning, and control. The system's modules must communicate and exchange information to achieve these tasks. Modern operating systems facilitate this by enabling each module to run as a separate software process and communicate across a network, either on the same machine or distributed across different devices. Modularity and inter-process communication has been a long-standing practice in robotic software design [6, 35, 36].

Message channels Ark uses a publisher-subscriber asynchronous message passing system for interprocess communication. In the Ark Network (see the central part of Figure 1), each node represents a Python script running in its own process. These nodes are each given a unique name and communicate with one another through messaging channels. Messages are transmitted across channels, allowing nodes to exchange information within the network.

The Ark Network is created programatically, each Python script implements a class that inherits from the `BaseNode` class. Communication between nodes is established by calling the `create_publisher` and `create_subscriber` methods. The user can choose an appropriate name for the messaging channel and define it with a string. The full channel name is then given by `NODE_NAME/CHANNEL_NAME` where `NODE_NAME` is the name of the node and `CHANNEL_NAME` is given by the user.

The Ark Network low-level communication model is designed to be modular, allowing different networking libraries to be easily swapped out. Currently, Ark uses the Lightweight Communications and Marshalling (LCM) library [37] as its backend for network communication. LCM is a middleware system with support for multiple programming languages. We chose LCM for its lightweight design and its built-in tools for data collection, debugging, and introspection.

The modularity of our implementation of networking infrastructure is valuable, as it allows us to easily adapt the low-level networking layer to suit different design goals in the future, depending on research directions pursued by us or the broader community. In future versions of Ark, we plan to support distributed training and potentially inference for backpropagation-based models, enabling more advanced machine learning workflows across networked nodes; such a goal can not be easily and efficiently handled by LCM in its current implementation.

We follow the LCM type specification language to define message types; i.e., each message channel is defined by a name and a message type. We provide a library called `ark_types` containing many message types common to robotics (e.g., a `joint_state_t` type or a `transform_t` type).

Another advantage of using LCM is its ability to facilitate integration of low-level languages such as C, C++, or Java into the Ark framework. Since nodes in Ark communicate via

LCM messaging channels, scripts written in alternative languages can interact by utilizing the standard LCM publisher/subscriber interface to communicate over the network. This approach can be useful in scenarios involving hardware devices–such as haptic interfaces–that are only accessible through C/C++/Java APIs provided by the manufacturer. In such cases, the user can expose the device to Ark by implementing the appropriate LCM publishers and subscribers. However, due to Ark's architectural design for coordinating between simulation and real-world environments (discussed later), using LCM as a bridge between alternative languages and Python may not always be the best choice. To address this, and recognizing that C/C++ are the dominant languages in hardware development, Ark also provides a set of tools and helper functions/classes to assist users in directly exposing C/C++ functionality to Python.

Services Asynchronous communication is not always suitable for all tasks. To address this, Ark provides a request-response mechanism known as services. This pattern ensures a clear association between requests and responses, making it ideal for operations that require acknowledgment–for example, triggering a calibration routine on a robotic arm. Ark services use the LCM message type specification for request and response types, allowing them to be selected dynamically. Similar to message channels, Ark services can be identified by a name, chosen by the user to suit the desired task.

In order to connect services over the network, Ark includes a central registry, that acts as a lightweight coordination and discovery hub. Information about the network can be retrieved (e.g., active services from other nodes) and enables various features such as runtime visualization and fault isolation. The registry has default service names, these are identified in list of services by the `__DEFAULT_SERVICE`. The Ark registry must be run by the user before the Ark Network can be established by other nodes.

Launcher As illustrated in Figure 1, multiple nodes can be executed and connected to form the overall Ark Network. Each node can be launched individually by starting its corresponding Python script from the terminal. However, manually launching a large number of processes can become cumbersome and error-prone. To address this, Ark provides a launcher utility that allows users to define the entire Ark Network within a single configuration file using the YAML format. This launcher script can then be executed once from the terminal to automatically start all specified subprocesses, simplifying the initialization of complex network setups.

Observation and Action Channels

We adopt the standard terminology from the reinforcement learning literature, referring to *observations* and *actions* as follows. An *observation* is information received from the environment, typically taken from sensors such as cameras or joint encoders. An *action* is a control signal or decision variable that influences the environment, for example by specifying joint velocities sent to a robot's actuators.

To promote ease of use and reduce the learning curve for users familiar with machine learning, Ark provides an interface inspired by the well-known OpenAI Gym (now Gymnasium) library. An environment class implements a `reset` function, which returns an observation and an information dictionary. Also a `step` function is given that accepts an action

as input and returns the next observation, a reward, termination and truncation flags, and an information dictionary.

We define observation and action spaces each by a collection of message channels running on the Ark Network. Each space is defined in the constructor of the environment class using a dictionary which maps the message channel name to it's type. Observation and action space classes are initialized as class attributes for the environment class that both automatically handle communication with the Ark network: the observation space subscribes to each channel, whereas the action space publishes to each given channel. By allowing the user to choose the observation/action spaces enables them to easily prototype different input/outputs for their policy model architecture. Also, each channel in the observation space, may be published at different sampling frequencies. Each observation returned thus contains the most recent message recieved on each channel.

Real World and Physics Simulation

As outlined in our three core design philosophies, the second principle (D2) focuses on ensuring that Ark can seamlessly operate across both real and simulated environments. This section provides further details on how this is achieved, including an overview of how Ark interfaces with various simulators.

Sim-Real switch A key capability of *Ark* is its ability to easily switch between simulated environments and real-world robotic systems using a single configuration flag, i.e., `sim = True, False`. This is made possible by Ark's distributed, node-based architecture, where each robot and sensor–whether physical or simulated–is implemented as an independent computational node. Ark distributes the physics simulators, using a configuration file as seen in Figure 2, which takes the chosen simulator and internally spins up nodes to mimic the interface from a real system. This ensures consistency across deployments and enabling transparent switching between them. Further details on hardware drivers are provided in the following section.

Simulator backend There are many physics simulators available to the robotics community, however there does not exist a single simulator that performs best in all desired features for all domains [38]. Each simulator is typically developed for a specific purpose or with a certain application in mind, e.g. manipulation, medical, marine, soft robotics, locomotion, etc. Therefore, instead of directly interfacing with a single simulator, we instead provide a simulator backend which enables users to integrate, in theory, any simulator they wish that fits their need. Currently, Ark supports PyBullet and MuJoCo due to their popularity in machine learning. In future work, we plan to integrate IssacSim, and will consider suggestions from the community.

The choice of backend used, as well as the switch between simulated and physical systems, is entirely managed through Ark's configuration infrastructure. By modifying a single YAML file, users can define the desired setup (e.g., real/sim and if sim then which simulator), and Ark will automatically initialize the appropriate drivers, ensuring consistent message schemas, channel names, and execution flow.

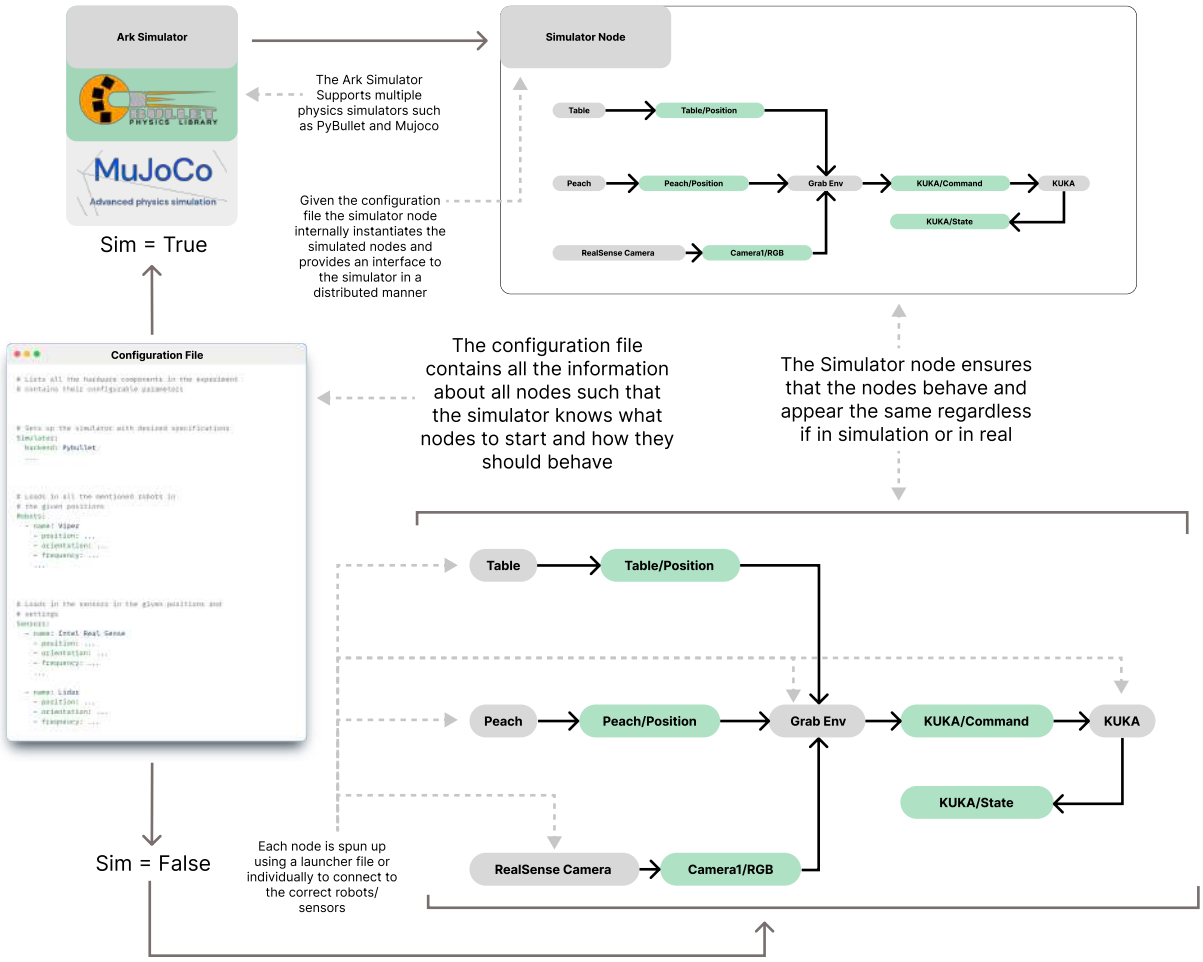

Figure 2. Technical diagram illustrating how Ark uses a unified configuration file to instantiate a distributed simulated system that mirrors real-world deployments. The YAML-based configuration specifies robots, sensors, environments, and networking parameters, which the Ark Simulator parses to launch corresponding simulated nodes. Each component, such as robot controllers, cameras, and sensor emulators, runs as an independent process, communicating through the same message passing protocol used in real deployments. This ensures that policies and pipelines developed in simulation operate identically when transferred to physical hardware, facilitating seamless sim-to-real transitions and reproducible experimentation

Robot and Sensor Drivers

While Ark provides some structure in its user interface (i.e., Gym-like interface as discussed above), it is very extensible and we have designed Ark with broad comparability across robots and sensors in mind. Recently developed frameworks such as LeRobot [39] and PyRobot [40], target only specific robot embodiments, Ark on the other hand is design to ensure that a wide range of hardware can be easily integrated. We enable generalizability across hardware by providing several interfacing methods.

Python drivers We provide an abstract Python base class, `ComponentDriver`, designed to standardize the integration of hardware components with Ark. To implement a driver, users create a subclass and override standard abstract methods such as `get_data` (for sensors) and `send_command` (for robots). Each driver integrates with Ark's global sim-real switch, automatically routing messages to either real or simulated hardware based on the global configuration settings.

C++ drivers As mentioned above, programming in C/C++ is sometimes necessary in robotic systems. For instance, certain hardware components provide only C/C++ interfaces, and in some cases, high sampling frequencies are required for real-time performance, such as in locomotion control. To support these scenarios, we provide a set of tools written in C++ using the `pybind11` library, enabling users to expose their hardware components to Ark. These tools ensure that hardware components with only C++ interfaces can be integrated with Ark in a consistent manner, following the same conventions as Python-based drivers.

ROS-Ark driver Arguably, ROS is the most widely used robotics platform today, with many research groups and developers around the world relying on it to build their robotic systems. In fact, some robots (e.g., ViperX arms) are only supported via manufacturer-provided ROS interfaces. To enable integration with the ROS ecosystem, Ark includes a ROS-Ark driver that facilitates bidirectional communication between ROS topics and Ark message channels. This allows users to run existing ROS setups while simultaneously leveraging Ark's interface, without requiring any modifications to the original ROS codebase. Moreover, the driver serves as a migration tool, helping users transition their existing ROS-based systems to Ark. Currently, based on our experience that most research labs (including our own) continue to use ROS 1, the ROS-Ark bridge supports only ROS 1. Support for ROS 2 may be considered in the future, depending on user demand.

Tools for Introspection and Debugging

Robot systems are often complex and rely on many intercommunicating processes. As a result, having a comprehensive suite of debugging tools, visualized in Figure 3, is essential, enabling users to efficiently investigate and resolve issues as they arise.

Ark Graph The Ark Graph tool offers a real-time visual representation of all active nodes, their published and subscribed channels, and available services.

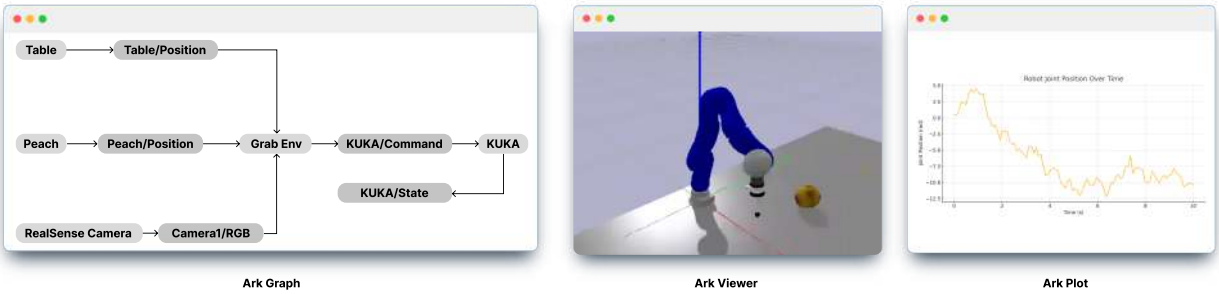

Figure 3. Graphical debugging tools provided by Ark. Ark Graph displays active nodes and their communication channels for network analysis. Ark Viewer renders live image streams to support camera calibration and inspection. Ark Plot visualizes real-time numerical data on any channel, aiding in system monitoring and debugging.

Ark Plot Ark Plot is a real-time plotting tool for visualizing numeric data on Ark message channels. It allows users to monitor the evolution of variables over time, making it useful for tasks such as tuning control parameters, diagnosing sensor behavior, and debugging system performance.

Ark Viewer Ark Viewer enables real-time visualization of image data transmitted over any LCM channel, making it especially valuable for configuring, monitoring, and debugging camera systems.

LCM Tools Another reason LCM was chosen as the communication library is the suite of built-in tools it offers for debugging and introspection. For example, `lcm-spy` is a graphical tool for viewing messages on an LCM network. Similar to network analysis tools like Ethereal/Wireshark or `tcpdump`, it allows users to inspect all received LCM messages and provides detailed information and statistics on the channels in use (e.g., number of messages received, message Rate in Hz, and jitter in ms).

Use Cases

In this section, we provide an overview of several use-cases for our proposed framework. These use-cases illustrate the ease of using and setting up Ark for multiple situations common to robot learning. All code to reproduce these use-cases will be made available.

Switching Between Simulation and Reality

Deploying learned robot policies in real-world scenarios can raise safety concerns. Moreover, many existing frameworks lack straightforward implementation pathways, resulting in adhoc and inconsistent solutions that do not generalize well across different systems or embodiments.

Ark addresses these challenges by providing a unified control interface built on Python/C++ drivers. It offers a configurable abstraction layer that enables seamless deployment of robotic policies across both simulated and physical environments. After defining

```yaml
# Lists all the hardware components in the experiment
# contains their configurable parameters


# Sets up the simulator with desired specifications
Simulator:
  backend: Pybullet
  ...


# Loads in all the mentioned robots in
# the given positions
Robots:
  - name: Viper
    - position: ...
    - orientation: ...
    - frequency: ...
    ...


# Loads in the sensors in the given positions and
# settings
Sensors:
  - name: Intel Real Sense
    - position: ...
    - orientation: ...
    - frequency: ...
    ...

  - name: Lidar
    - position: ...
    - orientation: ...
    - freqneucy: ...
```

Configuration

```python
env = ViperCokeCan()
state = env.reset()
SIM = True  # False

def policy(state):
    ...
    return action

while not done:
    action = policy(state)
    state, reward, done, _ = env.step(action)
```

Policy

```python
class ViperCokeCan(ArkEnv):
    def __init__(self):
        super().__init__(
            "Viper_Coke_Can",
            observation_channels=[("Viper/joint_commands", joint_commands_t)],
            action_channels=[("Viper/joint_states", joint_states_t)],
        )

    def step(self, action: Any, reward_function=None):
        # Log or print information before taking a step
        log.info(f"Taking step with action: {action}")

        # Takes the action and packes it into Ark Messages
        # Publishes the Ark Messages to the selected channels
        return super().step(action, reward_function)

    def reset(self):
        log.info("Resetting the environment")
        # Sends reset commands to all the robots and sensor
        return super().reset()
```

Environment

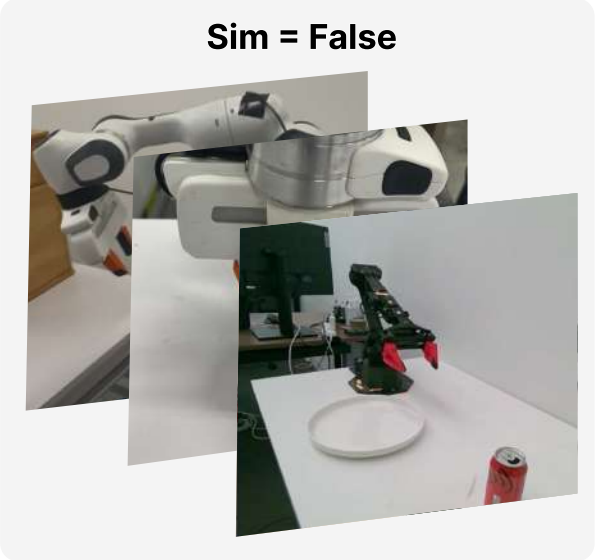

Figure 4. Seamless sim-real transition in Ark enabled by consistent observation and action space definitions. The environment configuration specifies sensor inputs (e.g., joint states, images) and actuator outputs (e.g., joint commands), which remain identical across both simulated and real systems. By toggling a single flag (sim = True/False), Ark automatically routes data through the different observation and action channels, allowing a single policy pipeline implementation to operate in both domains without modification. This unified interface simplifies development, debugging, and deployment across the sim-real boundary.

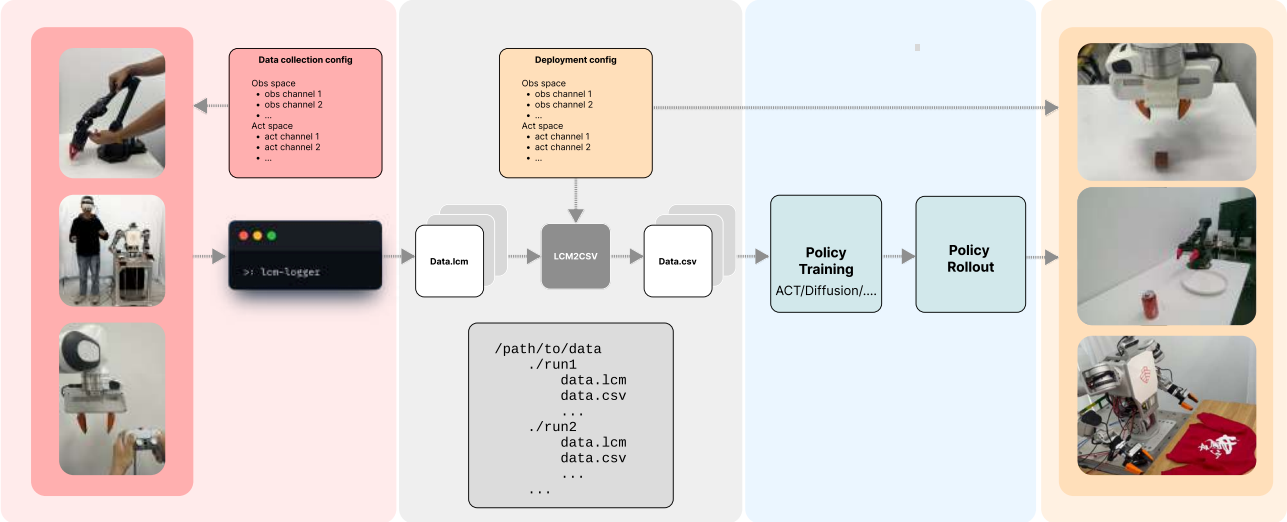

Figure 5. lcm-logger enables efficient data collection for imitation learning by recording demonstrations from a variety of control interfaces, including kinesthetic teaching, VR teleoperation, and gamepad input (left). Each demonstrations is saved as a separate CSV file (right), allowing users to accumulate diverse datasets across different input modalities rapidly.

the environment (i.e, robot, sensors, objects), the user need only specify a single flag in the configuration to switch between real and simulation, i.e., `sim = True, False`.

To demonstrate this facility, we implemented a pick-and-place task using a ViperX 300s fixed-base robotic arm equipped with a parallel gripper at the end-effector. The robot was tasked with picking an object and placing it onto a plate. The entire environment is specified by a single YAML file that defines object initial placements, camera and robot poses, as well as physics parameters such as gravity. This configuration is used by Ark to setup the physical environment (left part of Figure 4), and instantiate a simulation environment where the scene geometry mirrors the real-world setup (right part of Figure 4).

In this example, the observation space is the current joint position command, and the action space is the goal robot joint velocity command. A hand-crafted expert policy is used in both the real world and simulation.

Crucially, transitioning the same policy to the real robot required no changes to code or data structures–only the single configuration variable `sim` was toggled. Ark internally reroutes communication from the simulated drivers to the physical hardware while preserving the same observation and action channels. This abstraction ensures that all downstream code (e.g., the policy logic, environment wrappers, and logging infrastructure) remains unchanged, facilitating direct deployment and reproducibility.

Data Collection for Imitation Learning

Learning policies using imitation learning requires large demonstration datasets. Data collection typically requires humans to interact with the robot system to collect several demonstrations of a collection of tasks [41].

Two primary methods can be employed to collect data from the system: *kinesthetic teaching* and *teleoperation*. In kinesthetic teaching, the human physically interacts with

the robot to provide demonstrations (see the left part of Figure 5). This approach is generally considered intuitive for human operators [42] and is often preferred [43]; however, it may raise safety concerns due to the need for direct physical contact with the system. The second method, teleoperation, involves the human controlling the robot remotely via an interface such as a virtual reality headset and controllers or a gamepad (central and right parts in Figure 5). Teleoperation allows the operator to interact with the robot from a safe distance, but it introduces challenges such as limited visibility [44] and difficulty in mapping controller inputs to the robot's control dimensions [45]. As a result, effective teleoperation often requires a skilled operator [46].

Due to Ark's modular architecture and the use of distinct message types for each communication channel, setting up data collection pipelines is straightforward. This design enables users to easily swap out different interfaces as needed. LCM provides a utility called `lcm-logger`, which can be executed at any time to record data. It captures and writes all messages published on LCM channels to a log file. Ark includes built-in functionality to extract data from these log files and convert it into CSV format. Moreover, by utilizing the same observation and action space configuration defined in the environment class, the data can be extracted in a format consistent with that used during deployment. Since messages on different channels may be published at varying frequencies, Ark also provides tools to handle this asynchrony. These tools allow users to either interpolate the channel data or extract the most recent message at each time step.

Kinesthetic Teaching

Ark supports learning from human-guided demonstrations through kinesthetic teaching and also replay. Instead of deploying a pre-programmed expert policy, the ViperX 300s robotic arm is manually guided through the task by the human physically interacting with the system. During this process, the LCM logger records the channel data, capturing the full trajectory of the demonstration as it is executed. We also include a camera in the setup, whilst this is not used in the policy for this demonstration, it can be used to collect videos of the robot setup. Note, the observation and action spaces remain the same as in the previous section.

Once a demonstration is complete, the data is saved to a log file. This data can either be processed for use in training an imitation learning policy, or replayed on the system using the `lcm-logplayer`. This replay functionality allows the demonstration to be reproduced exactly as it was performed by the human demonstrator. Such playback is particularly useful in scenarios where the human appears in the camera frame–potentially introducing bias or noise into vision-based policies–or when force-based interactions are involved. In the latter case, it can be difficult to disentangle the forces that the robot should actively track from those resulting from physical contact during the demonstration. Additionally, we provide services to streamline environment resetting. For instance, users can link a reset service to a keyboard button press, making repeated demonstrations or evaluations more efficient and user-friendly.

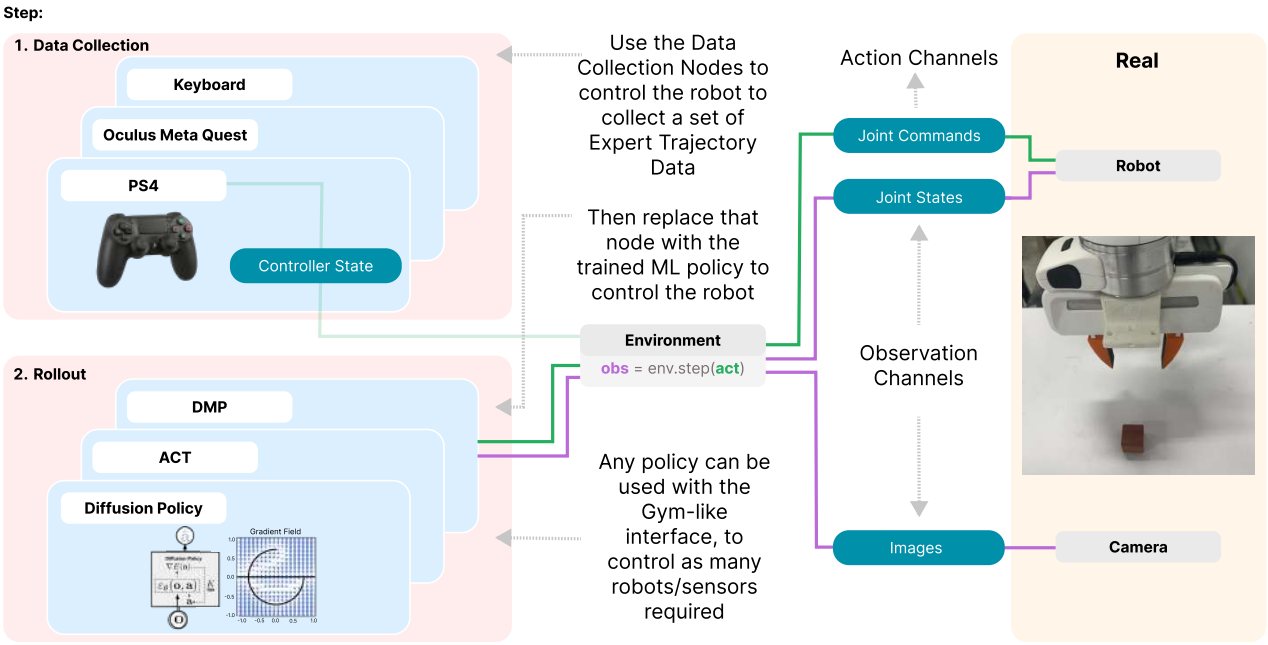

Figure 6. Ark's modular data collection nodes allow users to control a robot using various input devices (e.g., PS4 controller, VR headset) to generate expert demonstration trajectories. These nodes populate observation and action channels with data such as images, joint states, and commands. Once expert data is collected, the control node can be seamlessly replaced with a learned policysuch as a diffusion policyusing Ark's gym-like interface. This architecture supports flexible integration of any policy and hardware configuration, streamlining the transition from demonstration to autonomous control.

Teleoperation

Ark also supports teleoperation using input devices such as VR and gamepad controllers. In one setup (central part of Figure 5), a user controls the OpenPyRo-A1 humanoid robot by streaming, in real-time, 6-DoF poses from a VR controller over a local network. Using an inverse kinematic controller, these are converted to joint velocity goals that are sent to the robot actuators. A second setup (right part of Figure 5) has a similar architecture for the Ark Network, however in this case a PlayStation4 controller is used as the interface that controls the robot gripper pose.

Imitation Learning

In this section, we demonstrate several use-cases for implementing imitation learning. These use-cases focus on how to use Ark for data collection, training, and deployment. Also, we highlight the effectiveness of using the OpenAI Gym (Gymnasium) interface with Ark to allow easy adaptations to the policy. An overview of Ark's data collection nodes are shown in Figure 6.

We showcase two methods for imitation learning: (i) diffusion policy, (ii) action chunking with transformers, and Ark serves as the central infrastructure, providing modular components, standardized interfaces, and real-time communication, all of which streamline data collection and deployment of trained policies.

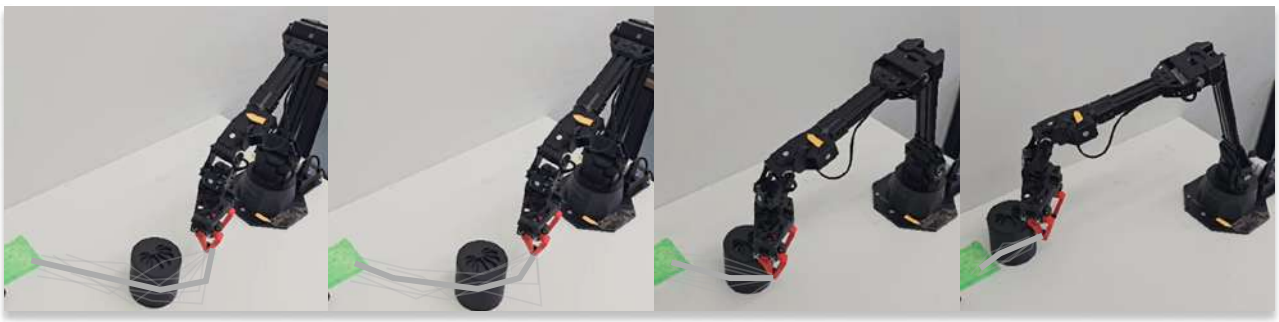

Figure 7. Sequential snapshots of a Viper X 300 s arm executing a learned diffusion policy. The line overlay shows the sampled action trajectories converging toward the target object, while the arm autonomously refines its actions at each timestep.

Diffusion Policy

First, we demonstrate deployment of a learned diffusion policy [16]. For this use case, a ViperX300s robotic arm and an Intel RealSense RGB camera are configured to perform a pushing task, shown in Figure 7. The observation space includes the robot's joint positions and a continuous RGB image stream from the camera. The action space consists of joint position commands sent to the ViperX 300s.

Data collection is facilitated entirely through Ark nodes as demonstrated in the previous section, which operate as independent, reusable processes. In this setup, four key nodes are deployed: a Controller Node for PS4 joystick input, an Environment Node that translates joystick commands into target end-effector poses, an Inverse Kinematics Node that converts these poses into joint commands, and Sensor Nodes that publish RGB images and joint states. Because Ark enforces strict message typing and channel separation via LCM, users can modify or replace nodes (e.g., switch cameras or use a scripted controller) without rewriting the rest of the system. This modularity enables reuse across tasks, which is particularly beneficial for research workflows where hardware configurations can change or various interfaces can be swapped out for different contexts.

After each demonstration, the environment is required to be reset. When the user presses the 'X' button on the controller, the trajectory is saved automatically, and the robot resets to a neutral state. This mechanism is built directly into Ark's service framework, enabling resets without requiring custom scripting. As a result, users can collect high-quality demonstration datasets with minimal setup, and without needing to manage state transitions or converting data formats manually.

Once the data is collected, Ark already specifies the observation and action space each as a collection of channels. This same configuration is used to extract data form the log files making it easy to implement data loaders in imitation learning training scripts.

Deployment within Ark replicates the data collection setup, easing engineering complexity. The trained diffusion policy is loaded into a policy node within the same environment used for data collection, replacing the joystick controller node. The policy receives the current RGB image and robot joint state from the observation channels and outputs the target end-effector position. This target position is published as a command through the same action channels used during demonstrations. Since the execution pathway remains unchanged, users do not need to modify the underlying infrastructure to test learned poli-

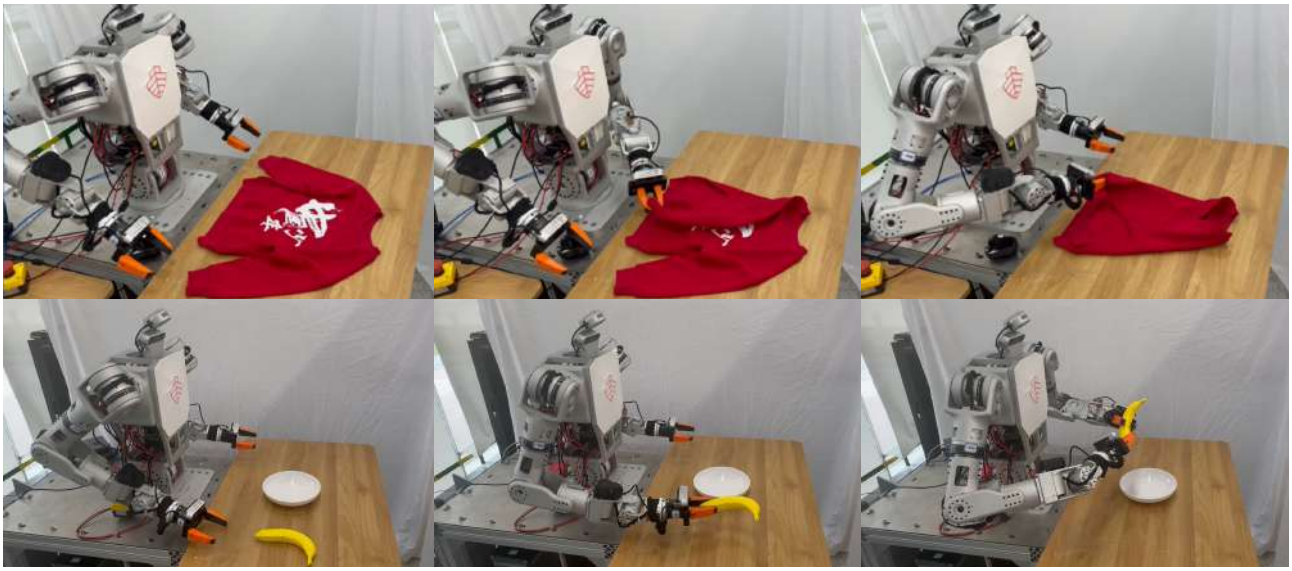

Figure 8. Execution of an ACT-based policy on the OpenPyro humanoid robot for two distinct tasks: cloth manipulation (top row) and object handover (bottom row). The policy produces precise, contact-rich behaviors that enable the robot to flatten a shirt and place a banana into a bowl.

cies, Ark ensures a consistent system setup across training and deployment.

Action Chunking with Transformers

In addition to diffusion policy, we have also implemented a demonstration based on action chunking with transformers [47]. For this demonstration, we use the OpenPyro-A1 humanoid platform [48] and teleoperate it with a virtual reality headset. Data was collected for two tasks, the first is cloth manipulation task (top row of Figure 8) and the second is object handover task (bottom row of Figure 8); snapshots in Figure 8 are from the videos of the robot performing these tasks using the learned policies.

Mobile-base Robots

Many real-world tasks, such as inspection, require a robot to navigate to various locations within an environment. To do this effectively, the robot must be able to both access a map of the environment and determine its position within that map. This process is commonly known as Simultaneous Localization and Mapping (SLAM), which enables the robot to construct a map of an unknown environment while simultaneously estimating its own location. Once a map has been created and the robot is accurately localized, planning algorithms can be employed to navigate the environment.

We have developed a mobile robotics pipeline in Ark that implements a variant of FastSLAM [49]. Using teleoperation to control the robot and data from its onboard LIDAR sensor, we employ FastSLAM to construct a map of the robot's environment. Once the map is built, we apply the A* search algorithm for path planning, incorporating a distance transform to maintain safe navigation margins between the robot and nearby obstacles. A proportional-derivative (PD) controller then guides the robot through each waypoint,

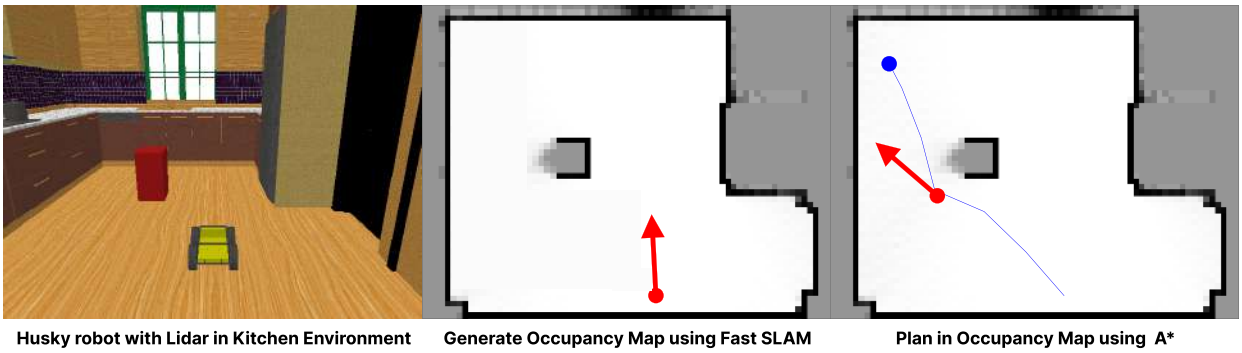

Figure 9. Using Ark's integrated SLAM and visualization tools, a Husky robot equipped with a LiDAR sensor navigates a kitchen environment. The robot first constructs an occupancy map using FastSLAM (center), facilitated by Ark's modular data streaming and map-building nodes. The final panel (right) shows an A* path generated within the occupancy map using Ark's planning and rendering utilities. This setup demonstrates how Ark enables end-to-end navigation workflowsfrom sensor integration and SLAM to path planning and visualizationwithin a unified framework.

translating the planned trajectory into low-level wheel velocity commands.

Map Building

To construct a 2D map of the environment while estimating the robot's pose, a teleoperation assisted SLAM pipeline was implemented using the Ark framework. The pipeline consists of two primary subsystems: a teleoperation controller and a probabilistic SLAM module, both implemented as Ark nodes that communicate over message channels.

The teleoperation node allows a user to control the robot by specifying desired linear and angular velocities. These high level commands are published over an Ark action channel and received by the low level controller node, which computes the corresponding left and right wheel velocities using differential drive kinematics.

Simultaneously, the LIDAR data and control signals are streamed to the SLAM node, which fuses these inputs to estimate the robot's pose and construct a map of the environment. The LIDAR data are represented as $N \times 2$ arrays of polar coordinates (distance, angle) centered at the LiDAR frame, while the control signals reflect the robot's commanded motion. These data are consumed by a node running FastSLAM, implemented using a Rao Blackwellized Particle Filter. Each particle maintains both a pose estimate and a local occupancy grid, with cells assigned probabilities between 0 (free) and 1 (occupied). Figure 9 illustrates Ark's integration with SLAM for map building and navigation.

Navigation

Given a map of the environment and the robot's pose within that map, the system can perform motion planning while avoiding obstacles. The A* search algorithm, a widely used grid-based planning method, has been adapted for robotic motion planning and is implemented in Ark, integrated with the map generated via SLAM. A* is particularly appealing

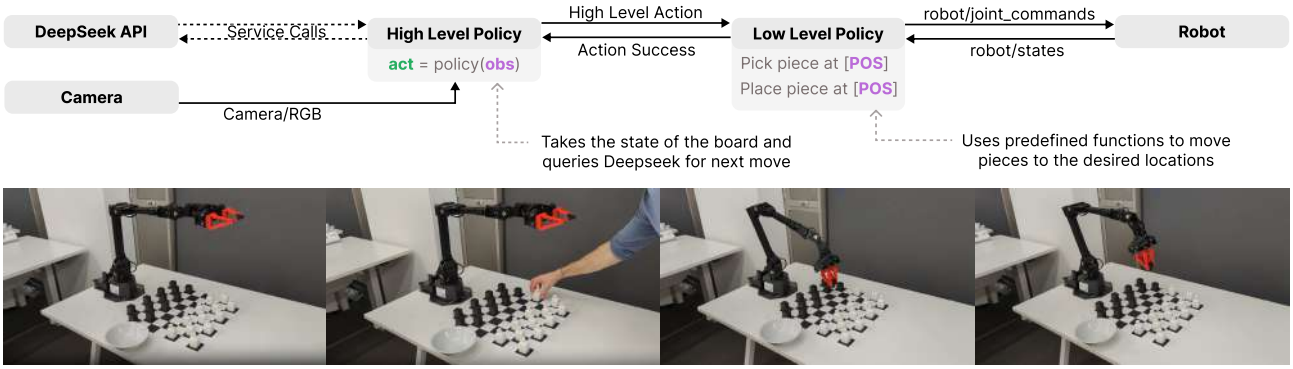

Figure 10. Deep Seek integration with Ark Framework to allow the Viper to play board games

because it is a complete algorithm–meaning that if a valid path from the start to the goal exists, A* is guaranteed to find it, provided the environment is discretized finely enough.

Upon receiving a goal position in world coordinates, the planning subsystem is triggered. This node receives as input both the target location and the probabilistic occupancy map produced during SLAM. To plan a safe and efficient trajectory, the planner first discretizes the occupancy map using a predefined threshold (typically 0.5) to distinguish between free and occupied cells. It then computes the distance transform of the grid to determine, for each cell, the shortest distance to the nearest obstacle. This metric is used to ensure that planned paths maintain a minimum clearance of at least half the robot's width plus a user-specified margin, reducing the risk of collision in narrow passages. The shortest feasible path from the robot's current location to the goal is computed using the A* algorithm. The resulting trajectory is a sequence of Cartesian waypoints (x, y) that avoid obstacles while maintaining smooth curvature and safety margins. To improve execution efficiency, the raw path is then downsampled based on a user-defined spatial resolution, reducing unnecessary intermediate waypoints and promoting smoother motion.

The control subsystem consumes both the planned path and the real-time pose estimates from the SLAM node. Implemented as a PD controller, the system directs the robot through each waypoint in order. For each waypoint, the controller first rotates the robot to face the target using angular control, followed by linear control to advance toward it. Once the current waypoint is reached within a defined tolerance, the next target is selected. The controller outputs both linear and angular velocities, which are then converted into left and right wheel commands using differential drive kinematics.

Embodied AI

Large language models (LLMs) and vision-language models (VLMs) have been shown to endow robots with impressive reasoning capabilities [31, 32]. Ark's modular design and Python-first architecture make it well suited for integrating large language models (LLMs) and vision-language models (VLMs) as high-level policy selectors within robot control loops; Python is the most common programming language used in machine learning. An agentic system was implemented using the Viper robotic arm, in which high-level semantic reasoning is performed by an LLM to perform reasoning tasks, shown in Figure 10. As our base LLM we use Deepseek-R1 [13]. The system implemented follows a code-as-policy paradigm [50]: each robotic manipulation skill–such as "pick piece", "place at location", and

"remove object"–are implemented as a parameterized policy that is callable as a Python function. These functions are exposed through a policy library, which the LLM selects from based on task context and scene understanding in the prompt.

Each node in the Ark Network–scene perception, language-based reasoning, and motion execution–is encapsulated as a standalone node. DeepSeek is integrated as an Ark node that exposes a service interface; the prompt is the request input and the output of the LLM is returned as the service response. At each step of the policy is a decision event, the perception node publishes scene observations (e.g., board state, object poses, RGB images) to a shared channel. These inputs are then passed as a structured query to the DeepSeek node via an Ark service call. The query includes the current state of the environment, a list of available policy functions, and optionally, a task prompt expressed in natural language. We tested Deepseek-R1 versus other LLMs (Qwen 2.5 and Llama 3) in a round-robin tournament and found DeepSeek-R1 to have the highest win-rate: Qwen 2.5 (26.6%), Llama 3 (30.0%), and Deepseek-R1 (43.3%). Unfortunately, each LLM was unable to beat humans.

Discussion

Ark has been purpose-built from the ground up with a design philosophy that will feel familiar to those experienced with machine learning software. It is guided by a number of core principles: simplicity and modularity, Python-first accessibility, and seamless integration with both real-world and simulated robotic systems. With its flexible, distributed architecture and lightweight communication layer, Ark enables reliable coordination of sensors, actuators, and AI models across a wide range of robotic embodiments. It offers researchers and developers a robust and customizable framework for rapidly prototyping, deploying, and iterating on real-world robotic systems.

Through a series of use-cases, Ark has been validated on several tasks important to robot learning research; from dexterous manipulation with expert-coded policies to language-conditioned visuomotor control powered by state-of-the-art foundation models. These implementations demonstrate Ark's role not only as an interface, but as a catalyst for advancing research in embodied AI. Its unified environment abstraction, standardized data pipelines, and native support for machine learning workflows enable teams to transition smoothly from simulation to real-world hardware, and from prototyping to full deployment.

Related Work

Over the years, several robotics frameworks have been developed to address different aspects of robot control, each with varying levels of modularity, language support, and real-time capabilities. We provide a comparison between these different robotic frameworks can be seen in Table 1. The comparison points in the table cover several points in each column. *Python*: indicates if Python is a supported language. *C/C++*: indicates if C/C++ are supported languages. *Gym*: indicates if the main user interface is designed to align with OpenAI Gym (Gymnasium). *IL*: means if imitation learning algorithms are directly integrated into the framework. *RL*: means if reinforcement learning algorithms are directly integrated into the framework (whilst Ark currently does not support RL, in the future we plan to implement this functionality). *Sim*: indicates whether simulators are integrated in

the framework. *ROS Dep*: indicates if ROS is a dependency of the framework, we see this as a negative attribute. *ROS Con*: indicates whether an optional ROS connection is provided (e.g., in our case, we provide the ROS-Ark Bridge). *PubSub*: indicates if the framework implements a modular publisher-subscriber networking approach. *Sim-Real*: indicates if specific switching mechanisms exist to switch between simulation and reality. *Pip*: indicates whether the framework is installable using only `pip`. *Data tools*: indicates whether tools are provided off-the-shelf for data collection/processing. *Inspection tools*: indicates whether tools are provided off-the-shelf for inspection and debugging. *Limited Embod*: indicates if the framework limits the user to a number of specific embodiments, this is a negative point.

YARP is a peer-to-peer communication framework focused on modularity and performance, primarily used in humanoid and legged robotics like iCub and MIT Cheetah. However, its ecosystem is limited by exclusive support for C++.

LCM, on its own, offers a lightweight publish-subscribe model optimized for low-latency, high-bandwidth messaging and has broad language support. While highly effective for communication, it provides only the messaging layer, requiring additional infrastructure for tasks such as system coordination, simulation integration, and machine learning workflows–gaps that Ark addresses.

OROCOS provides real-time libraries for control, kinematics, and filtering, with strong deterministic guarantees via CORBA integration–but it leaves much of the broader system design to the user.

ROS 1, once the de facto standard for robotics development and now officially end-of-life, offered a rich ecosystem of reusable tools, libraries, and drivers that greatly accelerated prototyping and system integration. However, it also suffered from several architectural limitations, including a lack of built-in security and poor reliability over lossy networks. Since Ark is currently focused on research rather than commercial deployment, security is not a primary concern at this stage. Additionally, switching between real and simulated environments in ROS 1 was not seamless and often required users to implement custom, ad hoc solutions.

In contrast to the frameworks/libraries mentioned above, Ark is designed to support modern machine learning-driven robotics. It promotes modularity and reusability but removes unnecessary complexity by providing a Python-first, lightweight, and distributed architecture. Unlike frameworks like YARP or OROCOS that focus on specific layers of the stack, Ark offers end-to-end integration–from low-level hardware communication to high-level policy control–using standardized channels and services. Its minimal setup and direct compatibility with machine learning tools make it particularly well-suited for rapid iteration and embodied AI research. Moreover, installing Ark is simple, it can be setup using pip and is compatible with conda environments.

Future Work

To better support the needs of embodied AI research, future development of Ark will focus on two key areas: reinforcement learning (RL) infrastructure and high-fidelity simulation capabilities.

While RL is a popular approach in modern robotics, Ark's current infrastructure offers only limited support for RL workflows. Upcoming enhancements will include native inte-

Table 1. Comparison of Ark versus alternatives. Note, red mark indicates a negative feature.

	Python	C/C++	Gym	IL	RL	Sim	ROS Dep	ROS Con	PubSub	Sim-Real	Pip	Data tools	Inspect. tools	Limited Embod.
Ark	●	●	●	●		●		●	●	●	●	●	●	
LeRobot	●		●	●	●	●				●	●	●		●
PyRobot	●		●			●	●	●		●				●
ROS 2	●	●				●	NA	NA	●				●	
Orocos		●							●					
YARP		●							●					

gration with popular RL libraries such as StableBaselines3 and RLlib, as well as support for parallelized environment execution. These improvements will enable researchers to train, evaluate, and deploy RL policies efficiently across both simulated and physical robotic platforms using a unified environment abstraction.

In parallel, we aim to significantly advance Ark's simulation stack. Although Ark currently supports integration with simulators like PyBullet and MuJoCo, it lacks advanced features such as domain randomization and differentiable physics–both crucial for robust policy learning and sim-to-real transfer. Future releases will focus on tighter integration with high-performance simulation backends, enabling more accurate, scalable, and versatile simulations.

Epilogue

In summary, Ark represents a significant step forward in bridging robotics and machine learning through a modern, modular, and accessible software architecture. By lowering the technical barriers to real-world robot deployment, while maintaining flexibility and extensibility for advanced research, Ark empowers a new generation of researchers to develop, test, and deploy intelligent robotic systems more efficiently. As it continues to mature–through enhanced simulation support, deeper RL integration, and expanded tooling–Ark is well-positioned to serve as a framework for embodied AI, catalyzing progress across the robot learning community.

References

[1] Nils J. Nilsson. “Shakey the Robot”. In: *SRI AI Center Technical Note* (1984).

[2] Richard Paul. “Robot Manipulators: Mathematics, Programming, and Control”. In: *Artificial Intelligence Center*. 1972.

[3] R. J. Popplestone, A. P. Ambler, and I. Bellos. “RAPT: A language for describing assemblies”. In: *Industrial Robot: An International Journal* 5.3 (Jan. 1978), pp. 131–137. DOI: 10.1108/eb004501.

[4] Peter I Corke. “A robotics toolbox for MATLAB”. In: *IEEE Robotics & Automation Magazine* 3.1 (2002), pp. 24–32.

[5] Brian Gerkey, Richard T. Vaughan, and Andrew Howard. “The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems”. In: *Proceedings of the International Conference on Advanced Robotics (ICAR)*. 2003.

[6] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan. 2009, p. 5.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (May 2015), pp. 436–444. DOI: 10.1038/nature14539.

[8] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012.

[10] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf.

[11] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[12] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].

[13] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL].

[14] O. Khatib. “A unified approach for motion and force control of robot manipulators: The operational space formulation”. In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53. DOI: 10.1109/JRA.1987.1087068.

[15] Michael Posa, Cecilia Cantu, and Russ Tedrake. “A direct method for trajectory optimization of rigid bodies through contact”. In: *The International Journal of Robotics Research* 33.1 (2014), pp. 69–81. DOI: 10.1177/0278364913506757. eprint: https://doi.org/10.1177/0278364913506757.

[16] Cheng Chi et al. “Diffusion policy: Visuomotor policy learning via action diffusion”. In: *The International Journal of Robotics Research* 0.0 (2024), p. 02783649241273668. DOI: 10.1177/02783649241273668. eprint: https://doi.org/10.1177/02783649241273668.

[17] Kevin Black et al. π_0*: A Vision-Language-Action Flow Model for General Robot Control*. 2024. arXiv: 2410.24164 [cs.LG].

[18] Oliver Kroemer, Scott Niekum, and George Konidaris. “A Review of Robot Learning for Manipulation: Challenges, Representations, and Algorithms”. In: *Journal of Machine Learning Research* 22.30 (2021), pp. 1–82.

[19] Xuan Xiao et al. “Robot learning in the era of foundation models: A survey”. In: *Neurocomputing* (2025), p. 129963.

[20] Aude Billard et al. “Robot Programming by Demonstration”. In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1371–1394. DOI: 10.1007/978-3-540-30301-5_60.

[21] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

[22] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[23] Greg Brockman et al. “OpenAI Gym”. In: (June 2016). DOI: 10.48550/arXiv.1606.01540.

[24] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[25] William J. Dally, Stephen W. Keckler, and David B. Kirk. “Evolution of the Graphics Processing Unit (GPU)”. In: *IEEE Micro* 41.6 (2021), pp. 42–51. DOI: 10.1109/MM.2021.3113475.

[26] Sharan Chetlur et al. *cuDNN: Efficient Primitives for Deep Learning*. 2014. arXiv: 1410.0759 [cs.NE].

[27] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[28] Paulo Canelas et al. “An experience report on challenges in learning the robot operating system”. In: *Proceedings of the 4th International Workshop on Robotics Software Engineering*. RoSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2023, pp. 33–38. DOI: 10.1145/3526071.3527521.

[29] Sophia Kolak et al. “It Takes a Village to Build a Robot: An Empirical Study of The ROS Ecosystem”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 430–440. DOI: 10.1109/ICSME46990.2020.00048.

[30] Nicholas Roy et al. *From Machine Learning to Robotics: Challenges and Opportunities for Embodied Intelligence*. 2021. arXiv: 2110.15245 [cs.RO].

[31] Christopher E Mower et al. “ROS-LLM: A ROS framework for embodied ai with task feedback and structured reasoning”. In: *arXiv preprint arXiv:2406.19741* (2024).

[32] Guowei Lan et al. “Experience is the Best Teacher: Grounding VLMs for Robotics through Self-Generated Memory”. In: *Under review* (2025).

[33] Yi Li et al. “Hamster: Hierarchical action models for open-world robot manipulation”. In: *arXiv preprint arXiv:2502.05485* (2025).

[34] D. L. Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. DOI: 10.1145/361598.361623.

[35] M. Montemerlo, N. Roy, and S. Thrun. “Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit”. In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*. Vol. 3. 2003, 2436–2441 vol.3. DOI: 10.1109/IROS.2003.1249235.

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

[37] Albert S. Huang, Edwin Olson, and David C. Moore. “LCM: Lightweight Communications and Marshalling”. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 4057–4062. DOI: 10.1109/IROS.2010.5649358.

[38] Jack Collins et al. “A Review of Physics Simulators for Robotic Applications”. In: *IEEE Access* 9 (2021), pp. 51416–51431. DOI: 10.1109/ACCESS.2021.3068769.

[39] Remi Cadene et al. *LeRobot: State-of-the-art Machine Learning for Real-World Robotics in Pytorch*. https://github.com/huggingface/lerobot. 2024.

[40] Adithyavairavan Murali et al. “PyRobot: An Open-source Robotics Framework for Research and Benchmarking”. In: *arXiv preprint arXiv:1906.08236* (2019).

[41] Aude Billard et al. “Survey: Robot programming by demonstration”. In: *Springer handbook of robotics* (2008), pp. 1371–1394.

[42] Arne Muxfeldt, Jan-Henrik Kluth, and Daniel Kubus. “Kinesthetic teaching in assembly operations–a user study”. In: *Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings 4*. Springer. 2014, pp. 533–544.

[43] Haozhuo Li, Yuchen Cui, and Dorsa Sadigh. *How to Train Your Robots? The Impact of Demonstration Modality on Imitation Learning*. 2025. arXiv: 2503.07017 [cs.RO].

[44] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. “Remote Telemanipulation with Adapting Viewpoints in Visually Complex Environments”. In: *Proceedings of Robotics: Science and Systems*. FreiburgimBreisgau, Germany, June 2019. DOI: 10.15607/RSS.2019.XV.068.

[45] Christopher E. Mower et al. “Comparing Alternate Modes of Teleoperation for Constrained Tasks”. In: *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*. 2019, pp. 1497–1504. DOI: 10.1109/COASE.2019.8843265.

[46] Christopher Mower. “An optimization-based formalism for shared autonomy in dynamic environments”. In: (2022).

[47] Tony Z. Zhao et al. *Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware*. 2023. arXiv: 2304.13705 [cs.RO].

[48] Helong Huang et al. *OpenPyRo-A1: An Open Python-based Low-Cost Bimanual Robot for Embodied AI*. https://openpyro-a1.github.io/. Technical report. 2025.

[49] Sebastian Thrun et al. “Fastslam: An efficient solution to the simultaneous localization and mapping problem with unknown data association”. In: *Journal of Machine Learning Research (JMLR)* (2004).

[50] Jacky Liang et al. “Code as policies: Language model programs for embodied control”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 9493–9500.