RIO: Flexible Real-time Robot I/O for Cross-Embodiment Robot Learning

Author Names Omitted for Anonymous Review. Paper-ID 928

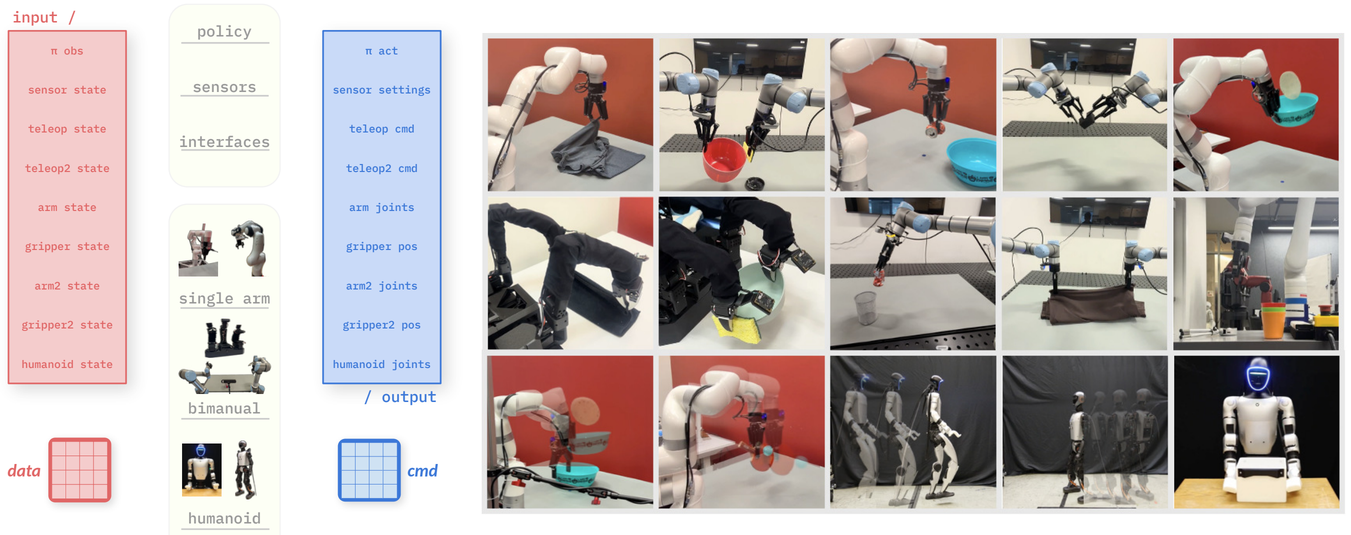

Fig. 1: **RIO**. We introduce a framework for flexible real-time Robot I/O (RIO) for cross-embodiment robot learning. RIO provides lightweight Python-based hardware drivers to coordinate diverse robot morphologies, sensors, teleoperation interfaces, and policies in a full-stack manner.

***Abstract*—Despite recent efforts to collect multi-task or multi-embodiment datasets, to design efficient recipes for training Vision-Language-Action models (VLAs), and to showcase these models on selected robot platforms, generalist robot capabilities and cross-embodiment transfer remain largely elusive ideals. This cross-embodiment robot learning paradigm remains limited by fragmented data-collection infrastructure, the lack of standardization on versatile data formats, and the significant engineering effort involved in reproducing hardware setups and organizing multiple control stacks for quickly deploying models on diverse robot platforms. As a result, most robot code tends to be highly specific to the exact robot setup that the user decided on, which adds major overhead when attempting to reuse, recycle, or share artifacts between users. To bridge this gap, we present Robot I/O (RIO), an open-source Python-based framework that provides flexible, lightweight components for robot control, teleoperation, data formatting, sensor configuration, and policy deployment across diverse hardware platforms and morphologies. RIO provides abstractions that enable users to make any choice (robots, sensors, teleoperation interfaces, middlewares, data formats, policies) and to switch between them, with minimal reconfiguration effort. We validate RIO on VLA deployment workflows across three morphologies (single-arm, bimanual, humanoid) and four robot hardware platforms with varying grippers and cameras. We showcase policy rollouts by collecting teleoperated data to fine-tune state-of-the-art VLAs, including $\pi_{0.5}$ and GR00T, on household tasks such as pick-and-place, folding, and bowl scrubbing. By open sourcing all our efforts, we hope the wider robotics community can accelerate their pace of robot learning on real-world robot hardware.**

I. Introduction

Vision-language-action models (VLAs) have recently emerged as a promising approach for training generalist robot policies, leveraging large-scale datasets to learn broadly capable robot behaviors. Despite their potential, achieving cross-embodiment generalization, the ability to transfer learned behaviors across different robot morphologies, remains a fundamental challenge. VLAs cannot be deployed out of the box on new embodiments; successfully reproducing and running these systems on new robots still demands substantial engineering effort. This challenge, however, extends well beyond VLAs.

Robotics practitioners have long contended with the fragmentation inherent in the field. Varying morphologies, diverse sensor configurations, heterogeneous hardware platforms, and manufacturer-specific driver code collectively result in robot infrastructure that is highly specific to a user's particular setup. This results in significant overhead when attempting to reuse code, share datasets, or build on each others works. Existing cross-embodiment datasets like Open X-Embodiment [36] are, in practice, aggregations of many individual collection efforts conducted across disparate infrastructure.

The cost of this fragmentation is growing. As robot hardware becomes increasingly affordable, more platforms are entering research labs and deployment settings. Yet the specialized nature of most robotics infrastructure means that each new platform carries substantial integration overhead. Consider a common scenario: a research team wishes to reproduce real-world results released by another group. To use the original control code, they would need to replicate the exact hardware setup one-to-one, as in efforts like DROID [26]. If they instead have a different robot arm, they face the burden of rewriting the entire control stack from scratch, before even trying to adapt any learned policies. This makes most robot learning hardware code *difficult to reuse*, and switching between platforms far harder than it should be.

What is the most important infrastructure for robot learning to advance? Beyond large datasets, we believe that a lack of flexible, reusable, accessible, and performant full-stack robot infrastructure has been a critical barrier to cumulative progress and collaboration within the field. Robot learning is missing reusable building blocks for hardware with flexible abstractions that have become standard in other areas of machine learning. Just as specialized high-performance GPU kernels within high-level auto-differentiation frameworks have enabled the rapid development and iteration of neural networks, robotics requires analogous foundational components for hardware and control that can be reliably shared, extended, and built upon across the community.

In this paper, we present the following contributions:

i) We introduce RIO, a flexible real-time Robot I/O framework for scalable cross-embodiment robot learning. RIO does not aim to be a comprehensive solution for robot learning, but rather a lightweight set of reusable building blocks that can be quickly combined to deploy policies on real robots, depending on each user's needed configuration. RIO is designed to be flexible, reusable, accessible, and performant, with abstractions such that the user is free to make any choice at each layer of the stack, and to switch between them with minimal effort.

ii) We validate RIO for the VLA deployment workflow spanning diverse embodiments across single arm, bimanual, and humanoid robots with different grippers and sensors. This includes different robots, sensors, teleoperation interfaces, middlewares, data formats, and policies.

iii) We demonstrate real-world deployment by collecting teleoperated data to fine-tune state-of-the-art VLAs such as $\pi_{0.5}$ and GR00T, on household tasks such as pick-and-place, folding, and bowl scrubbing.

II. Related Works

A. Generalist robot policies

Recent advances in vision-language-action models (VLAs) [6, 18, 4, 23, 40, 3] aim to leverage the robust image-to-language alignment learned by internet-scale pre-trained vision-language models (VLMs) [50, 17, 2, 1, 14] to train generalist robot policies. VLAs adapt VLMs to predict actions through imitation learning on robot datasets collected via human teleoperation of robots, scaling foundational work on imitation learning for visuomotor policy learning, such as ALOHA [51] and Diffusion Policy [12]. Due to the computational resources and data scale required, state-of-the-art VLAs are predominantly trained by industry labs with substantial infrastructure and engineering personnel. Open source efforts have sought to reproduce and democratize these results [44, 31, 39, 14, 27], providing fully open-source implementations and model weights. However, a significant limitation remains: current VLAs must in practice be fine-tuned for each robot setup. Released VLA model checkpoints are typically fine-tuned for specific embodiments, such as the Franka arm from DROID [26] or the WidowX arm from BridgeData V2 [42]. Consequently, end-users must either reproduce the exact hardware setup used during training [43], or undertake substantial engineering effort to implement their own robot control stack, before even attempting to adapt learned policies to their own platforms. In this work, we lower this barrier by introducing a flexible cross-embodiment robot control stack, validated on the VLA adaptation workflow and deployed across diverse robot configurations.

B. Cross-embodiment robot data

The effectiveness of scaling VLAs depends on access to large-scale robot demonstration data. Prior work has demonstrated that scaling robot data across both task diversity and robot embodiments [13, 45, 10, 16, 41, 47] shows promise at training better generalist robot policies, and cross-embodiment robot data may also enable learning directly from humans [25]. Training truly general robot policies requires diversity in both tasks and embodiments. Open X-Embodiment [36] aggregates 60 datasets spanning over 1 million robot trajectories across 22 embodiments. However, the heterogeneous collection techniques and sensor configurations across these datasets necessitate substantial curation for effective policy training, such as through filtering Ghosh et al. [18], Khazatsky et al. [26] or data mixture re-weighting [22]. In this work, we aim to facilitate the collection of high-quality cross-embodiment robot data, by developing flexible and reusable robot infrastructure.

C. Robot control stacks

Over the years, many robot control stacks have emerged [34, 28, 15, 8, 53, 29, 35, 24, 19, 33, 11] that are capable of cross-embodiment robot control. ROS [33] was developed to facilitate system compartmentalization and distributed communication. While this modular, distributed approach offers benefits for complex robotic systems, it requires compounding systems-level engineering to coordinate all modules together. Furthermore, ROS presents a high barrier to entry for researchers and practitioners new to robotics, as it requires wrangling its complex configuration management and build system. Despite this proliferation of frameworks, robot code remains highly platform-specific. We attribute this to two factors: first, most roboticists work with a single hardware platform, which incentivizes writing vendor-specific code quickly

Table I: **Comparison of cross-embodiment robot stacks.** We compare various cross-embodiment robot stacks on the basis of native platform support, at different layers of the robot learning pipeline, e.g., data-collection system support, robot hardware support, middleware, data formats, and policy architecture support. For example, some stacks combine robot arm and robot gripper drivers, making it difficult to use other end effectors on arms.

Framework	Humanoids	Bimanual	Single arm	Robot grippers	Teleop	Cameras	Middleware(s)	Data format(s)	Policies
Ark [15]	✓	✓	✓	✗	✓	✓	✗: LCM	✗: Pickle	✓
LeRobot [8]	✓	✓	✓	✗	✓	✓	✗: Threads/gRPC[1]	✗: LeRobotDataset	✓
ManiUniCon [53]	✗	✗	✓	✗	✓	✓	✗: Shm	✗: Zarr	✓
PAPRLE [29]	✓	✓	✓	✓	✓	✓	✗: ROS	✗: Pickle	n/a
PyRobot [35]	✗	✗	✓	✓	✓	✓	✗: ROS	✗: Pickle	n/a
RCS [24]	✗	✗	✓	✓	✓	✓	✗: RPC	✗: Parquet	✓
RoBits [19]	✗	✓	✓	✓	✓	✓	✗: ZMQ	✗: NPZ/JSON	n/a
UMI, DP [11, 12]	✗	✓	✓	✗	✓	✓	✗: Shm	✗: Zarr	✗: DP
RIO (ours)	✓	✓	✓	✓	✓	✓	✓: **any**	✓: **any**	✓

[1] *LeRobot uses Threads for hardware drivers and gRPC for asynchronous policy inference.*

rather than abstractable solutions; second, existing frameworks lack flexible abstractions at every layer of the stack to ensure cross-embodiment robot code is easy to write in the first place. As robot hardware becomes increasingly affordable [8, 51], more platforms are entering research labs and deployment settings. Hardware code released alongside Diffusion Policy (DP) [12], later extended for cross-embodiment robot support by UMI [11, 49], has found widespread adoption by robotics researchers with numerous community forks. LeRobot [8] has also found widespread adoption among the broader robotics community. Its popularity stems in part from its support for low-cost robot hardware along with Python-only implementations, which eliminates the need for complex build systems such as ROS and multi-language dependencies. By streamlining the development workflow, LeRobot has lowered the barrier to deploying real-world robot systems and enabled a wider range of practitioners to experiment with robot learning. In Table I, we compare the flexibility of RIO to a variety of existing cross-embodiment robot infrastructure, across every layer of the robot learning stack. RIO offers a reusable Python-based set of real-time robot infrastructure in a similar style to LeRobot and DP/UMI, while supporting reconfigurability at each layer to facilitate VLA deployment workflows across diverse robot morphologies. RIO enables combinatorial configuration of robots, teleoperation devices, cameras, middlewares, data formats, and policies, for maximum flexibility.

III. Robot I/O (RIO)

RIO is a Python-based framework for flexible real-time Robot I/O, with reusable components for robot control, teleoperation, data collection, and policy deployment across diverse robot embodiments. Users are free to make any choices at every layer of the stack (humanoid robots, robot arms, robot grippers, teleoperation interfaces, cameras, middlewares, data formats, policies) and to switch between them with minimal effort for reconfigurability. See Table II for a detailed list of currently supported hardware.

Figure 2 illustrates the overall system architecture of RIO. Section III-A describes the *Design Philosophy*, Section III-B introduce the client-server *Nodes* abstraction and *Middlewares* implementation for message passing, Section III-C outlines the reconfigurable instantiation of *Robot Stations*, Section III-D describes *Teleoperation* and *Data Collection* used to collect real-world robot trajectories across multiple embodiments, and Section III-E describes the implementation of asynchronous *Policy Inference* to obtain smooth real-world rollouts with observation/action chunking.

Table II: **Current hardware support** (*more incoming*). RIO provides flexibility across robot hardware (humanoid robots, robot arms, robot grippers), teleop interfaces, cameras, and middlewares for multi-process distributed communication. These can be combined in any configuration depending on the user's hardware requirements.

Humanoid Robots	Unitree G1, Booster T1
Robot Arms	UFactory (xArm5/6/7, 850, Lite6), UR (UR5e, UR7e), Franka (FR3, Panda), Kinova (Gen3), SO-100/SO-101
Robot Grippers	UFactory Gripper, Franka Gripper, Robotiq Gripper (2F-85/2F-140), DH-Robotics Gripper (AG-105-145)
Teleop Interfaces	Spacemouse, Gamepad, Keyboard, VR (Apple Vision Pro, Meta Quest), Leader-Follower (GELLO), Phone
Cameras	RealSense, ZED, UVC (Webcams, USB cameras), iPhone (Record3D)
Middlewares	Shared Memory, Thread, Portal, Zenoh, ZeroRpc

A. Design Philosophy

We describe the five design tenets of RIO: *flexible*, *reusable*, *accessible*, *performant*, and *consistent*.

- *Flexible.* RIO is agnostic to each component and does not make any locked-in choices. The user is free to choose between options at every layer, and switch between them with minimal effort.
- *Reusable.* RIO is composed of a lightweight set of reusable building blocks, that can be quickly combined and modified to support a user's desired configuration.
- *Accessible.* RIO is lightweight to install and uses Python-based hardware modules, with a command-line interface over a single configuration file.

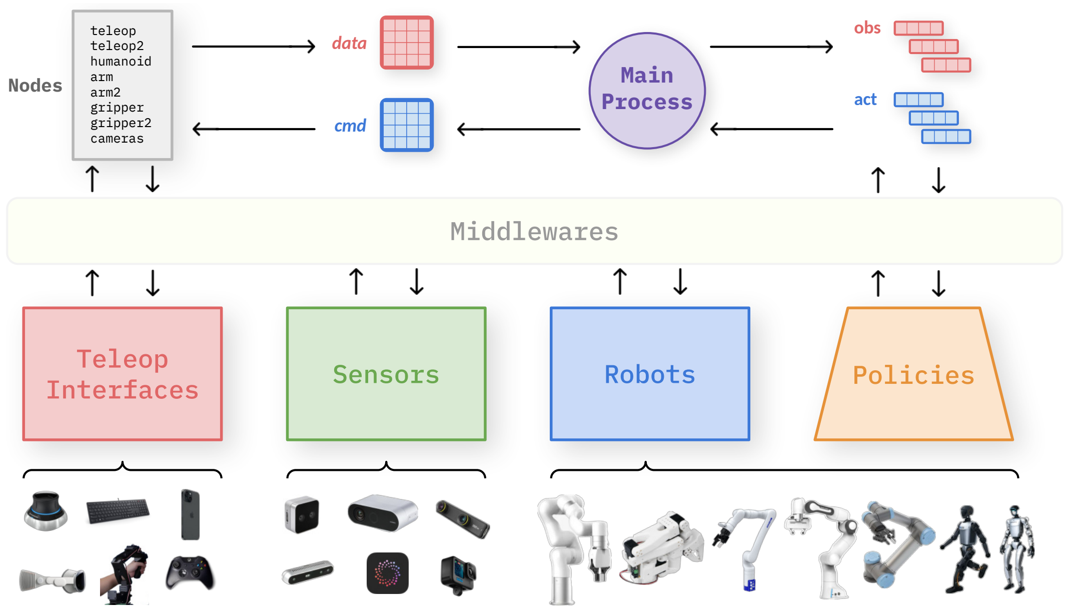

Fig. 2: **Architecture.** High-level overview of the architecture of RIO. Every component of the stack is flexible, meaning that the user is free to choose between different options (robots, sensors, teleoperation interfaces, middlewares, data formats, policies) and switch between them, with minimal effort.

- *Performant.* RIO is fully capable of high-frequency real-time robot control, and uses asynchronous policy inference to yield smooth robot trajectories.
- *Consistent.* RIO is designed to ensure consistent, scalable, reproducible data collection and robot learning.

Despite the proliferation of different robot infrastructures (Table I), most robot code has remained highly platform-specific. We attribute this to two factors: first, most roboticists work with a single hardware platform, which incentivizes writing vendor-specific code quickly rather than developing more abstract solutions; second, existing frameworks lack flexible abstractions at every layer of the stack to ensure that cross-embodiment robot code is easy to write in the first place. We design RIO to share full-stack robot control code that can be reconfigured and built upon by the community.

B. Nodes and Middlewares

Nodes for teleoperation interfaces, sensors, robots, and policies are implemented from the same template Node, requiring minimal boilerplate to enable flexible, real-time I/O across diverse hardware and deployment configurations.

Nodes. A Node dynamically inherits from a given Middleware that automatically handles message passing. Factory functions produce matched server-client Node pairs, for which its dynamic parent class implements the specified Middleware. Nodes support three execution patterns for publishing data and handling requests:

1) Publish-only: `pub()` publishes data in the run loop.
2) Request-only: `req()` handles requests in the run loop.
3) Combined: `pubreq()` publishes data and handles requests in a single loop.

For patterns (1) and (2), the complementary operation can be optionally run in a separate worker loop thread. For example, `pub()` in the run loop with `req()` in a worker loop thread, or vice versa. To implement a Node, the user defines the relevant methods: a `pub()` implementation calls `ring_buffer.put(..)` to publish data, while a `req()` implementation calls `request_queue.get()` to receive and process requests. Published data flows through a ring buffer that continuously streams state at a fixed frequency, providing time-synchronized access to sensor readings, robot poses, and other data. Requests flow through a queue that enables asynchronous command communication, allowing multiple clients to send timestamped commands independently and at arbitrary rates. For each, a server Nodes execute "publish/request", while a client Node automatically resolves "subscribe/reply". The user specifies `example_data` and `example_request` in each Node definition, which are used to infer buffer shapes and data types when initializing ring buffers and request queues. Each Node exposes a public API (defined via `__api__`), whose methods are automatically wrapped for serialization on the server side and deserialization on the client side, enabling transparent bidirectional communication. Because Nodes are middleware-agnostic, they can be paired with different middleware backends depending on deployment requirements. Process synchronization is managed through ready and exit events that signal when "publish/request" loops are ready or exited, ensuring that user logic in the main process blocks until every Node has fully initialized and that the process cleans up when it completes.

Middlewares. Nodes interact with Middleware through a common interface, hiding transport-level details. For network-based communication, middleware such as Zenoh or ZeroRpc handles serialization and transport over TCP or IPC. For high-throughput local communication, shared memory middleware uses a SharedMemoryManager to allocate ring buffers and request queues in shared memory, allowing zero-copy data exchange between processes. The shared memory implementation runs the Node's loops in a separate process and communicates buffer handles back to the parent through pipes, enabling multiple processes to access the same underlying memory regions without serialization overhead. Middlewares can be interchanged depending on the user's requirements. For instance, switching to the Thread middleware can help with debugging when orchestrating many Nodes on the same computer, since running everything multi-threaded within one process simplifies stack traces, breakpoints, and exception handling compared to multi-process or networked deployments. Alternatively, for embodiments such as mobile manipulator robots that may not have a powerful onboard computer, the user may use network-based middleware to communicate between multiple machines.

C. Robot Stations

We aggregate the instantiation of all nodes that make up an environment into a single station configuration file. A robot station is instantiated through a composable dataclass configuration that specifies the hardware topology of the deployment, defining the set of robots, end effectors, and sensors (e.g., {`arm`, `gripper`, `arm2`, `gripper2`, `wrist_camera`, `wrist_camera2`} for a bi-manual robot station). The effect is to simplify the main routine's logic in robot control loops: a context manager initializes all server Nodes with the specified middleware backend, while client Nodes are started within nested context managers that yield proxy objects for transparent communication. While the nodes internally use the specified middleware, queues, and ring buffers, the main routine interacts only with the APIs, resulting in simple, Python-based code. This pattern enables the same application logic to operate over arbitrary station configurations without modification.

D. Teleoperation and Data Collection

Teleoperation. We design three teleoperation scripts to support data collection. The first one controls relative end-effector poses (Keyboard, Phone, Gamepad, Spacemouse), and the second maps absolute joint positions using a leader-follower setup (ALOHA [51], GELLO [46]), for either single-arm or bimanual tabletop robot arms. The third script uses wrist pose retargeting from Apple Vision Pro [37] to control the upper-body of a humanoid robot, similar to [20, 21]. These scripts do not assume any particular hardware platform, so teleoperation devices and robots can be swapped based on each user's needs. To ensure smooth teleoperation across devices and control frequencies, we include interpolators and signal-processing filters (e.g., low-pass filters). These can also be used during policy inference to mitigate the effects of noisy actions.

Data collection. We define a recorder for logging robot demonstrations. To ensure consistent data collection across different hardware, we enforce standardized units: meters for world coordinates and radians for angular measurements. We adapt the RLDS-style format [38] to aggregate multiple data streams into a unified state representation, as shown in Figure 3. To support different robot platforms, we introduces the concept of morphologies—abstract descriptions of a robot's structure, each defining its own set of state keys. Each morphology overloads the observation field of the RLDS step with its specific keys. This design standardizes state reporting across platforms, regardless of the underlying hardware.

```python
@dataclass
class Camera:
    rgb: np.ndarray | None = None
    depth: np.ndarray | None = None
    meta: dict = field(default_factory=dict)

@dataclass
class Observation:
    proprio: np.ndarray  # Defaults to policy
    ↪  action space
    cameras: dict[str, Camera] =
    ↪  field(default_factory=dict)

@dataclass
class Step:
    timestep: int | None
    observation: Observation
    instruction: str | None
    action: np.ndarray | None
    meta: dict | None =
    ↪  field(default_factory=dict)
```

Fig. 3: **Observation Schema.** Standardized state reporting across different client instances and embodiments.

One challenge with scaling robotic data collection is the sheer storage required to manage it, given that a typical robot demo consists of the internal state of the robot, as well as multiple camera streams (often including depth) and other relevant sensors. Additionally, depending on the specific target learning architecture, the data may need to be exported and processed differently. To this end, we use RoboDM [9], a toolkit that employs flexible compression schemes and streamlined file structures, to efficiently record demos in a highly compressed, lightweight format that is quick to save and load. For training or finetuning, we encourage users to write minimal converters so that the exported demos can directly integrate with their intended dataloader format.

E. Policy Inference

Aside from providing lightweight building blocks for hardware components, we also want to reduce the overhead needed to swap between different policies. To reduce the challenge of integrating robot policies into our control stack, we design a high-level API for policies. For each policy, we only require a lightweight interface to instantiate the policy, convert observations from a standardized format to the policy-specific observation format, and run inference. We design a policy

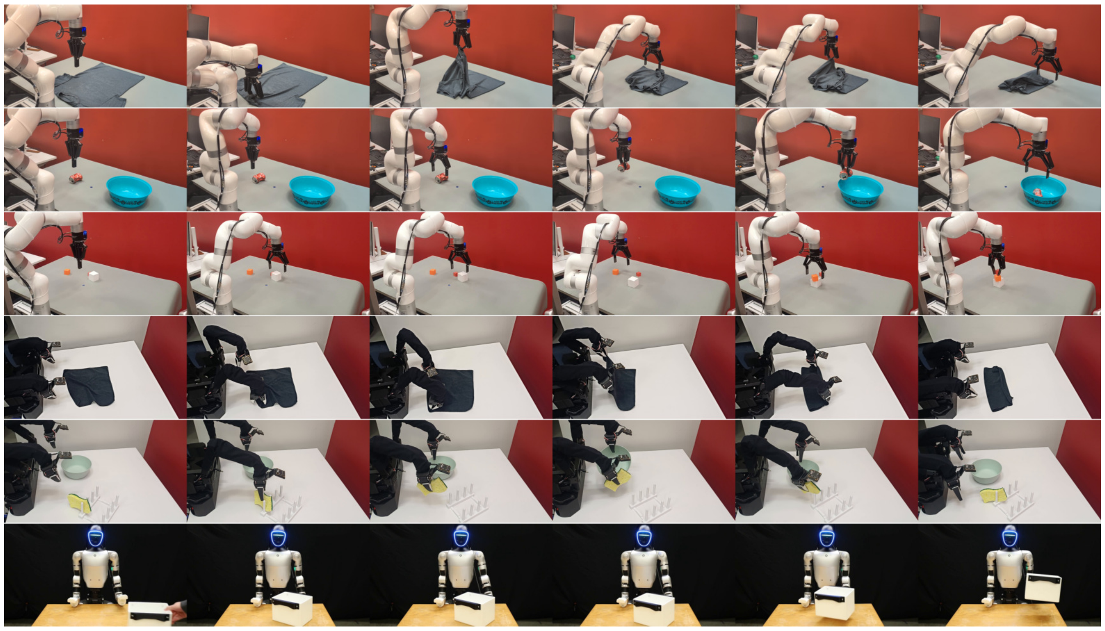

Fig. 4: **VLA manipulation trajectories.** We showcase rollouts on $\pi_{0.5}$ across 3 morphologies on 5 diverse tasks. This figure showcases a frame of the policy rollout at 0%, 20%, 40%, 60%, 80%, and 100% task completion.

Table III: **Policy deployment.** We deploy state-of-the-art VLAs ($\pi_{0.5}$, GR00T N1.5) across 3 morphologies (single arm, bimanual, humanoid), achieving $\geq$60% success across 20 trials on all tasks with finetuning on 50 teleoperated demonstrations. Policy rollout times closely match human demonstrations, with an average slowdown of just $\tilde{3}$.69s, and some tasks completing faster than demonstrations. Asynchronous inference maintains high GPU utilization throughout execution, showing that RIO efficiently saturates available compute.

Robot	Policy	Task	Success Rate	Task Completion Time (s)	Demo Time (s)	RAM (GB)	CPU (%)	GPU Util (%)	GPU Mem (%)
xArm7	BC $\pi_{0.5}$	Fold Shirt	92.5	41.96 $\pm$ 14.58	41.57 $\pm$ 9.25	22.5 $\pm$ 2.7	13.0 $\pm$ 1.4	56.7 $\pm$ 1.7	79.1 $\pm$ 0.0
xArm7	BC $\pi_{0.5}$	Place Can	95.0	16.08 $\pm$ 3.41	14.46 $\pm$ 2.00	24.8 $\pm$ 1.5	13.2 $\pm$ 1.4	54.6 $\pm$ 3.1	79.1 $\pm$ 0.1
SO-100	BC $\pi_{0.5}$	Fold Cloth	60.0	27.50 $\pm$ 5.51	22.43 $\pm$ 3.30	19.6 $\pm$ 0.5	14.1 $\pm$ 1.5	46.3 $\pm$ 10.0	78.6 $\pm$ 0.0
SO-100	BC $\pi_{0.5}$	Scrub Bowl	64.0	40.33 $\pm$ 13.68	27.66 $\pm$ 5.22	19.7 $\pm$ 0.7	15.0 $\pm$ 2.2	52.0 $\pm$ 4.8	78.6 $\pm$ 0.1
Unitree G1	BC GR00T N1.5	Pick Box	95.0	9.07 $\pm$ 6.10	10.38 $\pm$ 4.04	25.2 $\pm$ 0.1	26.6 $\pm$ 3.3	61.7 $\pm$ 4.7	33.8 $\pm$ 0.0
Unitree G1	RL PPO	Navigate	100.0	31.27 $\pm$ 6.56	n/a	23.0 $\pm$ 0.1	10.3 $\pm$ 0.4	5.1 $\pm$ 0.1	10.3 $\pm$ 0.1
Booster T1	RL PPO	Navigate	100.0	29.73 $\pm$ 4.49	n/a	22.6 $\pm$ 0.1	10.4 $\pm$ 0.4	5.3 $\pm$ 0.2	10.4 $\pm$ 0.3

wrapper node that uses this API to directly instantiate policies and asynchronously handle inference requests. By handling inference requests directly through our middleware, we avoid the additional overhead of a dedicated policy server. Additionally, we design a configurable, policy-agnostic, hardware-agnostic inference script that queries the policy wrapper, handles logic for continuously obtaining observations from hardware, post-processes actions for smoothness, and sends commands to hardware. This allows for seamless switching between different policies and hardware.

IV. Evaluation

We evaluate whether RIO can support the complete robot-learning workflow, from teleoperated data collection through to finetuning and deployment, across diverse embodiments. Specifically, our experiments address the following questions regarding RIO's core functionalities:

- *Is RIO performant for real-time policy deployment?*
- *Can RIO support data collection across diverse morphologies that require different teleoperation interfaces?*
- *Does RIO enable effective deployment and benchmarking of state-of-the-art VLAs across embodiments?*

Experimental Setup. All evaluations are performed on the same computer, equipped with NVIDIA GeForce RTX 4090 GPU, AMD Ryzen 7 5700X CPU, and 64 GB of RAM.

A. Policy Finetuning and Deployment

RIO targets two stages of the robot-learning pipeline: data collection for fine-tuning and deployment. We adopt a bring-your-own-training-stack approach: researchers collect demonstrations with RIO, export them to their preferred training format, and import the resulting weights back into RIO for deployment. This ensures consistency between collection and rollout while remaining agnostic to model architecture and training infrastructure.

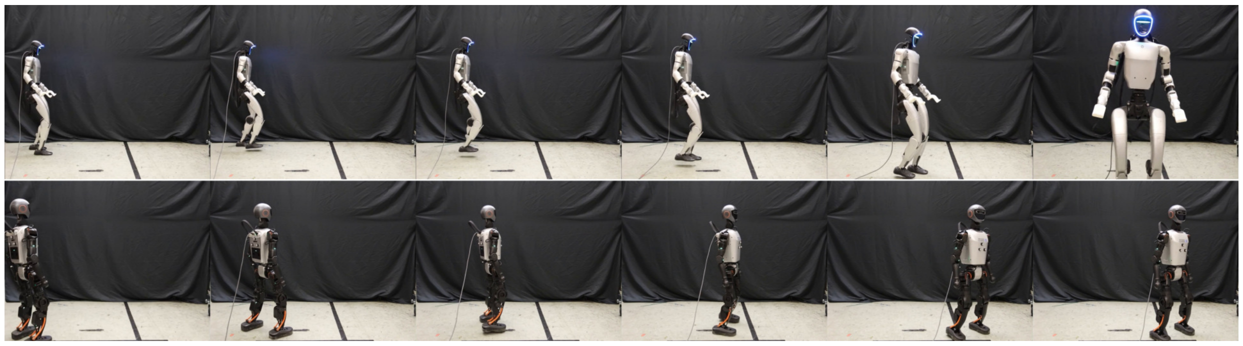

Fig. 5: **Humanoid locomotion trajectories.** We evaluate RL policies on both Unitree G1 (top) and Booster T1 (bottom), humanoid robots from different manufacturers with different hardware drivers added to our stack. RIO is capable of high-frequency real-time control for humanoid locomotion.

We validate this workflow across three morphologies (single-arm, bi-manual, humanoid) and four platforms, training two VLA policies ($\pi_{0.5}$, `GR00T` N1.5) and two RL policies (PPO for locomotion). For $\pi_{0.5}$, we finetune[1] from the DROID checkpoint for single-arm tasks and the ALOHA checkpoint for bi-manual tasks, training for 20K steps in each case. For `GR00T`, we use 150 demonstrations for the humanoid manipulation task. All demonstrations are collected at 50Hz and stored using a compressed format that can be exported to target training pipelines; for reference, 150 episodes with three camera views require only 1.31GB.

To verify policy integration, we report success-rate metrics and compare the average task completion time with the average demonstration time.

Single Arm Tabletop Manipulation. We use a UFactory xArm7 with a Robotiq 2F-140 gripper and three RealSense cameras to evaluate RIO's. We collect data for two pick-and-place tasks (place can in bin, folding a shirt) and two dynamic tasks (hitting a ball with a paddle, and flipping a tortilla).

For pick-and-place, we employ both a Spacemouse and a GELLO leader-follower interface. Empirically, end-effector-space devices such as the Spacemouse yield cleaner demonstrations for tasks where rotation is not a major factor. We note that training VLAs with binary gripper actions yields poor gripper control; we therefore apply trajectory interpolation and low-pass filtering to smooth collected actions. As shown in Table III, both tasks achieve success rates above 90%, with completion times within 2 seconds of demonstration time.

For dynamic tasks, we use the GELLO interface to capture complex motions. These tasks illustrate RIO's ability to collect data at high control rates (80Hz). Given that reliable dynamic task performance from VLAs remains an open research question, we only demonstrate teleoperation for these tasks.

Bi-manual Arm Tabletop Manipulation. We use SO-101 arms to evaluate RIO's support for bi-manual coordination. RIO natively supports robot-to-robot teleoperation; configuring the leader arms as teleoperation devices requires only a change to the station configuration file. We perform bi-manual cloth folding and bowl scrubbing, in which the operator must first grasp the objects before executing the task.

All bi-manual tasks achieve at least a 60% success rate. When comparing task completion time, we observe a greater departure from demonstration time than in the uni-manual case, with an average difference of -8.87 seconds. In particular, the *scrub bowl* task shows the largest difference. We empirically observe that this is not due to inference snags but rather to errors in policy rollout that lead to retry behavior.

Humanoids. To demonstrate the flexibility of RIO, we also validate that our system can perform high-frequency control for humanoid locomotion on two humanoid robots from different vendors: Unitree G1 and Booster T1 (Figure 5).

B. Performance Analysis

We evaluate RIO's real-time performance at two levels: round-trip middleware latency and end-to-end latency under realistic workload conditions.

Round-trip Middleware Latency. To establish an approximate lower bound on RIO's communication overhead, we measure round-trip latency across all supported middlewares. We follow Open Messaging Benchmark by defining latency as half the median round-trip time, removing the 1st and 99th percentiles to improve robustness to outliers. We use a synthetic 2048-byte payload, reflecting typical observation sizes. This benchmark isolates middleware performance since the main loop performs only ring buffer reads and writes. Table IV shows results across 5 supported backends; Zenoh and shared memory achieve sub-millisecond latency suitable for high-frequency control, while thread-based and ZeroRpc backends trade latency for simpler debugging or network flexibility.

[1]We use one checkpoint per-task, per embodiment.

Table IV: **Middleware latency.** We report round-trip latency across RIO's supported middlewares, mean ± stddev, averaged across 1,000 passes with 2048 bytes payload. Network-based backends (Zenoh, ZeroRpc) enable distributed multi-machine deployments, while local options (Shared Memory, Threads) minimize overhead for single-machine setups. This flexibility allows balancing performance, distribution, and system complexity depending on requirements.

Middleware	Latency (ms)
ZeroRpc	10.3192 ± 5.1707
Thread	5.0746 ± 0.0318
Shared Memory	0.1462 ± 0.0119
Zenoh	0.1227 ± 0.1058

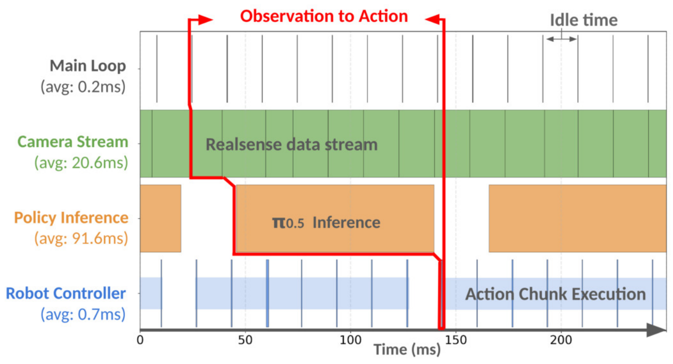

Fig. 6: **Node Profiling During Policy Deployment.** RIO distributes blocking operations (camera streaming, policy inference, robot control) across separate nodes, keeping the main loop free for precise timekeeping. We benchmark on a realistic payload: $\pi_{0.5}$ rollouts on an xArm7 station with three RealSense cameras streaming at 60Hz. Despite significant inference latency ($\tilde{9}$2ms), asynchronous action requests enable continuous control without blocking on model forward passes.

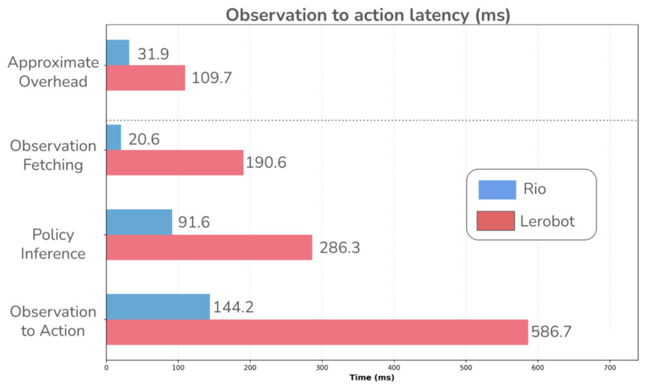

Fig. 7: **Observation-Action Latency.** We measure end-to-end latency from multi-camera observation to robot command execution duration $\pi_{0.5}$ deployment. RIO achieves over 3x lower latency than LeRobot, after accounting for differences in inference time. This efficiency enables smoother policy rollouts at higher control frequencies.

End-to-end profiling. To quantify performance under realistic conditions, we profile RIO during $\pi_{0.5}$ rollouts, receiving inputs from three Intel RealSense cameras (two D415s and one D405) at 640 × 480 resolution. Section IV-B shows the execution timeline across nodes. The main loop remains non-blocking, allowing for precise time-keeping, with blocking operations distributed to dedicated nodes. Asynchronous inference allows the system to preemptively request actions, maintaining continuous control despite the $\tilde{9}$2ms model forward pass. We compare observation-to-action latency against LeRobot, a widely adopted Python-based framework. Given minimal overlap between the platforms supported by RIO and LeRobot, we use a mock robot whose read/write operations have a 1 ms block time; the camera setup is kept the same as in previous tasks. As shown in Section IV-B, RIO achieves lower latency across the different stages. When isolating communication overhead by excluding observation fetching and policy inference, RIO outperforms the baseline with 3x lower latency. This efficiency stems from its streamlined architecture: while LeRobot introduces delays by threading observations before network transmission to an asynchronous server, RIO directly leverages the middleware for asynchronous inference.

V. Conclusion

The lack of flexible, reusable, accessible, performant, and consistent full-stack robot infrastructure has proven to be a critical barrier to cumulative progress and collaboration in robotics. In this work, we present RIO, a flexible real-time Robot I/O framework for cross-embodiment robot learning. RIO provides lightweight, middleware-agnostic building blocks for robot control, teleoperation, data collection, and policy deployment that can be freely combined and reconfigured across diverse hardware platforms. We validate RIO across the complete robot learning workflow, demonstrating its effectiveness on three morphologies (single-arm, bimanual, humanoid) and four robot platforms. We open-source RIO, and hope that it will lower the barrier for robotics practitioners to deploy, benchmark, and iterate on modern robot learning approaches across diverse hardware configurations.

Limitations and future directions. In this work, we focus on single-embodiment fine-tuning. Since cross-embodiment generalization remains an active area of research, we leave cross-embodiment fine-tuning of VLAs to future work. Additionally, while we demonstrate dynamic tasks through teleoperation, reliable dynamic task performance by VLAs remains an open research question that warrants further investigation. Future directions include systematic benchmarking of distribution shift across embodiments or extending support to include other robot hardware such as mobile manipulators and multi-fingered dexterous robot hands.

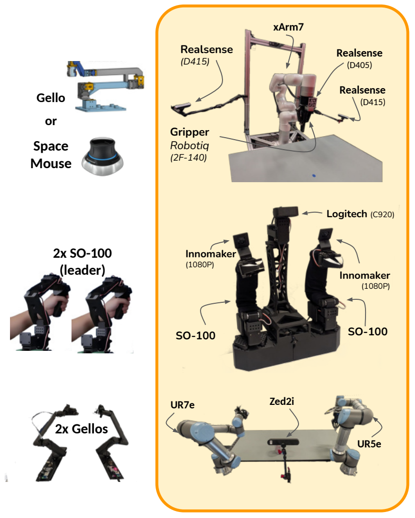

Fig. 8: **Example robot stations.** We illustrate single arm and bimanual robot stations with different cameras, controlled with different teleoperation interfaces, using RIO.

References

[1] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

[2] Lucas Beyer, Andreas Steiner, André Susano Pinto, Alexander Kolesnikov, Xiao Wang, Daniel Salz, Maxim Neumann, Ibrahim Alabdulmohsin, Michael Tschannen, Emanuele Bugliarello, et al. Paligemma: A versatile 3b vlm for transfer. *arXiv preprint arXiv:2407.07726*, 2024.

[3] Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

[4] Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A Vision-Language-Action Flow Model for General Robot Control. *arXiv preprint arXiv:2410.24164*, 2024.

[5] Kevin Black, Manuel Y Galliker, and Sergey Levine. Real-Time Execution of Action Chunking Flow Policies. *arXiv preprint arXiv:2506.07339*, 2025.

[6] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*, 2022.

[7] Herman Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, volume 3, pages 2523–2528. IEEE, 2001.

[8] Remi Cadene, Simon Alibert, Alexander Soare, Quentin Gallouedec, Adil Zouitine, Steven Palma, Pepijn Kooijmans, Michel Aractingi, Mustafa Shukor, Dana Aubakirova, Martino Russi, Francesco Capuano, Caroline Pascal, Jade Choghari, Jess Moss, and Thomas Wolf. Lerobot: State-of-the-art machine learning for real-world robotics in pytorch, 2024. URL https://github.com/huggingface/lerobot.

[9] Kaiyuan Chen, Letian Fu, David Huang, Yanxiang Zhang, Lawrence Yunliang Chen, Huang Huang, Kush Hari, Ashwin Balakrishna, Ted Xiao, Pannag R Sanketi, et al. Robo-DM: Data Management For Large Robot Datasets. *arXiv preprint arXiv:2505.15558*, 2025.

[10] Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025.

[11] Cheng Chi, Zhenjia Xu, Chuer Pan, Eric Cousineau, Benjamin Burchfiel, Siyuan Feng, Russ Tedrake, and Shuran Song. Universal Manipulation Interface: In-The-Wild Robot Teaching Without In-The-Wild Robots. In *Robotics: Science and Systems*, 2024.

[12] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025.

[13] Sudeep Dasari, Frederik Ebert, Stephen Tian, Suraj Nair, Bernadette Bucher, Karl Schmeckpeper, Siddharth Singh, Sergey Levine, and Chelsea Finn. RoboNet: Large-Scale Multi-Robot Learning. In *Conference on Robot Learning*, pages 885–897. PMLR, 2020.

[14] Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, et al. Molmo and pixmo: Open weights and open data for state-of-the-art vision-language models. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 91–104, 2025.

[15] Magnus Dierking, Christopher E Mower, Sarthak Das, Huang Helong, Jiacheng Qiu, Cody Reading, Wei Chen, Huidong Liang, Huang Guowei, Jan Peters, et al. Ark: An Open-source Python-based Framework for Robot Learning. *arXiv preprint arXiv:2506.21628*, 2025.

[16] Ria Doshi, Homer Rich Walke, Oier Mees, Sudeep Dasari, and Sergey Levine. Scaling Cross-Embodied Learning: One Policy for Manipulation, Navigation, Locomotion and Aviation. In *Conference on Robot Learning*, pages 496–512. PMLR, 2025.

[17] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. PaLM-E: an embodied multimodal language model. In *Proceedings of the 40th International Conference on Machine Learning*, pages 8469–8488, 2023.

[18] Dibya Ghosh, Homer Rich Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, et al. Octo: An Open-Source Generalist Robot Policy. In *Robotics: Science and Systems*, 2024.

[19] Markus Grotz, Mohit Shridhar, Tamim Asfour, and Dieter Fox. PerAct2: Benchmarking and Learning for Robotic Bimanual Manipulation Tasks. *arXiv preprint arXiv:2407.00278*, 2024.

[20] Tairan He, Zhengyi Luo, Wenli Xiao, Chong Zhang, Kris Kitani, Changliu Liu, and Guanya Shi. Learning human-to-humanoid real-time whole-body teleoperation. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8944–8951. IEEE, 2024.

[21] Tairan He, Zhengyi Luo, Xialin He, Wenli Xiao, Chong Zhang, Weinan Zhang, Kris M Kitani, Changliu Liu, and Guanya Shi. OmniH2O: Universal and Dexterous Human-to-Humanoid Whole-Body Teleoperation and Learning. In *Conference on Robot Learning*, pages 1516–1540. PMLR, 2025.

[22] Joey Hejna, Chethan Anand Bhateja, Yichen Jiang, Karl

Pertsch, and Dorsa Sadigh. ReMix: Optimizing Data Mixtures for Large Scale Imitation Learning. In *Conference on Robot Learning*, pages 145–164. PMLR, 2025.

[23] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, et al. $\pi_{0.5}$:a Vision-Language-Action Model with Open-World Generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[24] Tobias Jülg, Pierre Krack, Seongjin Bien, Yannik Blei, Khaled Gamal, Ken Nakahara, Johannes Hechtl, Roberto Calandra, Wolfram Burgard, and Florian Walter. Robot Control Stack: A Lean Ecosystem for Robot Learning at Scale. *arXiv preprint arXiv:2509.14932*, 2025.

[25] Simar Kareer, Karl Pertsch, James Darpinian, Judy Hoffman, Danfei Xu, Sergey Levine, Chelsea Finn, and Suraj Nair. Emergence of Human to Robot Transfer in Vision-Language-Action Models. *arXiv preprint arXiv:2512.22414*, 2025.

[26] Alexander Khazatsky, Karl Pertsch, Suraj Nair, Ashwin Balakrishna, Sudeep Dasari, Siddharth Karamcheti, Soroush Nasiriany, Mohan Kumar Srirama, Lawrence Yunliang Chen, Kirsty Ellis, et al. DROID: A large-scale in-the-wild robot manipulation dataset. In *Robotics: Science and Systems*, 2024.

[27] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P Foster, Pannag R Sanketi, Quan Vuong, et al. OpenVLA: An Open-Source Vision-Language-Action Model. In *Conference on Robot Learning*, pages 2679–2713. PMLR, 2025.

[28] Vikash Kumar, Rutav Shah, Gaoyue Zhou, Vincent Moens, Vittorio Caggiano, Abhishek Gupta, and Aravind Rajeswaran. Robohive: A unified framework for robot learning. *Advances in Neural Information Processing Systems*, 36:44323–44340, 2023.

[29] Obin Kwon, Sankalp Yamsani, Noboru Myers, Sean Taylor, Jooyoung Hong, Kyungseo Park, Alex Alspach, and Joohyung Kim. PAPRLE (Plug-And-Play Robotic Limb Environment): A Modular Ecosystem for Robotic Limbs. *arXiv preprint arXiv:2507.05555*, 2025.

[30] Obin Kwon, Sankalp Yamsani, Noboru Myers, Sean Taylor, Jooyoung Hong, Kyungseo Park, Alex Alspach, and Joohyung Kim. Paprle: Plug-and-play robotic limb environment: A modular ecosystem for robotic limbs. *IEEE Robotics & Automation Magazine*, 2026.

[31] Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. RDT-1B: a Diffusion Foundation Model for Bimanual Manipulation. In *The Thirteenth International Conference on Learning Representations*, 2025.

[32] Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running vlas at real-time speed. *arXiv preprint arXiv:2510.26742*, 2025.

[33] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science robotics*, 7(66):eabm6074, 2022.

[34] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006.

[35] Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. Pyrobot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[36] Abby O'Neill, Abdul Rehman, Abhiram Maddukuri, Abhishek Gupta, Abhishek Padalkar, Abraham Lee, Acorn Pooley, Agrim Gupta, Ajay Mandlekar, Ajinkya Jain, et al. Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6892–6903. IEEE, 2024.

[37] Younghyo Park and Pulkit Agrawal. Using apple vision pro to train and control robots, 2024. URL https://github.com/Improbable-AI/VisionProTeleop.

[38] Sabela Ramos, Sertan Girgin, Léonard Hussenot, Damien Vincent, Hanna Yakubovich, Daniel Toyama, Anita Gergely, Piotr Stanczyk, Raphael Marinier, Jeremiah Harmsen, et al. Rlds: an ecosystem to generate, share and use datasets in reinforcement learning. *arXiv preprint arXiv:2111.02767*, 2021.

[39] starVLA Community. StarVLA: A Lego-like Codebase for Vision-Language-Action Model Developing, January 2026. URL https://github.com/starVLA/starVLA.

[40] Gemini Robotics Team, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Travis Armstrong, Ashwin Balakrishna, Robert Baruch, Maria Bauza, Michiel Blokzijl, et al. Gemini robotics: Bringing ai into the physical world. *arXiv preprint arXiv:2503.20020*, 2025.

[41] RDT Team. Rdt2: Enabling zero-shot cross-embodiment generalization by scaling up umi data, September 2025. URL https://github.com/thu-ml/RDT2.

[42] Homer Rich Walke, Kevin Black, Tony Z Zhao, Quan Vuong, Chongyi Zheng, Philippe Hansen-Estruch, Andre Wang He, Vivek Myers, Moo Jin Kim, Max Du, et al. Bridgedata v2: A dataset for robot learning at scale. In *Conference on Robot Learning*, pages 1723–1736. PMLR, 2023.

[43] J Wang, M Leonard, K Daniilidis, D Jayaraman, and ES Hu. Evaluating pi0 in the Wild: Strengths, Problems, and the Future of Generalist Robot Policies, 2025. URL https://penn-pal-lab.github.io/pi0-Experiment-in-the-Wild.

[44] Junjie Wen, Yichen Zhu, Jinming Li, Minjie Zhu, Zhibin Tang, Kun Wu, Zhiyuan Xu, Ning Liu, Ran Cheng, Chaomin Shen, et al. Tinyvla: Towards fast, data-efficient vision-language-action models for robotic manipulation. *IEEE Robotics and Automation Letters*, 2025.

[45] Kun Wu, Chengkai Hou, Jiaming Liu, Zhengping Che, Xiaozhu Ju, Zhuqin Yang, Meng Li, Yinuo Zhao, Zhiyuan Xu, Guang Yang, et al. Robomind: Benchmark

on multi-embodiment intelligence normative data for robot manipulation. *arXiv preprint arXiv:2412.13877*, 2024.

[46] Philipp Wu, Yide Shentu, Zhongke Yi, Xingyu Lin, and Pieter Abbeel. Gello: A general, low-cost, and intuitive teleoperation framework for robot manipulators. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12156–12163. IEEE, 2024.

[47] Wei Wu, Fan Lu, Yunnan Wang, Shuai Yang, Shi Liu, Fangjing Wang, Qian Zhu, He Sun, Yong Wang, Shuailei Ma, et al. A Pragmatic VLA Foundation Model. *arXiv preprint arXiv:2601.18692*, 2026.

[48] Bin Xie, Erjin Zhou, Fan Jia, Hao Shi, Haoqiang Fan, Haowei Zhang, Hebei Li, Jianjian Sun, Jie Bin, Junwen Huang, Kai Liu, Kaixin Liu, Kefan Gu, Lin Sun, Meng Zhang, Peilong Han, Ruitao Hao, Ruitao Zhang, Saike Huang, Songhan Xie, Tiancai Wang, Tianle Liu, Wenbin Tang, Wenqi Zhu, Yang Chen, Yingfei Liu, Yizhuang Zhou, Yu Liu, Yucheng Zhao, Yunchao Ma, Yunfei Wei, Yuxiang Chen, Ze Chen, Zeming Li, Zhao Wu, Ziheng Zhang, Ziming Liu, Ziwei Yan, and Ziyu Zhang. Dexbotic: Open-Source Vision-Language-Action Toolbox. *arXiv preprint arXiv:2510.23511*, 2025.

[49] Mengda Xu, Han Zhang, Yifan Hou, Zhenjia Xu, Linxi Fan, Manuela Veloso, and Shuran Song. DexUMI: Using Human Hand as the Universal Manipulation Interface for Dexterous Manipulation. *arXiv preprint arXiv:2505.21864*, 2025.

[50] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. Sigmoid loss for language image pre-training. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 11975–11986, 2023.

[51] Tony Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware. In *Robotics: Science and Systems XIX*, 2023.

[52] Jinliang Zheng, Jianxiong Li, Zhihao Wang, Dongxiu Liu, Xirui Kang, Yuchun Feng, Yinan Zheng, Jiayin Zou, Yilun Chen, Jia Zeng, et al. X-vla: Soft-prompted transformer as scalable cross-embodiment vision-language-action model. *arXiv preprint arXiv:2510.10274*, 2025.

[53] Zhengbang Zhu, Minghuan Liu, Xiaoshen Han, and Zhengshen Zhang. Maniunicon: A unified control interface for robotic manipulation, 2025. URL https://github.com/Universal-Control/ManiUniCon.

Appendix

A. *Dynamic tasks with diffusion policy*

To validate RIO's ability to support high-frequency policy rollouts, we chose two dynamic tasks as a case study. Note that we train Diffusion Policy (DP) rather than VLAs, as the latter struggle to produce precise, fast action chunks.

- Ball Throwing: where the robot picks up a ball and throws it into a container.
- Tortilla flipping: where the robot must grasp a bowl containing a tortilla and move it such that the tortilla flips over.

We collect 50 teleoperated demonstrations using GELLO at 80 Hz. Table V shows the success rate across 20 trials; DP is able to achieve 66.7% for the flipping task and 100% for the throwing.

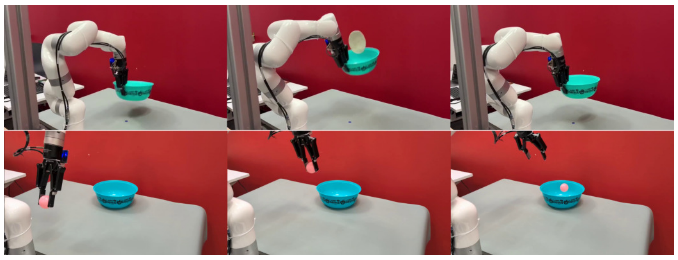
Fig. 9: **Dynamic tasks** We show keyframes from successful DP rollouts on the chosen tasks.

B. *Code specifics*

Template node. Our Node implementation is inspired by Diffusion Policy [12] and UMI [11], with a main loop that publishes state through a `ring_buffer` and processes requests received through a `request_queue`. For RIO, we provide code for a template node in Figure 12, which users can copy from to quickly implement new Nodes, such as to for a different robot or teleoperation interface. To support a range of middleware with seamless switching between them, we construct Nodes via factory functions that dynamically inherit from any middleware class that implements "publish/request" functionality. These factory functions can produce pairs of client and server nodes to automatically handle the "subscribe/response" protocol. Each middleware creates its own `ring_buffer` and `request_queue` based on `example_data` and `example_request`, along with internal functionality for message passing, that is abstracted away from the user.

Main Loop Example. RIO streamlines robot control development by generating matched Server and Client pairs from a single station configuration dataclass. The factory function introspects configuration fields and their corresponding configurations, dynamically imports modules, and instantiates node factories. Servers are launched in parallel using the server manager, while clients connect through the configured middleware layer. Robot and camera nodes can be optionally aggregated into an environment class that exposes Gym-style methods `reset()`, `step()`, `get_state()`), with the embodiment type automatically inferred from available components. Peripheral nodes not wrapped by the environment, such as teleoperation devices or visualizers, remain accessible via their configuration keys.

Within the main loop, users call node API methods directly, which internally leverage ring buffers and request queues for asynchronous interprocess communication. This architecture decouples timing constraints: servers publish sensor data and process commands at their native frequencies, while the control loop samples and issues commands at its own rate without blocking. The consistent pattern across applications enables rapid prototyping of teleoperation, policy deployment, and data collection workflows.

```python
from rio import time
from rio.envs.factory import make_env
from rio.middleware import ServerManager

# Factory function to create servers, clients,
↪  and environment based on configuration
servers, clients, env = make_env(cfg)

# Starts the servers with the desired middleware
with ServerManager(cfg.mw,
↪   list(servers.values())):
    # Start clients
    with (
        env,
        clients["teleop"]() as teleop,
    ):

    while True:
        # Query client APIs, all non-blocking
        cmd = teleop.poll()
        action = env.build_action(cmd)
        obs = env.step(action)
        time.precise_wait()
```

Fig. 10: **Example of a main loop with RIO.** Factory functions instantiate environments and custom clients from a single configuration file. Dynamic Inheritance forwards each component to the chosen middleware; once servers and clients are initialized, method calls pass through the storage structures (queues and ring buffers), avoiding blocking operations in the main loop.

The Embodiment Abstraction and State Reporting RIO introduces an embodiment abstraction layer that aggregates hardware-specific clients into coherent robot morphologies. The Base Embodiment class defines a common interface with methods for state retrieval, command execution, and action parsing. Concrete implementations such as SingleArm combine an arm client with an optional gripper and hand clients, while Bimanual pairs two arms with their respective end-effectors. During environment initialization, the factory function introspects each embodiment class's constructor signature and automatically matches required parameters against available clients from the station configuration. This design enables seamless transitions between different robot setups, from a single xArm to a dual-arm SO-100 configuration, without modifying application logic.

```
from ..schema import Observation

@dataclass
class BimanualObs(Observation):
    # Left arm (arm1)
    arm1_proprio_eef: np.ndarray | None = None
    arm1_proprio_joints: np.ndarray | None = None
    gripper1_position: float | None = None
    hand1_pose: np.ndarray | None = None
    hand1_joints: np.ndarray | None = None

    # Right arm (arm2)
    arm2_proprio_eef: np.ndarray | None = None
    arm2_proprio_joints: np.ndarray | None = None
    gripper2_position: float | None = None
    hand2_pose: np.ndarray | None = None
    hand2_joints: np.ndarray | None = None



@dataclass
class SingleArmObs(Observation):
    proprio_eef: np.ndarray | None = None
    proprio_joints: np.ndarray | None = None
    gripper_position: float | None = None

    hand_pose: np.ndarray | None = None
    hand_joints: np.ndarray | None = None
```

Fig. 11: **Example of observation schema.** Morphology-specific schemas extend a common base structure, enabling standardized state reporting across different robot configurations.

Each embodiment defines a dedicated observation structure that extends a common base schema, ensuring standardized data representation across morphologies. The embodiment queries all component states and camera data, returning a structured observation object, which is then wrapped into a step structure containing the timestep, instruction, observation, action, and metadata fields. This unified schema provides a consistent interface for downstream consumers such as policy networks, data recorders, and visualization tools, regardless of the underlying hardware configuration.

```
import numpy as np
from .. import time
from ..middleware import ClientFactory,
↪ ServerFactory
from ..node import Node


class Template(Node):
    __api__ = ["get_state", "send_req"]
    __pub__ = True
    __req__ = True

    def __init__(self, dtype=np.float32, *, freq:
    ↪ int = 100, **kwargs):
        self.dtype = dtype
        super().__init__(freq=freq, **kwargs)

    def __post_init__(self):
        self.example_data = {
            "state": np.array(...,
            ↪ dtype=self.dtype),
            "timestamp": time.now()}
        self.example_request = {"value":
        ↪ np.array(..., dtype=self.dtype)}
        self.run = self.pubreq
        super().__post_init__()

    def pubreq(self):
        rate = time.Rate(self.freq)
        self.pub_ready_event.set()
        self.req_ready_event.set()

        while not self.exit_event.is_set():
            # Publish state
            data = {"state": 0.0, "timestamp":
            ↪ time.now()}
            self.ring_buffer.put(data)

            # Fetch requests
            reqs = self.request_queue.get_all()
            for req in reqs:
                # Handle request...

            rate.precise_sleep()

    def get_state(self, k=None, out=None):
        return (
            self.ring_buffer.get(out=out)
            if k is None
            else self.ring_buffer.get_last_k(k=k,
            ↪ out=out)
        )

    def send_req(self, value):
        self.request_queue.put({"value": value})

def TemplateServer(mw, *args, **kwargs):
    return ServerFactory(mw, Template, *args,
    ↪ **kwargs)


def TemplateClient(mw, *args, **kwargs):
    return ClientFactory(mw, Template, *args,
    ↪ **kwargs)
```

Fig. 12: **Template node.** Nodes are constructed with a factory function by dynamic inheritance from any middleware class that implements publish/request functionality, allowing for seamless switching between different middlewares. Paired client-server nodes automatically handle subscribe/response.

Table V: **Policy deployment for dynamic tasks (Diffusion Policy).** We showcase that RIO can successfully roll out challenging dynamic tasks, achieving competent success rates across 20 demonstrations on a throwing and a flipping task. We also showcase that we can do so without a significant slowdown compared to the average demonstration time in the finetuning set.

Robot	Policy	Task	Success Rate (%)	Task Completion Time (s)	Demo Time (s)	RAM (GB)	CPU (%)	GPU Util (%)	GPU Mem (%)
xArm7	BC DP	Flip Tortilla	66.7	12.36 ± 2.57	7.59s ± 0.79	17.2 ± 0.1	8.8 ± 1.9	8.6 ± 3.7	9.3 ± 0.4
xArm7	BC DP	Throw Ball	100.0	14.73 ± 1.28	13.26 ± 2.23	15.8 ± 0.8	6.8 ± 0.1	0.6 ± 0.8	9.1 ± 0.6