RIO System Improvements and Integration

Bhaswanth Ayapilla
Robotics Institute
Carnegie Mellon University
bayapill@andrew.cmu.edu

1 Introduction

Robot I/O (RIO) [1] is an open-source Python-based framework that supports a wide range of robot embodiments, control policies, and teleoperation interfaces under a common infrastructure. As the framework matures, expanding its capabilities to support new hardware, control paradigms, and learning-based policies becomes increasingly important for both research and real-world deployment.

This report documents work carried out to improve and extend the RIO framework across several fronts. Teleoperation data was collected across single-arm and bimanual robot platforms for a variety of manipulation tasks, including pick and place, cloth folding, and bowl scrubbing, as well as on industrial robots at Bosch AI. This data was used for policy training and evaluation in support of the RIO paper submission. VR headset integration was completed for the Pico platform using the XRoboToolkit SDK, enabling immersive whole-body teleoperation within the RIO stack. A motion tracking policy built on top of the GentleHumanoid codebase was integrated into RIO as a Node, enabling the Unitree G1 humanoid to track reference motions from live VR input. LiDAR support for the G1 was added via a new RIO Node that subscribes to the robot's onboard Livox MID-360 sensor and publishes parsed point clouds. Finally, an industrial robot arm from Bosch AI (the Kassow KR series) was onboarded into the RIO stack via a C++ real-time bridge.

The remainder of this report is organized as follows. Section 2 reviews the relevant literature and systems. Section 3 describes the system integration and development work. Sections 4 through 6 cover the evaluation plan, experiments, and results, respectively. Section 7 concludes the report.

2 Literature Review

2.1 RIO System

Robot I/O (RIO) is an open-source Python-based framework developed at Carnegie Mellon University for flexible real-time robot control across diverse hardware platforms and morphologies. The core motivation behind RIO is to address the fragmentation that exists in robotics infrastructure, which is that most robot code is tightly coupled to a specific hardware setup, making it difficult to reuse, share, or extend across platforms.

RIO is built around five design principles: flexible, reusable, accessible, performant, and consistent. At its core, it provides a Node abstraction for hardware components, such as robots, sensors, teleoperation interfaces, and policies, that communicate through middleware backends ranging from shared memory for low-latency local communication to network-based options like Zenoh for distributed deployments. This design allows users to swap out any component of the stack with minimal reconfiguration effort.

A key contribution of RIO is its policy inference interface, which defines a lightweight protocol for integrating learning-based policies into the control stack. Instead of building around a specific model architecture or requiring a dedicated inference server, RIO only requires users to implement a small set of methods, making it straightforward to bring new policies online. The framework has been validated across single-arm, bimanual, and humanoid embodiments, and supports state-of-the-art

VLAs including $\pi 0.5$ and GR00T. The work in this report extends RIO's teleoperation and policy support.

2.2 Motion Tracking and GentleHumanoid

Motion tracking has emerged as a scalable training paradigm for humanoid whole-body control. Rather than specifying tasks through reward engineering, motion tracking policies are trained to imitate large corpora of retargeted human motion capture data, providing dense, frame-by-frame supervision. Common training datasets include AMASS, InterX, and LAFAN, which are retargeted to the target humanoid using tools like GMR (General Motion Retargeting), and filtered to remove motions that are too dynamic or otherwise incompatible with interaction scenarios.

GentleHumanoid, extends the standard motion tracking paradigm by incorporating upper-body compliance. Most existing RL-based tracking policies emphasize rigid tracking and tend to suppress external forces, whereas GentleHumanoid integrates impedance control into the tracking policy through a unified spring-based formulation that handles both resistive contacts and human-guided guiding contacts. The framework achieves coordinated compliance responses across the shoulder, elbow, and wrist while maintaining unified interaction modeling, and supports tunable force thresholds to ensure safe human-robot interaction. It is demonstrated on the Unitree G1 and generalizes across diverse motions and interaction types.

The `motion_tracking` codebase used in this project is built on top of the GentleHumanoid inference and deployment stack. It deploys the trained policy as an ONNX model that runs at 50 Hz, taking a flat proprioceptive observation vector as input and outputting joint position deltas relative to a default standing pose. The sim-to-real pipeline uses MuJoCo for sim2sim validation before deployment on physical hardware, and a ZMQ bridge for accepting teleoperation pose commands at runtime. This codebase structure makes it possible to slot the policy into RIO as an external inference node without modifying its internals.

2.3 VR Teleoperation Interfaces

VR-based teleoperation has become an increasingly practical approach for humanoid data collection and control, driven by the availability of consumer headsets with built-in body tracking. The core idea is to stream human pose estimates from a headset directly to the robot, either as raw joint targets or as SMPL body model parameters that are then retargeted to the robot's kinematic structure.

Systems like OmniH2O demonstrated that a universal whole-body teleoperation policy trained on retargeted AMASS data can support diverse human control interfaces including VR headsets, RGB cameras, and language commands within the same policy. More recent work on TWIST2 showed that using a PICO4U VR headset for real-time whole-body motion capture, combined with a custom robot neck for egocentric vision, enables holistic human-to-humanoid teleoperation at scale, with the ability to collect 100 demonstrations in under 15 minutes.

The NVIDIA GEAR-SONIC system formalizes this streaming pipeline. It uses the PICO whole-body motion-tracking interface to obtain full-body human pose estimates in SMPL format, which are transmitted over ZMQ and used directly as motion tracking targets for the deployed policy. This architecture supports multiple operating modes, allowing the operator to switch between a locomotion planner mode driven by velocity commands and a full-body SMPL pose streaming mode from the headset.

In this project, both the Meta Quest 3 and Pico platforms were integrated into the RIO stack as teleoperation sources. The XRoboToolKit SDK is used to stream SMPL poses from the headset to the host machine, which are then forwarded over ZMQ into RIO's teleoperation input pipeline. This allows the same VR teleoperation infrastructure to drive both the motion tracking policy and, in future work, higher-level VLA policies operating within the RIO framework.

3 System Integration & Development

3.1 Teleoperation Data Collection and Policy Evaluation

A significant portion of the work involved collecting high-quality teleoperation demonstration data across multiple robot platforms and task types, in support of the RIO paper submission. Data was collected on single-arm setups (UFactory xArm7) for pick-and-place tasks, and on bimanual setups (SO-101 and UR arms) for cloth folding and bowl scrubbing. All demonstrations were collected at 50 Hz using a Spacemouse as the primary teleoperation interface for single-arm tasks and a GELLO leader-follower interface for bimanual tasks.

The data was recorded in a compressed RLDS-style format using RoboDM, which supports multiple camera streams alongside proprioceptive state. For reference, 150 episodes with three camera views require only 1.31 GB of storage. This data was then used to fine-tune state-of-the-art VLAs: $\pi 0.5$ was fine-tuned from the DROID checkpoint for single-arm tasks and from the ALOHA checkpoint for bimanual tasks, each trained for 20,000 steps. GR00T N1.5 was fine-tuned using 150 humanoid manipulation demonstrations.

Policy evaluation results reported in the RIO paper show that single-arm pick-and-place tasks (fold shirt, place can) achieved success rates above 92% across 20 trials, with task completion times within two seconds of demonstration time. Bimanual tasks (cloth folding, bowl scrubbing) achieved at least 60% success. The humanoid pick-box task using GR00T N1.5 achieved 95% success. These results validate that demonstrations collected through RIO are high-quality and sufficient for effective VLA fine-tuning.

3.2 VR Headset Integration

The Pico VR headset was integrated into RIO as a teleoperation source using a two-layer architecture: a low-level RIO Node that wraps the XRoboToolkit SDK, and an application-level teleoperation script that consumes the node's output.

XRoboToolkit Node The XRoboToolkit Node runs a 100 Hz publish loop, continuously pushing controller poses, button states, headset pose, and optionally a full 24-joint body skeleton into a ring buffer. The key technical challenge was coordinate system conversion. The XRoboToolkit SDK is built on Unity, which uses a left-handed coordinate system, whereas RIO and the robot operate in a right-handed frame. Every pose coming from the headset must be converted before it can be used for robot control.

The coordinate transform applied is:

$$T = \begin{bmatrix} 0 & 0 & -1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Position is converted as $p_{\text{robot}} = p_{\text{unity}} T^\top$, and rotation as $R\_\text{robot} = T R_{\text{unity}} T^\top$. This transform is applied to every pose field including left controller, right controller, headset, and all 24 body skeleton joints.

Teleoperation Control Loop The teleoperation script consumes the XRoboToolkit node at 30 Hz. A deadman switch design is used for safety: the right controller's A button acts as the engagement trigger. On press, the system snapshots the current robot TCP pose and the current controller pose as a shared origin. All subsequent motion is computed as deltas from that snapshot, preventing sudden jumps when teleoperation is engaged mid-motion.

Position deltas are computed as robot_delta $= T(\text{current\_pos} - \text{start\_pos})$. Rotation deltas use $SO(3)$ composition: delta_rot $=$ current_rot start_rot^{-1}, applied on top of the robot's starting orientation. The final 6-DOF TCP command is sent to the arm each cycle. When body tracking is enabled, the node additionally publishes the full 24-joint body skeleton as arrays of shape $[24, 3]$ for positions and $[24, 4]$ for quaternions, which serves as the motion source for the whole-body motion tracking policy described in Section 3.4.

3.3 LiDAR Integration

The Unitree G1 humanoid carries a Livox MID-360 LiDAR that publishes point cloud data over Unitree's DDS middleware. A new G1Lidar RIO Node was built to subscribe to this sensor and expose parsed point clouds to the rest of the RIO stack.

PointCloud2 Parsing The LiDAR publishes data in raw ROS PointCloud2 format over the DDS topic `rt/utlidar/cloud_livox_mid360`. This format is a binary blob where each point's fields (x, y, z, intensity, etc.) are stored at specific byte offsets with different datatype codes. The parser reads the field map from the message metadata, extracts the XYZ bytes at the correct offsets using NumPy's `frombuffer` with the appropriate dtype, casts them to float32, and filters out any non-finite values. If the resulting cloud exceeds the configured maximum (default 50,000 points), random downsampling is applied to cap memory usage. The output is a clean (N, 3) float32 NumPy array published into the ring buffer at 10 Hz.

Node Integration The `G1Lidar` Node follows RIO's standard publish-only pattern. On startup, it initializes a `ChannelSubscriber` on the DDS topic and registers a callback that runs the parser whenever a new message arrives. A rate-limited loop keeps the Node alive and satisfies RIO's lifecycle contract. Once the subscriber is successfully initialized, `pub_ready_event` is set so the rest of the system knows data is incoming. The Node exposes a single method, `get_pointcloud()`, which returns the latest point cloud as an (N,3) `float32` array, or `None` if no message has been received yet.

The G1Lidar node is used in the full G1 integration script alongside the robot, WBC policy, Pico teleoperation interface, and Rerun visualizer. Point clouds are streamed to Rerun each cycle for live 3D visualization of the robot's environment.

3.4 Motion Tracking Integration

The `motion_tracking` codebase is a PPO-trained whole-body policy that tracks reference motion clips on the Unitree G1 at 50 Hz. It was not designed to work with RIO: it uses bare imports, assumes execution from inside `sim2real/src/`, and relies on a Controller object as a live state bus that its observation modules read directly. The integration goal was to make this policy run as a standard RIO Node without modifying the upstream codebase.

_MockController: State Format Adapter The `motion_tracking` observation modules read attributes such as `ctrl.qj_isaac`, `ctrl.quat`, and `ctrl.gyro` directly from the Controller object. `_MockController` replicates this exact interface but populates it from RIO's `G1_29` state dictionary instead of the original Unitree SDK. Two convention mismatches must be resolved on every inference cycle:

- **Joint ordering:** `G1_29` publishes joints in real SDK order, but the policy was trained in Isaac Gym which uses a different ordering. A JointMapper is applied on every update to remap the joint state arrays between the two orderings.
- **Quaternion convention:** `G1_29` internally converts orientation from the Unitree SDK's WXYZ convention to XYZW. The `motion_tracking` observation modules expect WXYZ. `_MockController` converts back on each update: `quat_wxyz = quat_xyzw[[3, 0, 1, 2]]`.

MotionTrackingPolicyNode `MotionTrackingPolicyNode` is the RIO Node that wraps the inference loop. Its `pubreq()` background thread performs the following sequence at startup: it injects `sim2real/src` into `sys.path` at runtime to satisfy the upstream codebase's bare import assumptions; lazily imports `TrackingPolicyRaw`; loads `controller.yaml` and `tracking.yaml` using `DictToClass`; builds a `_MockController` instance; instantiates the policy; and calls `fade_in()` to reset policy state.

On each cycle, the inference loop pulls the latest robot state from the request queue and calls `_MockController.update(obs)` to populate the controller interface. From there, `update_obs()` assembles the flat proprioceptive observation vector, `compute_action()` runs a forward pass through the ONNX model, and `post_step()` handles any cleanup. The policy outputs joint position deltas

in Isaac Gym ordering, relative to the default standing pose. Before sending these to the robot, the node adds `default_qpos_real` to recover absolute joint targets and reorders them into real SDK order. The resulting target is pushed to the ring buffer and retrieved by the main control loop via `get_action_latest()`.

G1_29 Modifications Three small modifications were made to the existing `G1_29` Node to support this integration. The first adds a `sim=True` flag that skips `enter_debug_mode()` on startup. Without this, `MotionSwitcherClient.CheckMode()` is called and crashes immediately when running against a loopback interface during sim2sim testing. The second adds optional per-joint `motor_kp` and `motor_kd` arrays so that `write_cmd()` uses the policy's training gains rather than `G1_29`'s hardcoded defaults; the `motion_tracking` policy was trained with significantly different values (e.g., hip: 99.1, ankle: 28.5, arms: 14.3) and using the wrong gains produces unstable motion. The third sends a `reserve[0] = 1` signal after Stage 1 completes, which `sim2sim.py` uses as a flag to unfreeze MuJoCo physics. Without it, the simulated robot stays frozen at its initial pose regardless of commands.

Sim2Sim Testing Sim2sim testing is supported via `motion_tracking`'s `sim2sim.py`, which runs a MuJoCo simulation over a DDS loopback interface acting as a fake robot. The runner script `play_g1_motion_tracking.py` wires `G1_29` and `MotionTrackingPolicyNode` together under RIO's `ServerManager`. The main loop is straightforward: read the latest robot state, pass it to `policy_node.step()`, retrieve the resulting joint targets via `get_action_latest()`, and forward them to the robot via `robot.moveJ(target, t_cycle_end)`. Running with `-g1-cfg.network-interface lo -g1-cfg.sim` points the stack at the loopback interface and disables the real hardware initialization path.

3.5 Bosch Robot Integration

The Kassow KR series is a 7-DOF collaborative arm used at Bosch AI. Its SDK (kord-api) is C++-only and UDP-based, and it has a torque deviation monitor that suspends the robot if a received position command differs from the current joint state by more than a small threshold. Every previous attempt at real-time control triggered this fault, because commands were arriving at the wrong time, with wrong values, or too many at once.

The C++ RT Bridge The core design decision was to isolate the real-time constraint entirely in C++. `KordBridge` is a C++ shared library that runs a dedicated `std::thread` which owns the kord session entirely. Python never calls kord directly; it only writes targets into a mutex-protected struct, and the RT thread picks them up each tick.

The RT thread does exactly three things in order on every tick: wait on `waitSync()`, send a command, then call `fetchData()`. The ordering matters more than it might seem. `waitSync()` signals that the robot's controller is ready for a command in a tight window, so the command needs to go out immediately after. The first version of the bridge called `fetchData()` before sending the command, which meant every command arrived one tick late. That single ordering mistake turned out to be the root cause of the torque deviation faults across multiple debugging sessions. The robot also runs at 250 Hz rather than the initially assumed 125 Hz, which was confirmed during experiments. The bridge is exposed to Python as `_kord_bridge.so` via nanobind, and attempts `SCHED_FIFO` real-time scheduling on the RT thread via `pthread_setschedparam`, failing gracefully if the process does not have `CAP_SYS_NICE`.

Command Modes The bridge supports four command types. `DJC` (directJControl) sends absolute joint position, velocity, and acceleration directly to the servo every tick. It is maximally responsive but very sensitive to timing — even one late command causes a torque spike. `VelCmd` has Python write a joint velocity and lets the RT thread integrate it into an absolute position reference at 250 Hz, which removes Python scheduling jitter from the position reference entirely. `StreamJ` is what ended up being used for teleoperation: Python writes an absolute target joint position and the RT thread sends it as a `moveJ` with `OT_VIAPOINT` blending, which makes it tolerant of Python timing variations. `MoveJ` and `MoveL` are one-shot point-to-point commands and are not suitable for streaming.

StreamJ Design Decisions Getting `StreamJ` to work reliably required working through several issues.

The original code used `OT_STOPPOINT`, which tells the robot to decelerate to a full stop at each commanded position before accepting the next one. With 2-second time budgets per command, even 0.4 seconds of key-pressing enqueued roughly 100 seconds of deferred motion. This is exactly why the robot only started moving after the script was stopped, since it was draining the backlog. Switching to `OT_VIAPOINT` tells the robot to blend through each waypoint on the way to the next one, which eliminates the queue buildup entirely.

A dirty flag was added so the bridge only sends a `moveJ` when Python has actually written a new target since the last send. Without this, the bridge floods the robot's motion buffer with 250 identical hold-position commands per second during idle periods, which causes the buffer to overflow and the robot to suspend after a few seconds. With the dirty flag in place, the command rate drops to 0 Hz during hold and the robot simply maintains its last position.

Even with the dirty flag, sending a command on every tick while a key is held resulted in 250 Hz, which was too fast for the robot's trajectory planner to keep up with. A throttle counter in the RT thread sends a `moveJ` only every N ticks. With `throttle=2` the effective command rate during motion is 125 Hz, which is stable. The throttle is stored as a `std::atomic<int>` so it can be adjusted from Python at runtime without recompiling.

The original streaming attempt used `TT_TIME=0.008`, which caused disconnections because telling the robot to reach a target in 8ms creates enormous acceleration demands if there is any position offset at the first command. Switching to `TT_JS_TARGET_SPEED` fixed this: bounding the joint speed at no more than X rad/s is physically safe regardless of the initial offset. However, at 125 Hz `TT_JS_TARGET_SPEED` eventually caused suspensions because the speed-constrained trajectory optimization was too slow to keep up with the command rate. `TT_TIME` is faster to plan since it is a fixed-duration interpolation with no optimization pass, and the switch made 125 Hz stable. The working operating point is 125 Hz `moveJ` commands at `throttle=2` with `TT_JS_TARGET_SPEED`.

RIO Node Integration `KassowArm` wraps the bridge as a RIO Node using the standard pub/req pattern. A pub thread polls bridge state at 125 Hz and pushes it into a ring buffer. A req thread drains incoming commands and calls the appropriate bridge methods. The public API exposes `get_state()`, `moveJ()`, `moveL()`, `streamJ()`, `directJControl()`, and `velocityJ()`, making `KassowArm` a drop-in replacement for any other arm in RIO.

4 Evaluation Plan

The evaluation covers two dimensions: functional correctness of each integration and end-to-end system performance.

For the VR headset integration, correctness is evaluated by verifying that controller poses stream at the expected 100 Hz, that coordinate frame conversion is accurate (robot and headset frames are visually aligned in Rerun), and that the deadman switch engages and disengages motion cleanly without jumps.

For the motion tracking policy integration, the primary evaluation is sim2sim validation: the policy should produce stable joint trajectories in MuJoCo when driven by the default UDP motion source, with the robot maintaining an upright pose throughout. End-to-end latency between the `G1_29` state publication and the policy's action reaching `robot.moveJ()` is also measured under both Thread and Shared Memory middleware.

For the LiDAR integration, correctness is evaluated by verifying that the parsed point cloud dimensions match the expected sensor output, that non-finite values are correctly filtered, and that the cloud is visible and spatially coherent in Rerun visualization.

For the Kassow arm integration, correctness is evaluated along two axes: frequency and fault behavior. The bridge exposes two counters in its state struct: `tick`, which increments on every `waitSync()` call, and `stream_j_sends`, which increments on every `moveJ` that reaches the robot. Dividing each by elapsed time gives the actual sync rate and command rate respectively. Correct behavior is: sync rate confirmed at 250 Hz, command rate at roughly 125 Hz during motion, and 0 Hz during hold.

The absence of torque deviation faults, disconnections, and command buffer overflows is used as the primary indicator of stable operation.

5 Experiments

Teleoperation data collection experiments were conducted across all supported task types. Single-arm pick-and-place and dynamic tasks were collected on the UFactory xArm7 with a Robotiq 2F-140 gripper and three RealSense cameras. Bimanual cloth folding and bowl scrubbing were collected on SO-101 arms using a GELLO leader-follower interface. All sessions ran at 50 Hz with three camera views.

Pico VR integration was tested with live controller streaming on a mobile xArm7 station. Coordinate frame alignment was verified visually by comparing the Rerun visualization of controller poses against the robot's workspace. The deadman switch was tested across repeated engagement and disengagement cycles to confirm no positional jumps occurred.

Motion tracking policy integration was tested via sim2sim using `motion_tracking`'s MuJoCo simulator over a DDS loopback interface. The policy was driven by the default UDP motion source (internal motion clip loop) to isolate policy behavior from VR input. Key debugging milestones included: resolving the `sim=True` flag to prevent initialization crashes on loopback, correcting the joint gain injection from `controller.yaml`, fixing the `reserve[0]` signal to unfreeze MuJoCo physics, and correcting `default_dof_pos` to target the half-squat pose rather than zeros during Stage 1.

Kassow arm teleoperation was tested on hardware at Bosch AI. The teleop script runs at 250 Hz to match the robot. Each cycle it reads keyboard or Spacemouse input, scales the delta by `max_rot_speed` (0.3 rad/s) and $dt = 1/250$ s, adds it to a running absolute joint target, clips to joint limits, and calls `streamJ()` if the delta is nonzero. Frequency counters were printed every 2 seconds throughout the session to monitor sync rate and command rate in real time. Several failure modes were encountered and resolved during this process, including the `fetchData()` ordering bug, the `OT_STOPPOINT` queue buildup, the motion buffer overflow from flooding during hold, and the `TT_TIME` vs `TT_JS_TARGET_SPEED` tradeoff at different command rates.

6 Results & Discussion

Teleoperation data collection contributed directly to the RIO paper submission. Policy evaluation results show that single-arm tasks achieved success rates of 92.5% (fold shirt) and 95% (place can) across 20 trials, with completion times within two seconds of demonstration time. Bimanual tasks achieved 60% (fold cloth) and 64% (scrub bowl). The humanoid pick-box task using GR00T N1.5 achieved 95% success. These results confirm that demonstrations collected through RIO are sufficient for effective VLA fine-tuning.

The Pico VR integration was successfully validated on the mobile xArm7 station. Controller poses streamed reliably at 100 Hz. Coordinate frame conversion was confirmed accurate through Rerun visualization. The deadman switch pattern proved effective in practice, eliminating positional discontinuities on engagement.

The motion tracking integration was validated through sim2sim testing. End-to-end policy inference runs stably through the full loop, with the simulated G1 maintaining an upright pose and tracking reference motions correctly. Live Pico VR teleoperation was also tested as the motion source: moving in the headset produced corresponding whole-body motion in the simulated robot, confirming that the VR streaming pipeline feeds through correctly into the policy. The sim flag, gain injection, `reserve[0]` signal, and default pose target were all resolved during debugging. Sim-to-real transfer on physical G1 hardware remains to be tested.

The LiDAR integration was validated on G1 hardware. Point clouds rendered in Rerun are spatially coherent and update in real time as the environment changes. Moving near the robot produced clearly visible changes in the point cloud, confirming that the sensor is capturing live geometry rather than stale or cached data. The `PointCloud2` parser correctly handles the binary field layout and produces clean `float32` XYZ arrays. Downsampling to 50,000 points keeps memory use reasonable without any noticeable drop in cloud quality.

The Kassow arm integration is stable on hardware. The robot runs at a confirmed 250 Hz sync rate, receives 125 Hz `moveJ` commands during motion, and holds position cleanly with no commands sent during idle periods. There are no torque deviation faults, no disconnections, and no command buffer overflows. The fix that unlocked stable operation was straightforward once identified: `waitSync()` must be followed immediately by the command send, with `fetchData()` called after. Reversing this order delays every command by one tick, which is enough to trigger hard faults consistently. Joint limits are enforced in Python before any command reaches the robot.

7 Conclusion

This report documents five contributions to the RIO framework: teleoperation data collection for VLA training, Pico VR headset integration, motion tracking policy integration for the G1 humanoid, LiDAR integration for the G1, and Bosch Kassow arm integration via a C++ real-time bridge. Each contribution follows RIO's Node architecture, making them composable with any other component in the stack without modification.

Across these integrations, a common pattern emerged: the most effective approach is to build a thin adapter layer that translates between the external system's interface and RIO's conventions, without modifying either side. This was most clearly demonstrated by `_MockController` in the motion tracking integration and by `KordBridge` in the Kassow arm integration, both of which isolate convention mismatches or language-level constraints at a well-defined boundary while leaving the rest of the system unchanged.

The work contributed to the RIO paper submission and to the broader goal of making the framework useful across a wider range of hardware and policy types. Several directions for future work follow naturally from the current integrations. The LiDAR node provides a foundation for mapping and SLAM on the G1 humanoid — the point clouds are already available as a standard RIO Node, so integrating an online mapping library is a straightforward next step. The `KordBridge` architecture is not specific to the Kassow arm; any robot with a C++-only real-time API and a strict heartbeat requirement can be onboarded using the same pattern, making it a reusable template for future industrial robot integrations within RIO. On the humanoid side, the remaining priority is sim-to-real transfer of the motion tracking policy on physical G1 hardware with live Pico VR teleoperation as the motion source.

References

[1] Pablo Ortega-Kral, Eliot Xing, Arthur Fender Coelho Bucker, Vernon Luk, Jason Kim, Owen Kwon, Angchen Xie, Nikhil Sobanbabu, Yifu Yuan, Megan Lee, Deepam Ameria, Bhaswanth Ayapilla, Jaycie Bussell, Guanya Shi, Jonathan Francis, and Jean Oh. RIO: Flexible Real-time Robot I/O for Cross-Embodiment Robot Learning. In *Robotics: Science and Systems (RSS)*, 2026.