OPENRUA: ROBOT-USE AGENTS ARE ZERO-SHOT VISUOMOTOR POLICIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are extending their reach into the physical world by writing and executing robot control programs. One might expect the agents to use the same mature software stack that engineers have developed over decades. Yet existing work surrounds these agents with specially engineered interfaces for robot use, wrapping learned policies as tools or supplying hand-crafted perception and control primitives. We introduce OpenRUA, a harness that bypasses bespoke abstraction layers by providing off-the-shelf coding agents with only terminal access to the robot's native software interface ROS 2. OpenRUA employs a zero-abstraction *workspace-as-harness* design, only offering ROS 2 documentation and basic tools while leaving the coding agent to organize its own work without orchestrating any agentic workflow. OpenRUA achieves success rates of 99.0% on CaP-Bench and 87.0% on LIBERO-PRO, demonstrating that an off-the-shelf coding agent can serve as a zero-shot visuomotor policy through the robot's native interface, without bespoke primitives or task-specific training. (1) For perception, the agent spontaneously writes programs that process raw sensory inputs and derive metric measurements in 96.80% of episodes. (2) For manipulation, the agent spontaneously builds motion-control clients in 95.87% of episodes and closed-loop control programs in 50.13% of episodes. Our code is available at https://anonymous.4open.science/r/OpenRUA-34210.

1 INTRODUCTION

Robot control has long been engineered by writing explicit programs that translate task specifications into physical motion (Fikes & Nilsson, 1971; Li et al., 1994; Siciliano et al., 2009). These programs execute on a multi-layered software stack, through which human engineers coordinate drivers, messaging, planning, and actuation to implement the required behavior (Macenski et al., 2022; García et al., 2020). However, this process requires substantial effort and human expertise, with further engineering often needed for new tasks or environments.

One path is to learn end-to-end *visuomotor policies*, such as *Vision-Language-Action* (VLA) models (Team, 2025; Kim et al., 2024; Intelligence et al., 2026; Bjorck et al., 2025) and *World Action Models* (WAMs) (Agarwal et al., 2026; Ye et al., 2026; Li et al., 2026), that map visual observations and language instructions directly to low-level actions. However, they require large-scale demonstration data, and their generalization remains limited beyond the training distribution (Zhou et al., 2025; Fei et al., 2025; Kim et al., 2026). For example, LIBERO-PRO (Zhou et al., 2025) reports that OpenVLA (Kim et al., 2024) and π_0 (Black et al., 2024), despite average success rates above 90% on standard LIBERO, achieve 0.0% success under position and task perturbations.

Another path is to *revisit* explicit programming for robot control (Fu et al., 2026), while replacing the human engineer with modern coding agents such as Claude Code (Anthropic, 2025) or Codex (OpenAI, 2025). Such an approach is known as *Code as Policies* (CaP) (Liang et al., 2023), where the policy itself can be inspected, edited, and debugged by the coding agent without retraining (Elmaaroufi et al., 2026; Lu et al., 2026). As a result, the coding agent acts as a *Robot-Use Agent* (RUA) (Isola, 2026), operating the robot the way it operates a computer (Team, 2025; Fu et al., 2026; Robocurve, 2026; Berman et al., 2026).

Despite this progress, existing RUA systems devote significant effort to engineering the tool interfaces and primitive libraries through which agents compose policies to operate robots. These

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

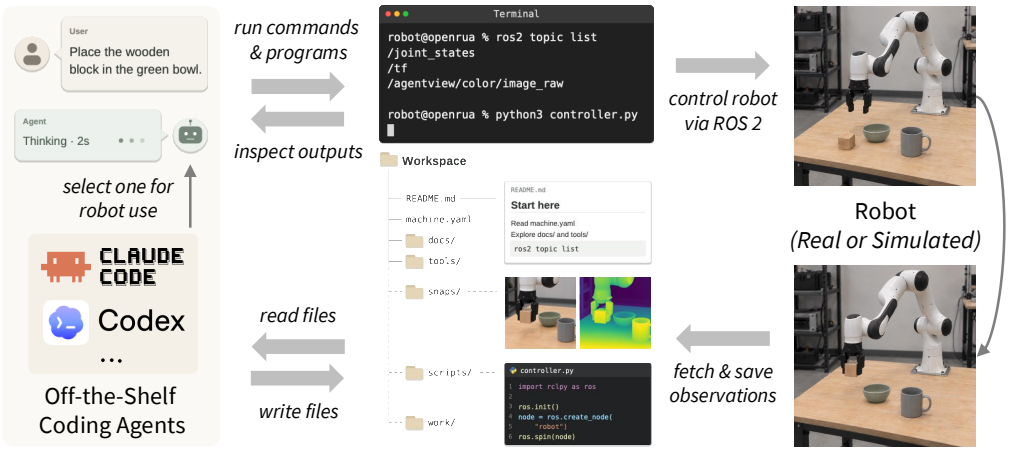


Figure 1: **OpenRUA: a zero-abstraction workspace-as-harness for robot-use agents.** Given a task such as placing the block in the bowl, OpenRUA enables any off-the-shelf coding agent to operate a real or simulated robot through its native ROS 2 interface. OpenRUA only provides a workspace with ROS 2 documentation and basic tools, making robot perception accessible through file I/O and manipulation through coding, without orchestrating any agentic workflow. Within this workspace, the agent exhibits emergent perception and control capabilities as it works to complete the task. It writes and executes `ros2cli` commands or `rclpy` programs that send control requests to the robot's controllers, here grasping and lifting the wooden block. The agent saves new RGB-D observations as workspace files and uses images, computed measurements, and terminal feedback to autonomously verify progress and refine its actions until the block is placed in the bowl.

systems either wrap learned visuomotor policies as callable tools (Zhang et al., 2026b; Galanti et al., 2026; Ding et al., 2026; Wang et al., 2026a) or hand-craft custom libraries of perception and control primitives based on human expertise (Fu et al., 2026; Lu et al., 2026; Elmaaroufi et al., 2026; Zhang et al., 2026a). However, it remains unclear whether such additional interface engineering is necessary for coding agents to operate robots effectively, given that robots have a mature ecosystem of software interfaces developed over decades for human engineers to access sensors and control motion (Macenski et al., 2022; García et al., 2020). This raises the question: *"Can an off-the-shelf coding agent serve as a zero-shot visuomotor policy through the robot's native software interface, without relying on bespoke primitives or learned policies?"*

In this work, we introduce OpenRUA, a simple yet effective harness for robot-use agents that builds on the robot's native interface, *Robot Operating System 2* (ROS 2) (Macenski et al., 2022), the de facto standard of the robotics community. OpenRUA bypasses bespoke abstraction layers by offering off-the-shelf coding agents only terminal access to ROS 2, allowing them to operate the robot as a human engineer would.

As illustrated in Figure 1, OpenRUA frames robot use as a live, interactive software project and employs a minimalist *workspace-as-harness* design. This design rests on the observation that frontier coding agents already operate autonomously on unfamiliar repositories to complete tasks (OpenAI, 2024; Merrill et al., 2026; Chu et al., 2026). We hypothesize that the orchestration with bespoke interfaces in existing RUA systems compensates for limited agent capability rather than reflecting a requirement of robot control, and that a capable coding agent needs only a workspace. Specifically, OpenRUA only offers ROS 2 documentation and basic tools in a workspace directory, leaving the coding agent to organize its own work without a prescribed agentic workflow. Within this workspace, OpenRUA recasts perception as file I/O and manipulation as coding: task-specific perception and control strategies emerge as the agent acquires observations, interprets images, computes metric measurements, and writes control programs. As a result, the coding agent itself serves as a zero-shot visuomotor policy: it goes from raw observations to motor commands, reasoning explicitly about geometry and iteratively refining its actions from execution feedback.

We conduct comprehensive experiments to evaluate OpenRUA on three widely used benchmarks, *i.e.*, CaP-Bench (Fu et al., 2026), LIBERO-PRO (Zhou et al., 2025), and RoboCasa365 (Nasiriany et al., 2026), in a zero-shot setting without access to privileged simulator state and the benchmark's

success predicates. With Claude Code powered by Claude Opus 5, OpenRUA achieves success rates of 99.0% on CaP-Bench and 87.0% on LIBERO-PRO, exceeding leading systems such as ASPIRE (Lu et al., 2026) and Harness VLA (Zhang et al., 2026b) by 18.0 and 4.6 percentage points, respectively. The results show that an off-the-shelf coding agent can serve as a zero-shot visuomotor policy through the robot's native software interface, without bespoke primitives or learned policies.

Further analysis reveals several findings about emergent agent capabilities: (1) For perception, the agent spontaneously writes programs that process raw sensory inputs (*e.g.*, cropping images, adding grids, and reconstructing point clouds from depth) and derive metric measurements (*e.g.*, computing coordinates for Cartesian motion targets) in 96.80% of episodes. (2) For manipulation, the agent spontaneously builds its own motion-control clients (*e.g.*, trajectory execution and gripper control) in 95.87% of episodes and closed-loop control programs (*e.g.*, adjusting motion or stopping based on sensor feedback) in 50.13% of episodes. (3) Increasing test-time computation through more interaction or greater reasoning effort improves task success without additional robot training or bespoke primitives, while performance eventually saturates at levels that vary across models.

2 Background and Related Work

We review visuomotor policies, robot-use agents, and CLI coding agents, and introduce the Robot Operating System; an extended version is provided in Appendix A.

Visuomotor Policies. A visuomotor policy maps raw sensory observations directly to low-level robot actions (Levine et al., 2016). Such policies are predominantly learned end to end from robot demonstrations, as in VLA models (Brohan et al., 2023; Kim et al., 2024; Team, 2025; Bjorck et al., 2025; Intelligence et al., 2026) and WAMs (Agarwal et al., 2026; Ye et al., 2026; Li et al., 2026). However, their generalization remains bounded by the training distribution (Zhou et al., 2025; Fei et al., 2025; Kim et al., 2026). OpenRUA instead casts an off-the-shelf coding agent as a zero-shot visuomotor policy, requiring no robot demonstration data or fine-tuning.

Robot-Use Agents. RUA (Isola, 2026) refers to LLM agents that operate a robot the way they operate a computer, through tool calls or executable programs, a paradigm that traces back to CaP (Liang et al., 2023; Singh et al., 2022). One line of work wraps learned visuomotor policies as callable tools, with the agent decomposing tasks and selecting tools while the underlying policies handle physical execution (ichter et al., 2023; Zhang et al., 2026b; Galanti et al., 2026). Another exposes low-level actions directly to the agent, which specifies end-effector poses step by step or writes control programs using predefined, hand-crafted primitives (Hu et al., 2026; Zhou et al., 2026; Fu et al., 2026; Elmaaroufi et al., 2026). OpenRUA instead provides the coding agent with only terminal access to the robot's native ROS 2 interfaces and a workspace, allowing it to build task-specific perception and control tools during execution rather than requiring them to be engineered in advance.

CLI Coding Agents. CLI coding agents such as Claude Code (Anthropic, 2025) and Codex (OpenAI, 2025) resolve real-world software engineering tasks through autonomous iteration (OpenAI, 2024; Merrill et al., 2026). These agents exhibit closed-loop agency over long horizons: they navigate codebases, edit and run code, issue CLI commands, interpret terminal output, and recover from errors. OpenRUA turns this competence toward robot control by treating the robot as a live software project, allowing the agent to apply these capabilities to physical tasks without adaptation.

Robot Operating System. ROS 2 connects robot components and exposes their sensing, state, and control capabilities via topics, services, and actions (Macenski et al., 2022). The native `ros2cli` command-line tools and `rclpy` Python client library allow developers to inspect these interfaces, acquire data, and send control requests. Prior work connects LLM agents to ROS through an additional abstraction layer of custom tools or predefined primitives, with evaluations limited to in-house tasks rather than standard robot control benchmarks (Royce et al., 2024; Mower et al., 2024).

3 OpenRUA: Workspace as Harness for Robot-Use Agents

Given a natural-language task, OpenRUA provides an off-the-shelf coding agent with only terminal access to the robot's native ROS 2 interface and a workspace on the robot's computer (Figure 1). The workspace provides ROS 2 documentation and basic starter tools for the agent to explore the robot, and stores the observations, programs, and notes the agent produces during execution. Within this

workspace, OpenRUA recasts perception as file I/O and manipulation as coding. We expect task-specific perception and control strategies to emerge as the agent explores this workspace, writes and tests programs, and uses execution feedback to refine its solutions, much as it would when working on an unfamiliar software project. Without a prescribed workflow, the agent itself decides when to observe, which commands or programs to write and execute, and how to verify task completion.

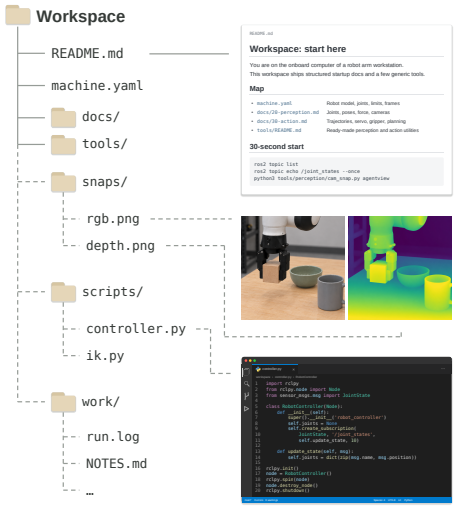
Workspace as Harness. OpenRUA only provides guidance for robot use through a simple workspace containing ROS 2 documentation and basic tools, in which the coding agent can approach the robot as an ordinary software project. As illustrated in Figure 2, the opening prompt specifies the task and asks the agent to survey the machine, complete the task, and verify the result. The documentation describes the robot's capabilities and how to access them, while the starter tools provide executable examples that the agent can inspect, use, or adapt. These materials provide a starting point rather than a prescribed workflow: the agent itself decides which resources to consult and how to organize its work using its built-in file and terminal capabilities (see Appendix C.2 for details).

The initial workspace provides `README.md`, which is the entry point, directing the agent to `machine.yaml` for the robot's configuration, including sensors, control interfaces, joint limits, and coordinate frames, and to the documentation for instructions on using these functionalities. A small set of starter tools offers ready-to-run implementations of basic operations. The tools are provided as source files; the agent can inspect and adapt them, use them as references, or write its own programs using the native ROS 2 interfaces.

During task execution, the workspace itself also serves as an external, persistent memory, preserving observations and intermediate results beyond the agent's context window. We expect the agent to autonomously organize images, programs, notes, and logs into folders such as `snaps/`, `scripts/`, and `work/`. By revisiting them, the agent can recover prior findings, reuse or revise earlier programs, and build on prior work as the task progresses.

```
You are working on a robot's onboard
computer.
Your task:  {task}
Survey the machine yourself to find out
what robot this is and what it can do;
work until the task is physically done,
verify it your own way, then finish.
The workspace contains starter docs and
tools you can use.
```

(a) The opening prompt.



(b) The workspace tree, with solid branches indicating starter documentation and tools and dashed branches indicating artifacts created by the agent during execution.

Figure 2: **A workspace for robot use.** (a) The opening prompt specifies the task and asks the agent to explore the machine, complete the task, and verify the result. (b) The workspace provides documentation and tools for getting started and stores the observations, programs, and records the agent produces during execution.

Perception as File I/O. OpenRUA reformulates robot perception as a standard file I/O task, allowing the agent to acquire data from the robot's native ROS 2 sensor topics and save it as workspace files. Depending on the robot's available sensors and interfaces, these files can contain (1) visual data (*e.g.*, RGB and depth images); (2) proprioceptive and kinematic states (*e.g.*, joint states and frame transforms); and (3) force and effort feedback (*e.g.*, force/torque readings and joint efforts); see Appendix C.3 for details. The agent can run the supplied starter tools, adapt their source code, or write its own `rclpy` programs to subscribe to the relevant topics and save the received data. The saved files are accessible through the agent's built-in file tools: images can be opened for visual inspection, while numerical arrays and structured text can be read by programs for computation. The agent is left to decide when to acquire new observations, which files to inspect, and what processing code to write based on the task and execution feedback. We examine the agent's perception strategies that emerge under this design in Section 4.3.

Table 1: **OpenRUA achieves 99.0% success on CaP-Bench and 87.0% on LIBERO-PRO in a zero-shot setting.** *Train.*, *Exp.*, and *Oracle* indicate training on robot demonstrations, experience accumulated before evaluation, and access to the benchmark's success predicate during execution, respectively. Reasoning effort is shown in parentheses; CC denotes Claude Code. Baseline sources and evaluation protocols are detailed in Appendices B and D.

(a) CaP-Bench

Method		Setting					Success Rate (%)							
		Agent	Model (Effort)	Train.	Exp.	Oracle	Lift	Stack	Wipe	Nut	Restack	2A-Lift	Handover	Avg.
Human Expert		–	–	–	–	–	93.0	73.0	100.0	80.0	47.0	94.0	91.0	82.6
Agent	CaP-X	CaP-Agent0	GPT-5.2 (none) Opus 4.5 (off) Gemini 3 Pro (high)	✗	✓	✗	97.0	98.0	100.0	0.0	89.0	74.0	20.0	68.3
	RHO	Codex	GPT-5.5 (xhigh)	✗	✓	✗	98.0	98.0	100.0	1.0	54.0	53.0	86.0	70.0
	ASPIRE	CC	Opus 4.6 (high)	✗	✓	✗	97.0	99.0	99.0	9.0	100.0	71.0	92.0	81.0
Agent	OpenRUA	CC	Opus 5	✗	✗	✗	100.0	100.0	100.0	100.0	100.0	99.0	94.0	99.0

(b) LIBERO-PRO

Method		Setting					Success Rate (%)									
		Agent	Model (Effort)	Train.	Exp.	Oracle	Object		Spatial		Goal		LIBERO-10		Average	
							P	T	P	T	P	T	P	T	6 Cells	8 Cells
Learned Policy	OpenVLA	–	–	✓	✗	✗	0.0	0.0	0.0	0.0	0.0	0.0	–	–	0.0	–
	π_0	–	–	✓	✗	✗	0.0	0.0	0.0	0.0	0.0	0.0	–	–	0.0	–
	$\pi_{0.5}$	–	–	✓	✗	✗	17.0	1.0	20.0	1.0	38.0	0.0	–	–	12.8	–
Agent + Policy	Pigey	Pigey-Agent	Opus 4.7 (off)	✓	✗	✗	54.0	54.0	66.0	80.0	44.0	22.0	–	–	53.3	–
	Harness VLA	Codex	GPT-5.5 (xhigh)	✓	✓	✓	91.0	94.0	69.0	81.0	66.0	75.0	49.0	52.0	79.3	72.1
		CC	Opus 4.7 (max)	✓	✓	✓	90.0	88.0	80.0	94.0	87.0	87.0	62.0	71.0	87.7	82.4
Agent	CaP-X	CaP-Agent0	GPT-5.2 (none) Opus 4.5 (off) Gemini 3 Pro (high)	✗	✗	✗	22.0	18.0	12.0	14.0	26.0	17.0	–	–	18.2	–
	RHO	Codex	GPT-5.5 (xhigh)	✗	✓	✗	12.2	37.2	53.0	61.6	50.6	55.6	–	–	45.0	–
	ASPIRE	CC	Opus 4.6 (high)	✗	✓	✗	98.0	95.0	51.0	60.0	81.0	45.0	–	–	71.7	–
	RATS	RATS-Agent	Gemini 3.1 Pro (high)	✗	✓	✓	61.0	63.0	29.0	31.0	43.0	36.0	–	–	43.8	–
Agent	OpenRUA	CC	Opus 5	✗	✗	✗	98.0	97.0	98.0	100.0	79.0	79.0	80.0	65.0	91.8	87.0

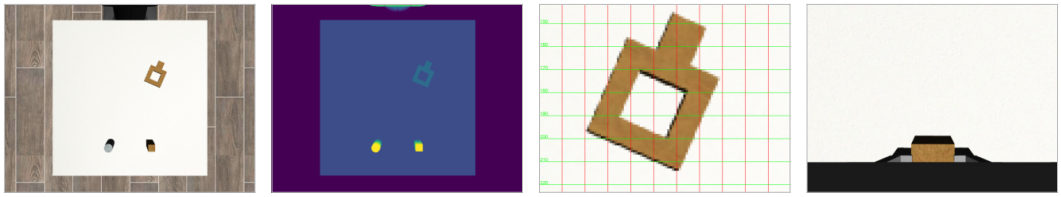
Manipulation as CLI Coding. OpenRUA reformulates robot manipulation as a CLI coding task, allowing the agent to write and execute `ros2cli` commands or `rclpy` programs in the terminal. These commands and programs send trajectory, velocity, or gripper control requests through native ROS 2 interfaces to the robot's controllers, which execute the requests to produce physical motion. The agent is free to issue commands step by step and inspect feedback between actions, or write programs that run closed-loop control using sensor feedback. The agent can inspect terminal output and acquire new observations to assess execution outcomes and revise its commands or programs. We examine the agent's control strategies that emerge under this design in Section 4.4.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate OpenRUA on three simulation benchmarks: (1) CaP-Bench (Fu et al., 2026) covers seven single-arm and bimanual manipulation tasks; we follow its protocol of 100 trials per task. (2) LIBERO-PRO (Zhou et al., 2025) evaluates robustness to position and task perturbations across four LIBERO suites; we run ten trials per task, following prior work (Zhang et al., 2026b;a; Galanti et al., 2026). (3) RoboCasa365 (Nasiriany et al., 2026) evaluates mobile manipulation on atomic and composite kitchen tasks; we use its 50 target tasks with ten trials per task, following prior work (Zhang et al., 2026b). We report success rates under each benchmark's official success predicate; task coverage, sampling, and scoring details are provided in Appendix B.

Agents and Models. We evaluate OpenRUA with Claude Code powered by Claude Opus 5, Sonnet 5, and Fable 5.1, all at high reasoning effort, and with Codex powered by GPT-6 Astra at medium and high reasoning effort. Each episode starts with a fresh workspace, without robot demonstrations,



(a) Overhead camera image. (b) Depth array segmented by height. (c) Enlarged crop with a pixel grid. (d) Wrist camera image after grasping.

Figure 3: **An example of agent-built perception in the CaP-Bench Nut task.** The agent combines visual understanding with numerical processing: (a) it reads the overhead camera image to identify the nut and two pegs; (b) it segments the depth array by height and uses camera calibration to compute their positions in world coordinates; (c) it enlarges a crop of the nut and adds a pixel grid to inspect the handle's orientation and width; and (d) it checks the grasp using the wrist camera image together with the measured finger gap.

task-specific rehearsal, or memory from earlier episodes. The agent has no access to privileged simulator state or the benchmark's success predicate (see Appendix B.2 for the evaluation protocol).

4.2 Main Results

Table 1 compares OpenRUA on CaP-Bench and LIBERO-PRO against four types of systems: human-written programs, learned visuomotor policies, agents that orchestrate learned policies, and agents that act as the policy through predefined primitives. We report the success rates of OpenRUA alongside the numbers that prior systems report in their papers. For each system, we also mark whether its policy is trained on demonstrations, whether it acts zero-shot on the evaluation tasks or draws on experience accumulated beforehand, and whether it can query the benchmark's success predicate during execution. More evaluation settings are detailed in Appendix D.

Robot-use agents are zero-shot visuomotor policies. With Claude Code powered by Claude Opus 5 (high), OpenRUA achieves success rates of 99.0% on CaP-Bench and 87.0% on LIBERO-PRO. It exceeds ASPIRE's 81.0% and Harness VLA's 82.4%, without robot-policy training, prior task experience, or access to the benchmark's success predicate during execution. These results show that an off-the-shelf coding agent can serve as a zero-shot visuomotor policy through the robot's native interface, without a learned robot policy or task-specific primitive library. On Nut, OpenRUA achieves 100% success, compared with 0–9% for CaP-X, RHO, and ASPIRE despite their synthesized skills or prior optimization. Accumulating skills over predefined primitives does not supply sensing capabilities absent from their interface, whereas native ROS 2 access allows the agent to incorporate force feedback directly into its insertion program. Detailed comparisons and analysis are provided in Appendices D.3 and D.4. Additional experiments on RoboCasa365 and analysis of generalization are provided in Appendix D.5.

Table 2: **The agent writes its own programs to process sensor images and compute metric measurements.** Percentage of episodes in which Claude Code with Opus 5 processes sensor images with self-written code or runs a self-written metric program. All successful and failed episodes are included. More analysis is provided in Appendix F.

Benchmark	Episodes	Image processing (%)	Metric programs (%)
CaP-Bench	700	47.43%	97.43%
LIBERO-PRO	800	94.75%	96.25%
Overall	1,500	72.67%	96.80%

4.3 How the Agent Perceives

We analyze the agent's programs and execution traces across 1,500 episodes on CaP-Bench and LIBERO-PRO to examine how it develops perception strategies during task execution. Table 2 reports how often the agent processes sensor images with self-written code or runs self-written metric programs; Figures 3 and 4 show a concrete example from the CaP-Bench Nut task. Additional analysis of image reads, program invocations, and computed motion targets is provided in Appendix F.

The agent builds its own image-processing tools for better visual understanding. The built-in visual capabilities of frontier models enable the agent to identify task-relevant objects, assess spatial

relationships, and check action outcomes. Yet interestingly, our analysis reveals that the agent develops its own image-processing tools to better understand the scene, spontaneously writing programs that transform raw sensory inputs into views tailored to the task. We observe this emergent behavior in 72.67% of episodes across CaP-Bench and LIBERO-PRO (Table 2), indicating that programmatic image processing is a common part of the agent's perception. For example, in Figure 3, the agent (a) identifies the nut and pegs in the overhead image, then (c) writes a program to crop the nut region from the image, enlarge it, and overlay a pixel grid, making the handle's orientation and width easier to inspect. Appendix C.4 offers further examples and analysis.

The agent builds its own tools for metric measurement to guide physical actions. Beyond visual understanding, manipulation requires numerical measurements such as target positions, object dimensions, and contact forces. Crucially, the agent constructs its own measurement programs during task execution, combining depth arrays with camera intrinsics and frame transforms to derive task-specific measurements. We observe this emergent behavior in 96.80% of episodes across CaP-Bench and LIBERO-PRO (Table 2), showing that self-written measurement programs are a common part of its approach to manipulation. For example, in Figure 3(b), the agent writes a Python program to segment the nut and pegs by height and compute their positions in world coordinates from the depth array, as shown in Figure 4. The resulting coordinates provide motion targets, while measurements such as the gripper's finger gap help the agent check execution outcomes. Appendix C.5 provides further details and examples.

Python

workspace › Python

```python
d = np.load("bird_depth.npy")
f = 579.4112549695428
zw = 2.0 - d
mask = ((zw > 0.825) & (zw < 1.3)).astype(
    np.uint8)
n, lab, stats, cent = (
    cv2.connectedComponentsWithStats(mask, 8))
for i in range(1, n):
    ...
    ys, xs = np.where(lab == i)
    wx = (ys - 240) * d[ys, xs] / f
    wy = (xs - 320) * d[ys, xs] / f
    ...
```

main* 0 errors 0 warnings Spaces: 4 UTF-8 Python

Figure 4: **An agent-written program for locating the nut and pegs from depth**. See Appendix H for further code details.

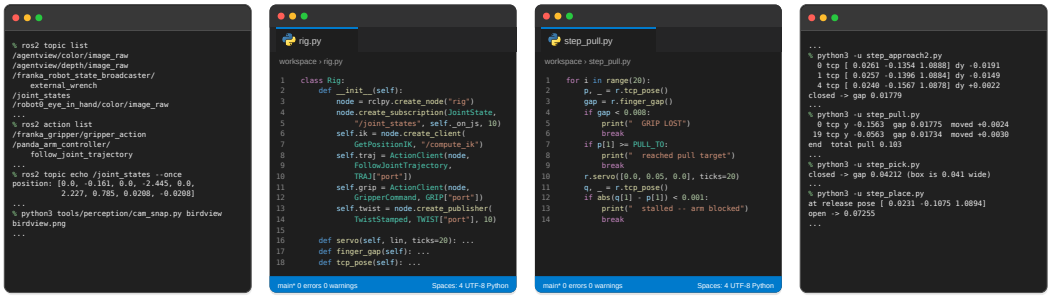
The agent autonomously pulls observations on demand. Rather than passively receiving sensory streams that continuously fill its context window, the agent actively decides which observations to fetch and when, based on the task and execution feedback. For example, in Figure 3, the agent selectively captures (a) an overhead image to locate the nut and pegs and (d) an eye-in-hand image to verify the grasp, rather than reading every camera frame throughout execution. In this way, the agent controls what sensory information enters its context, using selected observations to verify progress and guide subsequent actions. Further details are provided in Appendix C.3.

4.4 How the Agent Manipulates

We analyze the agent's programs and execution traces across the same 1,500 CaP-Bench and LIBERO-PRO episodes to examine how it constructs control tools and responds to execution feedback. Table 3 reports how often the agent writes motion-control clients and executes self-written feedback-control programs; Figures 5 and 17 show examples of tool construction and recovery from a failed action. Additional analysis and examples are provided in Appendix F.

The agent builds its own motion-control tools from native interfaces. The supplied starter tools provide ready-to-run implementations of basic robot motions. Interestingly, the agent develops its own control clients during task execution, writing `rclpy` programs that access trajectory, gripper, velocity, and inverse-kinematics interfaces. We find agent-written motion clients in 95.87% of episodes across CaP-Bench and LIBERO-PRO (Table 3), indicating that constructing control tools is a common part of the agent's work. For example, in Figure 5, the agent (a) inspects the robot's native interfaces and (b) writes a client library that it imports into subsequent action programs. This library lets it compose robot operations in reusable code as it works through the task. Appendix H provides the corresponding client and action programs.

The agent writes its own feedback-control programs to adjust physical actions. Beyond issuing individual motion commands, the agent constructs programs that repeatedly read sensor data and use those readings to adjust motion or determine when to stop. Crucially, the feedback conditions



(a) Inspect interfaces. (b) Write a client library. (c) Pull with feedback. (d) Inspect results.

Figure 5: **The agent constructs motion-control clients and feedback logic during a LIBERO-PRO drawer task.** (a) It inspects the robot's interfaces and acquires observations. (b) It writes a reusable client library for inverse kinematics, trajectory and gripper actions, and velocity commands. (c) It writes a pulling program that checks finger gap and displacement, stopping if the grip is lost or motion stalls. (d) It executes its programs and reads the resulting measurements to assess progress. Some lines are omitted to highlight the key operations. See Appendix H for the programs and Appendix C.6 for execution details.

are encoded in the agent's own control logic and evaluated by the running program. We observe executed programs with such feedback logic in 50.13% of episodes across CaP-Bench and LIBERO-PRO (Table 3). For example, in Figure 5(c), the agent writes a program that pulls the drawer in velocity bursts and checks the finger gap and displacement after each burst, stopping if the gap falls below 8 mm or the drawer ceases to move. In another episode, an agent-written descent program stops when the change in measured force exceeds its threshold (Figure 19(a)). These examples show task-specific feedback logic emerging in programs written during execution. Appendix C.6 provides the control traces, feedback conditions, and further examples.

The agent revises its actions when execution feedback reveals a failure. Alongside feedback within its programs, the agent inspects terminal output and new observations between actions to decide whether to continue or revise its approach. Importantly, its checks extend beyond whether a command completed to whether the intended physical change occurred. For example, in Figure 17, an initial grasp misses the ketchup bottle and knocks it over (b). The agent detects the failure from the finger gap and camera observations, reconstructs the fallen bottle's geometry from depth, and plans a new grasp based on its orientation (c). It then grasps the bottle and places it upright in the basket (d). The recovery is worked out during execution from the changed scene, rather than supplied as a task-specific recovery routine. Further cases and an analysis of unsuccessful episodes are provided in Appendices G and I.

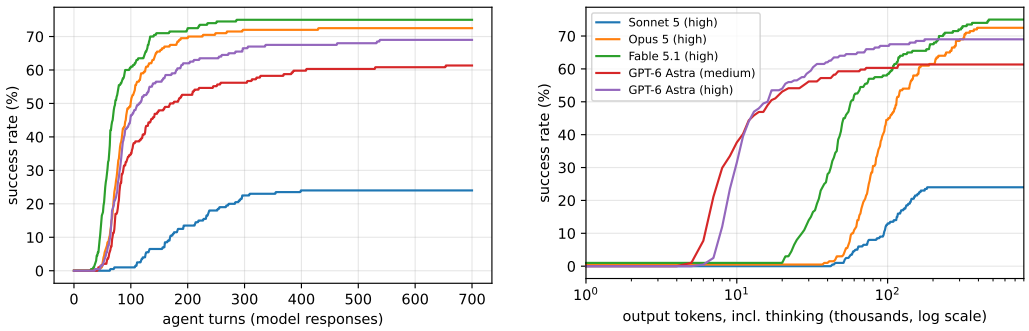
Table 3: **The agent writes its own motion-control clients and feedback-control programs.** Percentage of episodes in which Claude Code with Opus 5 writes a motion-control client or executes a self-written program that uses sensor feedback to adjust motion or determine when to stop. All successful and failed episodes are included. More analysis is provided in Appendix F.

Benchmark	Episodes	Motion-control clients (%)	Feedback-control programs (%)
CaP-Bench	700	97.00%	60.14%
LIBERO-PRO	800	94.88%	41.38%
Overall	1,500	95.87%	50.13%

4.5 Scaling with Coding Agents and Models

Figure 6 shows how success rates change with increasing test-time computation (*e.g.*, agent turns, output tokens, reasoning effort) across coding agents and models on the LIBERO-10 suite of LIBERO-PRO; detailed interaction and cost profiles are provided in Appendix E.

Test-time computation improves robot control through a zero-abstraction interface. Increasing the agent-turn budget from 100 to 200 raises the success rate of Claude Code with Opus 5 from 50% to 70% (Figure 6a). Increasing reasoning effort also improves performance: Codex with GPT-6 Astra reaches 69.0% at high effort, compared with 62.5% at medium effort. Both agents use the same workspace and native interface, so these gains require no additional tools or primitives to be engineered in advance. The gains nevertheless diminish as the curves flatten, while differences



(a) agent-turn budget (b) output-token budget (log scale)

Figure 6: Success rate on LIBERO-10 (both perturbation cells, 200 trials per model) as a function of the budget spent. A trial counts as solved within a budget if the benchmark predicate first held before it; one turn is one model response, and tokens are those the model wrote, including thinking. The right end of each curve is the success rate reported in Table 1; the runs behind the curves, and how the token position of a Claude Code episode is estimated, are described in Appendix E.

between models persist: Sonnet 5 reaches 24.0% despite continuing to gain successes up to about 400 turns, compared with 72.5% for Opus 5 and 75.0% for Fable 5.1. Thus, more interaction and greater reasoning effort improve performance under our native, zero-abstraction interface, while the achievable performance remains dependent on the underlying model. Detailed comparisons of interaction patterns, costs, and per-task results are provided in Appendix E.

5 Limitations

Although ROS 2 is a de facto standard for simulation and physical robots (Macenski et al., 2022), a shared interface does not establish real-world applicability. Our simulation results require validation on physical robots, particularly in three respects.

Timing. Following common simulation evaluation protocols (Fu et al., 2026; Galanti et al., 2026; Kim et al., 2024), physical time advances during action execution but pauses during agent deliberation (Appendix B.2). This excludes the effects of inference latency on environments that continue evolving during deliberation. Agent-written feedback controllers react within running programs but do not eliminate delays between agent decisions. Real-time responsiveness remains to be evaluated, especially for dynamic tasks.

Sensing. Raw sensor access without privileged object states does not eliminate the gap between simulated and real-world sensing. Sensor noise, missing depth values, and calibration errors may affect both visual interpretation and programmatic measurement. Few perception failures therefore do not establish real-sensor robustness; physical experiments must test whether agent-written programs can detect and accommodate these errors.

Safety. Task success does not establish safe execution: our failure analysis includes motions that would collide with the table. Although agents sometimes write their own force thresholds and stopping conditions, these protections are not guaranteed to be present or correct. Physical deployment requires independent motion limits, collision checks, and emergency stops; validating these controls while retaining the agent's flexibility remains future work.

6 Conclusion

We presented OpenRUA, a minimalist *workspace-as-harness* design providing off-the-shelf coding agents with native ROS 2 terminal access, machine documentation, and basic tools. It recasts perception as file I/O and manipulation as coding, without prescribing an agentic workflow. Experiments show that coding agents can serve as zero-shot visuomotor policies without learned robot policies or task-specific primitive libraries. Crucially, task-specific image-processing tools, metric measurement programs, motion-control clients, and feedback controllers emerge as the agent writes code during execution. These findings suggest that the robot's existing software stack can support agents in constructing their own tools without an expanding set of human-engineered abstractions.

AI USE STATEMENT

We used Claude and Codex in an iterative, author-guided process to refine the research hypotheses, conceptual framework, and experimental design, and to implement the method. These tools also assisted with qualitative analysis of agent behavior and failures, interpretation of results, literature search and organization, manuscript polishing, and figure preparation. References were selected and inserted by the authors. We checked AI-assisted code and analyses by inspecting generated files and numerical outputs and by running and debugging the code. The coding agents evaluated in this study and their configurations are described in the experimental setup and appendix. The authors take responsibility for the final content of the paper and accompanying artifacts.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide the implementation at https://anonymous.4open.science/r/OpenRUA-34210. Appendix B details the benchmark tasks, trial sampling, agent configuration, execution budgets, and scoring procedures. Appendix C.2 and Appendix C.2.1 describe the workspace documentation and starter tools, while Appendix E documents the configurations and analysis procedures for the scaling experiments.

ETHICS STATEMENT

All experiments in this study were conducted in simulation. Deploying agents that generate and execute robot-control programs on physical hardware poses risks of injury and property damage. Our results do not establish safe physical deployment; such deployment would require independent safety controls and validation, as discussed in Section 5.

REFERENCES

Niket Agarwal, Arslan Ali, Jon Allen, Martin Antolini, Adeline Aubame, Alisson Azzolini, Junjie Bai, Maciej Bala, Yogesh Balaji, Josh Bapst, et al. Cosmos 3: Omnimodal world models for physical ai. *arXiv preprint arXiv:2606.02800*, 2026.

Anthropic. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025. Product announcement, February 24, 2025. Accessed September 25, 2026.

Shmuel Berman, Michael Ilie, Jia Deng, and Daniel Freeman. Claude plays robotics. Anthropic Research, July 2026. URL https://www.anthropic.com/research/claude-plays-robotics. Research report, July 9, 2026. Accessed September 25, 2026.

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.

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.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

Siyi Chen, Hugo Hadfield, Alex Zook, Mikaela Angelina Uy, Chan Hee Song, Erwin Coumans, Xuning Yang, Faisal Ladhak, Qing Qu, Stan Birchfield, et al. Volo: A physical orchestrator for open-vocabulary long-horizon manipulation. *arXiv preprint arXiv:2606.07723*, 2026a.

Yanzhe Chen, Zechen Bai, Zhijun Cao, Wenzheng Zeng, Kevin Qinghong Lin, Yiqi Lin, Guoqiang Liang, Kevin Yuchen Ma, Qiming Huang, and Mike Zheng Shou. Show-harness: Just a vlm agent can play robots. *arXiv preprint arXiv:2609.10522*, 2026b.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. TerminalWorld: Benchmarking agents on real-world terminal tasks. *arXiv preprint arXiv:2605.22535*, 2026. URL https://arxiv.org/abs/2605.22535.

Xin Ding, Liang Mi, Mingzhe Huang, Zixuan Wang, Chao Zhang, Zixu Hao, Fu Chen, Xiangyu Li, Yikai Zheng, Yaoyu Guo, Weijun Wang, Kun Li, Hao Wu, Yunxin Liu, and Ting Cao. Zetta ζ: An efficient closed-loop embodied harness for self-evolving physical intelligence. *CoRR*, abs/2608.16590, 2026. doi: 10.48550/ARXIV.2608.16590. URL https://doi.org/10.48550/arXiv.2608.16590.

Karim Elmaaroufi, Justin Svegliato, Sarunas Kalade, Graham Schelle, Sanjit A. Seshia, and Matei Zaharia. RHO: your coding agent is secretly a roboticist. *CoRR*, abs/2606.16458, 2026. doi: 10.48550/ARXIV.2606.16458. URL https://doi.org/10.48550/arXiv.2606.16458.

Zhirui Fang, Qingchi Yu, Ziyang Chen, Longfei Li, Haoran Ma, Keru Zhou, Xinrun Xu, Samith Va, Yuxuan Hu, Peixuan Song, et al. Emerge-policy: A robot mind emerges beyond a single policy. *arXiv preprint arXiv:2608.29896*, 2026.

Senyu Fei, Siyin Wang, Junhao Shi, Zihao Dai, Jikun Cai, Pengfang Qian, Li Ji, Xinzhe He, Shiduo Zhang, Zhaoye Fei, et al. Libero-plus: In-depth robustness analysis of vision-language-action models. *arXiv preprint arXiv:2510.13626*, 2025.

Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Max Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Feifei Li, Guanya Shi, Jiajun Wu, Shankar S. Sastry, Yuke Zhu, Ken Goldberg, and Linxi "Jim" Fan. Cap-x: A framework for benchmarking and improving coding agents for robot manipulation. *CoRR*, abs/2603.22435, 2026. doi: 10.48550/ARXIV.2603.22435. URL https://doi.org/10.48550/arXiv.2603.22435.

Liane Galanti, Dhruv Shah, and Tri Dao. Addressing the orchestration gap in generalist robots via physical agency. *CoRR*, abs/2607.21725, 2026. doi: 10.48550/ARXIV.2607.21725. URL https://doi.org/10.48550/arXiv.2607.21725.

Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: a perspective from the service robotics domain. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (eds.), *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pp. 593–604. ACM, 2020. doi: 10.1145/3368089.3409743. URL https://doi.org/10.1145/3368089.3409743.

Hengyuan Hu, Priya Sundaresan, Jensen Gao, and Dorsa Sadigh. VIA: visual interface agent for robot control. *CoRR*, abs/2607.11119, 2026. doi: 10.48550/ARXIV.2607.11119. URL https://doi.org/10.48550/arXiv.2607.11119.

Jinbang Huang, Yuanzhao Hu, Zhiyuan Li, Ran Qi, Yixin Xiao, Zhanguang Zhang, Mark Coates, Tongtong Cao, and Yingxue Zhang. Roboharness: Memory-driven orchestration of heterogeneous robot policies for long-horizon planning. *CoRR*, abs/2607.18060, 2026. doi: 10.48550/ARXIV.2607.18060. URL https://doi.org/10.48550/arXiv.2607.18060.

brian ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In Karen Liu, Dana Kulic, and Jeff Ichnowski (eds.), *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 14–18 Dec 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Physical Intelligence, Bo Ai, Ali Amin, Raichelle Aniceto, Ashwin Balakrishna, Greg Balke, Kevin Black, George Bokinsky, Shihao Cao, Thomas Charbonnier, Vedant Choudhary, Foster Collins, Ken Conley, Grace Connors, James Darpinian, Karan Dhabalia, Maitrayee Dhaka, Jared DiCarlo, Danny Driess, Michael Equi, Adnan Esmail, Yunhao Fang, Chelsea Finn, Catherine Glossop, Thomas Godden, Ivan Goryachev, Lachlan Groom, Haroun Habeeb, Hunter Hancock, Karol Hausman, Gashon Hussein, Victor Hwang, Brian Ichter, Connor Jacobsen, Szymon Jakubczak, Rowan Jen, Tim Jones, Gregg Kammerer, Ben Katz, Liyiming Ke, Mairbek Khadikov, Chandra Kuchi, Marinda Lamb, Devin LeBlanc, Brendon LeCount, Sergey Levine, Xinyu Li, Adrian Li-Bell, Vladislav Lialin, Zhonglin Liang, Wallace Lim, Yao Lu, Enyu Luo, Vishnu Mano, Nandan Marwaha, Aikys Mongush, Liam Murphy, Suraj Nair, Tyler Patterson, Karl Pertsch, Allen Z. Ren, Gavin Schelske, Charvi Sharma, Baifeng Shi, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, Will Stoeckle, Jiaming Tang, Jimmy Tanner, Shalom Tekeste, Marcel Torne, Kyle Vedder, Quan Vuong, Anna Walling, Haohuan Wang, Jason Wang, XuDong Wang, Chris Whalen, Samuel Whitmore, Blake Williams, Charles Xu, Sukwon Yoo, Lili Yu, Wuming Zhang, Zhuoyang Zhang, and Ury Zhilinsky. $\pi_{0.7}$: a steerable generalist robotic foundation model with emergent capabilities, 2026. URL https://arxiv.org/abs/2604.15483.

Phillip Isola. Robot-use agents. https://web.mit.edu/phillipi/www/writing/robot-use-agents.html, September 2026. Blog post, September 7, 2026. Accessed September 25, 2026.

Mengzhao Jia, Yang Lin, Xixin Zhang, Zhihan Zhang, Xiaobai Liu, and Meng Jiang. Agent as policy for robotic manipulation, 2026. URL https://arxiv.org/abs/2609.12541.

Chanyoung Kim, Minwoo Kim, Minseok Kang, Hyunwoo Kim, and Dahuin Jung. Libero-para: A diagnostic benchmark and metrics for paraphrase robustness in vla models. *arXiv preprint arXiv:2603.28301*, 2026.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Paul Foster, Pannag R. Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. Openvla: An open-source vision-language-action model. In Pulkit Agrawal, Oliver Kroemer, and Wolfram Burgard (eds.), *Conference on Robot Learning, 6-9 November 2024, Munich, Germany*, volume 270 of *Proceedings of Machine Learning Research*, pp. 2679–2713. PMLR, 2024. URL https://proceedings.mlr.press/v270/kim25c.html.

Vaishak Kumar. Act-observe-rewrite: Multimodal coding agents as in-context policy learners for robot manipulation. *CoRR*, abs/2603.04466, 2026. doi: 10.48550/ARXIV.2603.04466. URL https://doi.org/10.48550/arXiv.2603.04466.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

Lin Li, Qihang Zhang, Yiming Luo, Shuai Yang, Ruilin Wang, Fei Han, Mingrui Yu, Zelin Gao, Nan Xue, Xing Zhu, et al. Causal world modeling for robot control. *arXiv preprint arXiv:2601.21998*, 2026.

Zexiang Li, Shankar Sastry, and R Murray. *A mathematical introduction to robotic manipulation*. CRC press Boca Raton, FL, 1994.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, pp. 9493–9500. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://doi.org/10.1109/ICRA48891.2023.10160591.

Haowen Liu, Xirui Li, Shaoxiong Yao, Peng Shi, Tianyi Zhou, Jia-Bin Huang, Furong Huang, and Jiayuan Mao. Guava: An effective and universal harness for embodied manipulation. *CoRR*, abs/2606.18363, 2026. doi: 10.48550/ARXIV.2606.18363. URL https://doi.org/10.48550/arXiv.2606.18363.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi "Jim" Fan, and Guanzhi Wang. ASPIRE: agentic /skills discovery for robotics. *CoRR*, abs/2607.00272, 2026. doi: 10.48550/ARXIV.2607.00272. URL https://doi.org/10.48550/arXiv.2607.00272.

Steven Macenski, Tully Foote, Brian P. Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Sci. Robotics*, 7(66), 2022. doi: 10.1126/SCIROBOTICS.ABM6074. URL https://doi.org/10.1126/scirobotics.abm6074.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Christopher E. Mower, Yuhui Wan, Hongzhan Yu, Antoine Grosnit, Jonas Gonzalez-Billandon, Matthieu Zimmer, Jinlong Wang, Xinyu Zhang, Yao Zhao, Anbang Zhai, Puze Liu, Davide Tateo, Cesar Cadena, Marco Hutter, Jan Peters, Guangjian Tian, Yuzheng Zhuang, Kun Shao, Xingyue Quan, Jianye Hao, Jun Wang, and Haitham Bou-Ammar. ROS-LLM: A ROS framework for embodied AI with task feedback and structured reasoning. *CoRR*, abs/2406.19741, 2024. doi: 10.48550/ARXIV.2406.19741. URL https://doi.org/10.48550/arXiv.2406.19741.

Soroush Nasiriany, Sep Nasiriany, Abhiram Maddukuri, and Yuke Zhu. Robocasa365: A large-scale simulation framework for training and benchmarking generalist robots. In *International Conference on Learning Representations*, volume 2026, pp. 98643–98667, 2026.

OpenAI. Introducing SWE-bench Verified. OpenAI, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Blog post, August 13, 2024. Accessed September 25, 2026.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, 2025. Product announcement, May 16, 2025. Accessed September 25, 2026.

Robocurve. GPT-6 Astra on robotic manipulation. https://openai.robocurve.org/gpt-6-astra/, 2026. Evaluation report, September 4, 2026. Accessed September 25, 2026.

Rob Royce, Marcel Kaufmann, Jonathan Becktor, Sangwoo Moon, Kalind C. Carpenter, Kai Pak, Amanda Towler, Rohan Thakker, and Shehryar Khattak. Enabling novel mission operations and interactions with ROSA: the robot operating system agent. *CoRR*, abs/2410.06472, 2024. doi: 10.48550/ARXIV.2410.06472. URL https://doi.org/10.48550/arXiv.2410.06472.

M Yunus Seker, Shobhit Aggarwal, Ruwan Wickramarachchi, Jonathan Francis, and Oliver Kroemer. Gta-2: A multi-vlm framework for synthesizing robot manipulation skills via grounded task axes. *arXiv preprint arXiv:2609.09808*, 2026.

Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. Advanced Textbooks in Control and Signal Processing. Springer, London, 2009. ISBN 978-1-84628-641-4. doi: 10.1007/978-1-84628-642-1.

Harsh Singh, Rocktim Jyoti Das, Mingfei Han, Preslav Nakov, and Ivan Laptev. MALMM: multi-agent large language models for zero-shot robotic manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2025, Hangzhou, China, October 19-25, 2025*, pp. 20386–20393. IEEE, 2025. doi: 10.1109/IROS60139.2025.11247340. URL https://doi.org/10.1109/IROS60139.2025.11247340.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Jiayi Su, Yixin Zheng, Mi Yan, Li Yi, Zhizheng Zhang, and He Wang. GPT-6 Astra as an embodied policy. https://github.com/anonymous-report-421/GPT-as-Policy, September 2026. Technical report, September 2026. Accessed September 25, 2026.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, et al. Agents' last exam. *arXiv preprint arXiv:2606.05405*, 2026. URL https://arxiv.org/abs/2606.05405.

Gemini Robotics Team. Gemini robotics 1.5: Pushing the frontier of generalist robots with advanced embodied reasoning, thinking, and motion transfer. *CoRR*, abs/2510.03342, 2025. doi: 10.48550/ARXIV.2510.03342. URL https://doi.org/10.48550/arXiv.2510.03342.

Terminal-Bench-Science Team. Terminal-Bench-Science: Evaluating AI agents on research workflows across scientific domains. https://github.com/harbor-framework/terminal-bench-science, 2026. Open-source repository. Accessed September 25, 2026.

Brian Y. Tsui, Alan Y. Fang, and Tiffany J. Hwu. Demonstration-free robotic control via LLM agents. *CoRR*, abs/2601.20334, 2026. doi: 10.48550/ARXIV.2601.20334. URL https://doi.org/10.48550/arXiv.2601.20334.

Peidong Wang, Zhiming Ma, Ying Chang, Xufang Luo, Yiqun Zhang, Zihan Wang, Xiaocui Yang, Shi Feng, Yuqing Yang, and Dongsheng Li. Self-evolving embodied agents via skill-harness evolution. *CoRR*, abs/2608.11350, 2026a. doi: 10.48550/ARXIV.2608.11350. URL https://doi.org/10.48550/arXiv.2608.11350.

Yuru Wang, Lejun Cheng, Yuxin Zuo, Sihang Zeng, Bingxiang He, Che Jiang, Junlin Yang, Yuchong Wang, Kaikai Zhao, Weifeng Huang, et al. Naturebench: Can coding agents match the published sota of nature-family papers? *arXiv preprint arXiv:2606.24530*, 2026b.

Shuangyu Xie, Kaiyuan Chen, and Ken Goldberg. Revisiting the "push-t" robot manipulation task with agentic robotics. *CoRR*, abs/2608.18227, 2026. doi: 10.48550/ARXIV.2608.18227. URL https://doi.org/10.48550/arXiv.2608.18227.

Yuhao Yang, Tianyu Fan, and Chao Huang. CLI-Anything: Towards agent-native computer use. *arXiv preprint arXiv:2606.03854*, 2026. URL https://arxiv.org/abs/2606.03854.

Seonghyeon Ye, Yunhao Ge, Kaiyuan Zheng, Shenyuan Gao, Sihyun Yu, George Kurian, Suneel Indupuru, You Liang Tan, Chuning Zhu, Jiannan Xiang, et al. World action models are zero-shot policies. *arXiv preprint arXiv:2602.15922*, 2026.

Junyi Zhang, Jiaxin Ge, Hanjun Yoo, Letian Fu, Zihan Yang, Yaowei Liu, Raj Saravanan, Shaofeng Yin, Justin Yu, Dantong Niu, Zirui Wang, Roei Herzig, Ken Goldberg, Yutong Bai, David M. Chan, Ion Stoica, Angjoo Kanazawa, Jiahui Lei, Haiwen Feng, and Trevor Darrell. Playful agentic robot learning. *CoRR*, abs/2606.19419, 2026a. doi: 10.48550/ARXIV.2606.19419. URL https://doi.org/10.48550/arXiv.2606.19419.

Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, Chunyang Zhu, Jiaxing Qiu, Yuchen Yan, Jiyuan Liu, Wenhao Tang, Zhengru Fang, Yi Nie, Changxu Wei, Yu Wang, Wenbo Ding, and Chao Yu. Harness VLA: steering frozen vlas into reliable manipulation primitives via memory-guided agents. *CoRR*, abs/2607.08448, 2026b. doi: 10.48550/ARXIV.2607.08448. URL https://doi.org/10.48550/arXiv.2607.08448.

Yunhao Zhao, Zhenyang Ni, Haoyang Chen, Ruohan Zhang, and Qi Zhu. Skills in weights, memory in code: Hybrid learning for memory-dependent robot manipulation. *CoRR*, abs/2608.09410, 2026. doi: 10.48550/ARXIV.2608.09410. URL https://doi.org/10.48550/arXiv.2608.09410.

Jian Zhou, Xunyi Zhao, Gengze Zhou, Zerui Li, Sihao Lin, Jiajun Liu, and Qi Wu. Embodied agents take control: Minimal-interface zero-shot agents rival industrial-scale policies in vision-and-language navigation. *CoRR*, abs/2607.26148, 2026. doi: 10.48550/ARXIV.2607.26148. URL https://doi.org/10.48550/arXiv.2607.26148.

Xueyang Zhou, Yangming Xu, Guiyao Tie, Yongchao Chen, Guowen Zhang, Duanfeng Chu, Pan Zhou, and Lichao Sun. LIBERO-PRO: towards robust and fair evaluation of vision-language-action models beyond memorization. *CoRR*, abs/2510.03827, 2025. doi: 10.48550/ARXIV.2510.03827. URL https://doi.org/10.48550/arXiv.2510.03827.

A Extended Related Work

Visuomotor Policies. A visuomotor policy maps raw sensory observations, such as camera images and proprioceptive states, directly to low-level robot actions (Levine et al., 2016). Today such policies are predominantly learned end to end: VLA models attach an action head to a pretrained vision-language backbone and fine-tune it on robot demonstrations (Brohan et al., 2023; Kim et al., 2024; Team, 2025; Bjorck et al., 2025; Intelligence et al., 2026), and WAM variants extend this line by jointly predicting future observations and actions (Agarwal et al., 2026; Ye et al., 2026; Li et al., 2026). However, they require large-scale demonstration data, and their generalization remains bounded by the training distribution: performance drops sharply under perturbations of objects, lighting, and poses (Zhou et al., 2025; Fei et al., 2025; Kim et al., 2026). OpenRUA instead casts an off-the-shelf coding agent as a zero-shot visuomotor policy, which requires no demonstration data or fine-tuning and scales directly with frontier coding agents.

Robot-Use Agents. RUA (Isola, 2026) refers to LLM agents that operate a robot the way they operate a computer, a paradigm that traces back to CaP, in which an LLM writes executable programs as robot policies (Liang et al., 2023; Singh et al., 2022). By composing predefined perception and control primitives, CaP brings the LLM's world knowledge about geometry, physics, and programming to robot control without any robot demonstrations (Kumar, 2026). Driven by frontier coding agents, today's RUA systems have extended CaP from one-shot code generation to an agentic loop: the agent acts through tool calls or generated programs, observes the outcome, and revises its next action (Team, 2025; Fu et al., 2026; Robocurve, 2026; Berman et al., 2026).

These systems broadly fall into two categories based on the interface through which the agent acts. One line of work wraps learned visuomotor policies into callable tools, yielding a hierarchical architecture in which the high-level LLM agent decomposes tasks and selects tools while the low-level policy handles physical execution (ichter et al., 2023; Zhang et al., 2026b; Galanti et al., 2026; Fang et al., 2026; Chen et al., 2026a). However, the tool interface hides low-level execution from the agent, which therefore cannot diagnose or recover when the underlying policy fails in out-of-distribution scenes (Chen et al., 2026b; Zhao et al., 2026; Ding et al., 2026; Huang et al., 2026). Another line of work exposes low-level actions directly to the agent, which itself serves as the robot policy (Jia et al., 2026; Berman et al., 2026; Robocurve, 2026; Su et al., 2026), either by issuing end-effector poses step by step (Hu et al., 2026; Chen et al., 2026b; Liu et al., 2026; Zhou et al., 2026) or by writing the control code with predefined primitives (Fu et al., 2026; Lu et al., 2026; Elmaaroufi et al., 2026). Nevertheless, these systems either read privileged simulator state (*e.g.*, ground-truth object poses) that is unavailable on real hardware (Singh et al., 2025; Tsui et al., 2026; Xie et al., 2026), or rely on a custom, hand-crafted library of perception and control primitives (*e.g.*, pose estimators and grasp samplers), which requires re-engineering for each new embodiment (Fu et al., 2026; Seker et al., 2026; Zhang et al., 2026a). OpenRUA instead provides the coding agent with the robot's native interface, ROS 2 (Macenski et al., 2022), the de facto standard of the robotics community, which is shared across embodiments and identical in simulation and on physical robots.

CLI Coding Agents. CLI coding agents, exemplified by Claude Code (Anthropic, 2025) and Codex (OpenAI, 2025), have emerged as general-purpose agents that resolve real-world software engineering tasks, as measured by benchmarks such as SWE-bench (OpenAI, 2024) and Terminal-Bench (Merrill et al., 2026). These agents exhibit strong closed-loop agency over long horizons: they navigate large, complex codebases, form and test hypotheses by editing and running code, issue CLI commands and interpret terminal output, and recover from their own errors through hours of autonomous iteration. Recently, this competence has extended beyond software engineering to operating desktop and web applications (Yang et al., 2026; Sun et al., 2026) and driving scientific discovery workflows (Wang et al., 2026b; Terminal-Bench-Science Team, 2026). OpenRUA turns this competence toward robot control by treating the robot as a live software project, in which the coding agent can transfer its strong capabilities to physical tasks without any adaptation.

Robot Operating System. ROS 2 provides a software framework that connects robot components and exposes their sensing, state, and control capabilities through common communication interfaces (Macenski et al., 2022). Topics carry streams such as camera images and joint states, services answer requests such as inverse kinematics queries, and actions support longer-running operations such as trajectory execution, with feedback and results. The native `ros2cli` command-line tools and `rclpy` Python client library allow developers to inspect these interfaces, acquire data, and send

control requests through terminal commands and programs, respectively. Robot controllers then execute the motion requests, while sensor and state interfaces provide feedback about the resulting physical changes. Prior work connects LLM agents to ROS through an additional abstraction layer of custom tools or predefined action libraries, with evaluations limited to in-house tasks rather than standard robot control benchmarks (Royce et al., 2024; Mower et al., 2024).

B Benchmarks and Evaluation Protocol

B.1 Benchmarks and Trial Sampling

The main evaluation contains 2,000 episodes: 700 on CaP-Bench, 800 on LIBERO-PRO, and 500 on RoboCasa365. All three benchmarks are evaluated in simulation using their task definitions and success predicates.

CaP-Bench. We evaluate Lift, Stack, Wipe, Nut, Restack, Two-Arm Lift, and Handover, with 100 trials per task (Fu et al., 2026). The last two tasks use two Panda arms; the others use one. Trial identifiers range from 1 to 100, following the benchmark's evaluation grid. The benchmark's reset implementation samples object placements randomly; these identifiers do not specify a fixed set of reproducible initial placements. Our trial count therefore matches the published protocol, but the initial states are not paired episode by episode with those of prior systems.

LIBERO-PRO. We evaluate the Object, Spatial, Goal, and LIBERO-10 suites under two perturbations: initial-position swaps (P in Table 1) and task-instruction changes (T) (Zhou et al., 2025). Each of the eight suite–perturbation cells contains ten tasks, and we use initial-state indices 0–9 from the released initial-state files for each task. This gives 100 trials per cell and 800 trials overall. The six Object, Spatial, and Goal cells form the 600-episode subset shared with several prior coding-agent evaluations; the two LIBERO-10 cells add 200 episodes. We retain the benchmark's perturbed scenes and task instructions and do not select initial states based on OpenRUA performance. The ten-trial count follows recent agent evaluations (Zhang et al., 2026b;a; Galanti et al., 2026); some earlier systems instead report 50 trials per task, as described in Appendix D.

RoboCasa365. We use the target50 evaluation set in the benchmark's target kitchen split: 18 Atomic-Seen tasks, 16 Composite-Seen tasks, and 16 Composite-Unseen tasks (Nasiriany et al., 2026). Seen and Unseen refer to whether the task templates occur in the benchmark's pretraining demonstrations, rather than whether OpenRUA has previously attempted them. Every task is evaluated for ten trials, using seeds 0–9, for 180, 160, and 160 episodes in the three groups. Each reset uses the benchmark's sampler to select an allowed kitchen layout and style, object instances, and placements; we do not force one trial per kitchen. Harness VLA uses the same task groups but evaluates ten held-out seeds per atomic task and five per composite task after a separate reference-seed exploration (Zhang et al., 2026b). Our ten-trial setting thus follows its atomic-task evaluation count and applies that count to all three groups, without reference-seed exploration. The GR00T results reproduced from the RoboCasa365 paper use 30 trials per task, as specified in its Appendix G.2.

B.2 Agent Configuration and Execution Protocol

The main experiments use Claude Code with Claude Opus 5 and high reasoning effort. Each episode uses the same workspace template for its robot configuration and a fresh agent session; files and task-specific findings are not carried between episodes. The agent receives the task instruction and the opening prompt shown in Figure 2a, with machine documentation and starter tools as described in Appendix C.2. No robot demonstrations or benchmark-specific rehearsal episodes are provided. The alternative agent and model configurations used in the scaling analysis are described in Appendix E.

Simulation and control. Simulation advances when control commands trigger environment steps and remains paused between those steps, including while the agent reasons or reads observations. A running control program can continue to issue commands and advance the simulation. Thus, model inference latency does not itself advance physical time. The robot exposes sensor, state, trajectory, gripper, and available velocity interfaces through ROS 2; the bridge executes these requests using

the simulator's robot controllers. The runs use joint-position control for arm trajectories and image observations rendered at 640×480 pixels. These execution settings differ from evaluations that feed low-resolution images to a learned policy and use its operational-space action representation; the benchmark scenes and success predicates remain those of the original tasks.

Budgets and termination. An episode ends when the agent finishes, reaches 500 agent turns, or exhausts 240 minutes of active wall-clock time, whichever occurs first. There is no additional simulation-step horizon in these runs. Active wall-clock time includes reasoning and tool execution but excludes periods when an episode is suspended because model access is unavailable. Resuming a suspended episode preserves its state and consumes the remaining budget rather than starting a new episode. These budgets are not assumed to match those of every baseline; comparisons use the published settings of each system.

Evaluation isolation. The agent runs in a separate sandbox with access to the robot's ROS 2 interfaces and its own workspace. The simulator implementation, privileged task-object poses, benchmark goal predicates, and evaluation records are not exposed in that workspace or as agent-callable tools. The evaluator checks success independently, and its verdict is neither returned to the agent nor used to terminate an episode early. The agent must decide when to finish using observations and execution feedback.

B.3 Scoring and Aggregation

The evaluator checks the benchmark's original success predicate at every simulation step. An episode counts as successful if the predicate is satisfied at any step before termination; once recorded, success is retained even if the final state later changes. The agent's own declaration of completion is not used as the success label. The first successful step and its timestamps are recorded for the budget analyses in Section 4.5.

Success rate is the fraction of evaluated episodes that satisfy this rule. CaP-Bench averages over seven equally sized task groups. For LIBERO-PRO, Table 1 reports both the six-cell average over Object, Spatial, and Goal and the eight-cell average that also includes LIBERO-10. The latter corresponds to all 800 episodes; missing baseline cells are not imputed. For RoboCasa365, the overall rate averages over all 50 tasks, weighting Atomic-Seen, Composite-Seen, and Composite-Unseen by their task counts of 18, 16, and 16. Each main configuration is evaluated once over its stated episode set; we do not interpret small numerical differences as evidence of statistical significance.

C Harness and Workspace

C.1 Native ROS Interfaces

ROS 2 organizes robot software into nodes that exchange typed messages through topics, services, and actions (Macenski et al., 2022). These communication mechanisms connect sensor drivers, state estimation, planning, and control components. The available names, message types, coordinate frames, and units depend on the robot and are described in the workspace documentation.

Communication and execution. Topics support publish–subscribe communication for sensor and state streams and for commands such as Cartesian velocities. Services return a response to a request, for example an inverse-kinematics solution for a target pose. Actions accept execution goals and provide feedback and a final result, with support for cancellation; joint-trajectory execution is one example. Controllers handle motion requests through the corresponding interfaces, while sensor and state messages report the robot's evolving physical state. The observation message types and file representations are detailed in Appendix C.3, and the trajectory, gripper, and velocity interfaces in Appendix C.6.

Terminal and programmatic access. The `ros2cli` tools can list available interfaces, inspect their types, read or publish messages, call services, and send action goals. For example, `ros2 topic list -t` lists topics and their message types, while `ros2 interface show <type>` displays the fields of an interface definition. The `rclpy` library provides Python access

to these same communication mechanisms through subscriptions, publishers, and service and action clients. A program can keep these connections active and process incoming messages through callbacks, allowing sensor readings and motion updates to be combined in a running feedback loop. Both entry points belong to the existing ROS 2 development tools; OpenRUA does not require an additional agent-specific API to invoke them.

C.2 WORKSPACE

The initial workspace contains `README.md`, `machine.yaml`, documentation, and ordinary source files for basic operations such as saving camera observations and sending motion requests. The machine description specifies the robot's sensors, control interfaces, joint limits, and coordinate frames. The README directs the agent to these resources; the agent can inspect or modify starter tools and write additional programs through its existing terminal and file capabilities. The workspace does not contain task solutions or findings from previous episodes.

The opening prompt is reproduced in Figure 2a; only its task placeholder changes between episodes. It asks the agent to survey the machine, carry out the task, and verify the result without prescribing a sequence of perception or manipulation steps. During an episode, the agent can retain observations, numerical measurements, programs, notes, and logs in the workspace. These files persist within the episode, including across a resumed session, but are not transferred to the next episode. The evaluator remains separate as described in Appendix B.2.

C.2.1 INITIAL MATERIALS AND FILE INVENTORY

We reproduce the complete workspace template below: six Markdown documents and six Python starter tools. The listings retain the original file contents; long lines are wrapped for typesetting, and Python line numbers are added for reference. These are the materials supplied before the agent begins the task; programs subsequently written by the agent are presented separately in Appendix H.

Provided guidance. The documentation includes recommendations for acquiring and checking observations, computing coordinates, and using sensor feedback, together with generic working patterns. The starter tools include pixel back-projection and an odometry-based mobile-base controller. These resources provide initial guidance and executable examples; the agent chooses which to use, modify, or replace when developing its task-specific solution.

File	Supplied content
`README.md`	Entry point, file map, and example discovery commands.
`docs/10-machine.md`	Machine-manifest fields, ROS tools, installed software, and simulation timing.
`docs/20-perception.md`	Joint states, transforms, force/torque, odometry, cameras, and calibration.
`docs/30-action.md`	Trajectory, velocity, gripper, mobile-base, and planning interfaces.
`docs/40-patterns.md`	Generic verification, motion, grasping, troubleshooting, and session guidance.
`tools/README.md`	Tool index with usage and operation descriptions for all six utilities.
`tools/perception/cam_snap.py`	Save an image topic as a PNG; also save a numerical array for depth.
`tools/perception/px2world.py`	Convert a selected depth pixel to world coordinates using intrinsics and TF.
`tools/action/fjt_send.py`	Send a joint-trajectory action goal and wait for its result.
`tools/action/gripper_cmd.py`	Send a gripper action goal and print the returned status.
`tools/action/ik_move.py`	Request inverse kinematics and execute the resulting joint trajectory.
`tools/action/base_goto.py`	Drive a mobile base toward a pose using odometry feedback.

Generated machine description. The workspace root additionally receives `machine.yaml`, generated from the resolved robot configuration at startup rather than copied from the static tem-

plate. It contains `schema`, `robot`, `frames`, `actuators`, `sensors`, `planning`, and `hand` fields. Actuator and sensor entries identify capability kinds, ROS ports and message types, with joint names, limits, coordinate frames, and gripper properties where applicable. Multiple arms yield separate actuator and force-sensor entries; a mobile base adds velocity and odometry interfaces. The manifest generator omits actuator entries without a port and sensor entries without a port or topic pattern, and adds explanatory comments for interface fields and units. Consequently, the concrete manifest varies with the machine; the documents and tool source below constitute the shared template. The opening prompt is reproduced in Figure 2a.

C.2.2 WORKSPACE DOCUMENTATION

The following listings reproduce the Markdown documents as text, including their command examples and operating guidance.

Listing 1: `README.md`

```
# Workspace: start here

You are on the onboard computer of a robot arm workstation. This
workspace ships structured startup docs and a few generic tools.

## Map

| Read | For |
|---|---|
| [machine.yaml](machine.yaml) | THIS machine's facts: robot model, ports, joints, limits,
    frames |
| [docs/10-machine.md](docs/10-machine.md) | how to read machine.yaml; software stack, runtime
     environment, timing |
| [docs/20-perception.md](docs/20-perception.md) | how to sense: joints, poses, force, cameras
     |
| [docs/30-action.md](docs/30-action.md) | how to move: trajectories, servo, gripper, mobile
    base, planning |
| [docs/40-patterns.md](docs/40-patterns.md) | reliable working patterns and troubleshooting |
| [tools/README.md](tools/README.md) | ready-made utilities (perception/ and action/) |

## 30-second start

```bash
ros2 topic list                                    # what the machine exposes
ros2 topic echo /joint_states --once               # where the arm is
python3 tools/perception/cam_snap.py agentview     # what the scene looks like
```

Docs are numbered in reading order; each capability section follows the
same shape: what it is → port → `ros2` CLI → `rclpy` → tool shortcut →
practical notes.
```

Listing 2: `docs/10-machine.md`

```
# 10. The machine

Top-down: where the facts live → software stack → runtime environment →
timing.

## 1. Machine facts live in `machine.yaml`

This doc set is generic; everything specific to THIS machine; robot
model, joint names and limits, port names, frames, planning setup; is
in **`machine.yaml`** at the workspace root. Read it first. Structure:

- `robot`; model and one-line description
- `frames`; the TF names for `world`, the arm `base`, the `hand`
- `actuators`; a list of typed write-ports, each entry:
  `kind` (`joint_trajectory` / `cartesian_twist` / `gripper` / ...),
  `port`, message `type`, plus kind-specific facts (joint names,
  limits, open/closed positions)
- `sensors`; a list of typed read-ports (`joint_state`, `wrench`,
  `camera_set`, ...)

Every key in it is defined in a comment on its own line, units included;
`machine.yaml` explains itself and needs nothing from this doc.
- `planning`; MoveIt entry points, if present

Each `kind` has a how-to section in [20-perception.md](20-perception.md)
or [30-action.md](30-action.md). The manifest is an index of the ROS
```

````
graph; `ros2 topic list` / `ros2 action list` show the same ports live
(and anything beyond the manifest).

## 2. Software stack

- **ROS 2** is the control middleware: everything is a topic, service
  or action. **`ros2` CLI** for shell access, **`rclpy`** for Python.
- The `ros2` CLI verbs that cover daily work:

  | Verb | Does |
  |---|---|
  | `ros2 topic list` / `info <t>` / `echo <t> --once` | find, inspect, read topics |
  | `ros2 topic pub [--rate N] <t> <type> '{...}'` | publish (once or streaming) |
  | `ros2 action list` / `send_goal <a> <type> '{...}'` | find and call actions |
  | `ros2 service list` / `call <s> <type> '{...}'` | find and call services |
  | `ros2 node list` | who is running |
  | `ros2 interface show <type>` | field layout of any message type |
  | `ros2 run tf2_ros tf2_echo <from> <to>` | read a TF transform |

- Read `machine.yaml` in Python:
  ```python
  import yaml
  M = yaml.safe_load(open("machine.yaml"))
  fjt = next(a for a in M["actuators"] if a["kind"] == "joint_trajectory")
  ```

## 3. Runtime environment

- ROS is **already sourced** in every shell (bash and sh alike);
  no need to `source` manually.
- Python 3 has `rclpy`, `numpy`, `scipy`, `cv2`, `PIL`, `cv_bridge`,
  `tf2_ros`, `control_msgs`, `moveit_msgs`, `PyKDL`, `urdf_parser_py`.
- **This directory (`/workspace`) is yours**: write scripts, images,
  notes here.
- **No internet.** `pip install` and web access fail; everything needed
  is installed.

## 4. Timing

The system clock (`/clock`) advances with robot activity, not wall time.
Consequences:

- `sleep` does not let the physical world settle;
- confirm every motion by re-reading sensors after the command
  completes, not by waiting.

## Timing facts (paused-clock machine)

- At session start the scene may not have settled yet (objects can sit
  slightly above their supports); the world only advances while commands
  execute; waiting wall-clock time changes nothing.
- A few sim-seconds of trajectory can take minutes of wall time to
  return under load. A client-side timeout is NOT a motion failure;
  verify against `/joint_states` before re-sending.
````

Listing 3: `docs/20-perception.md`

````
# 20. Perception

Two families: **A. proprioception** (the robot's own state) and
**B. vision** (cameras). Concrete port names come from `machine.yaml`
(`sensors` list); sections below are keyed by sensor `kind`. Every
section: what → port → CLI → rclpy → tool → notes.

## A. Proprioception

### A1. Joint state (kind `joint_state`)

- **What**: positions/velocities/efforts for every joint.
- **Port**: `machine.yaml → sensors[kind=joint_state].port`
  (`sensor_msgs/msg/JointState`).
- **CLI**: `ros2 topic echo <port> --once`
- **rclpy**: subscribe once;
  ```python
  import rclpy, yaml
  from sensor_msgs.msg import JointState
  port = next(s for s in yaml.safe_load(open("machine.yaml"))["sensors"]
              if s["kind"] == "joint_state")["port"]
  rclpy.init(); node = rclpy.create_node("js")
  got = []
````
````

```
  node.create_subscription(JointState, port, got.append, 1)
  while not got: rclpy.spin_once(node, timeout_sec=0.5)
  print(dict(zip(got[0].name, got[0].position)))
  ```
- **Notes**: message order is not fixed; always match by joint `name`
  (arm joint names: `machine.yaml` → actuators[kind=joint_trajectory].joints`).

A2. Pose (kinematics via TF)

- **What**: where any frame is relative to any other (`frames` in
  `machine.yaml` names the important ones: `world`, `base`, `hand`).
- **CLI**: `ros2 run tf2_ros tf2_echo <world> <hand>` (frame names from
  `machine.yaml` → frames`; prints once a second, Ctrl-C after the first).
- **rclpy**: `tf2_ros.Buffer` + `TransformListener`, then
  `buffer.lookup_transform(world, hand, Time())`.
- **Notes**: TF gives you where the ROBOT is; object locations you
  estimate from camera images (B) + geometry.

A3. Wrist force/torque (kind `wrench`)

- **What**: external wrench at the end effector; contact evidence.
- **Port**: `machine.yaml` → sensors[kind=wrench].port`
  (`geometry_msgs/msg/WrenchStamped`).
- **CLI**: `ros2 topic echo <port> --once`
- **Notes**: baseline is not zero (gravity); watch for *changes* on
  contact, not absolute values.

A4. Odometry (kind `odometry` (only on mobile machines))

- **What**: the mobile base's pose; absent from the manifest = fixed
  base, skip.
- **Port**: `machine.yaml` → sensors[kind=odometry].port`
  (`nav_msgs/msg/Odometry`; pose in the `world` frame, `child_frame_id`
  from the manifest entry's `frame`).
- **CLI**: `ros2 topic echo <port> --once`
- **Notes**: read before and after driving (30-action D); displacement
  is your evidence the base actually moved.

B. Vision (kind `camera_set`)

B1. Discover the cameras

- The manifest entry gives the topic pattern and the discovery command:
  `ros2 topic list | grep image_raw`. Each camera `<name>` publishes
  color, depth and intrinsics topics per that pattern.

B2. Grab a color frame

- **Tool**: `python3 tools/perception/cam_snap.py <name>` → `<name>.png`,
  then open the PNG (read the file).
- **rclpy**: subscribe to the topic, convert with
  `cv_bridge.CvBridge().imgmsg_to_cv2(msg, "bgr8")`.

B3. Depth and intrinsics

- Depth: same pattern on the depth topic (`cam_snap.py` accepts a full
  topic path; depth is 32-bit float meters; convert with
  `imgmsg_to_cv2(msg, "passthrough")`).
- Intrinsics: `ros2 topic echo /<name>/color/camera_info --once`
  (`k` = 3×3 camera matrix); pixels + depth → 3D points in the camera
  optical frame, then TF into `world`.

C. Graph introspection (beyond the manifest)

- `ros2 topic list` / `ros2 service list` / `ros2 action list` /
  `ros2 node list`
- `ros2 topic info <topic>` and `ros2 interface show <type>` explain any
  port you find.

Sensor-surface facts

- The TF arm chain may be DISCONNECTED from the world tree (lookups
  like world->hand fail with "unconnected trees"). For the hand pose
  use the FK service; TF stamps are sim-time and a cached buffer can
  serve stale poses after motion.
- `/robot_description` publishes with TRANSIENT_LOCAL durability; a
  plain subscription misses it; request that QoS.
- Camera topics may appear in the graph lazily; the manifest's
  camera_set entry is authoritative, `ros2 node info` shows the full set.
- Oblique (side-view) cameras systematically overestimate the height of
  low, flat objects near their edges; cross-view disagreement on such
```
```

```
  objects usually traces to this.
- The odometry topic publishes only as the sim advances.

**Tool**: `python3 tools/perception/px2world.py <camera> <u> <v>`;
depth + intrinsics + TF for one pixel, printed as world x y z.
```

Listing 4: `docs/30-action.md`

```
# 30. Action

Ways to move, coarse to fine, keyed by actuator `kind` in `machine.yaml`
(`actuators` list; a dual-arm machine simply has two `joint_trajectory`
entries; check what THIS machine has). Every section: what → port →
CLI → rclpy → tool → notes.

## A. Joint trajectory (kind `joint_trajectory` (the workhorse))

- **What**: track a joint-space trajectory; the primary way to move.
- **Port**: `machine.yaml → actuators[kind=joint_trajectory]` gives the
  action name, the joint names (in order) and their limits.
- **Tool**: `python3 tools/action/fjt_send.py <p1,...,pN> <seconds>`
- **rclpy**: `ActionClient` (`control_msgs/action/FollowJointTrajectory`)
  sending a trajectory whose points have `positions` (one per joint,
  manifest order) and `time_from_start`; several points = one smooth
  pass (see `fjt_send.py` for the skeleton).
- **CLI**: possible via `ros2 action send_goal`, but the YAML is
  unwieldy; prefer the tool/rclpy.
- **Notes**:
  - 2–4 s for large moves; too-short durations overshoot;
  - action result returning ≠ target reached exactly; re-read the
    joint state and compare;
  - nonzero `error_code` = tracking problem (see 40-patterns.md).

## B. Cartesian servo (kind `cartesian_twist`)

- **What**: stream end-effector velocities for small precise motion.
- **Port**: `machine.yaml → actuators[kind=cartesian_twist]`; topic,
  and the `frame` to stamp commands with (linear is m/s).
- **CLI**: `ros2 topic pub --rate 20 <port> geometry_msgs/msg/TwistStamped \
  "{header: {frame_id: <frame>}, twist: {linear: {z: -0.05}}}"`;
  Ctrl-C to stop the stream.
- **rclpy** (the world only advances while commands stream; publish in
  a loop, then re-check sensors):
  ```python
  import rclpy, yaml
  from geometry_msgs.msg import TwistStamped
  tw = next(a for a in yaml.safe_load(open("machine.yaml"))["actuators"]
            if a["kind"] == "cartesian_twist")
  rclpy.init(); node = rclpy.create_node("nudge")
  pub = node.create_publisher(TwistStamped, tw["port"], 10)
  msg = TwistStamped(); msg.header.frame_id = tw["frame"]
  msg.twist.linear.z = -0.05                # 5 cm/s downward
  for _ in range(20):                       # ~1 s of motion
      pub.publish(msg); rclpy.spin_once(node, timeout_sec=0.05)
  ```
- **Notes**: burst → sense → correct beats one long blind stream.

C. Gripper (kind `gripper`)

- **What**: open/close the fingers. The manifest entry carries this
  gripper's facts (`open_m`, `closed_m`, `stops_at`, `max_effort`),
  each with its definition written on the line.
- **Port**: `machine.yaml → actuators[kind=gripper]`
  (`control_msgs/action/GripperCommand`).
- **Tool**: `python3 tools/action/gripper_cmd.py <width_m>`
- **CLI** (fill `<port>`, `open_m`/`closed_m`, `max_effort` from the
  manifest entry):
  `ros2 action send_goal <port> control_msgs/action/GripperCommand \
  "{command: {position: <m>, max_effort: <n>}}"`
- **Notes**: after closing on something, finger positions in the joint
  state stop above `closed_m`; that gap is your grasp evidence.

D. Mobile base (kind `base_twist` (only on mobile machines))

- **What**: drive the base with body-frame velocities. Absent from the
  manifest = this machine's base is fixed; skip this section.
- **Port**: `machine.yaml → actuators[kind=base_twist]`; plain
  `geometry_msgs/msg/Twist` (`/cmd_vel` convention): `linear.x` forward,
  `linear.y` left (holonomic bases only; check `type`), `angular.z`
  counter-clockwise yaw.
```

```
- **Tool**: `python3 tools/action/base_goto.py <x> <y> [yaw]`;
  closed-loop on odometry; exits with the last pose as fact on
  stall.
- **CLI**: `ros2 topic pub --rate 10 <port> geometry_msgs/msg/Twist \
  "{linear: {x: 0.3}}"`; Ctrl-C to stop.
- **Semantics**: **one message = one control tick** of motion at that
  velocity. Messages queue up to a small depth (4); anything beyond is
  dropped, so flooding buys nothing. Queued ticks still execute after
  you stop publishing; pace yourself by odometry feedback, not by
  message count or wall time.
- **Notes**:
  - closed-loop is the only reliable pattern: publish a few ticks →
    read odometry (20-perception A4) → correct → repeat;
  - drive and arm motion are separate commands; position the base
    first, then work the arm;
  - operational norm for go-to/docking jobs: park the base directly in
    front of the target, within arm's reach (close enough to work on it
    without re-driving), squarely facing it. "Near" is not parked.

## E. Planning (MoveIt): `machine.yaml → planning`

- **What**: collision-aware planning and IK when raw trajectories are
  not enough (present when `planning.moveit` is true).
- **Ports**: `planning.move_action` (plan+execute,
  `moveit_msgs/action/MoveGroup`, group `planning.group`);
  `planning.ik_service` (`moveit_msgs/srv/GetPositionIK`); Cartesian
  pose → joint positions.
- **Tool**: `python3 tools/action/ik_move.py <x> <y> <z> <qx> <qy> <qz>
  <qw> [seconds] [--at tcp]`; IK then trajectory, loud on failure.
- **Pattern**: IK for the target pose, then A (trajectory) to go there;
  or `move_action` end-to-end when obstacles matter.
- **Notes**: check the returned `error_code` (1 = success); frame
  behavior is NOT what headers suggest; see "Planning facts" below.

## Actuation facts

- Gripper commands are PER-FINGER positions: the opening between the
  fingers is twice the commanded value (nominals in machine.yaml).
  What this machine does with the commanded value is its own fact, not
  a general rule: read `stops_at` on the manifest's gripper entry. Gap
  readings need a few settle ticks; the measured maximum opening can sit
  slightly under nominal.
- A trajectory tolerance-violation result on a long goal is often
  controller lag, not an obstruction: resending the same goal typically
  converges.
- Arm contact and reaction forces can displace the mobile base by
  centimeters; re-read odometry after arm work before trusting stored
  base poses.

## Planning facts (MoveIt)

- The planner's model frame is the ARM BASE even where message headers
  say "world" (machine.yaml planning.planning_frame). Leave `frame_id`
  EMPTY in IK requests; a base-frame label is rejected (-21).
- The planning scene starts EMPTY; no world collision objects exist
  until you publish them; collision meshes carry roughly a centimeter
  of padding, so trust measurements over nominals.
- IK/FK requests HANG if their robot_state carries non-arm joints
  (finger joints included): seed with the arm joints only.
- IK can fail silently (no error, no motion) and non-monotonically
  (nearby poses flip feasibility). Seed requests with the current
  configuration to keep branch continuity, and verify motion actually
  happened from `/joint_states`.
```

Listing 5: `docs/40-patterns.md`

```
# 40. Working patterns

Cross-cutting habits that save turns and time.

## P1. Act → verify, every time

The clock only moves with activity (10-machine.md §4), so verification
is sensor re-reads, never waits:

1. command (30-action) → 2. re-read the relevant sensor (20-perception)
→ 3. compare with what you intended → 4. correct or continue.

Motion: compare the joint state against your target. Manipulation:
finger gap + a fresh camera frame. World changes: before/after frames
```

```
from the same camera.

## P2. Coarse-to-fine motion

Large repositioning with one trajectory move (fast, accurate over big
spans), final approach with short servo bursts (small, sensed,
correctable). Avoid many small trajectory hops and long blind servo
streams alike.

## P3. Approach-grasp-verify (generic pick pattern)

1. Snapshot cameras; locate the target object in the image (pixels +
   depth + intrinsics → world position, 20-perception B3).
2. Move above it (IK → trajectory), open the gripper.
3. Descend with short servo bursts, watching a wrist camera if present.
4. Close the gripper; verify the grasp: finger gap above `closed_m`,
   wrench change on contact.
5. Lift and confirm visually (fresh frame) before moving on.

Placing is the mirror: position above the destination, descend, open,
retreat, verify with a fresh frame.

## P4. Troubleshooting table

| Symptom | Likely cause | Fix |
|---|---|---|
| trajectory result `error_code != 0` | duration too short / target beyond a joint limit |
    slow down; check limits (machine.yaml) |
| arm lags the servo stream | per-message motion is tiny | keep streaming in a loop; verify
    between bursts |
| gripper "closed" but nothing held | closed on air | reopen, re-approach lower/centered, use
    the wrist camera |
| image looks stale | pulled an old frame | snapshot again after the motion completed |
| a port seems missing | wrong name assumed | `ros2 topic list` / `ros2 action list`, then `
    ros2 topic info` |

## Session-runtime facts

- NEVER end your turn while a motion or background task is pending: an
  ended turn ends the session; nothing resumes it, and the paused
  world means what you were "waiting for" will never arrive. Poll to
  completion first.
- Long commands: run them detached writing to a file with `python3 -u`
  (pipes buffer output invisibly), then poll the file with short
  commands. Build service/action clients once and reuse them; every
  rebuild costs seconds.
```

Listing 6: `tools/README.md`

```
# tools/: minimal utilities

Six small Python tools for camera capture, geometric measurement, and
motion requests. Plain source files: read them, run them, adapt them,
or write your own programs using the native ROS interfaces (see docs).

| Tool | Usage | Operation |
|---|---|---|
| `perception/cam_snap.py` | `python3 tools/perception/cam_snap.py <camera_or_topic> [out.png
    ]` | save a camera frame (color → PNG; depth → visualization PNG + numerical `.npy`
    array) |
| `perception/px2world.py` | `python3 tools/perception/px2world.py <camera> <u> <v>` | convert
     a selected depth pixel to world coordinates using camera intrinsics and TF |
| `action/fjt_send.py` | `python3 tools/action/fjt_send.py <p1,...,pN> <seconds>` | send a
    joint trajectory and wait for its result; joint names and port come from `machine.yaml` |
| `action/gripper_cmd.py` | `python3 tools/action/gripper_cmd.py <width_m>` | command a
    gripper position and print the result; `width_m` is the per-finger position |
| `action/ik_move.py` | `python3 tools/action/ik_move.py <x> <y> <z> <qx> <qy> <qz> <qw> [
    seconds]` | request inverse kinematics, then execute the joint trajectory; see `docs/30-
    action.md` for planning-frame conventions |
| `action/base_goto.py` | `python3 tools/action/base_goto.py <x> <y> [yaw_rad]` | drive a
    mobile base toward a world pose using odometry feedback; requires base and odometry ports
     in `machine.yaml` |
```

C.2.3 STARTER TOOL SOURCE CODE

The six tools below are provided as ordinary Python source files. Their arguments select sensor inputs or motion targets; the agent can inspect or modify the implementations and write additional programs using the same native interfaces.

Listing 7: `tools/perception/cam_snap.py`

```python
#!/usr/bin/env python3
"""Save one frame from an image topic to a PNG.

Usage: python3 tools/cam_snap.py <camera_name> [out.png]
Reads /<camera_name>/color/image_raw (pass a full topic to use it as-is).
"""
import sys

import rclpy
from cv_bridge import CvBridge
from sensor_msgs.msg import Image

import cv2


def main() -> None:
    if len(sys.argv) < 2:
        raise SystemExit(__doc__)
    cam = sys.argv[1]
    topic = cam if cam.startswith("/") else f"/{cam}/color/image_raw"
    out = sys.argv[2] if len(sys.argv) > 2 else f"{cam.strip('/').split('/')[0]}.png"
    rclpy.init()
    node = rclpy.create_node("cam_snap")
    got: list = []
    node.create_subscription(Image, topic, lambda m: got.append(m), 1)
    while not got:
        rclpy.spin_once(node, timeout_sec=1.0)
    msg = got[0]
    if "FC" in msg.encoding or "16UC" in msg.encoding:  # depth image
        import numpy as np
        depth = CvBridge().imgmsg_to_cv2(msg, desired_encoding="passthrough")
        np.save(out.rsplit(".", 1)[0] + ".npy", depth)  # raw meters
        finite = depth[np.isfinite(depth)]
        lo, hi = (finite.min(), finite.max()) if finite.size else (0.0, 1.0)
        cv2.imwrite(out, ((depth - lo) / max(hi - lo, 1e-6) * 255).clip(0, 255)
                    .astype("uint8"))
        print(f"{out} (visualization; raw meters in .npy)")
    else:
        cv2.imwrite(out, CvBridge().imgmsg_to_cv2(msg, desired_encoding="bgr8"))
        print(out)
    node.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

Listing 8: `tools/perception/px2world.py`

```python
#!/usr/bin/env python3
"""Convert one image pixel to world coordinates (depth + intrinsics + TF).

Usage: python3 tools/perception/px2world.py <camera> <u> <v>
Prints "x y z" in the world frame for pixel column u, row v of the
camera's CURRENT depth frame. Exits nonzero if the pixel has no valid
depth. Pure geometry; which pixel is worth asking about is your call.
"""
import struct
import sys

import numpy as np
import rclpy
from sensor_msgs.msg import CameraInfo, Image
from tf2_ros import Buffer, TransformListener


def _grab(node, topic, msg_type, timeout=15.0):
    got = {}
    sub = node.create_subscription(msg_type, topic,
                                   lambda m: got.setdefault("m", m), 1)
    end = node.get_clock().now().nanoseconds / 1e9 + timeout
```

```
    while "m" not in got and node.get_clock().now().nanoseconds / 1e9 < end:
        rclpy.spin_once(node, timeout_sec=0.2)
    node.destroy_subscription(sub)
    if "m" not in got:
        raise SystemExit(f"no message on {topic} within {timeout}s")
    return got["m"]


def main() -> None:
    if len(sys.argv) != 4:
        raise SystemExit(__doc__)
    cam, u, v = sys.argv[1], int(sys.argv[2]), int(sys.argv[3])
    rclpy.init()
    node = rclpy.create_node("px2world")
    tfbuf = Buffer()
    TransformListener(tfbuf, node)

    depth = _grab(node, f"/{cam}/depth/image_raw", Image)
    info = _grab(node, f"/{cam}/color/camera_info", CameraInfo)
    if not (0 <= v < depth.height and 0 <= u < depth.width):
        raise SystemExit(f"pixel ({u},{v}) outside {depth.width}x{depth.height}")
    z = struct.unpack_from("<f", depth.data, (v * depth.width + u) * 4)[0]
    if not np.isfinite(z) or z <= 0:
        raise SystemExit(f"no valid depth at ({u},{v}): {z}")

    fx, fy = info.k[0], info.k[4]
    cx, cy = info.k[2], info.k[5]
    p_cam = np.array([(u - cx) * z / fx, (v - cy) * z / fy, z, 1.0])

    frame = f"{cam}_optical_frame"
    end = node.get_clock().now().nanoseconds / 1e9 + 10.0
    while node.get_clock().now().nanoseconds / 1e9 < end:
        rclpy.spin_once(node, timeout_sec=0.2)
        if tfbuf.can_transform("world", frame, rclpy.time.Time()):
            break
    t = tfbuf.lookup_transform("world", frame, rclpy.time.Time())
    q = t.transform.rotation
    x, y, zz, w = q.x, q.y, q.z, q.w
    R = np.array([
        [1 - 2 * (y * y + zz * zz), 2 * (x * y - zz * w), 2 * (x * zz + y * w)],
        [2 * (x * y + zz * w), 1 - 2 * (x * x + zz * zz), 2 * (y * zz - x * w)],
        [2 * (x * zz - y * w), 2 * (y * zz + x * w), 1 - 2 * (x * x + y * y)],
    ])
    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = [t.transform.translation.x, t.transform.translation.y,
                t.transform.translation.z]
    p = T @ p_cam
    print(f"{p[0]:.4f} {p[1]:.4f} {p[2]:.4f}")
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

Listing 9: `tools/action/fjt_send.py`

```
#!/usr/bin/env python3
"""Send one joint-space trajectory goal and wait for the result.

Usage: python3 tools/action/fjt_send.py <p1,p2,...,pN> <seconds>
Joint names and action port come from machine.yaml (kind
joint_trajectory); N must match that entry's joint count.
"""
import sys
from pathlib import Path

import rclpy
import yaml
from builtin_interfaces.msg import Duration
from control_msgs.action import FollowJointTrajectory
from rclpy.action import ActionClient
from trajectory_msgs.msg import JointTrajectoryPoint


def manifest_entry() -> dict:
    root = Path(__file__).resolve().parents[2]
    m = yaml.safe_load((root / "machine.yaml").read_text())
    return next(a for a in m["actuators"] if a["kind"] == "joint_trajectory")

```

```

def main() -> None:
    if len(sys.argv) != 3:
        raise SystemExit(__doc__)
    entry = manifest_entry()
    joints, action = entry["joints"], entry["port"]
    positions = [float(x) for x in sys.argv[1].split(",")]
    seconds = float(sys.argv[2])
    if len(positions) != len(joints):
        raise SystemExit(f"need {len(joints)} positions ({joints}), "
                         f"got {len(positions)}")
    rclpy.init()
    node = rclpy.create_node("fjt_send")
    client = ActionClient(node, FollowJointTrajectory, action)
    if not client.wait_for_server(timeout_sec=10.0):
        raise SystemExit(f"no action server at {action}")
    goal = FollowJointTrajectory.Goal()
    goal.trajectory.joint_names = joints
    pt = JointTrajectoryPoint(positions=positions)
    pt.time_from_start = Duration(sec=int(seconds),
                                  nanosec=int((seconds % 1) * 1e9))
    goal.trajectory.points = [pt]
    send = client.send_goal_async(goal)
    rclpy.spin_until_future_complete(node, send)
    result = send.result().get_result_async()
    rclpy.spin_until_future_complete(node, result)
    code = result.result().result.error_code
    print(f"done error_code={code}" + ("" if code == 0 else " (nonzero = tracking problem)")
     )
    node.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

Listing 10: `tools/action/gripper_cmd.py`

```
#!/usr/bin/env python3
"""Open/close the gripper to a target width and wait for the result.

Usage: python3 tools/action/gripper_cmd.py <width_m>
width_m is the desired PER-FINGER position (machine.yaml gripper:
open_m..closed_m). Command a target width, not 0: driving to 0 against
an object keeps squeezing it. Absent from the manifest = this machine
has no gripper.
"""
import sys
from pathlib import Path

import rclpy
import yaml
from control_msgs.action import GripperCommand
from rclpy.action import ActionClient


def main() -> None:
    if len(sys.argv) != 2:
        raise SystemExit(__doc__)
    width = float(sys.argv[1])
    root = Path(__file__).resolve().parents[2]
    m = yaml.safe_load((root / "machine.yaml").read_text())
    entry = next((a for a in m["actuators"] if a["kind"] == "gripper"), None)
    if entry is None:
        raise SystemExit("this machine has no gripper (see machine.yaml)")
    rclpy.init()
    node = rclpy.create_node("gripper_cmd")
    client = ActionClient(node, GripperCommand, entry["port"])
    if not client.wait_for_server(timeout_sec=10.0):
        raise SystemExit(f"no action server at {entry['port']}")
    goal = GripperCommand.Goal()
    goal.command.position = width
    goal.command.max_effort = float(entry.get("max_effort", 30.0))
    fut = client.send_goal_async(goal)
    rclpy.spin_until_future_complete(node, fut, timeout_sec=30)
    handle = fut.result()
    res_fut = handle.get_result_async()
    rclpy.spin_until_future_complete(node, res_fut, timeout_sec=120)
    res = res_fut.result().result
    print(f"reached_goal={res.reached_goal} stalled={res.stalled}")
```

```
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

Listing 11: `tools/action/ik_move.py`

```
#!/usr/bin/env python3
"""Move the hand to a world-frame pose: IK -> one trajectory -> done.

Usage: python3 tools/action/ik_move.py <x> <y> <z> <qx> <qy> <qz> <qw> \
           [seconds=4] [--at tcp|hand]
Pose is where the HAND frame goes; --at tcp aims the fingertip point
instead (offset from machine.yaml gripper.tcp_offset_m along hand -Z...
+Z; the offset is applied along the hand's approach axis). IK failures
exit loudly with the MoveIt error code; no solution means NO MOTION
happened. Approach direction, grasp logic, verification stay yours.
"""
import sys
from pathlib import Path

import numpy as np
import rclpy
import yaml
from moveit_msgs.srv import GetPositionIK
from sensor_msgs.msg import JointState


def manifest() -> dict:
    root = Path(__file__).resolve().parents[2]
    return yaml.safe_load((root / "machine.yaml").read_text())


def main() -> None:
    args = [a for a in sys.argv[1:] if not a.startswith("--")]
    if len(args) not in (7, 8):
        raise SystemExit(__doc__)
    x, y, z, qx, qy, qz, qw = map(float, args[:7])
    seconds = float(args[7]) if len(args) == 8 else 4.0
    at_tcp = "--at" in sys.argv and "tcp" in sys.argv

    m = manifest()
    traj_entry = next(a for a in m["actuators"]
                      if a["kind"] == "joint_trajectory")
    planning = m.get("planning", {})
    if at_tcp:
        off = float(m.get("hand", {}).get("tcp_offset_m", 0.0))
        # shift the target back along the hand's approach (+Z of the
        # hand frame, expressed in world via the quaternion)
        R = _quat_to_R(qx, qy, qz, qw)
        x, y, z = np.array([x, y, z]) - off * R[:, 2]

    rclpy.init()
    node = rclpy.create_node("ik_move")
    js = {}
    node.create_subscription(JointState, "/joint_states",
                             lambda msg: js.setdefault("m", msg), 1)
    end = node.get_clock().now().nanoseconds / 1e9 + 10
    while "m" not in js and node.get_clock().now().nanoseconds / 1e9 < end:
        rclpy.spin_once(node, timeout_sec=0.2)
    if "m" not in js:
        raise SystemExit("no /joint_states")

    cli = node.create_client(GetPositionIK, planning.get("ik_service",
                                                         "/compute_ik"))
    if not cli.wait_for_service(timeout_sec=10):
        raise SystemExit("IK service unavailable")
    req = GetPositionIK.Request()
    req.ik_request.group_name = planning.get("group", "panda_arm")
    # machine fact: leave frame_id empty; poses are interpreted in the
    # planner's model frame (the arm base), see machine.yaml planning
    req.ik_request.pose_stamped.header.frame_id = ""
    p = req.ik_request.pose_stamped.pose
    p.position.x, p.position.y, p.position.z = x, y, z
    p.orientation.x, p.orientation.y = qx, qy
    p.orientation.z, p.orientation.w = qz, qw
    # seed with ARM joints only: a full /joint_states (finger joints
    # included) can hang move_group's IK service indefinitely
    seed = JointState()
```

```
    arm = set(traj_entry["joints"])
    for n_, p_ in zip(js["m"].name, js["m"].position):
        if n_ in arm:
            seed.name.append(n_)
            seed.position.append(p_)
    req.ik_request.robot_state.joint_state = seed
    fut = cli.call_async(req)
    rclpy.spin_until_future_complete(node, fut, timeout_sec=60)
    res = fut.result()
    if res is None:
        raise SystemExit("IK service did not answer within 60s; no motion "
                         "sent (service load, not necessarily unreachable)")
    if res.error_code.val != 1:
        raise SystemExit(f"IK FAILED (error_code={res.error_code.val}) "
                         "; no motion sent")

    sol = dict(zip(res.solution.joint_state.name,
                   res.solution.joint_state.position))
    joints = traj_entry["joints"]
    positions = ",".join(f"{sol[j]:.6f}" for j in joints)
    rclpy.shutdown()
    # execute through the standard trajectory tool (same act path you
    # would use by hand)
    import subprocess
    r = subprocess.run([sys.executable,
                        str(Path(__file__).parent / "fjt_send.py"),
                        positions, str(seconds)])
    sys.exit(r.returncode)


def _quat_to_R(x, y, z, w):
    return np.array([
        [1 - 2 * (y * y + z * z), 2 * (x * y - z * w), 2 * (x * z + y * w)],
        [2 * (x * y + z * w), 1 - 2 * (x * x + z * z), 2 * (y * z - x * w)],
        [2 * (x * z - y * w), 2 * (y * z + x * w), 1 - 2 * (x * x + y * y)],
    ])


if __name__ == "__main__":
    main()
```

Listing 12: `tools/action/base_goto.py`

```
#!/usr/bin/env python3
"""Drive the mobile base to a world pose, closed-loop on odometry.

Usage: python3 tools/action/base_goto.py <x> <y> [yaw_rad] \
           [--tol 0.05] [--yaw-tol 0.1]
Streams velocity bursts, re-reads /odom between bursts, corrects.
Exits 0 at the target; exits 3 on stall, printing the LAST POSE as a
fact ("stalled at x y yaw"); why it stalled is yours to find out.
Absent base_twist in the manifest = this machine's base is fixed.
"""
import math
import sys
from pathlib import Path

import rclpy
import yaml
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry


def _pose(msg):
    p, q = msg.pose.pose.position, msg.pose.pose.orientation
    yaw = math.atan2(2 * (q.w * q.z + q.x * q.y),
                     1 - 2 * (q.y * q.y + q.z * q.z))
    return p.x, p.y, yaw


def _read_odom(node, odom, timeout=10.0):
    got = {}
    sub = node.create_subscription(Odometry, odom,
                                   lambda m: got.setdefault("m", m), 1)
    end = node.get_clock().now().nanoseconds / 1e9 + timeout
    while "m" not in got and node.get_clock().now().nanoseconds / 1e9 < end:
        rclpy.spin_once(node, timeout_sec=0.2)
    node.destroy_subscription(sub)
    if "m" not in got:
        raise SystemExit(f"no odometry on {odom}")
```

```
    return _pose(got["m"])


def main() -> None:
    args = [a for a in sys.argv[1:] if not a.startswith("--")]
    if len(args) not in (2, 3):
        raise SystemExit(__doc__)
    gx, gy = float(args[0]), float(args[1])
    gyaw = float(args[2]) if len(args) == 3 else None
    tol = float(sys.argv[sys.argv.index("--tol") + 1]) \
        if "--tol" in sys.argv else 0.05
    ytol = float(sys.argv[sys.argv.index("--yaw-tol") + 1]) \
        if "--yaw-tol" in sys.argv else 0.10

    root = Path(__file__).resolve().parents[2]
    m = yaml.safe_load((root / "machine.yaml").read_text())
    base = next((a for a in m["actuators"] if a["kind"] == "base_twist"), None)
    odom_entry = next((s for s in m["sensors"] if s["kind"] == "odometry"), None)
    if base is None or odom_entry is None:
        raise SystemExit("this machine has no mobile base (see machine.yaml)")

    rclpy.init()
    node = rclpy.create_node("base_goto")
    pub = node.create_publisher(Twist, base["port"], 10)
    latest = {}
    node.create_subscription(Odometry, odom_entry["port"],
                             lambda m: latest.update(p=_pose(m)), 10)
    end = node.get_clock().now().nanoseconds / 1e9 + 15
    while "p" not in latest and node.get_clock().now().nanoseconds / 1e9 < end:
        rclpy.spin_once(node, timeout_sec=0.2)
    if "p" not in latest:
        raise SystemExit(f"no odometry on {odom_entry['port']}")

    # Continuous stream, passive odometry: pauses reset the base's
    # velocity ramp, so publish steadily and let the same spin loop
    # deliver odometry updates on the side.
    no_progress = 0
    last_err = None
    for i in range(20000):
        x, y, yaw = latest["p"]
        dx, dy = gx - x, gy - y
        dist = math.hypot(dx, dy)
        dyaw = 0.0
        if gyaw is not None:
            dyaw = math.atan2(math.sin(gyaw - yaw), math.cos(gyaw - yaw))
        if dist < tol and abs(dyaw) < ytol:
            print(f"arrived at {x:.3f} {y:.3f} {yaw:.3f}")
            rclpy.shutdown()
            return
        bvx = math.cos(-yaw) * dx - math.sin(-yaw) * dy
        bvy = math.sin(-yaw) * dx + math.cos(-yaw) * dy
        n = max(dist, 1e-6)
        t = Twist()
        t.linear.x = bvx / n
        t.linear.y = bvy / n
        t.angular.z = max(-1.0, min(1.0, dyaw * 2))
        pub.publish(t)
        rclpy.spin_once(node, timeout_sec=0.03)
        if i % 100 == 99:  # progress check every ~100 ticks
            err = dist + abs(dyaw)
            if last_err is not None and err > last_err - 5e-3:
                no_progress += 1
                if no_progress >= 5:
                    print(f"stalled at {x:.3f} {y:.3f} {yaw:.3f} "
                          f"(remaining {dist:.3f} m, {dyaw:.2f} rad)")
                    rclpy.shutdown()
                    sys.exit(3)
            else:
                no_progress = 0
            last_err = err
    print("cycle budget exhausted before arrival")
    rclpy.shutdown()
    sys.exit(4)


if __name__ == "__main__":
    main()
```

C.3 Observation Acquisition and File Representations

The agent obtains observations through the interfaces described in `machine.yaml` and the workspace documentation. The acquisition methods below are alternatives the agent can select or combine according to its needs, rather than a prescribed sequence.

Available data and interface discovery. Camera images and depth are exposed as `sensor_msgs/msg/Image`, with camera intrinsics available through `sensor_msgs/msg/CameraInfo`. Robot state is available through `sensor_msgs/msg/JointState`, which associates joint names with positions, velocities, and efforts where provided. Depending on the machine, additional observations include wrist force and torque through `geometry_msgs/msg/WrenchStamped` and mobile-base odometry through `nav_msgs/msg/Odometry`. The machine description identifies the available sensors and ports; these modalities are not assumed to be present on every robot. The agent can inspect the running graph with `ros2 topic list`, examine a topic with `ros2 topic info`, and inspect its message definition with `ros2 interface show`. Small messages, such as joint states or camera intrinsics, can be read through `ros2 topic echo <topic> --once`, with the output optionally redirected to a workspace file.

Image and numerical files. The starter tool `tools/perception/cam_snap.py` subscribes to an image topic and saves one received frame. It accepts either a camera name or a full topic path and an optional output filename. For a color image, it converts the ROS message with `cv_bridge` and writes a PNG. For a depth image, it preserves the numerical array in a `.npy` file and writes a separate PNG visualization. The visualization rescales depth values for display; metric calculations use the original array and its documented units. The agent can inspect or modify this tool, or write an `rclpy` subscriber to save selected message fields in a format suited to subsequent processing. For joint data, values must remain associated with their joint names rather than be interpreted using an assumed message order.

Calibration, coordinate frames, and timing. Geometric calculations require the camera intrinsics and the relation between the camera frame and the desired reference frame. The agent can retrieve intrinsics from `CameraInfo` and query available frame transforms through `tf2`; the workspace documentation describes machine-specific alternatives when the required transform is unavailable. Saving RGB and depth separately does not itself guarantee temporal alignment: the starter snapshot tool captures a single topic and does not synchronize multiple streams. When a computation requires aligned observations, an agent-written acquisition program must account for message timestamps and the correspondence between image pixels, calibration, and coordinate frames. This is particularly relevant when either the robot or the observed scene moves between captures.

C.4 Visual Inspection and Programmatic Image Processing

Through its existing image-reading capability, the agent can inspect both camera images and views derived from sensor data by its own programs. The operations below illustrate ways it can prepare observations for visual inspection; the harness does not prescribe a fixed image-processing sequence.

Preparing a view for inspection. The agent can use ordinary image-processing libraries to crop a region of interest, enlarge it, or overlay grids and markers, then save and read the resulting image. In the Nut episode of Figure 3, its program extracts rows 130–234 and columns 335–444 from the overhead image, enlarges the crop by a factor of six, and draws a grid at intervals of ten original pixels. The grid labels retain the coordinates in the original image, allowing the agent to relate details in the enlarged view to the source pixels. The program saves this view as `nut_zoom.png`; Appendix H provides the code. These image operations expose existing visual detail and add reference marks; physical dimensions still require a geometric relation between image coordinates and the scene.

Checking action outcomes. Visual inspection also supports checking changes after an action. In Figure 3, the agent inspects an eye-in-hand image after grasping and compares the finger gap with its earlier measurement of the handle width. In the kettle episode of Figure 10, it crops the switch

before and after pressing and produces a difference image to inspect the change. Such views can be considered alongside numerical readings and terminal output when deciding whether to continue or revise an action.

Inspecting computed representations. Programs can also turn numerical results into images for the agent to inspect. For example, Figure 11 shows an agent reconstructing a dishwasher rack from depth, rendering its point cloud from a chosen viewpoint, and plotting cross-sections to examine the rail. Here, numerical processing constructs the representation and visual inspection helps the agent interpret it. The prevalence of these operations across episodes is reported separately in Appendix F.

C.5 Metric Measurements from Sensor Data

This subsection expands the numerical processing described in Section 4.3. The agent can combine measurements read from native interfaces with quantities computed from saved observations; the harness does not prescribe a task-specific measurement pipeline.

Direct readings and derived quantities. Joint positions, velocities, efforts, and available force/-torque readings provide numerical information about the robot and its interaction with the scene. Available frame transforms can be queried through `tf2`, for example with `ros2 run tf2_ros tf2_echo <reference> <source>`. These transforms describe the frames exposed by the robot's software; locating a task object without a corresponding tracked frame requires additional estimation from observations. The acquisition interfaces and file representations are described in Appendix C.3.

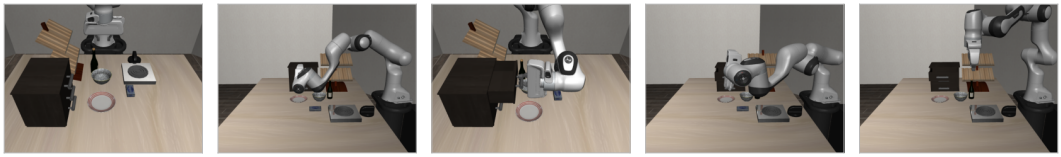
From depth pixels to physical coordinates. For a pixel (u, v) in a pinhole camera image, let z denote its depth along the camera's optical axis, in meters, and let K be the intrinsic matrix for that image. The corresponding camera-frame point and its reference-frame coordinates are

$$\mathbf{p}_c = zK^{-1}\begin{bmatrix} u & v & 1 \end{bmatrix}^\top, \qquad \mathbf{p}_r = R_{rc}\mathbf{p}_c + \mathbf{t}_{rc}, \tag{1}$$

where R_{rc} and $\mathbf{t}_{rc}$ transform coordinates from the camera optical frame to the chosen reference frame, such as the world or robot base frame. Here, u indexes columns and v indexes rows; the formula assumes pixels consistent with the pinhole calibration, with distortion accounted for where necessary. Depth values must use the documented scale and convention, and non-finite or non-positive values cannot be used as valid surface measurements. If target pixels are selected in a color image, their correspondence with depth pixels must be established before applying the calculation. Calibration, image timing, and frame transforms must refer to the observations being processed, as discussed in Appendix C.3. The agent can apply this calculation to selected pixels or a segmented region to estimate positions and extents.

The Nut example. In the episode shown in Figure 3(b), the agent loads `bird_depth.npy` and uses the overhead camera height of $2.0\,\mathrm{m}$ to compute world heights as $2.0 - z$. Its program selects points with world heights between 0.825 and $1.3\,\mathrm{m}$, forms connected components, and excludes components containing fewer than 50 pixels. It then converts the remaining pixels into world coordinates and reports each component's coordinate ranges and centroid. These thresholds and the simplified overhead-camera calculation belong to this episode and camera configuration, rather than to a fixed harness procedure. The code, including a second-camera calculation using an explicit rotation and translation, is reproduced in Appendix H.

Cross-checking and using measurements. Before moving, the agent repeats the geometric measurement from an oblique camera and compares the results in the same world frame. After grasping, it compares the measured finger gap of $30.35\,\mathrm{mm}$ with the previously estimated handle width of $30.4\,\mathrm{mm}$, alongside the eye-in-hand image in Figure 3(d). This comparison supplies evidence about the grasp rather than a universal success criterion. Likewise, force and effort readings can provide contact evidence, while accounting for their baseline and reference frame. How programs incorporate such readings during motion belongs to the control mechanism described in Section 4.4.



(a) survey the scene (b) before grasping the handle (c) after pulling (d) drawer open, reaching the box (e) box released in the drawer

Figure 7: The same episode as Figure 5 in the camera images the agent saved at its checkpoints: it took each of these to decide the next step or to verify the last one, and read them with its own vision.

C.6 Command Execution and Feedback Programs

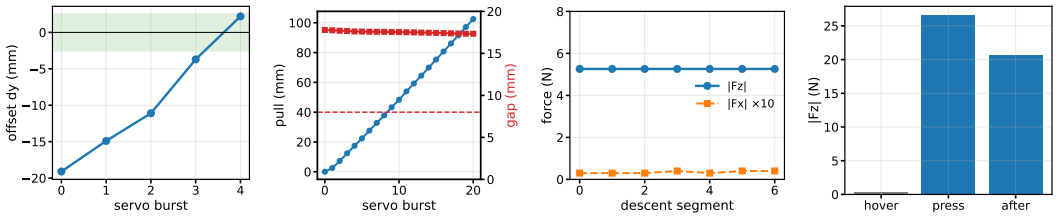
This subsection expands the two forms of interaction described in Section 4.4. Both can use the same native robot interfaces. The distinction is whether the agent inspects an outcome and chooses another action, or a running program makes the next adjustment from sensor feedback. A single task can combine both, and a program used for one step can itself contain a feedback loop.

Native control calls. The workspace documentation identifies the available action, service, and topic names, together with their message types and coordinate conventions. For joint motion, a client sends a `control_msgs/action/FollowJointTrajectory` goal containing joint names, target positions, and trajectory timing. A Cartesian target can first be converted to joint positions through an available inverse-kinematics service. Gripper requests use `control_msgs/action/GripperCommand`, while a velocity client publishes commands such as `geometry_msgs/msg/TwistStamped` in the documented reference frame. The agent can invoke action and service clients through `ros2cli` or construct them in `rclpy`; the appropriate robot controller executes the requested motion. In the drawer episode, the agent writes a reusable Python client for the IK service, trajectory and gripper actions, joint-state subscription, and velocity topic, then imports it into its action programs (Appendix H).

Checking a step and revising the next one. Terminal output can report request acceptance, execution results, and measurements printed by a program. The agent uses these together with physical observations when assessing progress. In the drawer episode, the first grasp closes on air: the program reports a finger gap of approximately 1.9 mm. The agent revises the approach target and runs a new program before attempting the pull; Figures 5 and 7 show its code and the corresponding visual checkpoints. The numerical evidence and stopping conditions are task-specific, rather than fixed success rules imposed by the harness.

Position feedback and guarded pulling. The revised approach program reads the current tool position, computes its offset from the grasp target, and applies proportional velocity corrections along the relevant axes. In this episode, it multiplies the errors by five, clips the two velocity components to 0.05 and 0.02 m/s, and exits the approach loop when the absolute error along the approach axis falls below 2.5 mm or the iteration limit is reached. Its velocity helper publishes bursts of 20 messages with a nominal rate parameter of 20 Hz; the state checks occur between bursts, so this parameter does not imply that the complete feedback loop runs at 20 Hz. The pulling program checks the finger gap and displacement, stopping if the gap falls below 8 mm, the pull target is reached, displacement after a burst is below 1 mm, or the iteration limit is exhausted. It prints the observed states and stop reason for the agent to inspect. Figure 8 plots the recorded behavior, and Appendix H gives the corresponding code.

Force feedback between trajectory segments. Feedback can also determine whether to continue a sequence of short position-controlled motions. In the CaP-Bench Restack example, the agent-written descent program records an initial force baseline, lowers the end effector in 2 mm increments, and reads the force after each segment. It stops when the norm of the force change exceeds 1.5 N or the specified lower position limit is reached, returning the final height and a contact flag. The threshold and step size belong to this program rather than to a harness-wide control policy. Figure 19 illustrates the descent, and Appendix H reproduces the program. Such feedback is processed directly by the running program; it need not be saved to an observation file or passed through the agent's



(a) approach the drawer handle (b) pull the drawer (c) insert the nut (d) wipe

Figure 8: Control loops the agent wrote, plotted from its own printed logs. (a)–(b) LIBERO-PRO, open the drawer and put the cream cheese inside: the agent approaches the handle with a proportional velocity command on the measured offset and stops within 2.5 mm (green band); it then pulls in bursts, reading the finger gap and the displacement after each, with two guards it wrote itself: abort if the gap collapses below 8 mm (dashed line, grip lost) or the displacement stalls. (c) CaP-Bench Nut: the insertion is executed in seven trajectory segments with the wrench read after each; the vertical load stays at the tool's weight and no lateral force appears. (d) CaP-Bench Wipe: the agent hovers, presses until the load confirms contact, then rasters 153 waypoints under position control and checks the load again.

context at each update. The agent can inspect the resulting logs and observations before selecting another action or revising the program.

D Compared Systems

D.1 Sources and Comparison Scope

All baseline results in Tables 1 and 4 are reproduced from the cited papers or their accompanying leaderboards; we do not rerun these systems. The comparisons therefore concern published system configurations, which can differ in model, reasoning budget, available tools, prior task experience, and evaluation protocol. They are not controlled ablations that isolate the effect of the interface alone. Models and reasoning-effort settings are listed alongside the results in the main tables.

Human-written programs. The CaP-Bench human reference row uses the S4 task-success values in Figure 17 of CaP-X (Fu et al., 2026): 93, 73, 100, 80, 47, 94, and 91 percent for Lift, Stack, Wipe, Nut, Restack, Two-Arm Lift, and Handover. Their arithmetic mean is 82.6%. This row represents the benchmark's expert-written programs under its low-level primitive interface, rather than a new human evaluation conducted for this paper.

Learned visuomotor policies. The OpenVLA, π_0, and $\pi_{0.5}$ results on the six LIBERO-PRO cells are reproduced from the comparison in CaP-X Table 2 (Fu et al., 2026). We leave the two LIBERO-10 cells unreported for these rows instead of combining results from a different evaluation source. The two GR00T N1.5 rows use the pretraining-only and pretraining-plus-full-target-data configurations in Table 2 of RoboCasa365 (Nasiriany et al., 2026). The latter includes demonstrations of the target Composite-Unseen tasks; the split name describes their absence from pretraining, not their absence from this post-training stage. We do not use the results from that paper's Table 1, which evaluates a different kitchen split. The RLDX-1 row is the direct frozen-policy baseline reported in Harness VLA Table 4 (Zhang et al., 2026b).

Agents that orchestrate learned policies. Pigey's six-cell LIBERO-PRO row comes from its Table 2, using a frozen $\pi_{0.5}$ backend and ten trials per task (Galanti et al., 2026). Its reported zero-shot protocol does not use task-specific memory or reference-seed exploration. Harness VLA's LIBERO-PRO and RoboCasa365 results come from Tables 3 and 4, respectively (Zhang et al., 2026b). It constructs task-specific memory through reference-seed exploration before evaluating on held-out seeds. On LIBERO-PRO, it evaluates ten held-out seeds for each task in all eight cells; on RoboCasa365, it uses ten for atomic tasks and five for each composite group. These counts and the reference-seed preparation differ from our fresh-session evaluation described in Appendices B.1 and B.2.

Agents that construct the policy. The CaP-X row denotes CaP-Agent0: its CaP-Bench values are the 3M configuration in Table 5, and its LIBERO-PRO values are from Table 2 (Fu et al., 2026). The RHO rows use its CaP-Bench S4 results in Table 1 and LIBERO-PRO results in Tables 6–8 (Elmaaroufi et al., 2026). RHO develops its repository on the first ten CaP-Bench trial identifiers per task and reports evaluation over all 100; for LIBERO-PRO, it develops on the unperturbed LIBERO tasks and evaluates on the perturbation suites. ASPIRE's rows come from Tables 2 and 3 (Lu et al., 2026). It develops a skill library before evaluation; on CaP-Bench it uses seeds 101–125 for development and evaluates generated programs on seeds 1–100. The RATS row is the full execution system in its Table 1 after 50 play iterations, evaluated on 60 LIBERO-PRO task variants with ten initializations each (Zhang et al., 2026a). We use its reported 43.8% full-system result rather than its separate plug-in configuration, and retain CaP-X's original baseline row rather than the RATS paper's rerun of that baseline.

D.2 Training, Experience, and Oracle Access

The table annotations distinguish three aspects of each reported configuration. *Training* indicates whether a robot policy is trained on demonstrations. *Experience* records additional preparation on the evaluation tasks or their underlying environments before the reported test episodes, such as rehearsed programs, accumulated skills, task-specific memory, or target-task adaptation. *Oracle* indicates whether the benchmark's success predicate is accessible during execution; access to rewards during an earlier development phase is a separate issue. These annotations describe the reported configurations and are not mutually exclusive. For OpenRUA, task-specific programs and notes are produced only within the current episode and are not prior experience under this definition.

Trial counts, initial states, stopping rules, and models differ across sources. For example, RHO reports 50 initial states per task on the six LIBERO-PRO cells, whereas our evaluation uses ten. Pigey scores simulation trials using the LIBERO predicate and its own completion protocol, including failure when the agent times out without declaring completion (Galanti et al., 2026); our scoring rule is specified in Appendix B.3. Accordingly, we retain source-specific evaluation settings rather than claim that every baseline shares our execution budget or termination rule.

D.3 Detailed Analysis of Main Results

The LIBERO-PRO percentages in this subsection refer to the six Object, Spatial, and Goal perturbation conditions.

Robot-use agents are zero-shot visuomotor policies. The programs written by human experts solve 82.6% of CaP-Bench, but each is written by hand for one task and does not carry over to the next. The learned visuomotor policies (*e.g.*, OpenVLA, π_0, and $\pi_{0.5}$) take the expert out of the loop and map camera images to motor commands directly, yet they are bound to their demonstrations: trained on these tasks, they stay at or below 12.8% on LIBERO-PRO once the scenes are perturbed. The prior agents act as the policy themselves, but rely on hand-crafted primitives and on experience: CaP-X, which writes programs over its own primitives, reaches 18.2% on LIBERO-PRO, and RHO, ASPIRE, and RATS reach 43.8–71.7% by accumulating policy repositories or skills over the provided primitives before evaluation (Elmaaroufi et al., 2026; Lu et al., 2026; Zhang et al., 2026a). Combining the two, Pigey places an LLM agent over a frozen $\pi_{0.5}$, so that the agent supplies the reasoning and the trained policy supplies the motor skill, and reaches 53.3%; Harness VLA does the same with a coding agent and reaches 87.7%, but is few-shot, since it builds task-specific memory through reference-seed exploration before testing, and reads the benchmark's success predicate during execution. OpenRUA lets the coding agent act as the policy directly: it perceives the scene through the robot's camera streams and acts through its motor interface, without relying on any trained policy, primitive library, or prior experience, and reaches 99.0% on CaP-Bench and 91.8% on LIBERO-PRO, the highest in both tables. A robot-use agent is thus a visuomotor policy on its own, and a zero-shot one.

D.4 Native Interfaces and the Limits of Accumulated Skills

Experience improves performance but leaves difficult tasks unresolved. The low Nut success rates are not results from agents without prior skill development. CaP-Agent0 uses nine synthe-

sized utilities distilled from successful CaP-Bench rollouts (Fu et al., 2026, Section 4 and Appendix H.1). RHO evolves multi-file policy repositories through repeated execution and reflective optimization (Elmaaroufi et al., 2026, Section 3), while ASPIRE accumulates validated repairs in a skill library and searches over candidate programs (Lu et al., 2026, Section 2). Despite this experience accumulation, their reported Nut success rates are 0%, 1%, and 9%, respectively, compared with 100% for OpenRUA. On the LIBERO-PRO Spatial suite, the corresponding position/-task results are 12%/14% for CaP-X, 53.0%/61.6% for RHO, and 51%/60% for ASPIRE, versus 98%/100% for OpenRUA. Experience improves many tasks, but the reported optimization and skill accumulation do not close these gaps.

The primitive interface constrains what experience can encode. ASPIRE explicitly identifies this limitation: programs remain bounded by the provided perception, planning, and control API; capabilities outside that API require inefficient approximation or a human extension of the interface (Lu et al., 2026, Section 5). Accumulating strategies over an interface does not by itself expose an unavailable sensor or control operation. CaP-X's documented S3/S4 simulation interface provides image observations, segmentation, grasp proposals, inverse kinematics, and joint and gripper moves, but does not list a force-reading or force-conditioned motion primitive (Fu et al., 2026, Appendix G). This restriction concerns the evaluated simulation interface; CaP-X separately describes force-wrench access for its real-robot setup. CaP-X also reports difficulties with insertion requiring continuous feedback, and attributes LIBERO-PRO failures to perception errors, angled grasp proposals, and the lack of collision-aware motion planning (Fu et al., 2026, Appendices A and J). Thus, the comparison concerns the capabilities available for programs to use as well as the experience accumulated in composing them.

Native feedback and agent-written control. OpenRUA exposes sensor and control interfaces through ROS 2, allowing the agent to write programs that monitor physical feedback during execution. The Nut example in Figure 8(c) executes insertion in seven trajectory segments with a force reading after each; Section 4.4 reports the broader use of agent-written feedback programs and motion clients. This provides a concrete mechanism for using sensing during manipulation without a pre-specified task primitive or a learned robot policy. Harness VLA takes a different route, composing analytic primitives with a frozen VLA for contact-rich execution; its strongest Spatial result in Table 1 is 94% (Zhang et al., 2026b). The consequences of relying on a learned policy under distribution shift are examined in Appendix D.5.

D.5 Generalization and Demonstration Coverage

Table 4 reports success rates on the 50 target tasks of RoboCasa365, which the benchmark splits into 18 atomic tasks, 16 composite tasks whose templates appear in its pretraining set (Seen), and 16 composite tasks whose templates are held out of it (Unseen); protocol details are in Appendix B.

OpenRUA maintains performance on unseen tasks, demonstrating generalization without relying on a learned robot policy. OpenRUA achieves comparable success rates on Seen and Unseen composite tasks, at 26.3% and 28.1%, respectively. In contrast, RLDX-1 drops from 21.3% to 5.0%, while Harness VLA, which orchestrates the RLDX-1 with a coding agent, drops from 61.0% to 13.8%. These results suggest that reliance on a learned robot policy can limit generalization beyond its training distribution, whereas OpenRUA handles both task groups by synthesizing and refining control programs during execution rather than relying on motor skills learned from robot demonstrations.

The effect of target-task demonstrations. The two GR00T N1.5 rows in Table 4 distinguish pretraining alone from pretraining followed by post-training on the target-task demonstrations (Nasiriany et al., 2026, Table 2). On the same 16 Composite-Unseen tasks, success rises from 0.2% to 42.1% after post-training on their demonstrations. This contrast shows the importance of demonstration coverage for the evaluated learned policy. The term *Unseen* refers to task templates absent from the benchmark's pretraining set; these templates are included in the target data used by the post-trained GR00T configuration. Its 42.1% therefore represents performance with target-task demonstration training, whereas the 28.1% of OpenRUA is obtained without such training.

Table 4: **OpenRUA generalizes to unseen composite tasks in RoboCasa365.** Success rates (%) are reported for atomic tasks and composite tasks whose templates are included in (Comp.-Seen) or held out from (Comp.-Unseen) the benchmark's pretraining data. Detailed evaluation protocols are provided in Appendix B.

Method		Setting					Success Rate (%)			
		Agent	Model (Effort)	Train.	Exp.	Oracle	Atomic-Seen	Comp.-Seen	Comp.-Unseen	Overall
Learned Policy	GR00T N1.5	–	–	✓	✗	✗	41.9	0.0	0.2	15.1
		–	–	✓	✓	✗	68.5	40.6	42.1	51.1
	RLDX-1	–	–	✓	✗	✗	60.0	21.3	5.0	30.0
Agent + Policy	Harness VLA	Codex	GPT-5.5 (xhigh)	✓	✓	✓	92.0	61.0	13.8	57.1
		CC	Opus 4.7 (max)	✓	✓	✓	79.4	47.5	15.0	48.6
Agent	OpenRUA	CC	Opus 5	✗	✗	✗	61.1	26.3	28.1	39.4

Adding an agent over a learned policy. RLDX-1 achieves 21.3% on Composite-Seen tasks and 5.0% on Composite-Unseen tasks in the frozen-policy evaluation reported by Harness VLA (Zhang et al., 2026b, Table 4). Harness VLA uses this same policy as its contact-rich execution primitive, together with analytic primitives, a coding agent, task-specific and global memory, and access to the benchmark's success predicate. With Codex, it reaches 61.0% on Seen tasks and 13.8% on Unseen tasks; with Claude Code, the corresponding rates are 47.5% and 15.0%. Both configurations improve on the underlying policy, including on Unseen tasks, but retain a pronounced Seen–Unseen performance gap. The comparison therefore shows that adding agent-level planning, memory, and verification does not remove this gap in the evaluated configurations: contact-rich execution still relies on the frozen robot policy. In OpenRUA, the coding agent instead constructs the control program during the episode, using current observations and the robot's native sensing and control interfaces. The same workflow yields 26.3% on Seen and 28.1% on Unseen composite tasks, without a learned robot policy or task-specific rehearsal.

Generalization and composite-task difficulty. Stable performance across Seen and Unseen templates does not imply that composite tasks are easy. The success rate of OpenRUA falls from 61.1% on Atomic-Seen tasks to 26.3% and 28.1% on the two composite groups; every compared system also performs worse on composite tasks than on atomic tasks. This atomic-to-composite comparison concerns the additional demands of composing and executing multiple steps, whereas the Seen–Unseen comparison concerns task-template coverage within the composite category. Appendix G examines the remaining difficulties, including incomplete long-horizon execution, physical constraints, mismatches between instructions and scoring criteria, and agent errors. Thus, the generalization result is that the absence of a task template from robot pretraining data is not accompanied by lower success for OpenRUA in this evaluation, even though composite manipulation remains challenging.

E Scaling Details

Runs behind Table 5. All runs use the same harness, workspace, and prompt on the two LIBERO-10 perturbation cells (10 tasks $\times$ 10 initial states each). Claude Code with Opus 5 and Fable 5.1 ran the full 200 episodes once at high reasoning effort, Claude Code's default. Claude Code with Sonnet 5 ran the full 200 episodes at high effort. Codex with GPT-6 Astra ran the full 200 episodes twice, once at medium reasoning effort and once at high; the two runs are reported as two rows. The turn and cost profiles use the episodes with tool-call timestamps: all 200 for GPT-6 Astra at high effort, 192 for medium (the first eight episodes of that run were made before rollout logging was enabled).

Gains from test-time computation. The curves in Figure 6 are computed retrospectively from the first successful step of each recorded episode: an episode contributes to the success rate at a given budget if success occurred within that budget. The agent-turn and output-token curves therefore describe when the recorded runs succeed, rather than separate runs in which the agent is instructed to work under each plotted budget. For Opus 5, the share solved within 100, 150, and 200 turns is 50%, 66%, and 70%, respectively; for Fable 5.1, it is 61%, 71%, and 72%. The reasoning-effort

Table 5: Behavior profile of OpenRUA with different models on LIBERO-10 (both perturbation cells, 200 trials per model). Turns and tokens per turn are **medians** over episodes: turns are model responses per episode, in parentheses for successful / failed episodes; tokens per turn are output tokens including thinking. Cost is the **mean** per episode at API list prices, since a batch of episodes costs its mean times its size; cost per success is the total cost divided by the number of successful episodes. Distributions are in Appendix E.

Agent	Model (Effort)	Success (%)	Turns, median (succ. / fail.)	Tokens / turn, median	Cost ($), mean	Cost per success ($)
CC	Sonnet 5 (high)	24.0	301 (219 / 334)	512	22.4	93.3
CC	Opus 5 (high)	72.5	103 (97 / 131)	1,058	13.7	18.9
CC	Fable 5.1 (high)	75.0	73 (72 / 84)	760	8.4	11.2
Codex	GPT-6 Astra (medium)	62.5	98 (103 / 89)	90	11.8	19.1
Codex	GPT-6 Astra (high)	69.0	93 (97 / 86)	122	10.0	14.4

comparison uses separate 200-episode runs with the same Codex agent and GPT-6 Astra model: increasing effort from medium to high raises success from 62.5% to 69.0%. These improvements occur without changing the initial workspace, adding task-specific primitives, or training a robot policy. The output-token curves provide a complementary view of the computation consumed before success; the estimation of Claude Code token positions is described below.

Diminishing returns and model differences. The Opus 5 and Fable 5.1 curves flatten within about 300 turns, whereas Sonnet 5 continues to gain successes up to about 400 turns and finishes at 24.0%. Within 100 turns, Fable 5.1 solves 61% of episodes, Opus 5 solves 50%, and Sonnet 5 solves 1%. These differences show that models vary both in final success and in the interaction needed to reach it. Failed Claude Code episodes are longer than successful ones: median turns are 131 versus 97 for Opus 5, 84 versus 72 for Fable 5.1, and 334 versus 219 for Sonnet 5. For Sonnet 5, 22% of episodes reach the 500-turn limit, and none of those episodes succeeds. Together, the curves and episode lengths show diminishing returns within the evaluated budgets, with substantial differences remaining between models. They do not establish that extending a failed episode beyond the evaluated limit could never lead to success.

Model performance and execution cost. As shown in Table 5, success rises from 24.0% with Sonnet 5 to 72.5% with Opus 5 and 75.0% with Fable 5.1, while median turns fall from 301 to 103 and 73. At API list prices, the corresponding cost per success falls from $93.3 to $18.9 and $11.2, even though Sonnet 5 has the lowest per-token prices among these models. Thus, lower token prices do not necessarily translate into lower cost per successful task: episode length and success rate also matter. For GPT-6 Astra, increasing reasoning effort raises median output tokens per turn from 90 to 122, but reduces mean episode cost from $11.8 to $10.0 and cost per success from $19.1 to $14.4. A larger output per turn therefore need not increase the cost of completing a task; total cost also includes input tokens and cache usage, as detailed below.

Compatibility across coding agents. Claude Code and Codex use the same workspace, documentation, and native ROS 2 interface without agent-specific changes to the harness. Their interaction profiles nevertheless differ: Codex with GPT-6 Astra produces a median of 90 or 122 output tokens per turn at medium or high effort, compared with 760 for Fable 5.1 and 1,058 for Opus 5. At high effort, failed Codex episodes are shorter than successful episodes, with medians of 86 and 97 turns, respectively, unlike the Claude Code runs. These profiles show that compatibility with the native interface does not require agents to follow the same interaction pattern. The comparison varies both the coding agent and the model, so the observed differences characterize the evaluated combinations rather than isolating either factor.

Per-cell success. LIBERO-10 swap cells 0–9 / task cells 0–9, solved out of 10: Sonnet 5: 8, 7, 0, 1, 0, 2, 1, 8, 0, 0 / 8, 3, 0, 0, 0, 3, 0, 4, 3, 0; Opus 5: 9, 9, 10, 10, 9, 10, 8, 9, 6, 0 / 9, 8, 10, 5, 8, 0, 6, 10, 0, 9; Fable 5.1: 9, 9, 10, 6, 9, 10, 3, 10, 9, 5 / 10, 8, 10, 5, 9, 7, 7, 7, 0, 7; GPT-6 Astra, medium: 10, 9, 10, 10, 5, 3, 0, 9, 8, 2 / 10, 8, 10, 9, 2, 4, 0, 9, 0, 7; GPT-6 Astra, high: 10, 9, 10, 9, 10, 1, 0, 9,

9, 8 / 10, 9, 10, 7, 5, 2, 1, 9, 0, 10. Across the five evaluated configurations, five of the twenty cells have a spread of at most three successful episodes. Comparing Opus 5 with GPT-6 Astra at high reasoning effort, the absolute difference exceeds three successful episodes in four cells (swap 5, 6, 9 and task 6).

Token budget (Figure 6b). The Codex rollout log records the running output-token total after every model response, so the token at which the predicate first held is read directly. Claude Code's transcript reports only the episode total; the split over turns is estimated from the thinking-token estimates the CLI streams between messages plus the visible text and tool inputs (at four characters per token), and the cumulative series is scaled so that the episode ends at the reported total. The endpoint of every episode is therefore exact; the shape of the Claude curves within an episode is an estimate.

Cost accounting. Costs are computed from the token counts each CLI reports, at API list prices on 2026-09-20 (USD per million tokens): Opus 5 input 5, cache read 0.5, one-hour cache write 10, output 25; Fable 5.1 input 10, cache read 0.25, cache write 20, output 50; Sonnet 5 input 2, cache read 0.2, cache write 4, output 10; GPT-6 Astra input 10, cached input 1, output 50. The runs themselves were billed on subscriptions; list prices are used only for comparability.

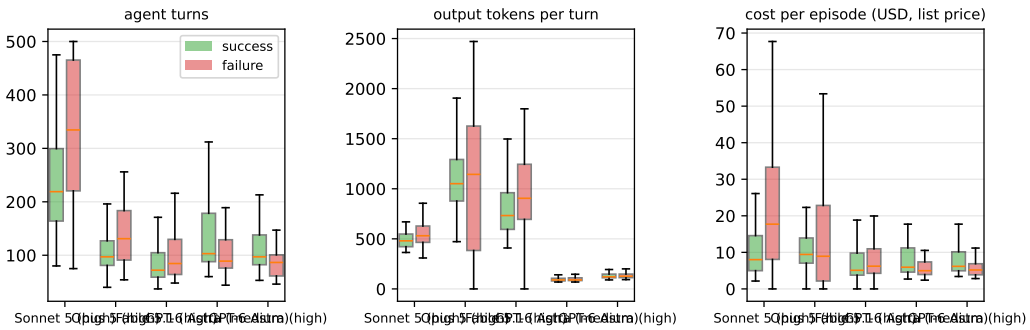


Figure 9: Distributions behind Table 5 on LIBERO-10: agent turns, output tokens per turn, and cost per episode at API list prices, for successful and failed episodes of each model. Boxes span the interquartile range; whiskers extend to the most extreme values within 1.5 times that range.

F Agent Behavior Statistics

Episode-level prevalence of agent-built perception. The two columns in Table 2 count distinct behaviors, each at most once per episode. For the image-processing column, an episode is counted when the agent reads an output image attributed to its own code for cropping, enlargement, annotation, masking, depth rendering, or combining sensor images. A metric-program episode contains an invocation of agent-written code matching depth or array processing together with camera-intrinsic or coordinate-transform operations; invoking the supplied `px2world.py` alone does not qualify. Across CaP-Bench and LIBERO-PRO, 1,090 of 1,500 episodes contain the former behavior and 1,452 contain the latter; both occur in 1,050 episodes (70.00%). These measures use source-pattern and execution-trace matching, which can miss indirect implementations and do not establish that every computed measurement is correct.

Perception activity and use of computed coordinates. Table 6 provides complementary statistics across all three benchmarks: image reads and metric-program invocations per episode, the fraction of image reads attributed to views computed from sensor data by agent-written programs, and the fraction of Cartesian motion targets traced to computed coordinates. These measures describe activity and use within episodes, whereas Table 2 reports the prevalence of the two behaviors across episodes.

The agent looks at images to recognize and verify, and computes to measure. Of the motion commands that carry a Cartesian target, 90–94% use coordinates that an agent-written metric pro-

Table 6: How the agent perceives, per benchmark (Claude Code with Opus 5, all mainline episodes). Images read = `Read` calls on image files per episode (median); of these, the share processed from sensor data by agent-written programs (crops, zooms, grids, overlays) rather than saved directly from a camera. Metric programs = runs of agent-written programs that compute geometry from depth, intrinsics or transforms (median per episode). Targets traced = share of motion commands with a Cartesian target whose coordinates appear in the output of an earlier metric program (within 1.5 mm) or are computed inside the motion program itself.

Benchmark	Images read	Processed images	Metric programs	Targets traced to computed coordinates
CaP-Bench	14	16%	13	94%
LIBERO-PRO	21	36%	21	93%
RoboCasa365	31	40%	35	90%

gram had computed from depth, intrinsics, or transforms, or that the motion program computes itself; the remainder are observation and parking poses set by hand. The agent does not estimate a position from a picture and move there. What it reads images for is recognition and checking, and it prepares them for that purpose: 16–40% of the images it reads are views processed from sensor data by its own programs, mostly crops and enlargements of the object of interest (68–94% of them) and, in 3–11%, views with a pixel grid or markers drawn on so that it can read positions off the image. In the episode of Figure 3, the agent recognizes the parts in the overhead image, segments them by height in the depth array, measures the handle on a gridded crop, cross-checks the coordinates from a second camera, and verifies the grasp with the eye-in-hand view and the finger gap. The depth-processing program from this episode is shown in Figure 4. Read purposes, the techniques the pipelines use, and the use of the starter tools are in Appendix F; the programs behind these pipelines are listed in Appendix H.

What the agent reads images for. Table 7 splits the image reads of Table 6 by the event that precedes them: before the first motion command (understanding the scene), after a motion command with no metric computation in between (checking its outcome), or after a metric computation (inspecting an intermediate result such as a crop or a segmentation). It also groups the processed images by the operations applied to their sensor inputs; a program that both crops and draws a grid counts under both, so the shares do not sum to one.

Table 7: Image reads by purpose (share of all reads) and processed images by kind (share of reads of processed images; multi-label).

	Purpose			Processed images				
Benchmark	Scene	Verify	Inspect	Crop / zoom	Grid / markers	Depth view	Mosaic	Mask overlay
CaP-Bench	32%	37%	31%	80%	3%	3%	1%	2%
LIBERO-PRO	38%	35%	27%	94%	11%	4%	2%	2%
RoboCasa365	18%	47%	36%	68%	10%	5%	6%	2%

What the perception pipelines are built from. Table 8 lists the techniques found in agent-written perception code, as the share of episodes in which each appears. Figures 10 and 11 show two more pipelines the agent built for itself: frame differencing to check whether a switch moved, and a point cloud reconstructed from depth and rendered from a chosen viewpoint, with cross-sections to locate a rail.

Episode-level prevalence of agent-written control tools. Table 3 covers all 700 CaP-Bench and 800 LIBERO-PRO episodes, including successes and failures. The motion-client column counts episodes whose agent-written workspace files contain native trajectory, gripper, velocity, or inverse-kinematics client code, excluding supplied tools and documentation. These files contain `rclpy` and client or publisher construction; the count establishes the presence of client source, not its successful execution. The feedback-program column counts episodes that invoke agent-written motion programs in which sensor-derived values affect a motion argument, loop condition, or branch; read-

Table 8: Techniques in agent-written perception code (share of episodes in which the technique appears).

Technique	CaP-Bench	LIBERO-PRO	RoboCasa365
Depth threshold / height band	97%	99%	98%
Centroid / bounding box	100%	96%	94%
Point cloud / plane fit	83%	69%	88%
Connected components / contours	68%	69%	56%
Color threshold	48%	24%	41%
Crop / zoom for its own reading	60%	99%	98%
Frames stacked for comparison	59%	34%	70%
Cross-section / slice plots	18%	35%	50%
Frame differencing	18%	14%	24%
Multi-camera cross-check	33%	16%	23%
Point cloud rendered to an image	6%	6%	15%

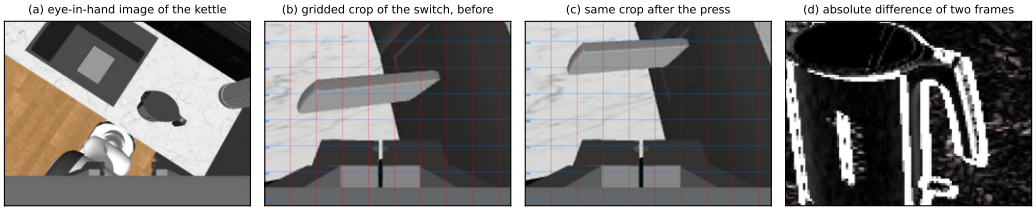


Figure 10: Checking a state change by frame differencing (RoboCasa365, turn on the electric kettle). (a) The agent reads the eye-in-hand image to find the kettle and its switch. (b) It crops the switch and draws a pixel grid to read the lever's position. (c) After pressing, it produces the same crop again. (d) It subtracts the two frames; the lever and its shadow light up where they moved, which is how the agent decides whether the press took effect.

ing a sensor alone does not qualify. The respective counts are 679 and 421 on CaP-Bench, and 759 and 331 on LIBERO-PRO, giving pooled rates of 1,438/1,500 (95.87%) and 752/1,500 (50.13%). The classifications use source and execution-trace patterns; indirect implementations can be missed, and feedback programs need not run high-frequency velocity servos.

Manipulation activity and control modes. Table 9 retains the finer task-group breakdown and RoboCasa365 results, distinguishing motion-command frequency, the share of commands issued through agent-written programs, and episode-level use of feedback, force readings, and velocity control.

The agent puts the sensor in the loop when the task needs it, and steps through the task when it does not. On the contact-rich CaP-Bench tasks, 99% of motion commands run a program the agent wrote, two thirds of the episodes contain a program in which a sensor reading decides the next motion, and 86% read the force sensor; on the LIBERO-PRO pick-and-place cells, a third of the motion commands go through the one-shot starter tools, a feedback program appears in 37% of the episodes, and the rest is done step by step, one command and one check at a time. The loops it writes are rarely velocity servos (3–12% of episodes); the usual form is a sequence of short trajectory segments with a sensor read between them, which is what Figure 8 shows: a proportional approach to a drawer handle, a pull guarded by the finger gap, a nut inserted in seven monitored segments, and a wipe that presses until the load confirms contact. The control layer behind these loops is the agent's own: 95–97% of episodes write their own trajectory, gripper, and inverse-kinematics clients, and the starter motion scripts are read as reference more often than they are run (Appendix F); the client library and the control loops of Figure 5 are provided in Appendix H.

Starter tools versus agent-written tools. The workspace ships six small scripts (Section 3); Table 10 reports how often each was executed. Only the camera snapshot is used in every episode, because the `ros2` command line cannot save an image topic to a file; the motion scripts are read

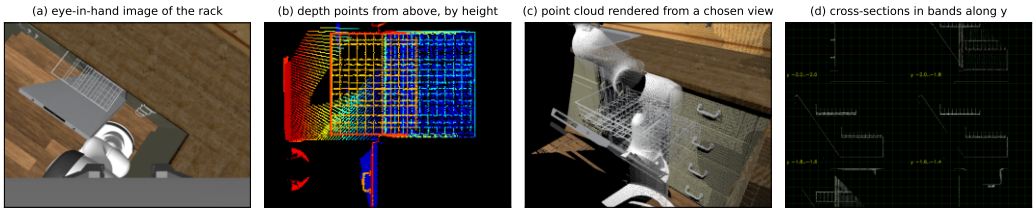


Figure 11: Reconstructing an articulated object from depth (RoboCasa365, slide the dishwasher rack). (a) The eye-in-hand image shows the rack and its rail. (b) The agent projects the depth points onto the counter plane, colored by height. (c) It renders the reconstructed point cloud from a viewpoint it chose, to see the rack's structure. (d) It cuts the cloud into bands along one axis and plots each cross-section, to locate the rail it has to push along.

Table 9: How the agent manipulates (Claude Code with Opus 5, all mainline episodes). Motion commands = commands that move the arm, gripper, or base (median per episode); of these, the share issued by running a program the agent wrote rather than a starter tool. Feedback program = an agent-written program in which a sensor value read inside the program decides the next motion, a loop condition, or a branch; force = the program reads the wrench topic; velocity servo = the feedback loop streams Cartesian velocities. The final three columns report shares of episodes; Through own programs reports the share of motion commands.

Tasks	Motion commands per episode	Through own programs	Feedback program	Reads force	Velocity servo
CaP-Bench, contact-rich (Nut, Wipe, Restack, 2A-Lift, Handover)	14	99%	67%	86%	4%
CaP-Bench, Lift and Stack	9	86%	44%	97%	6%
LIBERO-PRO, six perturbation cells	20	68%	37%	64%	3%
LIBERO-10	26	84%	54%	44%	3%
RoboCasa365	50	86%	71%	63%	12%

more often than they are run, and the agent writes its own clients instead: 96–99% of episodes contain agent-written sensor subscriptions and 95–97% agent-written motion clients (median 6–14 program files per episode, most organized as importable modules).

Table 10: Starter tools: share of episodes that executed each tool (mean calls per episode in parentheses) and share that only read its source.

Tool	Purpose	CaP-Bench	LIBERO-PRO	RoboCasa365
cam_snap	save an image topic to a file	100% (7.9)	100% (12.5)	100% (17.8)
px2world	back-project a pixel with depth	29% (0.6), read 24%	75% (1.9), read 10%	57% (1.6), read 24%
fjt_send	send one joint trajectory	8% (0.1), read 33%	63% (2.8), read 13%	32% (2.0), read 30%
gripper_cmd	open / close the gripper	9% (0.2), read 25%	56% (1.8), read 17%	22% (1.5), read 29%
ik_move	IK to a pose and move	3% (0.1), read 41%	41% (1.2), read 36%	8% (0.6), read 50%
base_goto	drive the base to a pose	–	–	62% (3.0), read 11%

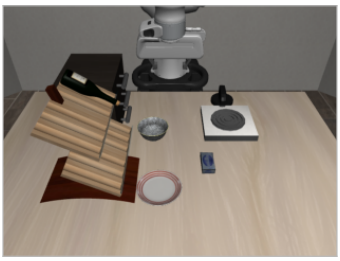
G Failure Study

We read every failed episode of OpenRUA with Opus 5 in Tables 1 and 4, 413 in all, and assigned each to the stage at which its decisive error lies (Table 11; the procedure and the class definitions are given below, and Table 12 lists the sub-causes).

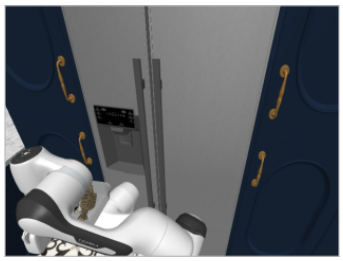
What counts as done, and how the machine behaves, are in the demonstrations rather than in the instruction. Classes A, B and D together hold 238 of the 413 failures (58%). In class A the agent did what the instruction says, measured the result, and announced its own reading of the task; the benchmark predicate needed a quantity the instruction never mentions. Told to open a drawer, it pulled 62 to 136 mm and stopped before the bottle standing in the drawer's path, where

Table 11: Where the failed episodes fail. Every failed trial of OpenRUA (Claude Code, Opus 5) in Tables 1 and 4 was read and assigned the stage at which the decisive error lies; counts are episodes; the classes are defined below and Table 12 lists the sub-causes.

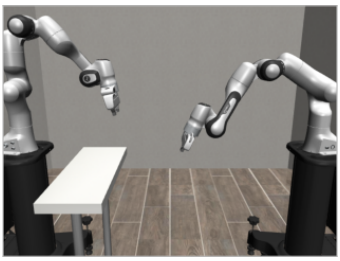
Cause of failure	LIBERO-PRO	CaP-Bench	RoboCasa365	All
A Success criterion not stated in the instruction	42	1	36	79
B Predicate contradicts the instruction	17	0	2	19
C Object or force beyond the machine's limits	12	0	63	75
D Budget spent on machine behaviour not stated anywhere	15	0	125	140
E Perception, control, or waiting error of the agent	18	6	76	100
Failed episodes	104	7	302	413



(a) A: "on the rack", laid with the base uphill; the predicate reads the base at the foot of the rack



(b) D: back with the tray, the freezer door has closed itself again



(c) E: arm 1 commanded to 0.008 m on a 0.030 m handle; hammer on the floor

Figure 12: One failed episode per benchmark, in the images the agent saved: LIBERO wine rack, RoboCasa365 freezer trays, CaP-Bench handover. Figures 14 to 16 show the full checkpoint sequence of each and Figure 13 a fourth case for class B.

the predicate needs the slide past 140 mm (14 seeds); told to put a bottle on a rack, it laid the bottle along the slope with its base uphill because a bottle across the slope would roll, where the predicate reads the base against a 22 mm band at the foot of the rack (seven seeds, Figure 12(a)); told to scrub a board briefly, it scrubbed until the force trace showed contact over 3 to 10 cm, where the predicate needs a 10 cm sweep (nine seeds). Class B is the same gap taken to the limit: in 19 episodes the predicate contradicts the instruction, scoring the pot the BDDL file itself places in its right init region when the instruction says left, or a 55 mm caddy slot for a 100 mm mug. Class D is the same gap on the machine side: the base has no brake and is pushed 0.15 to 0.39 m whenever the arm meets a door or a counter, freezer doors close on their own while the robot fetches the next tray (Figure 12(b)), and a faucet lever carries two hinges of which one is scored; nothing in the instruction or in the interface says so, the agent found each of these by running into it, and then spent the budget writing a station-keeper or reopening a door. A policy trained on demonstrations inherits the criterion: in LIBERO's own demonstrations the drawer ends at a median 156 mm of its 160 mm travel, the plate ends inside the region box, and the bottle's base rests at the foot of the rack (see the procedure below). A zero-shot agent receives one sentence, and has to guess the criterion and rediscover the machine in every episode. Benchmarks for instruction-following agents should state the criterion in the instruction.

The physical losses trace to the gripper and the base rather than to the agent's control. Class C holds 75 failures (18%). On LIBERO the twelve are objects at the edge of the gripper's envelope: a 106 mm bowl and 103 mm mug rims against a 72 to 80 mm opening, and plate rims that a two-finger pinch extrudes on lift; the same tasks succeed in their other seeds, so the limit is a margin, not a wall. On RoboCasa365 the 63 are the binary gripper and the mobile base: the gripper keeps closing on a soft or thin object and squeezes it out during a carry (the finger gap on a straw decays from 14 to 1 mm over one base drive), contact forces above the base's 250 N friction move the robot and lift its torso by up to 0.34 m, and an object that reaches the floor is lost for good because the arm cannot reach below about $z = 0.3$ m.

The agent's own errors sit at the last centimetres of control, and rarely in perception. Class E holds 100 failures (24%). Control errors account for 46: the handover agent commanded 0.008 m per finger on a 0.030 m handle it had measured, against the documentation's warning, and squeezed the hammer out the instant the other arm let go (Figure 12(c)); elsewhere an IK branch jump drove the hand below the table, a release ran before the arm had arrived, or a recovery sweep knocked over the object already placed. Waiting accounts for 27: turns or wall clock spent polling background jobs, and turns that ended while a monitor was still armed. Perception accounts for 27, three on LIBERO-PRO and 24 on RoboCasa365; most of the latter are a target outside the parked robot's view that was never searched for, or a joint the sensors do not expose, such as the scored hinge of a faucet, so the measurement side of Section 4.3 holds in the failures as well.

Procedure. The study covers every failed episode of OpenRUA with Claude Code and Opus 5 in the main runs: 104 on LIBERO-PRO, 7 on CaP-Bench and 303 on RoboCasa365. For each episode we read the transcript and the agent's workspace, recovered the benchmark predicate (the BDDL goal and region definitions on LIBERO, the task class's success check on RoboCasa365, the stated condition on CaP-Bench), and compared the end state the agent measured with that predicate; for RoboCasa365 the scene was re-instantiated from the episode seed to recover quantities the predicate uses but the agent cannot observe, such as region boxes and target poses. One RoboCasa365 episode is excluded because every shell command failed in the sandbox and the robot was never commanded. To see what the demonstrations teach, we replayed the final state of each of the 50 LIBERO demonstrations for three of the tasks in the benchmark environment: the middle drawer ends at 127 to 160 mm of travel (median 156, the joint's limit is 160), the plate ends within 4 cm of the centre of the stove-front region in 49 of 50, and the bottle's base rests in the rack's 22 mm band in the 25 demonstrations whose final state satisfies the predicate. Each episode receives one class, the stage at which the decisive error lies, and a record of whether its last message announces completion.

Classes. A: the agent completed and verified its own reading of the instruction, and the predicate needs a quantity or a region the instruction does not state. B: the predicate contradicts the instruction (it scores another object, or a region the named object cannot occupy). C: the object or the required force lies at or beyond the limits of the gripper, the base or the arm's reach. D: the episode is still in progress at the cap, with no decisive error, and the budget went into machine behaviour that no instruction or interface states (a base without a brake, doors that close themselves, fixtures whose scored joint is not the one that moves). E: an error of the agent itself, in perception (a target never searched for, a misread joint), in control (a wrong gripper width, a release before arrival, a trajectory through an obstacle), or in waiting (budget spent polling background jobs). Table 12 lists the subcauses.

By benchmark. On LIBERO-PRO, classes A and B hold 59 of 104 failures, and every one of these episodes ends with a completion message that states the agent's own criterion; the 17 class-B episodes come from two task variants whose predicate names a different object than the instruction or a region the object cannot fit. On CaP-Bench six of the seven failures are the handover, all with the same control error. On RoboCasa365, classes C and D hold 188 of 302 failures: the base is a velocity-servo joint with 250 N of friction and no brake, so contact forces above that move the robot, and the arm cannot reach the floor, so a dropped object ends the episode; 76 episodes spent their budget on doors, handles and levers, and 49 on holding station. Figures 13 to 16 show one episode for each of classes B, A, D and E in the images the agent saved.

Limits. The study reads one model's episodes; the other models in Table 5 were not classified. Classes were assigned by reading and are recorded with a confidence level; five LIBERO-PRO episodes (a bottle-in-drawer task) could not be resolved between class A and a residual drawer opening below the point-cloud resolution and are counted under A.

H Programs the Agent Wrote

The programs below are taken verbatim from mainline episodes (Claude Code with Opus 5); only imports and repeated blocks are elided, marked . . . . They are the kind of code behind the numbers in Tables 2 and 3.



(a) start: two identical pots (b) end: the pot the agent resolved as "left" is on the burner (c) overhead check, the other pot untouched (d) the agent's crop: axis 1.2 mm from the burner centre

Figure 13: Class B, LIBERO-PRO task variant "put the left moka pot on the stove" (nine of ten seeds; seed 2 shown). The BDDL file starts `moka_pot_2` in `moka_pot_left_init_region` and `moka_pot_1` in `moka_pot_right_init_region`, and its goal is `(On moka_pot_1 flat_stove_1_cook_region)`. The agent checked the image-right direction of all seven cameras, moved the pot in the left region, placed it 1.2 mm from the burner centre with the stove carrying its full weight, and is scored on the other pot.



(a) pre-place pose over the rack (b) released: bottle along a groove (c) side view after settling (d) front view after settling

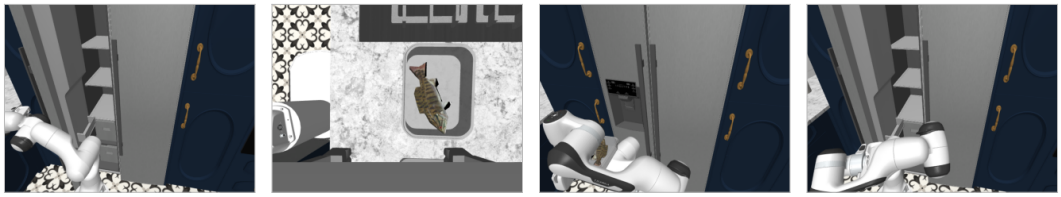
Figure 14: Class A, LIBERO "put the wine bottle on the rack" (seven of seven failed seeds; seed 5 shown). The agent modelled the shelf as a 28.6° corrugated plane, wrote "laying the bottle along a groove was the only stable option: cross-ways it rolls off", laid it along the slope and measured it at rest. The predicate reads the bottle's base against `wine_rack_1_top_region`, a band 22 mm wide at the foot of the slope; the base ended 40 mm up the slope from that band and 10 mm outside its width. In the benchmark's demonstrations the bottle lies along the slope with its base in that band.

Perception: segment the depth array by height and back-project (CaP-Bench Nut). The agent saved the overhead depth topic as an array, thresholded height above the table, and turned each connected component into world coordinates with the camera intrinsics it had read from the `camera_info` topic.

```
d = np.load("bird_depth.npy")          # depth in meters, from cam_snap.py
f = 579.4112549695428                  # focal length, from /birdview/color/camera_info
zw = 2.0 - d                           # height above the floor (camera at z = 2.0)
mask = ((zw > 0.825) & (zw < 1.3)).astype(np.uint8)
n, lab, stats, cent = cv2.connectedComponentsWithStats(mask, 8)
for i in range(1, n):
    if stats[i, 4] < 50: continue
    ys, xs = np.where(lab == i)
    zz = zw[ys, xs]
    wx = (ys - 240) * d[ys, xs] / f; wy = (xs - 320) * d[ys, xs] / f
    print(f"comp{i} area={stats[i,4]} zmax={zz.max():.4f} zmin={zz.min():.4f}")
    print(f"   world x[{wx.min():.4f},{wx.max():.4f}] y[{wy.min():.4f},{wy.max():.4f}]"
          f" centroid=({wx.mean():.4f},{wy.mean():.4f})")
```

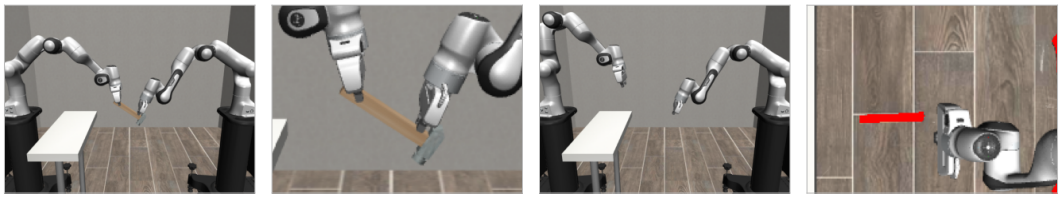
Perception: cross-check from a second camera (same episode). Before moving, the agent repeated the measurement from an oblique camera, transforming its depth image into the world frame with the camera pose it had read from the transform tree.

```
d = np.load("ag_depth.npy"); f = 579.4112549695428
# agentview optical frame: world pos (0.5,0,1.35) quat (0.6531,0.6531,-0.2710,-0.2710)
q = np.array([0.6531, 0.6531, -0.2710, -0.2710]); q = q / np.linalg.norm(q)
x, y, z, w = q
R = np.array([[1-2*(y*y+z*z), 2*(x*y-z*w),   2*(x*z+y*w)],
```



(a) freezer door opened (b) tray grasped at the counter, 0.8 m away (c) back at the freezer: door shut again (d) door opened a third time

Figure 15: Class D, RoboCasa365 “move the trays into the freezer” (ten of ten failed seeds hit the wall clock; seed 4 shown). The door closes by itself while the robot drives to the counter, so every round trip starts over; the agent’s own label for the move after (c) is “Retract arm from the closed door”. The base also pushes the door shut and slides toward the fridge when the arm pulls, since it has no brake. Nothing was dropped or misread; the episode is a loop the budget ends.



(a) both arms on the hammer (b) the agent’s crop: handle at 33° (c) after arm 0 opens: nothing in either gripper (d) the agent’s floor mask: hammer 0.5 m out of reach

Figure 16: Class E, CaP-Bench handover (four of six failed seeds share the mechanism; seed 58 shown). The agent measured the handle at 0.030 m and commanded arm 1 to 0.008 m per finger; the workspace documentation says “Command a target width, never 0 against an object; 0 keeps squeezing and can eject it”. The instant arm 0 opened, arm 1’s fingers went from 0.0123 to 0.0010 m and the hammer was squeezed onto the floor. Arm 0 had also held the hammer 0.21 m from the head, so it drooped from 3° to 33° during the carry.

```
              [2*(x*y+z*w),   1-2*(x*x+z*z), 2*(y*z-x*w)],
              [2*(x*z-y*w),   2*(y*z+x*w),   1-2*(x*x+y*y)]])
t = np.array([0.5, 0.0, 1.35])
H, W = d.shape
u, v = np.meshgrid(np.arange(W), np.arange(H))
Z = d.astype(np.float64)
P = np.stack([(u-320)*Z/f, (v-240)*Z/f, Z], -1)    # camera frame
Wd = P @ R.T + t                                   # world frame
...
blob((zz > 0.94) & (zz < 0.96), "PEGTOP")
blob((zz > 0.834) & (zz < 0.846), "NUTTOP")
```

Perception: cropping and annotating a camera image for inspection (same episode). To read the nut’s orientation and handle width, the agent cropped the overhead image, enlarged it, and drew a pixel grid with coordinate labels, then read the result with its own vision.

```
img = cv2.imread("snap_birdview.png")
c = img[130:235, 335:445]
c = cv2.resize(c, None, fx=6, fy=6, interpolation=cv2.INTER_NEAREST)
# draw grid every 10 original px
for i in range(0, 111, 10):
    cv2.line(c, (i*6, 0), (i*6, c.shape[0]), (0, 0, 255), 1)
    cv2.putText(c, str(335 + i), (i*6 + 2, 12), 0, 0.35, (0, 0, 255), 1)
for j in range(0, 106, 10):
    cv2.line(c, (0, j*6), (c.shape[1], j*6), (0, 255, 0), 1)
    cv2.putText(c, str(130 + j), (2, j*6 - 2), 0, 0.35, (0, 255, 0), 1)
cv2.imwrite("nut_zoom.png", c)
```

Manipulation: a client library (LIBERO-PRO, open the drawer and put the cream cheese inside). The agent wrapped the robot’s native interfaces once, in a module it then imported from

Table 12: Sub-causes behind Table 11. Counts are failed episodes of OpenRUA (Claude Code, Opus 5). Thresholds are those of the benchmark predicates (LIBERO: articulated-object joint ranges and region boxes in the BDDL files; RoboCasa365: the task's success check).

Class	Sub-cause	LIBERO	CaP	RoboCasa
A	Joint threshold: "open" needs > 140 mm or $\geq 95\%$ of travel, a mixer head $\geq 99\%$; lift needs > 0.10 m	14	1	6
	Region box differs from the words: "in front of the stove", "right of the plate", "on the rack", "in the microwave"	19	0	0
	Tolerance of `On`: body origins within 30 mm (a mug's origin sits 18 mm toward its handle)	2	0	0
	Final check coarser than the predicate: drawer read as shut from a point cloud, door "closed" needs $< 0.5\%$ of travel, lid seated within 4 cm and 7°	7	0	10
	Stand-off after release (> 0.15 / 0.25 m), contact and timer clauses, sweep ≥ 0.10 m, "do not touch the food", navigation pose within 0.20 m	0	0	20
B	Predicate names the other object ("left moka pot" scores the pot in the BDDL's right init region)	9	0	0
	Target region cannot hold the object (100 mm mug, 55 mm caddy slot)	8	0	0
	"Right drawer" resolved relative to the spawn pose	0	0	2
C	Object at the edge of the gripper envelope; binary gripper extrudes soft, thin or round objects	12	0	54
	Base pushed or lifted by contact forces (velocity-servo base, 250 N friction, no brake)	0	0	7
	Object out of the arm's reach after a drop (floor below $z \approx 0.3$ m)	0	0	2
D	Budget spent on fixtures: self-closing doors, round-bar handles, faucet levers, toaster lever	0	0	76
	Budget spent holding station against base recoil and drift; cap with no decisive error	15	0	49
E	Perception: second candidate of the same kind never resolved; target outside the parked view never searched; a joint the sensors do not expose (faucet hinge, knob modulo 180°, button geometry)	3	0	24
	Control: half the measured handle width commanded, IK branch jump, release before arrival, recovery sweep knocks the placed object	12	6	28
	Waiting: turns or wall clock spent polling background jobs; turn ended while a monitor was armed	3	0	24
	Failed episodes	104	7	302

every step program: the inverse-kinematics service, the trajectory and gripper actions, the joint-state subscription, and the velocity stream.

```
class Rig:
    def __init__(self):
        rclpy.init()
        self.node = rclpy.create_node("rig")
        self._js = {}
        self.node.create_subscription(JointState, "/joint_states", self._on_js, 10)
        self.ik = self.node.create_client(GetPositionIK, "/compute_ik")
        self.fk = self.node.create_client(GetPositionFK, "/compute_fk")
        self.traj = ActionClient(self.node, FollowJointTrajectory, TRAJ["port"])
```

```
        self.grip = ActionClient(self.node, GripperCommand, GRIP["port"])
        self.twist_pub = self.node.create_publisher(TwistStamped, TWIST["port"], 10)
        ...
    def servo(self, lin, ticks=20, rate_hz=20.0):
        """Stream a world/base-frame twist (m/s) for `ticks` messages."""
        msg = TwistStamped()
        msg.header.frame_id = TWIST["frame"]
        msg.twist.linear.x, msg.twist.linear.y, msg.twist.linear.z = (
            float(lin[0]), float(lin[1]), float(lin[2]))
        for _ in range(ticks):
            msg.header.stamp = self.node.get_clock().now().to_msg()
            self.twist_pub.publish(msg)
            rclpy.spin_once(self.node, timeout_sec=1.0 / rate_hz)
```

Manipulation: approach the handle under feedback (same episode). After a first grasp had closed on air, the agent rewrote the approach as a proportional servo on the measured offset and used the finger gap as evidence of a grasp.

```
"""Reopen, drive the fingers the rest of the way onto the bar, close.

First attempt stopped with the fingertips at y=-0.1354, still 4 mm in
front of the bar's front surface (-0.1392), so the fingers closed on air
(gap 0.0019). Push to y=-0.1545 this time and keep servoing until the
error is actually gone.
"""
GRASP = np.array([0.0365, -0.1545, 1.0880])
r = Rig()
print("open ->", round(r.gripper(0.04), 4))
for i in range(25):
    p, _ = r.tcp_pose()
    dy = GRASP[1] - p[1]
    dz = GRASP[2] - p[2]
    print(f"  {i} tcp {np.round(p, 4)} dy {dy:+.4f}")
    if abs(dy) < 0.0025:
        break
    r.servo([0.0, np.clip(dy * 5, -0.05, 0.05), np.clip(dz * 5, -0.02, 0.02)], ticks=20)
gap = r.gripper(0.0)
print("closed -> gap", round(gap, 5))
print("grasp evidence: gap > closed_m(0.0) means something is held")
```

Manipulation: pull the drawer with guards (same episode).

```
"""Pull the grasped top drawer open along +y, watching the grip."""
PULL_TO = -0.020  # world y for the fingertips; ~0.14 m of travel
r = Rig()
p0, _ = r.tcp_pose()
for i in range(20):
    p, _ = r.tcp_pose()
    gap = r.finger_gap()
    if gap < 0.008:
        print("  GRIP LOST (gap collapsed)"); break
    if p[1] >= PULL_TO:
        print("  reached pull target"); break
    r.servo([0.0, 0.05, 0.0], ticks=20)
    q, _ = r.tcp_pose()
    print(f"  {i} tcp {np.round(q, 4)} gap {r.finger_gap():.5f} moved {q[1]-p[1]:+.4f}")
    if abs(q[1] - p[1]) < 0.001:
        print("  stalled -- drawer at its travel limit or arm blocked"); break
p, _ = r.tcp_pose()
print("end", np.round(p, 4), "total pull", round(p[1] - p0[1], 4))
```

Manipulation: insert in monitored segments (CaP-Bench Nut). The insertion is a sequence of short joint-trajectory segments; after each, the program reads the wrench and the finger gap, and it aborts if the planned path dips below a floor it computed from the arm's link heights.

```
def go(name, zt, secs, prev, phi=PHI):
    tw = (XY[0], XY[1], zt)
    qs = plan.solve(tw, phi, seeds(prev, tw))       # own IK search
    if qs is None:
        raise SystemExit("IK FAIL " + name)
    mz, t = plan.path_min_z(prev, qs)
    floor = min(plan.link_zs(prev) + plan.link_zs(qs)) - 0.004
    if mz < floor:
        raise SystemExit("PATH minz %.3f floor %.3f %s" % (mz, floor, name))
    code = r.goto_q(qs, secs)                        # FollowJointTrajectory
    act = np.array(r.arm_q())
```

```
    pa, _ = kin.fk_tcp(act)
    w = r.wrench()                                  # external wrench topic
    print(f"{name} z={zt:.3f} code={code} tcp={np.round(pa+rc.BASE_IN_WORLD,4)} "
          f"gap={r.finger_gap():.5f} F={np.round(w[:3],2)} T={np.round(w[3:],3)}")
    return act

for z, s in [(0.99, 1.5), (0.965, 1.0), (0.945, 1.0), (0.92, 1.0),
             (0.89, 1.0), (0.86, 1.0), (0.833, 1.0)]:
    q = go("descend", z, s, q)
```

Perception: ray-cast a colour mask onto the table plane (CaP-Bench Wipe). Each connected blob of the brown mask is cast from the camera centre onto the table plane; a blob counts as spill only if nine tenths of it lands on the table, and the hits are rasterized into a 5 mm grid that is intersected across three cameras.

```
grid = np.zeros((len(GX), len(GY)), dtype=bool)
n_lab, lab = cv2.connectedComponents(mask)
for i in range(1, n_lab):
    vs, us = np.nonzero(lab == i)
    if len(us) < 30: continue
    dirs = (R @ np.vstack([(us - cx) / fx, (vs - cy) / fy, np.ones(len(us))])).T
    s = (Z0 - C[2]) / dirs[:, 2]          # ray length to the table plane
    good = s > 0
    P = C + dirs[good] * s[good, None]
    on = ((P[:, 0] > TAB[0]) & (P[:, 0] < TAB[1]) & (P[:, 1] > TAB[2]) & (P[:, 1] < TAB[3]))
    # a blob is spill only if essentially ALL of it lands on the table
    if on.mean() < 0.9: continue
    P = P[on]
    ix = np.clip(((P[:, 0] - TAB[0]) / CELL).astype(int), 0, len(GX) - 1)
    iy = np.clip(((P[:, 1] - TAB[2]) / CELL).astype(int), 0, len(GY) - 1)
    grid[ix, iy] = True
```

Manipulation: a force-guarded descent (CaP-Bench Restack).

```
def guarded_down(self, xy, z_start, z_floor, step=0.002, fthresh=1.5, yaw=YAW, label="
    guarded"):
    """Creep down in `step` increments, stopping at the first real change
    in the wrist wrench.  Returns (contact_z, contacted)."""
    f0 = self.force()
    seed = self.a.arm_joints()
    z = z_start
    while z > z_floor + 1e-9:
        z = max(z - step, z_floor)
        seed = self.ik_tcp(xy, z, yaw, seed)
        self.a.send_traj([(seed, 0.35)], "")
        t = self.tcp()
        df = self.force() - f0
        n = np.linalg.norm(df)
        print(f"    z={t[2]:.4f} dF={np.round(df, 2)} |dF|={n:.2f}", flush=True)
        if n > fthresh:
            print(f"  {label}: CONTACT at z={t[2]:.4f} (|dF|={n:.2f})", flush=True)
            return t[2], True
```

Manipulation: move both arms at once (CaP-Bench two-arm lift). Both trajectory goals are sent before either result is awaited.

```
def move_both(self, traj, label=""):
    """traj: {arm: (list_of_q, list_of_times)}; both arms execute concurrently."""
    handles = {}
    for arm, (qs, ts) in traj.items():
        f = self.fjt[arm].send_goal_async(self._goal(arm, qs, ts))
        rclpy.spin_until_future_complete(self.node, f)
        gh = f.result()
        if not gh.accepted:
            raise SystemExit(f"{arm}: goal rejected")
        handles[arm] = gh.get_result_async()
    codes = {}
    for arm, fut in handles.items():
        rclpy.spin_until_future_complete(self.node, fut)
        codes[arm] = fut.result().result.error_code
    print(f"[move {label}] error_codes={codes}", flush=True)
    self.refresh()
    return codes
```



(a) Check scene motion. (b) First grasp fails. (c) Remeasure the fallen bottle. (d) Place the bottle in the basket.

Figure 17: **The agent revises its grasp after a failed attempt in a LIBERO-PRO task.** (a) It compares camera frames and detects that the scene has not settled. (b) Its first grasp misses and knocks the ketchup bottle over; finger-gap readings and a camera view reveal the failure. (c) It uses a wrist-camera observation and a reconstructed point cloud to estimate the fallen bottle's orientation and plan a new grasp. (d) It places the bottle upright in the basket and checks the result. The panels show RGB-D observations processed by agent-written code.

Manipulation: level a drooping handle (CaP-Bench handover). After the handover the hammer drooped; the agent measures the handle's pitch from its top face, counter-rotates the wrist, and re-solves so the handle centre lands at the target height.

```
pitch, mid, R = measure(r)
T = r.hand_world("right")
p_h = T[:3, :3].T @ (mid - T[:3, 3])          # handle midpoint in the hand frame
dR = Rx(-pitch)
R_new = dR @ T[:3, :3]
off = R_new @ p_h
origin = np.array([mid[0], mid[1], Z_TARGET]) - off
q = r.solve("right", origin, R_new, seed=r.q("right"), other_q=r.q("left"),
            table_z=0.845, grasp_contact=True)
dq = np.abs(q - r.q("right")).max()
print(f"== levelling, dq_max={dq:.3f}")
r.send_q("right", q, float(np.clip(2.5 + dq * 2.5, 2.5, 7.0)))
```

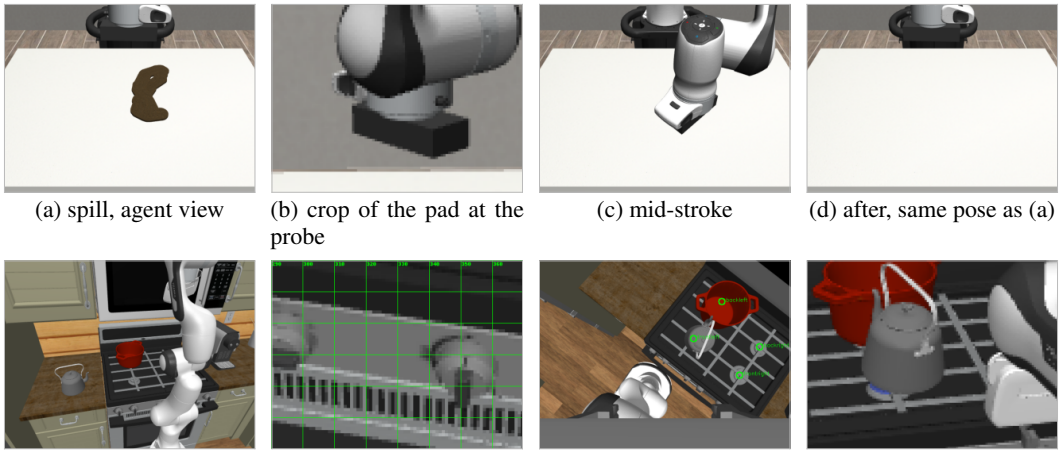
I More Episodes

This appendix adds episodes from all three benchmarks to the ones shown in Section 4.3 and Section 4.4, in the same three forms: sensor images the agent saved directly or processed with its own code, control loops plotted from its printed logs, and its terminal and programs. Text and code are verbatim; for illustration, lines between the ones shown are omitted and marked . . . .

Recovery from a failed grasp.

Perception pipelines. Figure 18 shows two more pipelines. On CaP-Bench Wipe (a–d) the agent masks the spill by colour in three cameras, ray-casts each blob onto the table plane, keeps only blobs that land entirely on the table, rasterizes them into a 5 mm world grid, and intersects the three grids; after the stroke it repeats the measurement and compares the same camera view at the same arm pose. On RoboCasa365 (e–h, boil the kettle) it crops the stove knobs and draws a pixel grid to pick knob pixels, projects the burner centres it computed from depth back into the wrist image to check them, and gates a colour threshold by depth-derived world coordinates to find the flame.

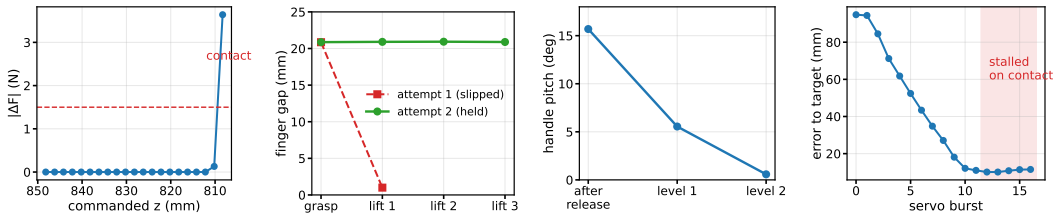
Control loops. Figure 19 plots four more loops from the agents' logs. (a) CaP-Bench Restack: a force-guarded descent in 2 mm steps that stops at the first change in the wrist wrench above 1.5 N. (b) CaP-Bench two-arm lift: the finger gap through a staged lift; in the first attempt the gap collapsed from the bar thickness to 1 mm and the agent aborted, re-measured the displaced pot, dragged it back with one arm, and grasped 7 mm deeper; in the second attempt the gap held through all three stages. (c) CaP-Bench handover: after the receiving arm took the hammer and the first arm released it, the handle drooped by 15°; the agent measured the pitch from the handle's top face and levelled it in two iterations. (d) RoboCasa365 weigh the ingredients: a velocity servo toward a jar converges until the hand meets the jar and the error stalls at 11 mm, which the program reports as contact.



(a) spill, agent view (b) crop of the pad at the probe (c) mid-stroke (d) after, same pose as (a)

(e) kitchen, left camera (f) gridded crop of the knobs (g) burner centres projected back (h) flame found under the kettle

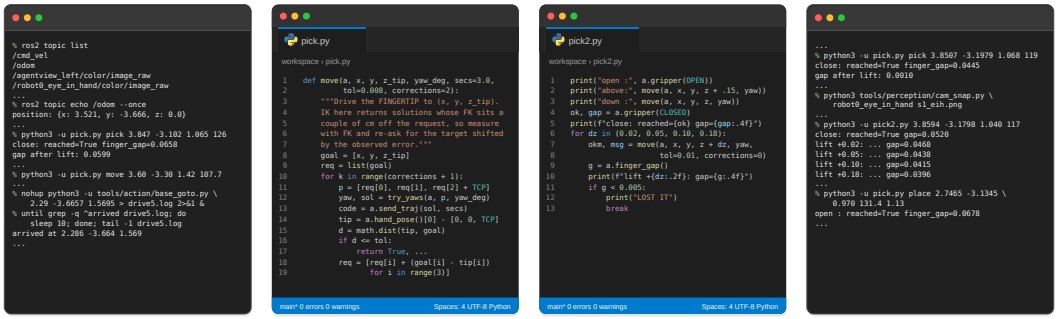
Figure 18: Two more perception pipelines: CaP-Bench Wipe (a–d) and RoboCasa365 boil the kettle (e–h). Panels (b), (f), (g), and (h) show sensor observations processed by agent-written code for visual inspection.



(a) guarded descent (Restack) (b) staged lift (two-arm lift) (c) levelling (handover) (d) servo to contact (RoboCasa365)

Figure 19: Four more control loops, plotted from the agents' printed logs.

A mobile-manipulation episode. Figures 20 and 21 show RoboCasa365 portion the hot dogs: two buns and two sausages from a bowl onto two plates about 1.5 m apart, so every transfer needs a base drive. The agent drives the base with the starter script and absorbs the base motion in its arm helper, which re-reads the base pose inside every inverse-kinematics call; it measures each move with forward kinematics and re-asks for the target shifted by the observed miss; after a sausage slipped, it rewrote the lift to read the finger gap at four heights. The second plate was outside the cameras' view until the first drive revealed it.



(a) drive, then reach (b) move with an FK-measured correction (c) lift in stages, watching the gap (d) a failed grasp and the retry

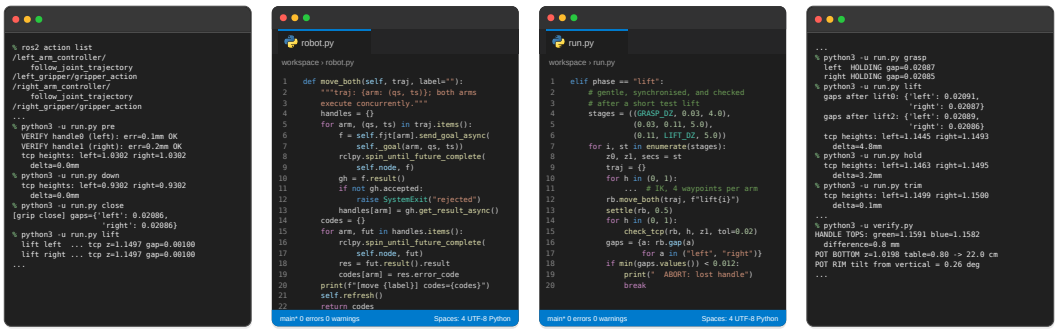
Figure 20: RoboCasa365 portion the hot dogs, in the agent's terminal and programs.



(a) bowl with the items (b) after the first drive: a second plate (c) empty gripper after a failed grasp (d) bun wedged in the open fingers (e) both plates set

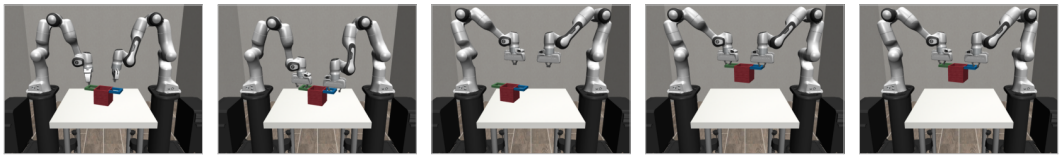
Figure 21: The same episode in the images the agent saved at its checkpoints; (c) and (d) are crops it made to diagnose two failures it then recovered from.

A two-arm episode. Figures 22 and 23 show CaP-Bench two-arm lift. With no motion planner on this machine, the agent wrote its own forward and inverse kinematics for the Panda, validated forward kinematics against the transform tree to 0.5 mm, and wrote a robot class whose `move_both` sends both trajectory goals before waiting for either, so the arms move together. It runs the task as one command per phase, each printing a verification, and lifts in three stages with an abort if either gripper's gap collapses; the first attempt slipped, the second held, and a final camera measurement put the two handle tops within 0.8 mm.



(a) one command per phase, verified (b) send both goals, then wait for both (c) staged lift with an abort guard (d) the second attempt, and the check

Figure 22: CaP-Bench two-arm lift, in the agent's terminal and programs.



(a) pot between the arms (b) both grippers on the handles (c) first lift: slipped (d) second lift (e) held level

Figure 23: The same episode from the front camera at the agent's checkpoints.

A failed episode. Figure 24 shows a LIBERO-10 episode (put both the alphabet soup and the cream cheese in the basket) that the benchmark scored as failed although the agent reported success. Two cans stand on the table; the agent cropped and identified only the tomato can in front, never inspected the blue can behind the milk carton, grasped the tomato can as the alphabet soup, placed it and the cream cheese cleanly, and verified placement, not identity. Its final message reads "Done — both objects are in the basket"; in the last image the blue alphabet-soup can is still on the table. Appendix G counts such false completions.



(a) scene: two cans (b) the only can it inspected (c) after the first pick (d) its check of the basket (e) "done": the blue can remains

Figure 24: A false completion on LIBERO-10: the wrong can, verified for placement but not identity.