ROBOARCHITECT: AUTOMATING THE DESIGN OF ROBOTIC AGENTS WITH PERSISTENT MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent work shows that a frozen robot policy performs far better inside an agent harness, where control code decides when to query the policy, consult vision tools, or command the arm directly. Yet today these harnesses are built by hand and redesigned for each new robot or benchmark. We show that harness design can be automated, offering a strong alternative to finetuning on a novel setup. In ROBOARCHITECT, a coding model observes a robot's rollouts, diagnoses failures, and iteratively revises the harness around the frozen policy. The harness evolves on a set of episodes for its optimization and is evaluated on held-out scenes and tasks. Without updating any model weights, ROBOARCHITECT raises average success from 9.7% to 43.3% on unseen scenes (4.5×) and from 12.7% to 40.2% on unseen tasks (3.2×), across multiple benchmarks and policies. The gains are largest where the benchmark is far from the policy's training data: wrapping a general-purpose policy, the evolved harness yields a specialist that often outperforms even the finetuned version. Intuitive control strategies are automatically designed, such as lifting a jammed arm clear or reaching in horizontally when an obstacle blocks the way from above. The harness also evolves its own persistent memory, carrying learned experience, such as effective grasp depths, into later decisions. These results suggest that substantial headroom in robotic manipulation lies in the agent harness, and its design can be effectively automated and optimized without weight updates.

1 INTRODUCTION

Language models have become far more useful inside an agent harness: surrounding code that mediates when and how to call the model, what context to supply, and how to handle its output. Robot policies are undergoing the same shift. This shift is particularly important for robotics, where adapting a policy to new tasks or hardware often requires collecting new demonstrations and finetuning model weights (Kim et al., 2024; Zang et al., 2025; Park et al., 2026). We posit that **harness optimization offers an effective alternative to finetuning, turning a general policy into a specialist** for a specific robot setup (§4.2).

Recent agentic robot systems like VoLo (Chen et al., 2026c) and HarnessVLA (Zhang et al., 2026e) show strong gains from wrapping frozen policies in hand-written harnesses (Xu et al., 2026; Zhao et al., 2026b; Gao et al., 2026; Hu et al., 2026). Today, these harnesses are still manually designed, often requiring substantial engineering effort for each task and embodiment. Building such a harness follows a familiar trial-and-error loop: run the agent on the robot, observe where it fails, diagnose the cause, modify the control code, and test again. This work is slow to perform by hand, and long recordings of robot rollouts are difficult for a human engineer to inspect exhaustively. We ask **whether this complete engineering loop, from diagnosing failure traces to refining the harness design to fix them, can be automated**.

In this work, ROBOARCHITECT uses a coding model to automatically design and optimize the harness (§3). Given access to a frozen policy, auxiliary vision and language tools, and step-by-step rollout recordings, the coding model diagnoses failure modes and proposes an updated harness to address them. Like a finetuned policy, a harness optimized on specific rollouts could overfit to them. We therefore split each benchmark into "training" (for harness optimization) and held-out episodes,

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.

evaluating on *unseen scenes*, where tasks are shared but scene arrangements differ, and *unseen tasks*, where held-out episodes contain entirely different tasks that require shared manipulation skills (§4.1).

Across diverse benchmarks and policies, our auto-optimized harnesses raise the average success rate on unseen scenes from 9.7% to 43.3% (4.5×), and on unseen tasks from 12.7% to 40.2% (3.2×) (Fig. 1). The strongest gains occur when policies are deployed in out-of-domain settings, where ROBOARCHITECT can effectively substitute for new training data: wrapping a general policy yields a specialist that often outperforms the finetuned version of the same model. For example, in settings where the vision-language-action (VLA) policy contributes no useful signal, the harness learns to stop querying it and instead drives the arm from its own perception, effectively becoming a code-as-policy agent. While code-as-policy (Liang et al., 2023; Lu et al., 2026; Elmaaroufi et al., 2026) and learned VLA policies (Kim et al., 2024; Intelligence et al., 2025) have traditionally been treated as distinct paradigms, ROBOARCHITECT unifies them empirically (§4.2). Unlike weight updates, each harness revision is also fully interpretable: it targets specific failure modes with intuitive control logic (Figs. 3 and 6), making subsequent revisions easy to inspect and build upon, and such iterative refinement proves to be key to generalization (§4.2).

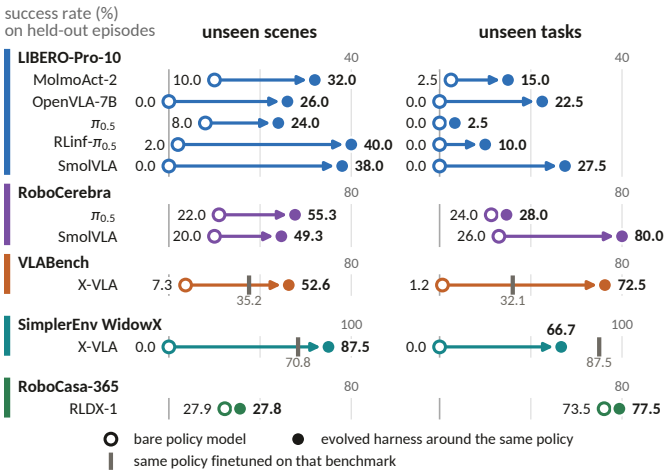


Figure 1: **Wrapping a frozen policy in an evolved harness raises its held-out success rate from 9.7% to 43.3% on unseen scenes and from 12.7% to 40.2% on unseen tasks, often past the same policy finetuned on the benchmark** (§4.2). All scores are evaluated on the held-out episodes never seen during the harness's optimization. *Unseen scenes:* held-out episodes evaluate the same tasks under varied scene arrangements. *Unseen tasks:* held-out episodes evaluate distinct tasks to test cross-task skill transfer (§4.1).

We also equip the harness with **persistent memory**, allowing experience to accumulate across episodes. No schema or memory rules are specified; ROBOARCHITECT is entirely free to decide what to store, in what form, and how to retrieve it. As a result, diverse forms of memories emerge from the harness optimization to record key pragmatic clues, such as effective grasp depths that inform later control decisions.

As ROBOARCHITECT proves highly effective in solving tasks on one robot embodiment at a time, we further use its harness evolution to build **a unified, generalizable agent that operates across different robot platforms** (§4.2). In this single harness optimized on a training pool drawn from diverse benchmarks, shared control strategies and memory structures emerge naturally across embodiments.

Our findings suggest that substantial performance headroom in robotic manipulation lies in the agent harness, as an alternative to finetuning. ROBOARCHITECT automates its design and optimization, providing a practical way to adapt robot policies to new environments without updating weights.

2 RELATED WORK

Vision-language-action policies. VLA policies are pretrained vision-language models finetuned for robot control, mapping visual observations and language instructions directly to robot actions (Brohan et al., 2023; Kim et al., 2024; Black et al., 2026; Intelligence et al., 2025; Shukor et al., 2025; Zheng et al., 2026; Fang et al., 2026; Kim et al., 2026). In practice, their performance on new, out-of-distribution tasks can depend heavily on finetuning (Zang et al., 2025; Park et al., 2026), which requires collecting demonstrations for every new task or hardware configuration. Such finetuned manipulation skills often fail to generalize beyond initial training distributions (Zhou et al., 2025).

Our held-out protocols test exactly this transfer (§4.1), on which the evolved harnesses maintain substantially better performance (§4.2). A complementary line of work keeps the policy frozen and adds inference-time mechanisms: retrieving demonstrations or successful action segments to condition the policy (Park et al., 2026; Zhao et al., 2026a), or selecting among the policy's candidate actions with a learned value function (Xie et al., 2026b; Zhang et al., 2026c). In contrast, ROBOARCHITECT **evolves complete agent interfaces from scratch that combine various mechanisms**, including persistent memory, programmatic control, and policy orchestration, all custom-fit to each setup (§4.3 and §4.4).

Robotic agents. Code-as-policy methods (Liang et al., 2023; Singh et al., 2022; Huang et al., 2023) bypass trained policies, and instead use executable programs as the primary robot control, composing perception, reasoning, and motion primitives. Recent coding agents go further, using iterative code generation to solve manipulation tasks (Tsui et al., 2026; Kumar, 2026; Xie et al., 2026a). However, CaP-X (Fu et al., 2026) shows that their success depends heavily on the hand-written primitives they are given. Other recent work retains trained VLAs but places them inside agent-level orchestration. VoLo (Chen et al., 2026c) treats the policy as an interruptible tool, guiding it alongside vision models and programmatic motions. HarnessVLA (Zhang et al., 2026e) assigns high-level planning to an LLM agent and restricts the frozen policy to contact-level primitives. Several systems further decompose tasks into subproblems with memory or verification mechanisms (Xu et al., 2026; Gao et al., 2026; Zhao et al., 2026b). Hu et al. (2026) find that components in such hierarchical VLA architectures (planner, controllers, and switching logic) must be co-designed to maximize performance. These works establish that the interface around a VLA matters, but their agent architectures were specified manually in advance. ROBOARCHITECT instead automates harness design and optimization, including the VLA/code division of labor (§4.2) and persistent memory (§4.4).

Automated code optimization. Coding models are increasingly used in auto-research routines that evaluate agent outputs, analyze execution traces, and iteratively refine agent implementations (Hu et al., 2025; Zhang et al., 2026b; Lee et al., 2026; Zhang et al., 2026a; Wei et al., 2026; Zhou, 2026; Cai et al., 2026). Within robotics, most LLM-driven optimization loops generate artifacts that train policy weights: reward functions (Eureka (Ma et al., 2024)), training configurations revised from execution logs (ENPIRE (Xiao et al., 2026)), or VLM-decomposed exploration data distilled into policies via finetuning (EXIMO (Sukhija et al., 2026)). These systems **automate finetuning**, and are thus complementary to ROBOARCHITECT, which **automates the agent built around a frozen policy**. Another line of work accumulates reusable code skills: repairing and archiving control functions (Lu et al., 2026), searching over repositories of control code (Elmaaroufi et al., 2026), assembling computation graphs from skill libraries (Chen et al., 2026a), or collecting skills during self-directed play (Zhang et al., 2026d). Crucially, these skill systems optimize programs that **entirely replace the robot policy**. In contrast, ROBOARCHITECT develops the harness *around* a frozen policy, allowing the code to **autonomously divide control between the policy and its own direct commands** (§4.2). Berman et al. (2026) similarly find that frontier coding models are more capable when given higher-level access to pretrained robot policies than when controlling low-level motion directly, an arrangement ROBOARCHITECT often evolves into on its own.

3 ROBOARCHITECT

ROBOARCHITECT optimizes the harness around a frozen policy, as illustrated in Fig. 2. It starts from a harness that passes the policy's actions through unchanged and repeats an empirical optimization loop: the harness runs on the training episodes, a coding model observes the recorded rollouts, diagnoses the failures and refines the harness. These "training" episodes serve only to optimize the harness: **no weights are updated**, and the final harness is evaluated on separate held-out episodes (§4.1). Over iterations, the harness grows into a highly performant robotic agent that, as needed, follows the policy, consults auxiliary tools, or commands the arm directly, and it can carry experience in a persistent memory. Below, we describe its key components.

Harness interface. A robotic agent harness is a program that sits between environment observations and low-level robot execution commands. At each control step, the harness receives the current environment observation, including camera images, depth maps, gripper pose, and gripper status, along with the natural language task instruction. It outputs low-level action vectors executed by the

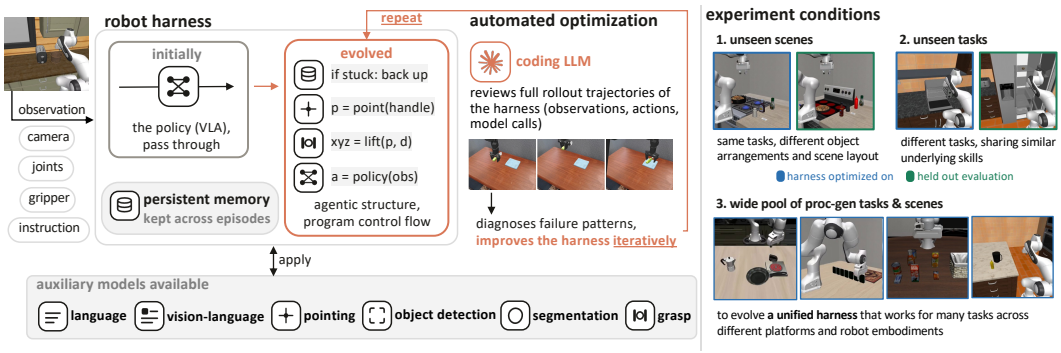


Figure 2: **ROBOARCHITECT turns a frozen policy into a performant robotic agent without updating its weights: a coding model iteratively refines the harness from execution traces.** Starting from a bare policy (VLA), ROBOARCHITECT evolves a robotic agent equipped with auxiliary tools and persistent memory. The right side illustrates experiment conditions (§4.1) that test how well the evolved harness generalizes.

robot controller. Initially, the harness is a simple pass-through of the bare VLA policy: it forwards observations to the frozen policy and returns the policy's predicted actions directly. The *Bare Policy* columns of Tab. 1 are thus the standard evaluation of the policy alone.

Auxiliary tools. Beyond the base policy, the harness can query auxiliary pretrained tools (§4.1), including language models, vision-language models, object detectors, segmentation models, pointing models, and grasp pose generators. The harness also has access to geometric utilities that project pixel-depth pairs into 3D robot base coordinates, and generate relative trajectory primitives. Together, these tools give the harness alternatives to relying on the frozen policy alone.

Optimization loop. The optimization follows the same trial-and-error loop as a human engineer (§1). Every evaluation logs complete trajectory traces, including per-step camera frames, the robot's readings, and all model and tool outputs. A coding model reviews these execution traces from the current harness, diagnoses failure modes, and proposes a candidate harness revision, where it may follow the policy, invoke any of the tools, or command the arm programmatically. The automated optimization loop thus settles the balance between policy and programmatic control (§4.2). The candidate is then re-evaluated on the same training set (§4.1) and accepted only if its performance surpasses that of the current harness, following the empirical validation in prior work on self-improving agents (Zhang et al., 2026b;a). We thus ensure a monotonic optimization trajectory on the training set, capped at a maximum number of iterations (§4.1). We compare iterative refinement with a single-attempt automated agent design in §4.2.

Persistent memory. We also provide the harness with a persistent memory mechanism that carries information across episodes and tasks. **Unlike fixed hand-crafted memory schemes** in recent robotic agents (Zhang et al., 2026e; Xu et al., 2026; Zhao et al., 2026b; Gao et al., 2026), our persistent memory structure is **autonomously evolved based on accumulated rollout evidence**, and fitted to each setup. As discussed in §4.4, ROBOARCHITECT discovers diverse forms of persistent memory.

4 EXPERIMENTS

4.1 SETUP

Benchmarks and policies. We evaluate on LIBERO-Pro-10, which is LIBERO-10 (*i.e.,* LIBERO-Long) (Liu et al., 2023) with object locations swapped as in LIBERO-Pro (Zhou et al., 2025), RoboCerebra (Han et al., 2025), VLABench (Zhang et al., 2025), RoboCasa-365 (Nasiriany et al., 2024) and SimplerEnv WidowX (Li et al., 2024). The unified harness (§4.1) is also tested on SimplerEnv Google (Li et al., 2024). §C describes each benchmark. The VLA policies are MolmoAct-2 (Fang et al., 2026), OpenVLA-7B (Kim et al., 2024), $\pi_{0.5}$ (Intelligence et al., 2025), RLinf-

$\pi_{0.5}$ (Chen et al., 2026b; Zang et al., 2025), SmolVLA (Shukor et al., 2025), X-VLA (Zheng et al., 2026), and RLDX-1 (Kim et al., 2026), with exact checkpoints in Tab. 5.

Training-source distance. We characterize each setup by the distance between the policy's training data and the evaluation benchmark (Tab. 1): (1) in-distribution, where the policy was tuned on this exact benchmark; (2) swapped, with object locations permuted as in LIBERO-Pro; (3) cross-benchmark, a different benchmark on the same simulator and robot; and (4) out-of-domain, with only general robot pretraining and benchmark unseen. For comparison with finetuning, we also evaluate public checkpoints of the same policy finetuned on the target benchmark on the same held-out test set (rows marked $\hookrightarrow$ in Tab. 1).

Held-out evaluation protocols. Every harness is evaluated on held-out episodes never seen during the harness's optimization (*i.e.,* "training"), under the following protocols:

(1) ***Unseen scenes***. Training and held-out splits share identical tasks but differ in initial scene arrangements and layouts (Fig. 8, left). The evolved harness must generalize beyond specific object positions or spatial configurations.

(2) ***Unseen tasks***. Training and held-out splits contain different tasks that share underlying manipulation skills (Fig. 8, right). The evolved harness must generalize at the skill level rather than memorizing task-specific routines.

(3) ***Unified harness across benchmarks***. Under the protocols above, individual agents are evolved separately for each setup, allowing them to specialize to the challenges of specific embodiments. As an extension, we also evolve a **single unified harness that works across multiple benchmarks simultaneously**, spanning different robot embodiments, simulators and action layouts, with the goal of discovering shared cross-embodiment agent mechanisms and persistent memory structures. We take the benchmarks LIBERO-Pro, VLABench, SimplerEnv WidowX and SimplerEnv Google as test bed (Franka Panda, WidowX and the Google mobile manipulator). We use the following policy configurations: (1) the *general* X-VLA checkpoint, never tuned on any of these benchmarks and thus operating entirely out of distribution; (2) *X-VLA specialists*, one checkpoint finetuned per benchmark. To evolve the unified harness, we build a training pool of **procedurally generated episodes with new scene arrangements, objects, instructions or tasks** outside standard benchmark evaluations (Fig. 9, top), and evaluate on the benchmarks' own standard episodes (Fig. 9, bottom).

Evaluation metrics. We report standard benchmark metrics. On VLABench, we use the official progress score, which assigns partial credit for grasping the right object and for each goal condition met; on all other benchmarks, we report success rate averaged over tasks. For simplicity, we call all these scores success rates (%) in this paper, although the VLABench numbers are progress scores.

Harness evolution configurations. We run at most 5 optimization iterations, using `claude-opus-5` as the coding model, invoked via the Claude Code client at maximum effort. Besides evaluation rollouts, each iteration takes the coding model 69 minutes and $29 of API usage on average (§C). The **auxiliary tools** are Qwen3-30B-A3B-Instruct (Yang et al., 2025) as the language model, Molmo2-8B (Clark et al., 2026) or Qwen3-VL-8B-Instruct (Bai et al., 2025) as the vision-language model, Grounding-DINO (Liu et al., 2024) for object detection, SAM 3 (Carion et al., 2026) for segmentation, Molmo2-8B for pointing, and GraspGen (Murali et al., 2025) for grasp pose generation. Further implementation details are provided in §C.

4.2 Harness Optimization Unlocks Substantial Robot Manipulation Headroom

An evolved harness turns a general policy into a specialist. Without any weight updates, RoboArchitect yields substantial gains across nearly all setups (Tab. 1), and converts poorly performing cross-benchmark and out-of-domain policies (§4.1) into functional specialists. Tab. 1 compares harness design and finetuning directly: on VLABench under both protocols and on SimplerEnv WidowX under unseen scenes, **the evolved harness around the general policy outperforms the finetuned weights**, carrying the frozen policy past a full weight update; it falls short only on SimplerEnv WidowX under unseen tasks. The gains **transfer strongly to unseen scenes and unseen tasks**: on *unseen scenes*, they generalize across novel scene arrangements, with the only minor exception of RLDX-1 on RoboCasa-365, which improved only on its optimization episodes (Fig. 10); every setup also transfers positively to *unseen tasks* (Tab. 1). This generalization reflects the structure of

Table 1: **ROBOARCHITECT's iteratively evolved harness raises mean held-out success by +345% on unseen scenes and +216% on unseen tasks.** *Bare Policy*: the VLA policy alone. *Harness (ours)*: the same frozen policy inside the harness ROBOARCHITECT designed, after a single design attempt (*One-Shot*, iteration 1) or after iterative refinement over up to 5 iterations (*Iterative*). Δ: gain of the iterative harness over the bare policy. Rows marked ↪ show the finetuned version of the same policy evaluated on the same held-out set. Training-source distance between the policy and the evaluation benchmark is color-coded: black = in-distribution; gold = spatially swapped; orange = cross-benchmark; dark red = out-of-domain. All scores are success rates in %.

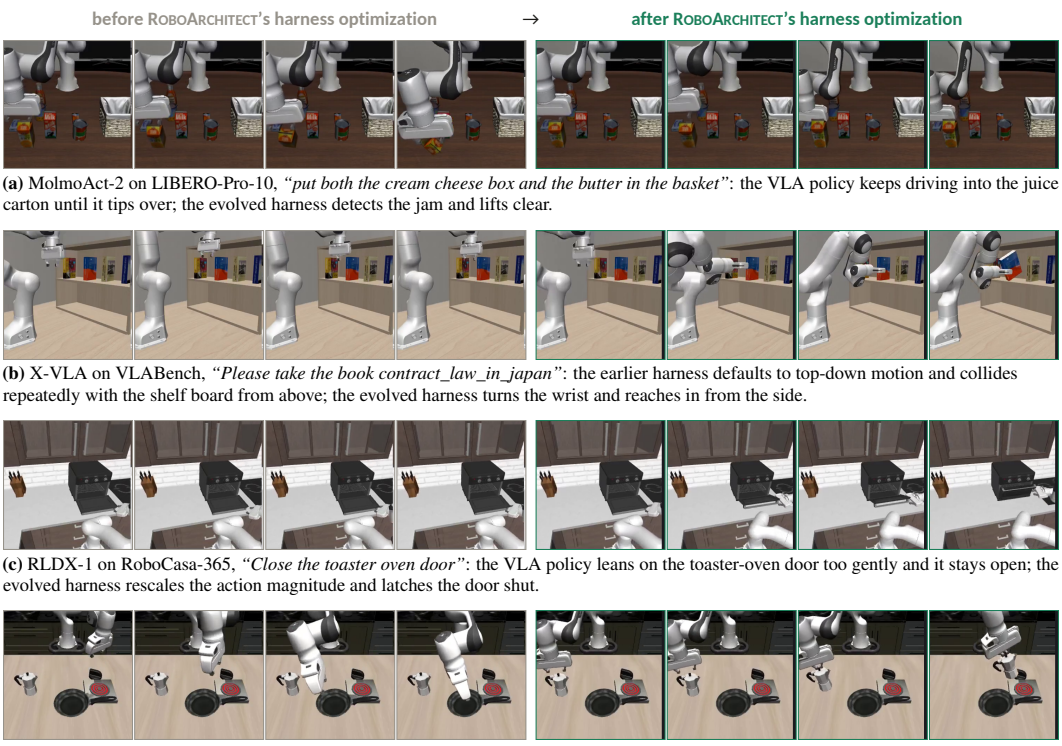
			Unseen Scenes (%)				Unseen Tasks (%)			
			Bare	Harness (ours)		Δ	Bare	Harness (ours)		Δ
Benchmark	Policy	Trained On	Policy	One-Shot	Iterative		Policy	One-Shot	Iterative	
LIBERO-Pro-10	MolmoAct-2	LIBERO	10.0	12.0	**32.0**	+22.0	2.5	2.5	**15.0**	+12.5
	OpenVLA-7B	LIBERO-10	0.0	2.0	**26.0**	+26.0	0.0	0.0	**22.5**	+22.5
	$\pi_{0.5}$	LIBERO	8.0	4.0	**24.0**	+16.0	0.0	0.0	**2.5**	+2.5
	RLinf-$\pi_{0.5}$	LIBERO-130	2.0	2.0	**40.0**	+38.0	0.0	0.0	**10.0**	+10.0
	SmolVLA	RoboCerebra	0.0	0.0	**38.0**	+38.0	0.0	10.0	**27.5**	+27.5
RoboCerebra	$\pi_{0.5}$	LIBERO	22.0	20.0	**55.3**	+33.3	24.0	12.0	**28.0**	+4.0
	SmolVLA	RoboCerebra	20.0	34.3	**49.3**	+29.3	26.0	16.0	**80.0**	+54.0
VLABench	X-VLA	general robot data	7.3	32.8	**52.6**	+45.2	1.2	–	**72.5**	+71.3
↪ *X-VLA finetuned on VLABench*			35.2	–	–	–	32.1	–	–	–
SimplerEnv WidowX	X-VLA	general robot data	0.0	77.1	**87.5**	+87.5	0.0	–	**66.7**	+66.7
↪ *X-VLA finetuned on WidowX*			70.8	–	–	–	87.5	–	–	–
RoboCasa-365	RLDX-1	RoboCasa-365	27.9	29.3	**27.8**	-0.2	73.5	76.5	**77.5**	+4.0
Mean (all setups)			9.7	21.4	**43.3**	+345%	12.7	–	**40.2**	+216%
Mean (cross-benchmark and out-of-domain)			7.3	32.5	**58.4**	+696%	6.3	–	**48.7**	+671%

Table 2: **A unified evolved harness serves different benchmarks: around the general X-VLA it substantially raises held-out success, and around the finetuned X-VLA specialists it preserves their competence.** Columns and labels read as in Tab. 1.

			Training Pool (%) procedurally generated				Held-Out (%) standard benchmark episodes			
			Bare	Harness (ours)		Δ	Bare	Harness (ours)		Δ
Policy	Trained On	Benchmark	Policy	One-Shot	Iterative		Policy	One-Shot	Iterative	
General X-VLA	general robot data	**LIBERO-Pro**	0.0	3.5	**29.4**	+29.4	0.1	3.5	**24.1**	+24.0
		VLABench	7.1	14.8	**17.5**	+10.4	7.5	17.3	**21.6**	+14.2
		SimplerEnv WidowX	0.0	65.8	**68.3**	+68.3	0.0	45.8	**24.0**	+24.0
		SimplerEnv Google	0.0	22.0	**51.2**	+51.2	0.0	20.0	**45.6**	+45.6
		Mean over all benchmarks	1.8	26.5	**41.6**	+2228%	1.9	21.6	**28.8**	+1417%
Specialist X-VLA	finetuned on each benchmark	**LIBERO-Pro**	82.9	83.2	**82.8**	-0.1	8.4	8.1	**8.0**	-0.4
		VLABench	13.4	14.9	**23.6**	+10.2	30.4	33.5	**35.1**	+4.7
		SimplerEnv WidowX	80.0	80.0	**80.8**	+0.8	72.9	72.9	**74.0**	+1.0
		SimplerEnv Google	55.9	55.9	**55.4**	-0.4	71.9	71.7	**70.9**	-1.0
		Mean over all benchmarks	58.0	58.5	**60.7**	+5%	45.9	46.5	**47.0**	+2%

the revisions themselves: control routines work from **measured geometry and the robot's own sensing**, not from static object names or positions written into the code (§4.3), and memories are often indexed by **measured physical dimensions** (§4.4). These results show that the domain gap traditionally closed through demonstration collection (Kim et al., 2024; Intelligence et al., 2025) can also be closed through automated harness design.

The harness dynamically balances policy and programmatic control. In several setups, the evolved harness bypasses the frozen policy entirely on benchmarks a general policy was never trained for. For example, the general X-VLA on SimplerEnv WidowX commands a fully open gripper at every step, so the harness learns to **disregard the policy entirely** and evolves into a vision-based code-as-policy agent (Fig. 6i; details in §B). On its optimization episodes, this harness raised success from 0% to 100% (Figs. 10 and 11), and achieved 87.5% on unseen scenes and 66.7% on unseen tasks. In contrast, the unified harness evolved on finetuned policies (§4.2) **keeps the VLA specialists in charge and intervenes only when they fail**. As shown in Fig. 7a, if late in an episode the gripper under the VLA policy's control has never approached the instructed object, the harness takes over via



(a) MolmoAct-2 on LIBERO-Pro-10, *"put both the cream cheese box and the butter in the basket"*: the VLA policy keeps driving into the juice carton until it tips over; the evolved harness detects the jam and lifts clear.

(b) X-VLA on VLABench, *"Please take the book contract_law_in_japan"*: the earlier harness defaults to top-down motion and collides repeatedly with the shelf board from above; the evolved harness turns the wrist and reaches in from the side.

(c) RLDX-1 on RoboCasa-365, *"Close the toaster oven door"*: the VLA policy leans on the toaster-oven door too gently and it stays open; the evolved harness rescales the action magnitude and latches the door shut.

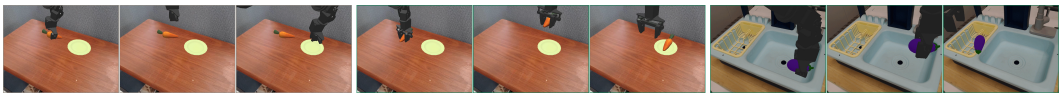
(d) RLinf-$\pi_{0.5}$ on LIBERO-Pro-10, *"turn on the stove and put the moka pot on it"*: the VLA policy closes on the frying pan standing at the moka pot's usual location memorized in the original LIBERO training data; the evolved harness locates the named object moka pot.

Figure 3: **Evolved control routines fix concrete failure patterns.** Each row shows an episode under an earlier harness (starting with bare VLA policy) on the left, and under the revised RoboArchitect harness on the right. Row (a) shows detecting and clearing arm jams, (b) synthesizing missing motion primitives, (c) rescaling contact forces, and (d) redirecting to target objects instead of memorized locations. Fig. 5 shows more examples. See discussions on these drivers in §4.3 and §B.

the pointing tool, and directly commands the arm to reach and grasp it. The harness also learns to hand control back to the VLA policy if it is already closing in on the object (Fig. 7b). Most evolved harnesses settle between these two ends: **the policy retains the motion it executes well, while the code takes over the decisions the policy gets wrong**, such as lifting a jammed arm clear before re-querying the policy (Fig. 3a), or carrying the gripper to the named object before the policy grasps it (Fig. 3d). Crucially, we never instructed the coding model how to divide control; each harness arrived at its structure from rollout evidence alone.

A unified harness works across robots. We next test whether a single harness can generalize across embodiments (§4.1; results in Tab. 2, full evolution trajectories in Fig. 12). The general X-VLA model scores near zero across all benchmarks, as it is entirely out of distribution (Tab. 2). The harness then evolves into a pure code-as-policy agent, replacing the general VLA entirely (as in §4.2). It raises every benchmark from (near) zero, reaching 41.6% on the training pool and 28.8% on held-out evaluation. This unified harness reads everything that differs between robots at runtime from the action specification and proprioception, never hardcoding embodiment-specific constants (§4.3 and §4.4). It thus **operates across embodiments without domain-specific finetuning**. Using finetuned specialists as base policies, the unified harness yields smaller gains. These policies already perform well (58.0% training, 45.9% held-out), and so our auto-evolved harness **correctly preserves their native competence**. As such, the unified harness strikes the same balance as individual setups (§4.2): it evolves into a pure code-as-policy agent when wrapping the general VLA, but correctly preserves native competence when wrapping finetuned specialists.

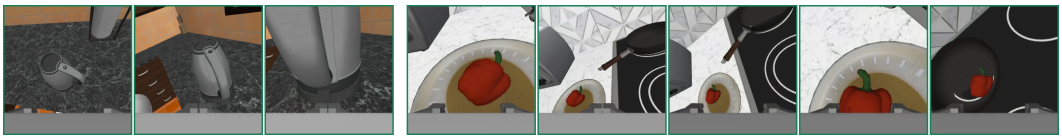
Iterative refinement further improves the automatically designed harness. A coding model can already design an agent from scratch in one attempt. But the agent itself generates new evidence through its own execution, and the same optimization process of RoboArchitect can use that



(a) X-VLA on SimplerEnv WidowX, *"put carrot on plate"* (left and middle) and *"put eggplant into yellow basket"* (right): the memory records the fractions of object height that successfully lifted the object; closing at 18% (raised to a 0.8 cm minimum) on the carrot lifts nothing, while closing at 45% of the object height (1.4 cm on the carrot, 3.9 cm on the eggplant) lifts both.



(b) OpenVLA-7B on LIBERO-Pro-10, *"put the white mug on the plate and put the chocolate pudding to the right of the plate"*: the memory records how often each grip style (pose) has worked for each object; the white mug is **spread from the inside** and stays upright, while the pudding box is **pinched**.



(c) RLDX-1 on RoboCasa-365, *"**press** down the lever to turn on the electric kettle"* (left) and *"**pick** the bell pepper from the plate and place it in the pan"* (right): the memory records that "pressing" tasks rarely *hold* objects while "picking" tasks do; the harness keeps pressing on the lever with closed fingers, but backs off and retries when the fingers close on nothing over the pepper.



(d) General X-VLA with the unified harness on SimplerEnv Google, *"pick coke can"* (left and middle) and on LIBERO-Pro, *"pick the akita black bowl on the cookie box and place it on the plate"* (right): the memory records the direction in which each robot's fingers close, and the harness turns the wrist accordingly; the Google robot closes its fingers across the can whether it lies sideways (left) or points toward the camera (middle), while the Franka arm grips the bowl's rim with one finger inside and one outside, following its correct wrist orientation.

Figure 4: **Persistent memories emerge from harness evolution and effectively inform action decisions.** Each row illustrates one form of memory mechanism; rows (a) and (b) show reusable execution experience, (c) task and environment context, and (d) embodiment-specific calibration. See discussions on these memories in §4.4 and §B.

evidence to refine it further. The harness after the first iteration (*One-Shot* in Tabs. 1 and 2) represents a **one-shot agent design**: the coding model sees the bare VLA policy's rollout failures (iteration 0) and produces an agent in a single attempt (iteration 1). Subsequent iterations observe the behavior of this agent and refine its control logic based on newly exposed failure modes (§3), up to iteration 5 (*Iterative* in Tabs. 1 and 2). Figs. 10 to 12 illustrate these optimization trajectories, showing **steady, mostly monotonic progression** over iterations. As shown in Tab. 1, the iteratively evolved harness outperforms the one-shot design on the training set in every setup (Figs. 10 and 11) and on held-out sets, except RLDX-1 on RoboCasa-365 under unseen scenes. For example, the code-as-policy agent on VLABench is built in its first attempt, and only its own rollouts reveal subtler failures, which later revisions repair (Fig. 6g,h; details in §B). **Iterative refinement also improves generalization**: while some one-shot designs fall below the bare VLA policy on unseen scenes or tasks, the iteratively evolved harness surpasses it on nearly every held-out set (Tab. 1). The improvement is not limited to adding new tools: later revisions also refine the division of control between the policy and programmatic logic, as discussed in §4.2. These results show that the harness is not merely automatically designed, but can **iteratively self-improve** by inspecting the execution traces of its own previous designs.

4.3 Evolved Control Routines Driving the Performance Gains

The evolved harnesses address concrete failure modes with targeted control routines. Figs. 3 and 6 show representative revisions across benchmarks. The harnesses learn to **detect and clear arm jams** that the policy cannot recover from (Fig. 3a), and to **synthesize missing motion primitives**, such as a side approach under a shelf board or a handle pull on a cabinet (Fig. 3b, Fig. 5e). On RoboCasa-365, they **rescale contact forces** when the policy pushes a door too gently (Fig. 3c), and **tether the mobile base** to active workspaces when it drifts away (Fig. 5f). Other routines make perception more reliable:

the harness redirects the arm to the target object when a policy overfit to its training scenes reaches for a memorized location (Fig. 3d), and **adapts to perception limitations** by working down ranked candidates and checking for firm grasps from depth (Fig. 6g,h). The unified harness also **adapts to each robot's setup**, such as its speed: it ends each move when the arm arrives, not on a fixed time budget (Fig. 7c), which contributed to raising the training-pool success rate on the slower Google robot from 11.0% to 30.2%. **§B details each routine.**

4.4 Persistent Memory Emerges from Harness Evolution

Across nearly all setups, RoboArchitect's evolved harnesses develop persistent memory structures that carry experience across episodes. Tab. 3 lists representative memory entries and how often each held-out episode reads the memory. Rather than following any predefined memory schema, RoboArchitect discovers diverse forms of memories that capture **reusable execution experience, task and environment context, and embodiment-specific calibration**.

Memories **learn from the harness's own grasps**: one records the *grasp heights* that lifted objects, relative to each object's measured height, and **reuses** them on new objects (Fig. 4a); another remembers the *grip style* (*pinch* or *spread*) that held each object (Fig. 4b). Indexed by measured physical properties or object identity, this experience carries over to novel objects and new tasks. Memories also track **implicit task and environment clues**, such as which instructions need *holding* objects (Fig. 4c) and the 150-step scene-reset period of RoboCerebra. The unified harness **calibrates itself to each robot**, recording which way the robot's fingers close to orient later grasps (Fig. 4d). **§B details each type of memory**.

Table 3: **Auto-designed persistent memories are frequently accessed and yield highly useful information to guide later actions.** *Reads per Episode*: average frequency of memory reads in held-out episodes. *Actionable Reads*: the share of these reads that find stored info.

Setting	Memory Entries	Reads per Episode	Actionable Reads
X-VLA on VLABench	finger depths that have been lifting objects	3.4	98%
X-VLA on SimplerEnv WidowX	height at which to close the fingers, per measured object height	3.5	96%
MolmoAct-2 on LIBERO-Pro-10	average depth of successful grasps below an object's top	9.9	99%
$\pi_{0.5}$ on RoboCerebra	scene-reset period of this simulation	13.0	95%
RLDX-1 on RoboCasa-365	task instructions that need a grip	11.0	92%

Importantly, these memories are **data structures within the designed robot agent**, not the coding model's own context: they accumulate information during the agent's rollouts and persist across episodes at execution time. As shown in Tab. 3, the harnesses **actively consult their memory** many times per episode on average, and **over 90% of these reads are actionable**, that is, they successfully find a stored entry, even on held-out episodes unseen during harness optimization. Because every evaluation starts with an empty memory to be accumulated, these entries' designs effectively carry over to held-out episodes and remain highly useful.

5 Conclusion

We introduced RoboArchitect to automate the design and optimization of agent harnesses around frozen robot policies. Across diverse benchmarks and frozen policies, the evolved harnesses substantially improve performance on held-out scenes and tasks. Even when the policy is untuned for the target environment, RoboArchitect can adapt it to a specialist in out-of-domain settings, and can even outperform its finetuned version. Crucially, the harness continues to self-improve beyond its initial design, using the behavior of previous versions as new evidence for subsequent refinements. Emergent from harness optimization, we observe intuitive control routines that resolve specific failure modes, and persistent memories that carry experience across episodes to inform later decisions. RoboArchitect can also evolve a unified harness that works across multiple embodiments, adapting itself to each robot. Together, these results suggest that substantial headroom in robot manipulation resides in the agent harness, and that this headroom can be unlocked by automatically evolving the harness for novel setups.

AI USE STATEMENT

In this work, we used generative AI tools to autonomously design and implement agents as an object of our study: every harness revision reported here was written by `claude-opus-5` inside the optimization loop of §3. We also used generative AI tools to orchestrate the split of the benchmarks' tasks into training and held-out sets, and the procedural generation of the unified harness's training pool from the benchmarks' own scenes and objects. Additionally, we used generative AI tools to assist in the design of figures and tables in the paper, and to polish the writing of this paper.

REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Shmuel Berman, Michael Ilie, Jia Deng, and Daniel Freeman. How claude performs on robotics tasks. Anthropic Research Blog, `https://www.anthropic.com/research/claude-plays-robotics`, July 2026.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control, 2026. URL `https://arxiv.org/abs/2410.24164`.

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.

Qianshu Cai, Yonggang Zhang, Xianzhang Jia, Huajiang Zheng, Wei Xue, Jun Song, Xinmei Tian, and Yike Guo. Moss: Self-evolution through source-level rewriting in autonomous agent systems. *arXiv preprint arXiv:2605.22794*, 2026.

Nicolas Carion, Laura Gustafson, Yuan-Ting Hu, Shoubhik Debnath, Ronghang Hu, Didac Suris Coll-Vinent, Chaitanya Ryali, Kalyan Vasudev Alwala, Haitham Khedr, Andrew Huang, et al. Sam 3: Segment anything with concepts. In *International conference on learning representations*, volume 2026, pp. 138846–138923, 2026.

Kaiyuan Chen, Shuangyu Xie, Letian Fu, Justin Yu, William Pacini, Sandeep Bajamahal, Hudson Kim, Jaimyn Drake, Daehwa Kim, Haoru Xue, et al. Gap: A graph-as-policy multi-agent self-learning harness for variational automation tasks. *arXiv preprint arXiv:2607.05369*, 2026a.

Kang Chen, Zhihao Liu, Tonghe Zhang, Zhen Guo, Si Xu, Hao Lin, Hongzhi Zang, Xiang Li, Quanlu Zhang, Zhaofei Yu, Guoliang Fan, Tiejun Huang, Yu Wang, and Chao Yu. π_{RL}: Online rl fine-tuning for flow-based vision-language-action models, 2026b. URL `https://arxiv.org/abs/2510.25889`.

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*, 2026c.

Christopher Clark, Jieyu Zhang, Zixian Ma, Jae Sung Park, Rohun Tripathi, Sangho Lee, Mohammadreza Salehi, Jason Ren, Chris Dongjoo Kim, Yinuo Yang, et al. Molmo2: Open weights and data for vision-language models with video understanding and grounding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 28652–28668, 2026.

Karim Elmaaroufi, Justin Svegliato, Sarunas Kalade, Graham Schelle, Sanjit A Seshia, and Matei Zaharia. Rho: Your coding agent is secretly a roboticist. *arXiv preprint arXiv:2606.16458*, 2026.

Haoquan Fang, Jiafei Duan, Donovan Clay, Sam Wang, Shuo Liu, Weikai Huang, Xiang Fan, Wei-Chuan Tsai, Shirui Chen, Yi Ru Wang, et al. Molmoact2: Action reasoning models for real-world deployment. *arXiv preprint arXiv:2605.02881*, 2026.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, et al. Cap-x: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026.

Hongbo Gao, Zeyu Ni, Xin Wen, Siyu Xu, and Ruifeng Li. Agm: Achievement-grounded memory for closed-loop agents with frozen vla policies. *arXiv preprint arXiv:2608.29537*, 2026.

Songhao Han, Boxiang Qiu, Yue Liao, Siyuan Huang, Chen Gao, Shuicheng Yan, and Si Liu. Robocerebra: A large-scale benchmark for long-horizon robotic manipulation evaluation, 2025. URL https://arxiv.org/abs/2506.06677.

Jiaheng Hu, Mohit Shridhar, Caden Lu, Dhruv Shah, Hao-Tien Lewis Chiang, Jie Tan, and Annie Xie. What matters in orchestrating robot policies: A systematic study of hierarchical vla agents. *arXiv preprint arXiv:2606.10267*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023.

Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a vision-language-action model with open-world generalization, 2025. URL https://arxiv.org/abs/2504.16054.

Dongyoung Kim, Huiwon Jang, Myungkyu Koo, Suhyeok Jang, Taeyoung Kim, Beomjun Kim, Byungjun Yoon, Changsung Jang, Daewon Choi, Dongsu Han, et al. Rldx-1 technical report. *arXiv preprint arXiv:2605.03269*, 2026.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

Vaishak Kumar. Act-observe-rewrite: Multimodal coding agents as in-context policy learners for robot manipulation. *arXiv preprint arXiv:2603.04466*, 2026.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Xuanlin Li, Kyle Hsu, Jiayuan Gu, Karl Pertsch, Oier Mees, Homer Rich Walke, Chuyuan Fu, Ishikaa Lunawat, Isabel Sieh, Sean Kirmani, et al. Evaluating real-world robot manipulation policies in simulation. *arXiv preprint arXiv:2405.05941*, 2024.

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 *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Qing Jiang, Chunyuan Li, Jianwei Yang, Hang Su, et al. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. In *European conference on computer vision*, pp. 38–55. Springer, 2024.

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, 2026. URL https://arxiv.org/abs/2607.00272.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, et al. Eureka: Human-level reward design via coding large language models. In *International conference on learning Representations*, volume 2024, pp. 26516–26560, 2024.

Adithyavairavan Murali, Balakumar Sundaralingam, Yu-Wei Chao, Wentao Yuan, Jun Yamada, Mark Carlson, Fabio Ramos, Stan Birchfield, Dieter Fox, and Clemens Eppner. Graspgen: A diffusion-based framework for 6-dof grasping with on-generator training. *arXiv preprint arXiv:2507.13097*, 2025.

Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, Abhishek Joshi, Ajay Mandlekar, and Yuke Zhu. Robocasa: Large-scale simulation of everyday tasks for generalist robots. *arXiv preprint arXiv:2406.02523*, 2024.

Jeongeun Park, Juhan Park, Taekyung Kim, Sungjoon Choi, Dongyoon Han, and Sangdoo Yun. Retrieve, don't retrain: Extending vision language action models to new tasks at test time. *arXiv preprint arXiv:2606.15631*, 2026.

Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, et al. Smolvla: A vision-language-action model for affordable and efficient robotics. *arXiv preprint arXiv:2506.01844*, 2025.

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.

Bhavya Sukhija, Oliver Groth, Mohit Shridhar, Tim Hertweck, Michael Bloesch, Markus Wulfmeier, Abbas Abdolmaleki, and Martin Riedmiller. Eximo: Vlm guided exploration of vla policies. *arXiv preprint arXiv:2608.19891*, 2026.

Brian Y. Tsui, Alan Y. Fang, and Tiffany J. Hwu. Demonstration-free robotic control via llm agents, 2026. URL https://arxiv.org/abs/2601.20334.

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, et al. Evo-harness: Context-to-harness skill compilation for self-evolving agents. *arXiv preprint arXiv:2608.15071*, 2026.

Wenli Xiao, Jia Xie, Tonghe Zhang, Haotian Lin, Letian Fu, Haoru Xue, Jalen Lu, Yi Yang, Cunxi Dai, Zi Wang, et al. Enpire: Agentic robot policy self-improvement in the real world. *arXiv preprint arXiv:2606.19980*, 2026.

Shuangyu Xie, Kaiyuan Chen, and Ken Goldberg. Revisiting the "push-t" robot manipulation task with agentic robotics, 2026a. URL https://arxiv.org/abs/2608.18227.

Xinyi Xie, Zican Hu, Zhanyu Liu, Yicheng Dong, Wenhao Wu, Zhenhong Sun, Haoran Li, Chunlin Chen, Zhi Wang, and Pichao Wang. Look before you leap: Distilling tree search into action evaluation for frozen vla models. *arXiv preprint arXiv:2607.03751*, 2026b.

Bingxin Xu, Yuzhang Shang, and Emilio Ferrara. Don't drop the baton: Long-horizon robot manipulation via agentic subtask exploration and transition-aware memory. *arXiv preprint arXiv:2608.16889*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Hongzhi Zang, Mingjie Wei, Si Xu, Yongji Wu, Zhen Guo, Yuanqing Wang, Hao Lin, Peihong Wang, Liangzhi Shi, Yuqing Xie, et al. Rlinf-vla: A unified and efficient framework for reinforcement learning of vision-language-action models. *arXiv preprint arXiv:2510.06710*, 2025.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026b.

Jiachen Zhang, Junnan Nie, Junyi Lao, Wei Cheng, Chenghao Liu, Jiaxin Jiang, and Songfang Huang. What frozen vlas already know about success: A probing study of value-like structure in foundation robot policies. *arXiv preprint arXiv:2605.28527*, 2026c.

Junyi Zhang, Jiaxin Ge, Hanjun Yoo, Letian Fu, Zihan Yang, Yaowei Liu, Raj Saravanan, Shaofeng Yin, Justin Yu, Dantong Niu, et al. Playful agentic robot learning. *arXiv preprint arXiv:2606.19419*, 2026d.

Shiduo Zhang, Zhe Xu, Peiju Liu, Xiaopeng Yu, Yuan Li, Qinghui Gao, Zhaoye Fei, Zhangyue Yin, Zuxuan Wu, Yu-Gang Jiang, et al. Vlabench: A large-scale benchmark for language-conditioned robotics manipulation with long-horizon reasoning tasks. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 11142–11152. IEEE, 2025.

Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, Chunyang Zhu, Jiaxing Qiu, Yuchen Yan, Jiyuan Liu, Wenhao Tang, et al. Harness vla: Steering frozen vlas into reliable manipulation primitives via memory-guided agents. *arXiv preprint arXiv:2607.08448*, 2026e.

Jianchao Zhao, Huoren Yang, Yusong Hu, Yuyang Gao, Qiguan Ou, Cong Wan, SongLin Dong, Zhiheng Ma, and Yihong Gong. Retrieve-then-steer: Online success memory for test-time adaptation of generative vlas. *arXiv preprint arXiv:2605.10094*, 2026a.

Yunhao Zhao, Zhenyang Ni, Haoyang Chen, Ruohan Zhang, and Qi Zhu. Skills in weights, memory in code: Hybrid learning for memory-dependent robot manipulation. *arXiv preprint arXiv:2608.09410*, 2026b.

Jinliang Zheng, Jianxiong Li, Zhihao Wang, Dongxiu Liu, Xirui Kang, Yuchun Feng, Yinan Zheng, Jiayin Zou, Yilun Chen, Jia Zeng, et al. X-vla: Soft-prompted transformer as scalable cross-embodiment vision-language-action model. In *International Conference on Learning Representations*, volume 2026, pp. 60580–60606, 2026.

Tailin Zhou. Hierarchical self-improvement: A framework for task-specific evolvable agent harnesses. *arXiv preprint arXiv:2608.08466*, 2026.

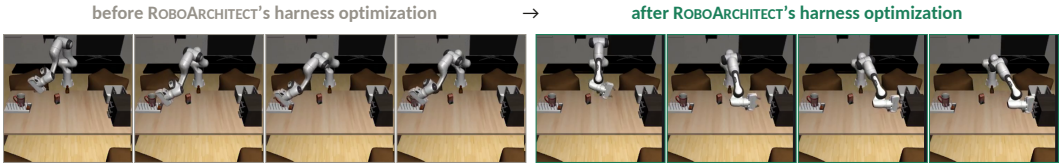
Xueyang Zhou, Yangming Xu, Guiyao Tie, Yongchao Chen, Guowen Zhang, Duanfeng Chu, Pan Zhou, and Lichao Sun. Libero-pro: Towards robust and fair evaluation of vision-language-action models beyond memorization. *arXiv preprint arXiv:2510.03827*, 2025.

A LIMITATIONS

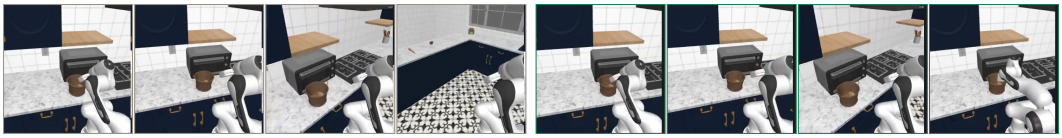
Scaling cap of iterations. As shown in Figs. 10 to 12, harness performance on the training episodes improves consistently across iterations. However, this work uses a fixed maximum of 5 optimization iterations (§4.1). More iterations may yield further gains, the point at which returns saturate remains to be explored. Nevertheless, even with a moderate budget of only 5 iterations (§C), ROBOARCHITECT achieves substantial gains that transfer to held-out settings.

More complex manipulation. A natural next step is extending ROBOARCHITECT to tasks with richer contact dynamics, such as dexterous in-hand manipulation or tool use, where the space of failure modes is larger than in pick-and-place. The range of control routines and memory structures that automated evolution can discover in these settings is an interesting open question.

Sim-to-real transfer. All evaluations run in simulation. Because our harness reads only the standard input space (*e.g.,* camera images, depth and the robot's own readings), its interface should carry over to a real robot. Testing our simulation-evolved harnesses on real robots and investigating the number of further improvement iterations on a real deployment are future work.

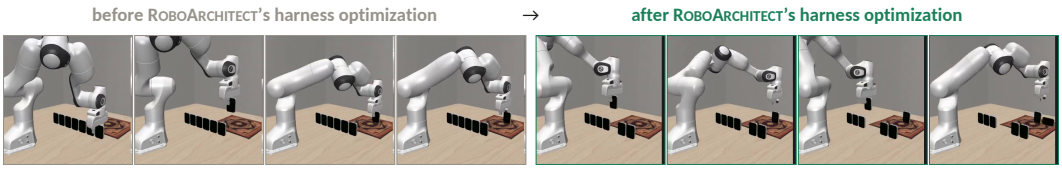


(e) SmolVLA on RoboCerebra, *"Open the wooden cabinet top region"*: the VLA policy fails entirely, missing required motions; the evolved harness correctly locates the cabinet, closes on the handle and pulls the drawer out.

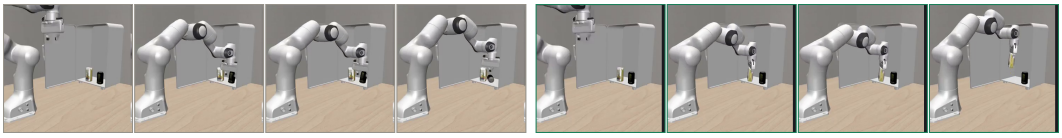


(f) RLDX-1 on RoboCasa-365, *"Open the cabinet, pick up the bread from the cabinet and place it in the basket. Then move the basket to the dining counter."*: the VLA policy drifts the base away and never returns to the counter; the evolved harness detects the drift and retraces back to the counter.

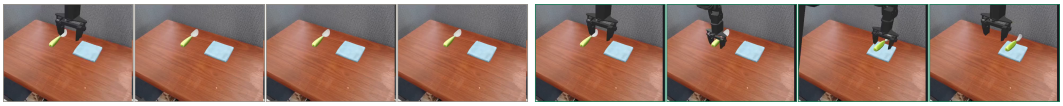
Figure 5: **More evolved control routines.** Rows read as in Fig. 3. Row (e) shows synthesizing missing motion primitives, and (f) tethering mobile bases to active workspaces.



(g) X-VLA on VLABench, *"Pick up the mahjong of 1 pin"*: the earlier harness commits to just a single tile; the evolved harness works down its ranked candidate list of multiple tiles.



(h) X-VLA on VLABench, *"It's a hot day and I could use some refreshment"*: the earlier harness judges its grip empty after lifting the drink and puts it down; the evolved harness verifies grasps from the depth image under the fingertips and keeps hold of the drink.



(i) X-VLA on SimplerEnv WidowX, *"put the spoon on the towel"*: the VLA policy (un-tuned and out of distribution) fails entirely; RoboArchitect replaces it with a code-as-policy agent that picks and places the spoon correctly.

Figure 6: **Evolved harnesses also enhance the robotic agent's perception precision and reliability.** Panels read as in Fig. 3, discussions in §4.3. Rows (g) and (h) show adapting to perception limitations, and (i) replacing an unusable policy with a code-as-policy agent (§4.2).

B Evolved Control Routines and Persistent Memories in Detail

This appendix details RoboArchitect's auto-evolved control routines of §4.3 and the persistent memories of §4.4; Figs. 5 to 7 show more of the revisions.

B.1 Control Routines

Detecting and clearing arm jams. When the arm is jammed against an object, the bare policy may continue issuing similar commands because its visual observations remain nearly unchanged during a jam, so it tips objects over and cannot recover. The evolved harness learns to monitor discrepancies between commanded and actual gripper movements; when disagreement persists, it lifts the arm clear before re-querying the policy (Fig. 3a).

Synthesizing missing motion primitives. Some tasks require motion primitives absent from the base policy or the earlier harness, such as approaching an object from the side rather than from above. For retrieving a book from the shelf on VLABench, the earlier harness defaults to top-down motion

and collides with the overhead board. The evolved harness detects overhead obstructions via depth maps and computes horizontal approach vectors to reach beneath them (Fig. 3b). The same pattern appears elsewhere: on RoboCerebra, the harness adds a wrist-rotation primitive for pouring with $\pi_{0.5}$, while for cabinet opening with SmolVLA, it locates the cabinet with the pointing model, finds its handles in the depth image, then aligns and pulls (Fig. 5e).

Rescaling contact forces. On RoboCasa-365, some failures arise from **insufficient contact force** rather than trajectory misalignment. For example, the toaster-oven door requires closure within 0.5% of its opening range, but the policy RLDX-1 pushed too gently. The evolved harness rescales action magnitudes while preserving direction, providing sufficient force to latch the door (Fig. 3c).

Tethering mobile bases to active workspaces. On RoboCasa-365, the harness also learns to constrain **mobile-base drift** around active workspaces. It defines workspace anchors as locations where the last gripper activations occurred, gives the base a driving budget, and retraces the base to the latest anchor once the budget is spent (Fig. 5f).

Redirecting to target objects instead of memorized locations. For spatially shifted scenes such as LIBERO-Pro, policies overfit on original LIBERO consistently reach toward wrong items at memorized locations (Zhou et al., 2025). ROBOARCHITECT independently diagnosed this memorization, and evolved the harness to query the pointing model for the target item. With RLinf-$\pi_{0.5}$, it learns to locate the instructed moka pot rather than reaching to its memorized position (Fig. 3d).

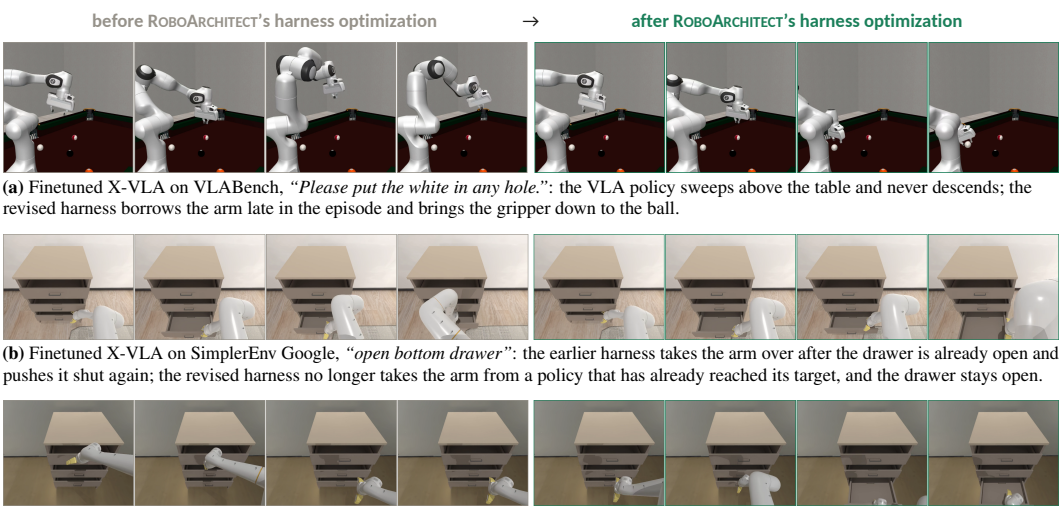
Replacing an unusable VLA policy. The base (un-finetuned) X-VLA on SimplerEnv WidowX commanded a fully open gripper at every step, producing no useful control signal. ROBOARCHITECT learned to **disregard the policy entirely**, as its commands do not change with the robot's observations. Instead, the harness evolved into a complete vision-based picking agent, akin to code-as-policy (Liang et al., 2023; Tsui et al., 2026; Elmaaroufi et al., 2026): parsing instructions, detecting target objects via open-vocabulary detectors, estimating 3D poses from depth maps, computing grasp vectors, and executing trajectories directly (Fig. 6i).

Adapting to perception limitations. When the VLA policy is unusable, the harness evolves into a pure code-as-policy agent (§4.2). Crucially, such agents continue to **refine their own perception and control mechanisms** across iterations (§4.2). On VLABench, the vision tools could not reliably distinguish mahjong tile faces. Rather than relying on a single vision prediction, the harness maintains a ranked candidate list, sequentially interacting with candidate tiles until the task completes (Fig. 6g). Vision-based grasp verification could also be unreliable, causing the agent to prematurely put down objects it was holding. The harness thus replaces **visual grasp checks** with **a depth check under the fingertips**, verifying grasps by the presence of a surface just below the fingers (Fig. 6h).

Per-robot control-rate independence. The unified harness (§4.2) evolves mechanisms that adapt control to each robot across platforms (see Fig. 7). Each motion phase used to end on a fixed time budget, which on a slower platform (*e.g.,* the 3 Hz Google robot, Fig. 7c) expires while the arm is still in transit. The evolved harness ends a move only when the arm arrives or stops making progress, measured by the distance between the arm's position and its target rather than by elapsed time, and reaches handles on vertical surfaces from the front along the measured surface normal. The revision that introduced these changes raised the agent's performance on the Google robot from 11.0% to 30.2% on the training pool.

B.2 PERSISTENT MEMORIES

Learning execution parameters from experience. The harnesses store empirical execution statistics that inform later control decisions. On SimplerEnv WidowX, the harness records the grasp heights that have successfully lifted an object, as a function of measured object height, so that effective grasp parameters transfer to novel objects (Fig. 4a). On LIBERO-Pro-10, the MolmoAct-2 harness instead averages the depth of successful grasps below the object top, as a share of the object's height, and uses this statistic to parameterize fallback grasps. The OpenVLA-7B harness similarly records the grip style, *pinch* or *spread*, that succeeds for each object, so future grasps select the proven pose when the object's shape allows both (Fig. 4b). Crucially, across these designs, memory is indexed by **actual, measured physical properties or object identity** rather than specific task names, so experience generalizes to novel objects across different tasks.



(a) Finetuned X-VLA on VLABench, *"Please put the white in any hole."*: the VLA policy sweeps above the table and never descends; the revised harness borrows the arm late in the episode and brings the gripper down to the ball.

(b) Finetuned X-VLA on SimplerEnv Google, *"open bottom drawer"*: the earlier harness takes the arm over after the drawer is already open and pushes it shut again; the revised harness no longer takes the arm from a policy that has already reached its target, and the drawer stays open.

(c) General X-VLA on SimplerEnv Google, *"open bottom drawer"*: the earlier harness presses the hand onto the drawer front and times out on a clock set for a faster robot; the revised harness ends moves on arrival and reaches the handle from the front along the surface normal.

Figure 7: **The unified harness learns to take over the arm command from a failing policy and to leave a progressing one alone, and calibrates itself to each robot's configurations.** Each row shows an episode under the harness before (left) and after (right) an iteration's revision.

Tracking implicit task and environment clues. Persistent memory also captures recurring clues in the task instruction and environment. On RoboCasa-365, the harness associates instruction verbs with the need to *hold ("pick up")* an object: once sufficient evidence accumulates for a *"picking"* task, it enables an *empty-grasp recovery routine* that retries when fingers close on nothing, while leaving the routine disabled for *"pressing"* tasks where a closed empty gripper is intentional (Fig. 4c). On RoboCerebra, the harness discovers and stores the environment's 150-step scene-reset period, allowing later episodes to anticipate resets and take over a placement early enough to finish it before the reset. Memories are also used to cache inference outputs, such as VLM/LLM parses of multi-step instructions, so that repeated executions avoid redundant computation.

Persistent self-calibration across embodiments. In the unified harness (§4.2), persistent memory additionally stores embodiment-specific calibration. Because finger motion differs across robots, the harness opens and closes the empty gripper, observes the resulting motion in the depth image, and **stores the kinematic mapping in memory**. This calibration guides grasp orientation across different embodiments (*e.g.,* on the Franka arm and on the Google robot) (Fig. 4d). The same routine records the resting finger configuration, providing a reference for detecting empty grasps on robots reporting finger positions.

C Implementation Details

Episode splits. Fig. 8 shows the split of a benchmark's episodes under the *unseen scenes* and *unseen tasks* protocols, and Fig. 9 shows the training pool and the held-out episodes of the unified harness (§4.1).

Cost of automated harness optimization. A complete evolution of 5 iterations (§4.1) takes under 6 hours of coding-model time and stays under $150 on average, with no demonstrations collected and no model weights updated. Through RoboArchitect, the engineering loop that is slow by hand (§1) becomes an inexpensive routine easily repeatable for every new robot setup.

Benchmarks. LIBERO-Pro-10 evaluates long-horizon task sequences from LIBERO (Liu et al., 2023) with initial object locations swapped while preserving original natural language instructions. These spatial swaps follow the LIBERO-Pro benchmark protocol (Zhou et al., 2025) designed to distinguish genuine manipulation skill from spatial memorization via overfitting on LIBERO. RoboCerebra (Han et al., 2025) tests on a sequence of short manipulation tasks with scene resets every

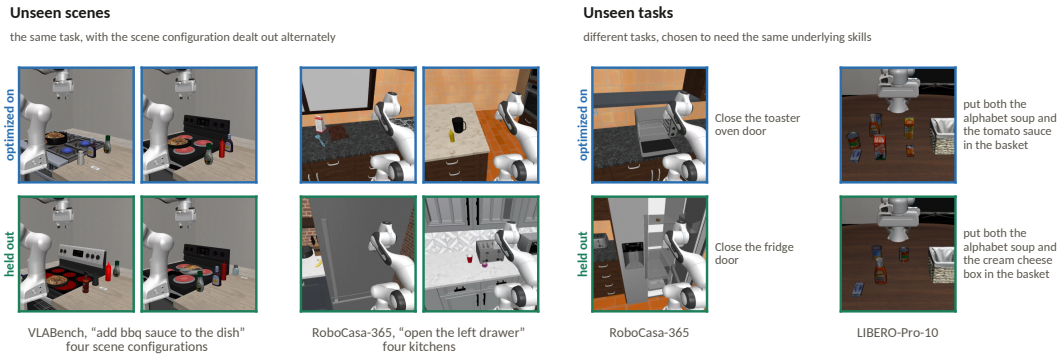


Figure 8: **Held-out episodes differ from the optimization episodes either in scene arrangement (*unseen scenes*) or in the task itself (*unseen tasks*)** (§4.1). *Left, unseen scenes:* the same tasks with scene configurations dealt out alternately. *Right, unseen tasks:* different tasks that require similar underlying skills.

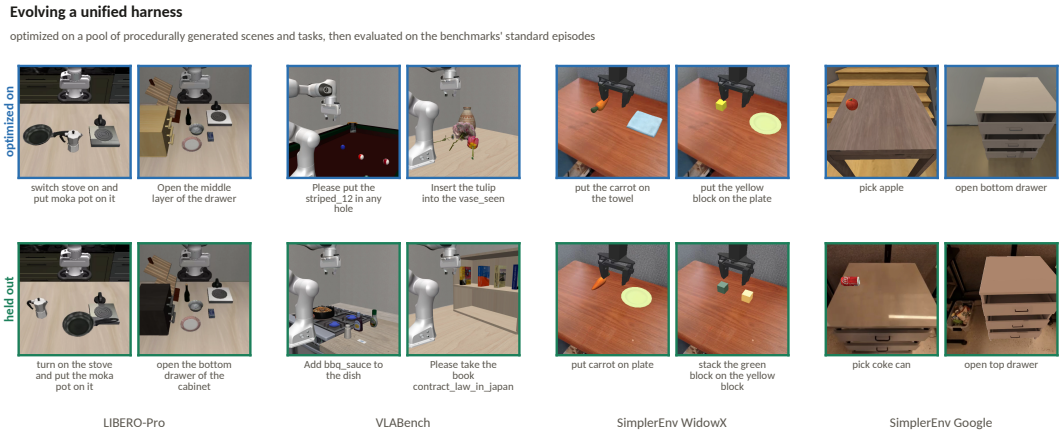


Figure 9: **The unified harness is evolved only on a training pool of procedurally generated episodes and evaluated on the benchmarks' own standard episodes** (§4.1 and §4.2). *Top, optimized on:* procedurally generated scenes, objects, instructions & tasks form the training pool. *Bottom, held out:* the benchmarks' own standard episodes, unseen during harness evolution.

150 control steps, challenging the ability to handle sudden environment changes. VLABench (Zhang et al., 2025) requires identifying specific target items among visually similar distractors with a focus on visual grounding. RoboCasa-365 (Nasiriany et al., 2024) places a mobile manipulator in varied kitchen environments, requiring coordinated mobile navigation and arm control. SimplerEnv WidowX (Li et al., 2024) deploys a WidowX manipulator within photorealistic scenes, testing cross-embodiment transfer under visual domain shift. SimplerEnv Google (Li et al., 2024) deploys the Google mobile manipulator under the same visual domain shift as SimplerEnv WidowX, and is included in the test bed of the unified harness (§4.1).

Optimization ("training") and held-out tasks under the *unseen tasks* protocol. Tab. 4 lists, for each benchmark, the tasks used for the harness evolution and the held-out tasks under the *unseen tasks* protocol (§4.1). The held-out tasks share underlying manipulation skills with the harness optimization tasks. The task split of a benchmark is shared by all the policies evaluated on it (Tab. 1). Every task is run for the same number of trials (*i.e.,* "episodes"), as listed next to each benchmark. The unified harness (§4.2) is evolved on a training pool of 2,201 procedurally generated episodes, and evaluated on the 2,160 standard episodes.

Episode length. An episode ends when its task succeeds or when it reaches the step limit. The step limits are: 520 steps on LIBERO-Pro-10, 150 per sub-task on RoboCerebra (135 for the first, so 585 to 2,235 for a whole chain), 200 on VLABench (100 for the poker-card tasks), 1,200 on SimplerEnv WidowX, and a limit set per task on RoboCasa-365, from 300 to 2,900. The unified harness also runs the *-goal*, *-object* and *-spatial* suites of LIBERO-Pro, with 300, 280 and 220 steps, and SimplerEnv Google, with 160 steps for picking a can and moving an object near another, 226 for opening or closing a drawer, and 400 for placing an apple in a closed drawer. On SimplerEnv WidowX and SimplerEnv Google, episodes always run to the limit and are scored on their final state. All the scores we compare, including those of the finetuned references, are evaluated under these same limits.

Table 4: **Optimization and held-out tasks under the *unseen tasks* protocol.** On RoboCerebra a task is a chain of sub-task instructions stated here in brief. On VLABench and RoboCasa-365 the instruction of a task changes with the sampled objects. *Trials* is the number of trials (*i.e.,* episodes) per task.

Benchmark	Trials	Optimization Tasks	Held-Out Tasks
LIBERO-Pro-10	10	*turn on the stove and put the moka pot on it*; *put the black bowl in the bottom drawer of the cabinet and close it*; *put both the alphabet soup and the tomato sauce in the basket*; *put both the cream cheese box and the butter in the basket*; *put the white mug on the left plate and put the yellow and white mug on the right plate*	*put the yellow and white mug in the microwave and close it*; *put both moka pots on the stove*; *put both the alphabet soup and the cream cheese box in the basket*; *put the white mug on the plate and put the chocolate pudding to the right of the plate*
RoboCerebra	10	cream cheese, popcorn and butter, each into a white storage box; cream cheese into a white storage box, then the frypan onto the rack; plate, popcorn and cookies rearranged on the dining set, then wine poured onto the plate; chocolate pudding from the cabinet top onto the plate, then wine poured onto the plate; chocolate pudding and butter into a black bowl, the stove turned on, then the frypan onto the stove	cookies into a basket, tomato sauce and bbq sauce poured onto the frypan, wine poured into a mug, then another mug into a wooden tray; cream cheese and popcorn into a white bowl, then the frypan onto the cabinet top; the short cabinet opened, butter into the cabinet, then tomato sauce poured onto the plate; the short cabinet opened, cream cheese into the cabinet, chocolate pudding into a black bowl, then wine poured into a mug; white bowl from the cabinet top, cookies and chocolate pudding into the bowl, the microwave opened, then the bowl into the microwave
VLABench	10	select the named fruit; select the named mahjong tile; select the named poker card	select the named drink out of a fridge; select the n-th largest poker card; select the mahjong tile of the unique type
SimplerEnv WidowX	24	*put the spoon on the towel*; *stack the green block on the yellow block*	*put carrot on plate*; *put eggplant into yellow basket*
RoboCasa-365	50	close the toaster oven door; open the cabinet door; open the drawer; pick an object from the counter and place it in the cabinet; press the start button on the microwave	close the fridge door; pick an object from the plate and place it in the pan on the stove; fully slide the top dishwasher rack out; press down the lever to turn on the electric kettle

Policy checkpoints. All the policies are evaluated from their public checkpoints, as listed in Tab. 5.

Table 5: **Every frozen policy is a public checkpoint, used without any weight update** (§4.1).

Policy	Checkpoint	Used In
MolmoAct-2	`allenai/MolmoAct2-LIBERO`	Tab. 1
OpenVLA-7B	`openvla/openvla-7b-finetuned-libero-10`	Tab. 1
$\pi_{0.5}$	`lerobot/pi05_libero_finetuned`	Tab. 1
RLinf-$\pi_{0.5}$	`RLinf/RLinf-Pi05-LIBERO-130-fullshot-SFT`	Tab. 1
SmolVLA	`lerobot/smolvla_robocerebra`	Tab. 1
RLDX-1	`RLWRLD/RLDX-1-FT-RC365`	Tab. 1
X-VLA (general)	`2toINF/X-VLA-Pt`	Tabs. 1 and 2
X-VLA finetuned on VLABench	`2toINF/X-VLA-VLABench`	Tabs. 1 and 2
X-VLA finetuned on WidowX	`2toINF/X-VLA-WidowX`	Tabs. 1 and 2
X-VLA finetuned on LIBERO	`2toINF/X-VLA-Libero`	Tab. 2
X-VLA finetuned on Google Robot	`2toINF/X-VLA-Google-Robot`	Tab. 2

D Harness Evolution Trajectories

Figs. 10 to 12 show the complete optimization update chain for every experimental setting under both held-out evaluation protocols (§4.1) and the unified agent design (§4.2).

E Prompt Template Given to Coding Model

Listing 1 gives the exact prompt that the coding model receives at every iteration of the optimization loop (§3). Crucially, the prompt prescribes no agent structure, leaves the coding model free to keep or drop the VLA policy (§4.2), and does not specify the contents of the persistent memory (§4.4).

Listing 1: Prompt template given to the coding model

```
You are a robotics engineer. Your job is to improve one Python program, `scaffold.py`,
which is the robot's control agent: every step of an episode it receives what the robot can
see and returns the numbers that move the robot. An episode is one attempt at one task,
from reset until the robot succeeds or runs out of steps.

Several frozen models are available to `scaffold.py`. One is a robot policy, trained to
turn a camera picture plus a task sentence into robot motion; the others see or read. You
cannot change any of them and you must not train or fine-tune anything. How you use them,
and whether you use them at all, is yours to decide: the numbers you return can come from a
model, from arithmetic you wrote, or from any mixture.

Everything between the observation arriving and the action leaving is yours to change.

## What to aim at
Make the robot succeed on more episodes. Work out what is going wrong first, from the
recordings described below, and only then change the code. Say what the failure is before
you fix it.

Whatever you change must be a general improvement to the robot agent. It has to keep
working on tasks and scenes you have never seen, not just the ones in these recordings.

How much you change is your call -- one line or a rewrite. Write a short note of what you
changed and why into `../agent_workspace/CHANGES.md`: what you saw in the recordings, what
you concluded, and what you did about it. Finish by saying the same thing in your last
message.

## The one file that runs, and where everything else goes
`scaffold.py` in your current directory is the whole robot agent, and it is the only file
that runs. Edit it in place. Keep the agent in that one file only.

Keep `create_scaffold()`, the `SCAFFOLD_CONFIG` dict and the `act(request, tools)`
signature. `SCAFFOLD_CONFIG` keeps its two lists, and no name may be in both. A name in
`required_capabilities` is loaded every time. A name in `optional_capabilities` is loaded
only when your file mentions it. A name in neither is never loaded and calling it fails.
`"vla"` is the robot policy and follows the same rule, so move it to
`optional_capabilities` if you stop calling it.

Everything else you produce -- notes, analysis scripts, plots, extracted data, rendered
pictures -- goes in `../agent_workspace/`. That folder is yours, put scratch files there
rather than in `/tmp`.
```

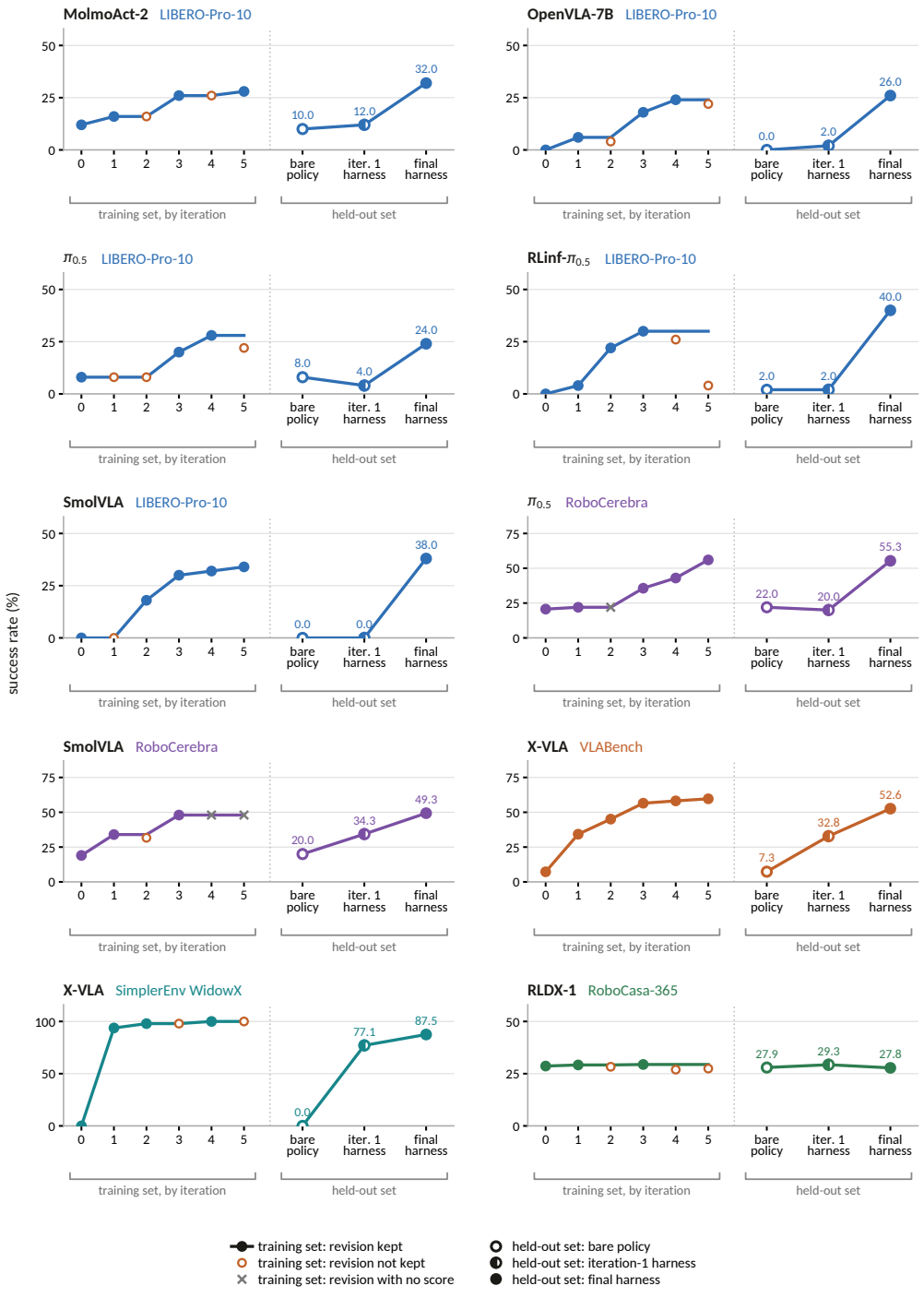


Figure 10: **Under *unseen scenes*, the harness improves over iterations on the training set, and the final harness outperforms the one-shot harness on nearly every held-out set.** Left of the dotted line is the training episode set, iteration by iteration; iteration 0 is the bare VLA policy, filled markers denote accepted harness revisions that improved task success over prior iterations, and open markers denote revisions that were not accepted. Right of the dotted line are the scores on the held-out set: the bare policy, the harness after iteration 1 (a single-attempt agent design) and the final iteratively refined harness. The held-out set evaluates identical tasks under different scene arrangements (§4.1).

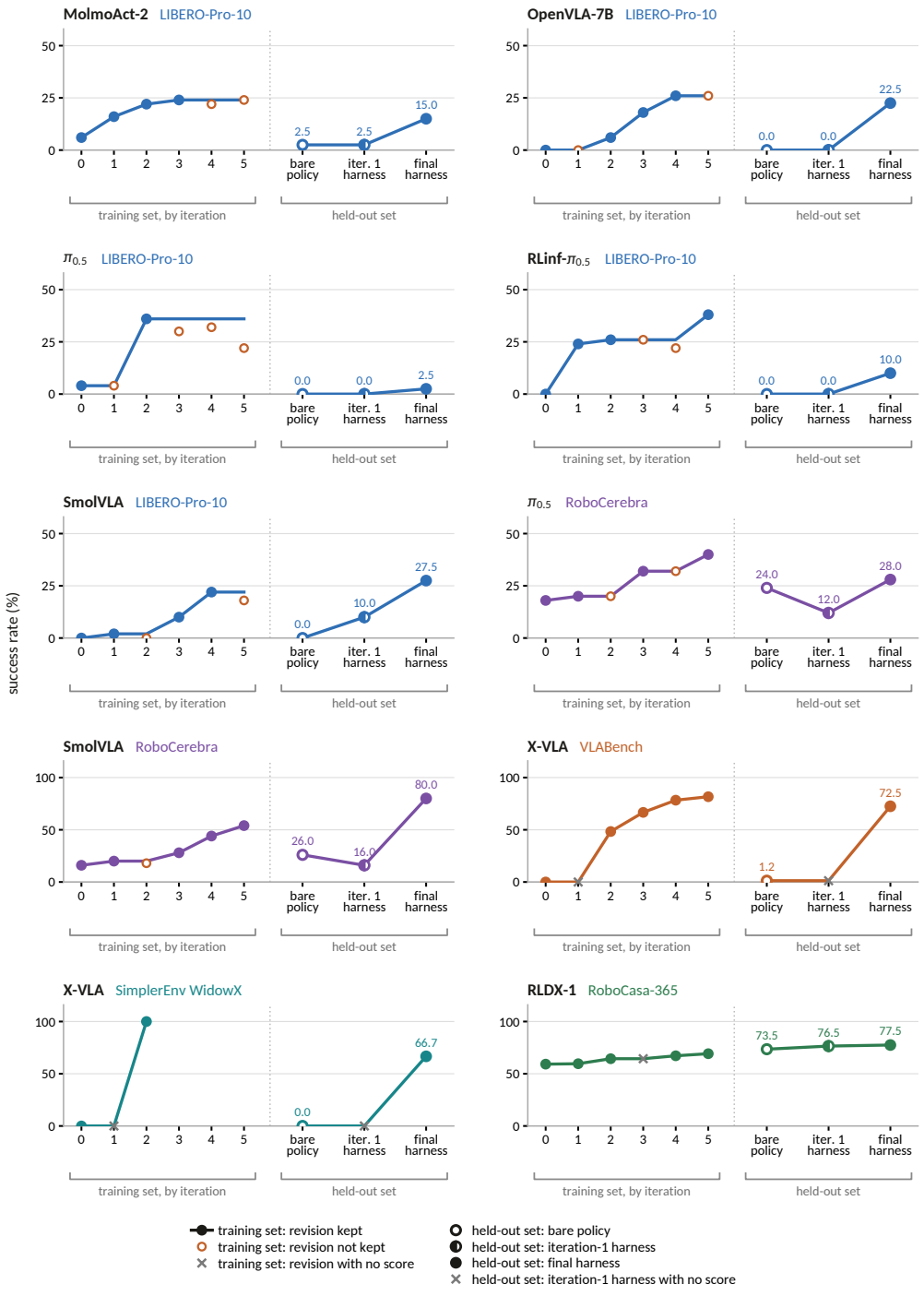


Figure 11: **Under *unseen tasks*, the harness improves over iterations on the training set, and the final harness outperforms the bare VLA policy and the one-shot harness on every held-out set.** Labels and panels read as in Fig. 10. A cross marks a revision that could not be evaluated; when this is the first revision, no single-attempt harness is available for held-out evaluation. The held-out set evaluates unseen tasks requiring shared underlying skills (§4.1).

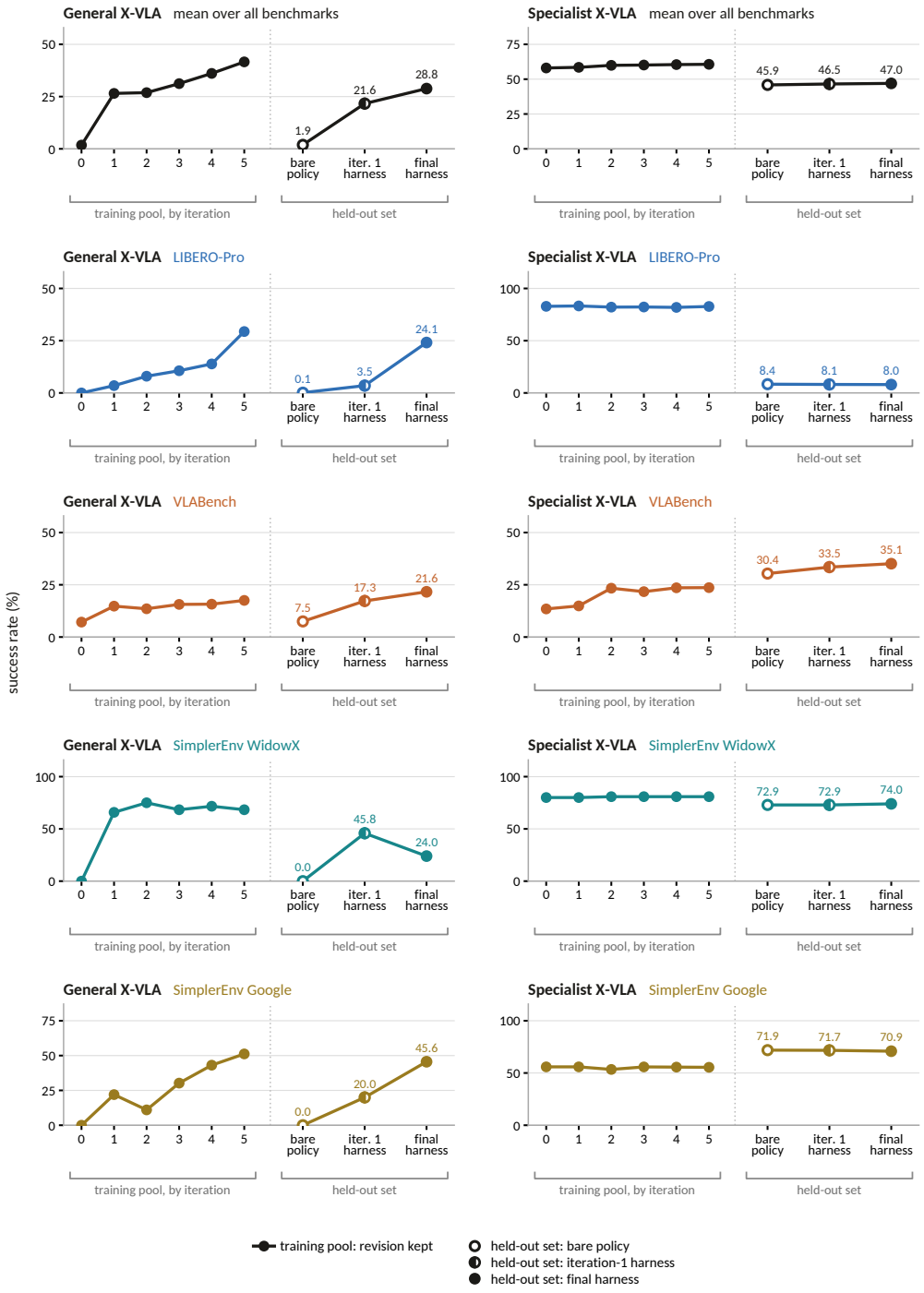


Figure 12: **Around the general X-VLA, the unified harness improves on every benchmark of the training pool over iterations; around the finetuned specialists, it leaves their already competent scores intact.** Left column, using the general X-VLA checkpoint as policy; right column, using benchmark-finetuned X-VLA specialists as policy. The first row is the mean over the benchmarks and each later row is one benchmark. Left of the dotted line shows the performance on the procedurally generated training pool, iteration by iteration (iteration 0 is the bare policy; filled markers denote accepted harness revisions). Right of it are the scores on the benchmark's standard episodes (unseen in the training pool): the bare policy, the harness after iteration 1 (a single-attempt agent design) and the final iteratively refined harness.

```
Python values stored on the scaffold object may or may not survive into the next episode.
Clear anything episode-specific in `reset(session_id)`. The one supported way to carry
something from one episode to the next is the memory described under "Memory that carries
across episodes" below.

## The interface (read `scaffold.py` first)
`act(request, tools)` is called once per step and must return exactly one
`CanonicalActionChunk`. Its `spec` must be this setup's own action spec. Its `start_step`
must equal `request.observation.step_index`. Its values must obey this setup's limits,
which the last section lists. The class is defined in `⟨package⟩/protocol/action.py`.

`request.observation` is everything the robot can see this step:
  - `.instruction` -- the task sentence, word for word as the task gives it.
  - `.cameras` -- a mapping from camera name to a camera picture.
  - `.proprioception` -- the robot's own readings, typically its gripper pose and how open
the gripper is. Which entries exist depends on the setup. Look one up by name with
`.by_name(...)`, or read `.vectors`, where each entry has a `.spec` describing what its
numbers mean and `.values` holding them.
  - `.step_index` -- which step of the episode this is.
There is nothing else in it. No object positions, no goal, no score.

Each camera has `.rgb` (uint8, height x width x 3). Where this setup has depth sensing it
also has `.depth_m` (float32, height x width, in metres, one value per colour pixel),
`.depth_valid`, `.intrinsics` (3x3) and `.camera_to_world` (4x4). Those four together let
you turn a pixel into a point in the robot's own coordinates.

The exact definitions are in `⟨package⟩/agent/api.py` (the request and result types),
`⟨package⟩/agent/tools.py` (the `tools` object) and `⟨package⟩/protocol/observation.py` (the
observation). Read them rather than guessing.

## The models you can call
Every call goes through the `tools` object. `⟨package⟩/agent/tools.py` is the implementation
and `⟨package⟩/agent/api.py` holds every request and result type, with the exact fields and
the checks each one applies. Read both before writing a call.

- `tools.vla(...)` -- the robot policy. You can build the whole agent around it, use it on
some steps only, or never call it at all. Three of its arguments are worth knowing about
before you read the file:
  - `instruction=` is the sentence the policy is given. The policy service feeds it to the
model only on a step where it actually runs the model, so changing the text on any other
step does nothing until then. See `refresh` next, and read `⟨package⟩/policies/` for the
service this setup uses.
  - `refresh=True` throws away whatever the policy service had left over and makes it run
the model again on this step's picture. The service runs the model once, gets several
future actions back, and hands them out one per step; that is why a new instruction can sit
unused for several steps. A refresh is a change in behaviour, not just a recomputation:
anything the discarded actions were carrying is gone with them.
  - `context=` is a tuple of strings sent to the policy service. It is ignored unless the
last section says this setup's policy reads it.
  - If you want the policy to know something, it has to be in the instruction text or in
the observation.

- `tools.vision(...)` -- a text model that can see pictures. Describe a scene, compare two
views, judge whether something is done.

- `tools.language(...)` -- a text model. It cannot see pictures.

- `tools.detect(...)` -- an object detector. Give it a picture and words; it returns boxes
with labels and scores.

- `tools.point(...)` -- a pointer. Give it a picture and an instruction; it returns pixel
coordinates.

- `tools.segment(...)` -- a segmenter. Give it a picture and at least one box or point; it
returns outlines.

Check with `tools.has("detection")` and so on before calling; the last section lists which
models are available here. Calling one that is not running raises `ToolUnavailableError`,
and so does a call that fails for any other reason, such as a timeout or a malformed
request. Any exception that escapes `act` ends that episode and it counts as a failure, so
guard your calls.

`tools.record(event_type, status, detail, capability, result)` writes a line into this
episode's recording. Every model call is recorded for you, answer included, so you do not
have to log those yourself -- use `record` for your own decisions: what you concluded, what
you chose to do, and why.

## Two ready-made modules you can import
Your scaffold runs with this project on its import path, so `from ⟨package⟩...` works. numpy
and PIL are available; no simulator package and no PyYAML are, so `⟨package⟩.benchmarks.*`
```

```
will not import. Two modules are there for you to use, and both are plain arithmetic with
no hidden state.

- `⟨package⟩.agent.geometry` -- turns a pixel into a point in the robot's own coordinates,
and back. Call `has_3d(camera)` first: it is False where this setup has no depth. Then
`depth_at`, `pixel_to_world`, `world_to_pixel`, `point_cloud`. `pixel_to_world` returns
`None` rather than raising when there is no usable depth at that pixel, so check the
result. The module already accounts for which way up this setup stores its picture.

- `⟨package⟩.agent.motion` -- ready-made movement commands you can return instead of a
model's answer. `make_controller(spec)` gives you `move_to`, `nudge`, `hold` and
`set_gripper`; each returns a float32 array of action numbers for one step, which you wrap
in a `CanonicalActionChunk` yourself. `make_controller` returns `None` where the action
layout is not one it can drive, so check for that; the last section says whether it can
drive this setup. On a setup whose action is a change from the current pose these commands
leave the wrist rotation alone; on a setup whose action is a target pose they reuse the
rotation from the last chunk you passed to `controller.note(chunk)`.

These commands are a convenience, not a limit. You may also compute the action numbers
yourself and return them, as long as the chunk carries this setup's own spec and its values
obey this setup's limits.
Read both modules before using them.

## If you call the policy, call it on every step
The policy service remembers which step it last produced an action for and refuses a
request that jumps ahead, with `policy_act: previous action is not observed as executed`.
Do not try to remove that check.

So there are exactly two shapes that work: call `tools.vla` on every step, and on any step
return something else instead of its action if you want to -- or never call it at all for
the whole episode. Mixing the two, by calling it, skipping some steps and calling it again,
fails every episode. Throwing away an action you asked for costs only the time it took to
compute.

None of this applies to the other models. You can call or skip those freely, on any step.

## The recordings are your evidence
`../public_input.json` gives you the folders that hold them.
  - `incumbent_episode_traces_dir` -- the recordings of the `scaffold.py` you have now,
running the whole episode set.
  - `previous_rejected_candidate_episode_traces_dir` -- present only sometimes. It is a
change that was already tried and did not do better, with its own full recordings in the
same shape.

Read them yourself with Bash, Grep and Read, and analyse them with python. Each episode
folder holds:
  - `trace.jsonl` : line 0 names the task, says whether the episode succeeded and how it
ended, and lists every camera with its size and the picture file that holds it. Every later
line is one step, with `step`, `instruction` (what the policy was given), `action` (the
numbers that were returned), `state` (the robot's own readings that step: gripper pose, and
where the setup reports it, how far apart the fingers are), `depth` (per camera: how much
of the depth picture was usable, and its smallest, middle and largest distance) and
`events`.
  - Each event is one thing that happened that step. A model call carries `result`, which
holds what that model actually answered -- the detector's boxes with labels and scores, the
pointer's pixels, a summary of each mask, the text a text model wrote. Events your scaffold
wrote with `tools.record(...)` appear the same way.
  - `camera-<name>.mp4` : every rendered frame of that camera for the whole episode. Pull
any step out with ffmpeg, for example `ffmpeg -i camera-main.mp4 -vf "select=eq(n\\,63)"
-vsync 0 -vframes 1 ../agent_workspace/step63.png`, and then open the PNG as an image. Line
0 of the trace names the file for each camera. These are compressed, so a pixel can be off
by a shade.
  - `episode.json` : whether that episode succeeded, how many steps it took, and exactly
which task, scene and seeds it used.
  - `private_metrics.json` : extra measurements for that episode, on the setups that
produce any. Many produce none, so do not assume this file exists.

## Running the simulator yourself
You can re-run any recorded episode in the simulator and look at whatever you like. It
reads the task, scene and seeds from `episode.json` and the actions from `trace.jsonl`, so
it repeats the episode that was recorded. The last section gives the exact command for this
setup.

## Rules your scaffold has to follow
- While you work you may read anything you need: the recordings, the simulator source, the
policy code. Keep to this project and the paths `runtime_paths.json` names; nothing else on
this machine is part of your task. But the scaffold you ship must not read the simulator's
own answers while an episode is running -- no live simulator state, no true object or goal
positions, no success check, no expert actions. In practice that means none of
`_check_success`, `sim.data`, `body_xpos`, `site_xpos`, `geom_xpos`. Do not smuggle the
answer into the scaffold.
```

- Do not write a solution for the particular tasks in these recordings. A general rule names no particular task and no particular object, and would behave the same way on a task you have never seen. A lookup does not: a table keyed by task name or id, a coordinate you read off a recording and typed in, a special case for one episode. In particular, a place you move the gripper to must be worked out from this episode's own observation -- from what the camera sees, from depth, from the robot's own readings -- and never typed in as a number. A constant offset is a different thing and is fine: `nudge(observation, (0, 0, 0.03))` means "lift three centimetres from wherever the gripper is now", which behaves the same on any task.

- Do not change how success is decided or how the result is counted. That is not in your file, and reaching for it will not work.

Memory that carries across episodes
Your scaffold has a memory that survives from one episode to the next.

- `tools.remember(key, value)` stores one value under one name. `key` is a string. `value` may be any mixture of numbers, strings, true/false, null, lists and dictionaries whose keys are strings; an array or an image is not one of those. Storing the same name again replaces what was there.
- `tools.recall(key)` returns what was stored under that name, or `None` if nothing was.
- `tools.memory_keys()` lists the names in use.
- `tools.forget(key)` removes one.

All the episodes of one scoring run -- one whole pass over the episode set -- share one memory, and it begins empty: the first episodes fill it and the later ones can use it. Episodes run several at a time, so you cannot count on the order in which they write. Nothing carries over into the next scoring run.

The memory is saved as `memory.json` beside the recordings, and `../public_input.json` gives the exact paths under `incumbent_memory_file` and, where there is one, `previous_rejected_candidate_memory_file`. Read it. The recordings tell you what the robot did; this tells you what it remembered, how many times it was read, and how many of those reads found anything.

Whether to use the memory at all, what to put in it, and when to read it are yours to decide. What you store follows the same rule as everything else: it must be worked out from what the robot could see, and it must not be an answer written down for a particular task.

Facts about THIS setup
- Models available on this setup: ⟨*available auxiliary tools*⟩. One is loaded only if `SCAFFOLD_CONFIG` lists it and your file mentions it, so name in a `tools.<name>(...)` call, a `tools.has("<name>")` check or one of its request types every model you intend to use. Check with `tools.has(...)` before calling.

- Never available here (tools.has(...) is False even if you call it, and calling one raises): ⟨*unavailable tools*⟩.

- Cameras: ⟨*cameras*⟩. Depth is on for: ⟨*cameras with depth*⟩. Those cameras carry .depth_m (metres, one value per colour pixel), .depth_valid, .intrinsics and .camera_to_world. Use ⟨*package*⟩.agent.geometry rather than projecting by hand.

- The robot is ⟨*robot embodiment*⟩.

- The action the policy returns, and the action you must return: ⟨*action layout of this robot*⟩.

- ⟨*further facts about this setup*⟩

- To re-run an episode in the simulator yourself, this is the command for this setup. Fill in the episode folder from `../public_input.json`:

  ⟨*replay command*⟩

  It writes one PNG per camera per step, optionally the depth as `.npy`, and a `replay.json` saying whether the episode succeeded. `--steps` stops early; leave it out for the whole episode. Point `--actions` at a different `trace.jsonl` to run a DIFFERENT action sequence in the same scene, which is how you check what would have happened if the robot had done something else.

- How the per-episode outcomes become this setup's single number is in ⟨*package*⟩/evaluation/scalars.py, under `⟨*benchmark metric*⟩`. Read it: it decides whether a gain on one task can pay for a loss on another.