

HarnessPAI: An Evolving Harness for Physical AI

Darwin Agent Team

See [Contributions and Acknowledgments](#) section for a full author list.

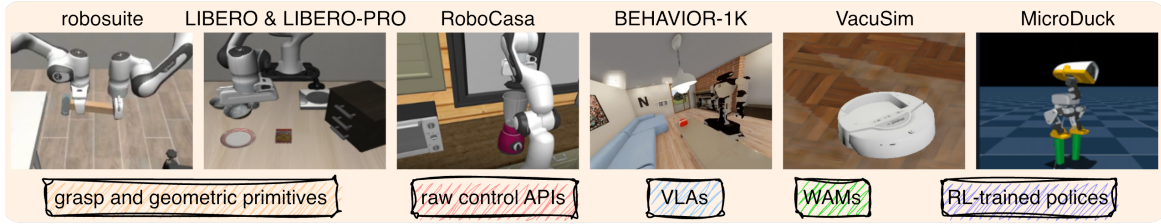


Abstract

Physical AI aims to build embodied agents that *perceive* the world, *understand and reason* about it, and decide how to *act*. Yet the field has focused primarily on the last component: the *action model* that maps observations to low-level controls. The prevailing training recipe can erode the perceptual and reasoning capabilities needed for robust behavior, leaving even strong action models vulnerable to scene perturbations and long-horizon tasks. We introduce **HarnessPAI**, a model- and embodiment-agnostic **Harness** framework for **Physical AI** that treats code as the executable and evolvable interface that organizes the underlying action primitive. The framework separates two timescales: within a rollout, it executes *open-loop* at the program level, with a fixed program guiding and checking execution; across rollouts, it evolves *closed-loop*, using execution feedback to revise the program and distill failures into reusable skills. Across desktop robot arms, household robots, a robot vacuum, and a legged walking agent, **HarnessPAI** improves on both pure action models and code-as-policy baselines without retraining the underlying model: a **61.6-point** gain over $\pi_{0.5}$ on LIBERO-PRO and a **27.2-point** gain over WorldDreamer on RoboCasa atomic tasks. Once a program is selected, rollout execution requires no online high-level LLM deliberation. Beyond execution, the converged program is also a cheap and reliable expert-data collector, and fine-tuning $\pi_{0.5}$ on collected expert data lifts success rate on LIBERO-PRO by **38.8 points**. Our results suggest that the frontier of Physical AI depends not only on stronger action models, but also on executable harnesses that integrate perception, task understanding and reasoning, and action execution into a unified, verifiable, and feedback-driven system.

Website: <https://darwin-agent.github.io/HarnessPAI>.

7 Experimental Settings · 4 Embodiment Categories · 5 Downstream Interfaces



HarnessPAI: An Evolving Harness for Physical AI



Figure 1 Overview of **HarnessPAI**, a general physical harness framework.

Contents

1	Introduction	3
2	Related Work	4
2.1	Post-training VLAs and WAMs	4
2.2	Code as Policy	5
2.3	Harnesses for VLAs and WAMs	5
2.4	Sim-to-Real Transfer and Real-Robot Validation	6
3	Motivation	6
3.1	Why Do VLA/WAM Models Need a Harness?	7
3.2	Why Open-Loop Execution?	9
3.3	Why Use Code as a Physical AI Harness?	10
4	Framework	11
4.1	Open-Loop Execution	11
4.2	Closed-Loop Evolution	12
4.3	Evolution Mechanism	13
4.4	Skill Library	15
4.5	Model Feedback	17
5	Experiment	18
5.1	Experimental Setup	18
5.2	Main Results	19
5.3	Harness Zero-shot	24
5.4	Evolution Process	25
5.5	Cost	27
5.6	Harness Trajectories for Post-Training	27
6	Discussion	29
6.1	Why the Framework Works	29
6.2	When the Framework Works	30
6.3	Where the Expert Data Comes From	30
7	Conclusion	30
8	Future Work	31
	Contributions and Acknowledgments	36
A	Main Benchmark Results Table	37
B	Detailed Results	37
B.1	LIBERO	37
B.2	LIBERO-PRO Perturbations	37
B.3	RoboCasa	43
B.4	Harness Zero-shot	44
C	Skills Case	44

1 Introduction

Physical AI seeks to build embodied agents that can *perceive* their surroundings, *understand and reason* about their goals, and *act* effectively in the physical world (Chen et al., 2026c, Ding et al., 2026, Fu et al., 2026, Lu et al., 2026, Zhang et al., 2026b). Recent foundation models have substantially advanced the first two capabilities, but closing the loop through action remains the hardest step, because physical behavior must satisfy constraints imposed by geometry, dynamics, contact, and safety over extended horizons and under uncertainty. This challenge has made learned action models a central focus of Physical AI (Intelligence et al., 2025b, Kim et al., 2024, Ye et al., 2026).

Two complementary architectures have emerged as prominent approaches to learned action. Vision–Language–Action models (VLAs) map visual observations and language instructions directly to robot actions (Black et al., 2024, Brohan et al., 2023, Intelligence et al., 2025b, Kim et al., 2024, NVIDIA et al., 2025, Zitkovich et al., 2023), while World Action Models (WAMs) additionally model how the environment evolves under those actions (Cen et al., 2025, Ye et al., 2026). Their complementary strengths point toward a unified architecture that combines semantic grounding, world prediction, and action generation. Strategies for adapting these models to downstream physical tasks broadly follow two directions. Post-training updates model parameters using additional demonstrations or interaction. However, fine-tuning can narrow the learned action distribution and hurt generalization, and its gains remain bounded by the base model itself. Their performance remains constrained by the scale, diversity, and quality of available embodied data, as well as by the difficulty of optimizing semantic understanding and physical control jointly. Indeed, even VLAs that approach perfect success on standard benchmarks can fall sharply under controlled scene and task perturbations (Zhou et al., 2025). Harness-based approaches instead keep the action model fixed and improve the system around it through external perception, reasoning, orchestration, and verification (Li et al., 2026, Zhang et al., 2026b). We take the latter view, treating the learned model as an action primitive organized and complemented by an executable harness.

In language and multimodal AI, harnesses have turned foundation models into capable agents by organizing tools, memory, reasoning, and iterative execution (Chen et al., 2026a, Liu et al., 2026a, Qiao et al., 2026, Wu et al., 2026). Recent approaches such as Harness VLA and RoboHarness extend this systems perspective to Physical AI, retaining a frozen action model as a low-level primitive and placing memory, task decomposition, skill selection, and failure recovery around it, which preserves learned motor competence while making high-level behavior more adaptive (Li et al., 2026, Zhang et al., 2026b). However, current physical harnesses still rely heavily on online reasoning and memory accumulation: an agent must repeatedly interpret observations, retrieve experience, and decide what to do, even when executing the same task again. Executable code offers a natural representation for consolidating and reusing this system-level knowledge, yet code-as-policy approaches replace learned manipulation with hand-designed APIs, sacrificing contact-rich control (Fu et al., 2026, Liang et al., 2023, Lu et al., 2026). What is missing is an executable harness that retains the learned action model, compiles perception, reasoning, action-model invocation, and verification into reusable programs, and evolves those programs from physical feedback across rollouts.

We introduce **HarnessPAI**, a model- and embodiment-agnostic harness framework for Physical AI that treats code as the executable and evolvable interface organizing the underlying action primitive. The framework separates execution and improvement into two complementary timescales. Within each rollout, it performs *open-loop execution*: an agent instantiates an executable program fixing the task decomposition, perception calls, geometric reasoning, action-model invocations, and success checks, which then runs directly without further high-level deliberation, delegating contact-rich manipulation to the underlying action model. Across rollouts, it performs *closed-loop evolution*: observations, execution traces, failures, and outcomes are collected as physical feedback to diagnose errors, revise the program, and distill reusable failure-to-repair into a graph structured skill memory. Code is therefore both an action interface and an evolvable representation of agent behavior—readable enough to diagnose, modular enough to edit, and executable enough to ground each revision in the physical environment. We further show that the evolved program attains high reliability and quality on tasks of the same type, and serves as a reliable expert-data collector whose successful trajectories improve the wrapped action model itself.

Compared with recent physical harnesses, **HarnessPAI** changes the unit of adaptation from transient online reasoning to persistent executable behavior. Existing systems often retrieve memories, replan, or intervene through repeated agent calls during deployment. **HarnessPAI** instead compiles successful reasoning into code, validates it across scenes, and reuses the resulting program when the task recurs. This removes online deliberation from the critical path and amortizes its cost across executions, while preserving the contact-rich motor competence of learned action models.

Rapid advances in general-purpose agents make this distinction increasingly important. A frontier model may reason, write code, and invoke available tools, but its success still depends on those tools being grounded in physical perception, embodiment-specific control, and reliable feedback. Physical actions are continuous, contact-rich, and often difficult to reverse, requiring an execution substrate that is fast, reproducible, and explicitly verifiable. **HarnessPAI** provides this substrate. Stronger models can improve code generation and diagnosis during evolution, but they do not replace the validated programs, physical primitives, and accumulated repair knowledge needed at deployment. **HarnessPAI** therefore complements advances in foundation models by turning transient model reasoning into persistent physical capability.

We evaluate **HarnessPAI** in seven experimental settings spanning four broad embodiment categories—robot arms, a humanoid upper body on a mobile base, a mobile robot vacuum, and a legged walking agent. It improves substantially on the action model it wraps in every one of them. On LIBERO-PRO, where policies above 90% on standard LIBERO fall sharply under perturbation, it holds 96.5%, 8.8 points above the strongest harness baseline. Fine-tuning $\pi_{0.5}$ on the self-collected trajectories lifts LIBERO-PRO success from 34.9% to 73.7%: the harness does not only improve the system around the model, it also improves the model itself. Our core contributions are:

- **Executable harness abstraction.** We introduce **HarnessPAI**, a model- and embodiment-agnostic framework that organizes perception, task-level reasoning, verification, and learned physical actions through executable programs.
- **Two-timescale evolution.** We develop a two-timescale execution and evolution procedure: programs execute without online high-level revision within a rollout and are revised across rollouts using execution feedback.
- **Structured skill memory.** We introduce a node-centric skill memory that stores failure-to-repair knowledge and transfers it across tasks sharing the same workflow stages.
- **Empirical validation and model feedback.** We evaluate the framework across heterogeneous embodiments and action backends, and show that converged harness trajectories can improve the wrapped action model through post-training.

2 Related Work

Physical AI aims to build embodied agents that *perceive* the world, *understand and reason* about it, and decide how to *act*. Foundation models have largely supplied the first two capabilities, while effort has concentrated on the third: the *action model* that maps observations to low-level controls, and where the competence to act should live. Existing approaches differ primarily in where adaptation is stored: in model parameters, in executable programs, or in an external runtime harness.

2.1 Post-training VLAs and WAMs

A growing body of work brings the success of LLMs and VLMs to embodied control, yielding a diverse family of VLAs such as RT-1/RT-2 (Brohan et al., 2023, Zitkovich et al., 2023), OpenVLA (Kim et al., 2024), the π -series (Black et al., 2024, Intelligence et al., 2025a,b, 2026), GR00T (NVIDIA et al., 2025), and Octo (Team et al., 2024), which map visual observations and language instructions to robot actions. These models span several action-generation designs: OpenVLA autoregressively predicts discretized action tokens with an open 7B VLM backbone (Kim et al., 2024); π_0 and $\pi_{0.5}$ condition a flow-matching action expert on a pretrained vision-language backbone, with $\pi_{0.5}$ extending the setting toward open-world and

long-horizon manipulation (Black et al., 2024, Intelligence et al., 2025b); and GR00T N1 combines a vision-language reasoning module with a diffusion-transformer controller for generalist humanoid robotics (NVIDIA et al., 2025). Despite this progress, VLAs remain markedly more fragile than their LLM/VLM counterparts, frequently failing on tasks that lie only modestly outside their training distribution (Xie et al., 2026, Zhou et al., 2025). A complementary family of WAMs couples action generation with prediction of future visual states, using video prediction to learn the consequences of actions and to support long-horizon control (Cen et al., 2025, Cheang et al., 2024, Wu et al., 2024, Ye et al., 2026, Yuan et al., 2026, Zhu et al., 2025). WorldVLA jointly models action and future observations in an autoregressive action-observation formulation (Cen et al., 2025); DreamZero scales this idea with a pretrained video-diffusion backbone and jointly predicts visual futures and actions from heterogeneous robot data (Ye et al., 2026); and Fast-WAM shows that much of the benefit can come from video modeling during training, without rendering future frames at test time (Yuan et al., 2026). WAMs therefore enrich the action model with predictive physical structure, but their adaptation still occurs through training the model and its associated objectives. **HarnessPAI** is orthogonal to these model-side approaches: it keeps the action model fixed during harness evolution and stores task-specific adaptation in executable programs and reusable skills rather than in model parameters.

2.2 Code as Policy

In code-as-policy systems, the generated program serves as the primary task policy: it translates a natural-language instruction into a sequence of calls to perception, geometry, motion, or simulation APIs (Fu et al., 2026, Huang et al., 2023, Liang et al., 2023, Lu et al., 2026, Singh et al., 2023). This representation makes high-level behavior compositional and inspectable: a generated program can sequence primitives, branch on sensed state, and invoke reusable skills instead of emitting an undifferentiated action sequence. In the original code-as-policy formulation, language models generate Python policies that combine visual affordances with a library of low-level robot interfaces (Liang et al., 2023). Subsequent work has explored more structured skill composition and agentic program synthesis. CaP-X studies coding agents for robot manipulation as a benchmark and improvement problem, making the quality of generated code, tool use, and iterative debugging explicit evaluation axes (Fu et al., 2026). ASPIRE discovers and reuses robot skills through an agent that writes, executes, and self-corrects programs against an explicit skill library (Lu et al., 2026). However, these methods usually obtain low-level action competence from hand-designed primitives or direct control APIs. Such interfaces can provide precise geometric control, but they may lack the contact-rich robustness and learned adaptability of pretrained action models, especially when object contacts, intermediate states, or scene conditions differ from the assumed execution pattern. **HarnessPAI**, in contrast, uses code to organize, invoke, and verify a learned action model rather than relying solely on control APIs, preserving its physical competence while moving reusable perception, reasoning, and task logic into an evolvable executable harness with structured skill memory.

2.3 Harnesses for VLAs and WAMs

Harness methods preserve pretrained physical competence while adding an external layer for planning, memory, verification, and recovery (Chen et al., 2026b, Liu et al., 2026b, Xu et al., 2026, Zhang et al., 2026b). Harness VLA steers a frozen VLA through memory-guided task decomposition and skill selection, while Zetta likewise keeps the base policy frozen but self-evolves code-based runtime critics and recovery skills through action-, rollout-, and update-timescale loops (Ding et al., 2026, Zhang et al., 2026b). Show-Harness instead exposes semantic action units that a VLM selects directly and embodiment-specific interpreters deterministically ground into robot motion, enabling one interface across tasks and embodiments (Chen et al., 2026b). RoboHarness further combines experience retrieval, failure attribution, external intervention, and memory consolidation for in-context adaptation (Li et al., 2026). Extending harnesses to WAMs, Harness-WAM maintains an evidence-grounded scene belief and structured task graph to organize finite-horizon WAM skills, verify progress, and recover from failures in long-horizon tasks (Gu et al., 2026). Depending on the method, these systems may invoke online reasoning, memory retrieval, semantic action selection, or runtime critic-recovery loops during deployment. This can improve adaptability, but it also places high-level model inference on the execution path. **HarnessPAI** makes the complete task program the unit of adaptation: a validated program executes directly when the task recurs, while code evolution for new tasks retrieves reusable skills from structured skill memory and can transfer across VLA, WAM, and other action backends.

Method family	Adaptation location	Coding agent during rollout	API-based control	Learned policy usage	Evolution across rollouts
VLA/WAM post-training	Model parameter	✗	✗	✓	Training only
Code-as-policy	Program	✗	✓	✗	Sometimes
VLA harness	Runtime/memory	✓	✓	✓	Usually
HarnessPAI	Program/memory	✗	✓	✓	Designed for persistent evolution

Table 1 Qualitative comparison of action-model and executable-harness paradigms, corresponding to Fig. 2.

2.4 Sim-to-Real Transfer and Real-Robot Validation

Sim-to-real work addresses the reality gap through visual and dynamics randomization, real-data simulator adaptation, and real-to-sim-to-real calibration (Chebotar et al., 2019, Peng et al., 2018, Tobin et al., 2017, Torne et al., 2024); recent methods also use language models for transfer design and online correction, while SIMPLER evaluates sim-real correspondence (Jiang et al., 2025, Li et al., 2025, Ma et al., 2024). Unlike these policy-focused approaches, **HarnessPAI** must transfer executable assumptions about perception, geometry, verification, timing, and recovery under sensing error, latency, contact variation, and hardware faults, motivating safety-gated validation on real robots.

Figure 2 gives a visual overview of these paradigms; Table 1 summarizes how they differ in where adaptation resides and how execution is organized.

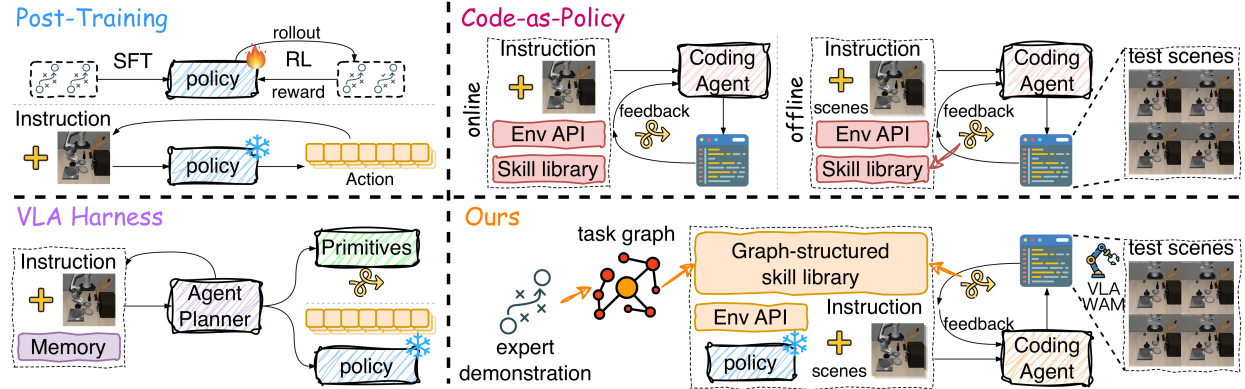


Figure 2 Our paradigm versus prior ones. (1) *Post-training*: an action model is trained on expert trajectories by SFT or by RL on a reward; at inference, the frozen model takes an instruction and image and outputs an action chunk. (2) *Code as policy*: an *online* variant evolves one program per task and seed, where the coding agent reads the instruction and image, calls APIs and the skill library, runs the code, revises it from the output, and evolves again; an *offline* variant instead evolves a single program over several selected seeds and runs that same program directly on scenes unseen during evolution. (3) *VLA harness*: a coding agent serves as the planner, receiving the image, instruction, and memory and deciding whether to call an environment primitive or the action-model policy. Our **HarnessPAI** instead first builds a task graph from expert trajectories, derives the code and skill-library architecture from that graph, and uses a coding agent as the planner that invokes VLA/WAM actions *through code*, so code manages perception, understanding and reasoning, and action execution; for one or many tasks it evolves code over multiple environment seeds and tests a single program on many unseen test scenes.

3 Motivation

The introduction argued that a large fraction of Physical AI research is organized around a single component—the *action model*. Before presenting **HarnessPAI**, we make three questions precise and show why the action-model-centric design answers each of them poorly: why a VLA or WAM is not, on its own, a sufficient physical agent (Section 3.1); why a physical agent should execute *open-loop* at the program level and aim

to succeed on the first attempt (Section 3.2); and why *code* is the right material in which to organize the missing capabilities (Section 3.3).

3.1 Why Do VLA/WAM Models Need a Harness?

Almost all recent VLA and WAM models are assembled the same way: a pretrained vision-language model (VLM) supplies semantic perception, and a separately trained *action head*—typically a diffusion, flow-matching, or diffusion-transformer model—maps the VLM’s representations into continuous action chunks or latent actions. The π_0 family learns a flow-matching action head on top of a PaliGemma vision-language backbone (Black et al., 2024, Intelligence et al., 2025b); GR00T N1 conditions a diffusion-transformer action head on vision-language tokens (NVIDIA et al., 2025); Octo fits a diffusion action head over a transformer backbone (Team et al., 2024); and WorldDreamer learns a World Action Model that jointly predicts future scene evolution and action chunks (Wang et al., 2024). Even the autoregressive alternatives—RT-2 and OpenVLA—follow the same division, casting actions as additional output tokens of a VLM (Kim et al., 2024, Zitkovich et al., 2023). Regardless of whether actions are emitted through diffusion denoising, flow matching, or discretized tokens. Despite architectural differences, these systems commonly expose an action-centric interface, with most downstream optimization focused on predicting or evaluating physical actions.

Action models learn demonstrated behavior through behavior cloning and can also be optimized for task rewards through reinforcement learning. However, strong action-task performance does not guarantee reliable target grounding and task completion when the instruction or scene changes. The following examples illustrate this limitation, but do not establish whether it results from degradation of vision and language capabilities during training. Fig. 3 shows $\pi_{0.5}$ succeeding in an in-distribution scene, while Figs. 4 and 5 show it failing under a language and a visual perturbation, respectively.

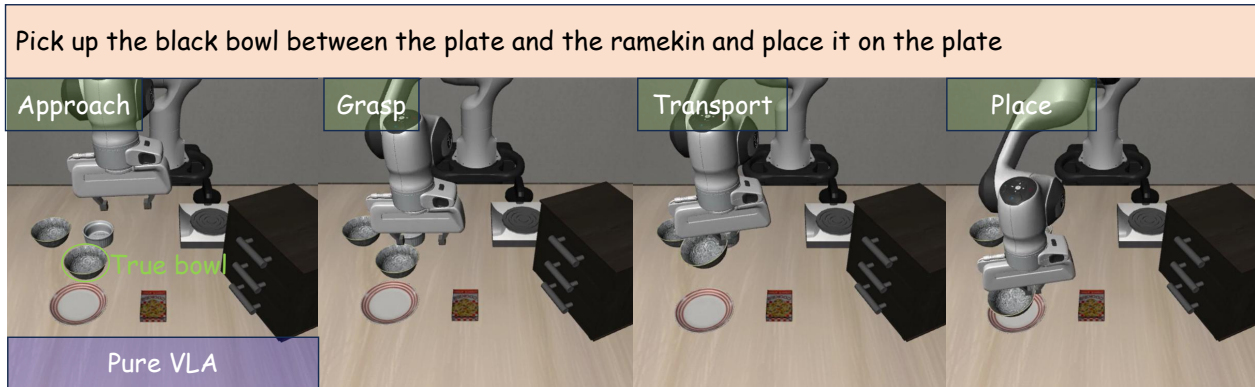


Figure 3 In-distribution success of a strong VLA ($\pi_{0.5}$) on LIBERO. Given the in-distribution instruction “pick up the black bowl between the plate and the ramekin,” the VLA approaches, grasps the correct bowl, transports it, and places it on the plate.

Fig. 4 illustrates a language perturbation. The demonstrated task is “pick up the black bowl between the plate and the ramekin.” Adding “not” before “between” flips the intended referent to a *different* bowl. A model that understood the instruction would re-derive the referent and grasp the correct object. $\pi_{0.5}$ does not: it still approaches and grasps the bowl between the plate and the ramekin despite the added negation. The “VLA + Mask” row shows that the illustrated masking intervention still does not enable the model to execute the modified instruction correctly. This particular masking intervention does not resolve the failure, suggesting that simply modifying the visual input is insufficient in this case. This indicates that the visual intervention is insufficient to resolve the instruction-following error, but does not by itself determine whether the error arises from language understanding, visual grounding, or the mapping from instructions to actions.

Fig. 5 illustrates a visual perturbation. Here the instruction is unchanged, but the positions of the plate and the ramekin are swapped. A visually grounded agent would re-locate the referenced objects and adjust its motion accordingly. Instead, in the illustrated rollout, $\pi_{0.5}$ grasps and transports the bowl but attempts to place it at the plate’s *original* location rather than its current location. The model completes the grasp and

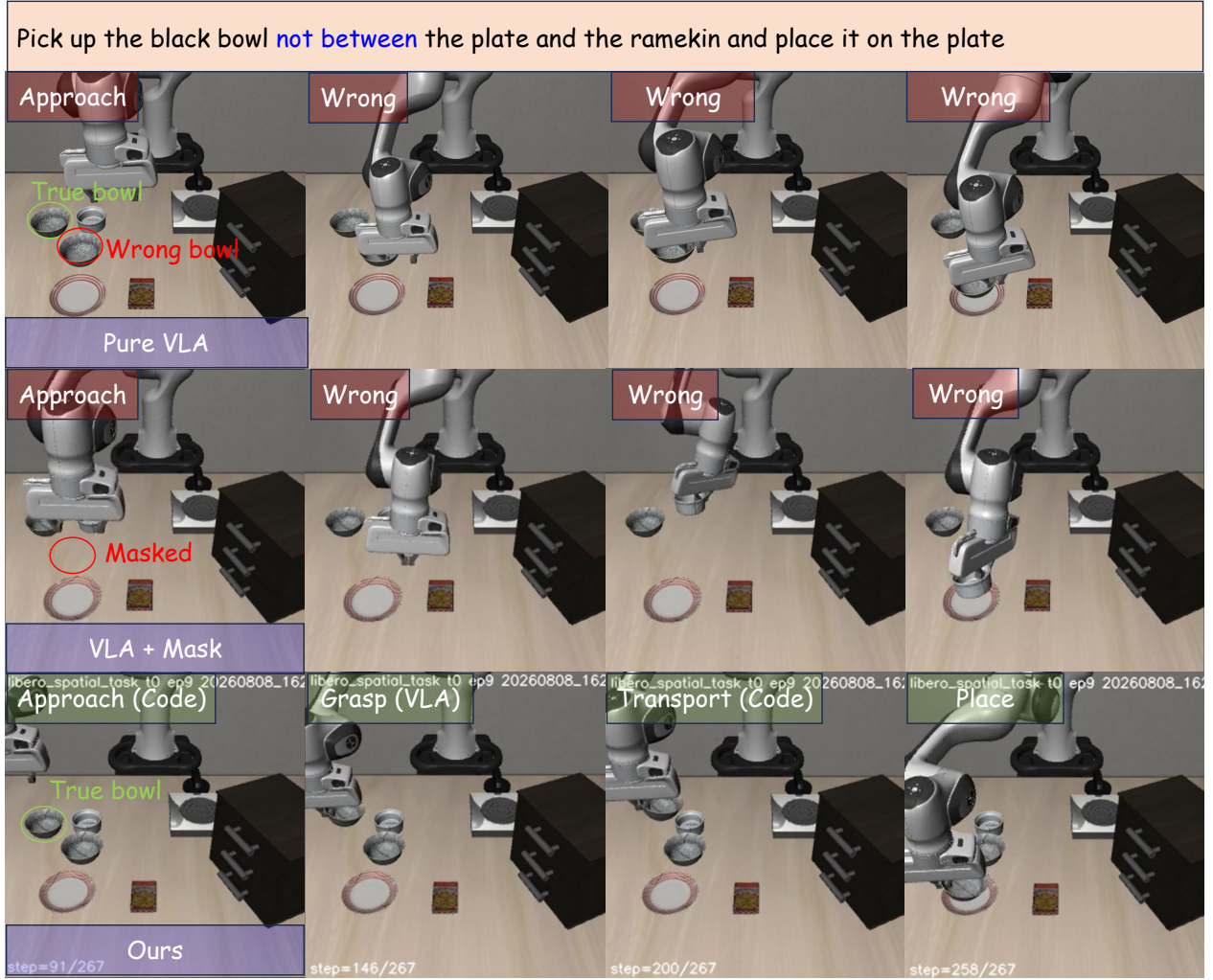


Figure 4 Language perturbation on LIBERO. Adding “not” before “between” flips the intended target to a different bowl, yet $\pi_{0.5}$ still grasps the original bowl, and externally masking the scene does not help. **HarnessPAI** re-grounds the instruction in code and grasps the correct bowl.

transport but fails to update the placement target to reflect the changed layout. This is exactly the collapse that LIBERO-PRO exposes—models above 90% on standard LIBERO lose most of their accuracy under such scene perturbations (Zhou et al., 2025).

Even when the robot reaches the vicinity of the target, a further failure mode appears in fine manipulation. The diffusion and flow-matching action heads that most modern policies use are generative models over continuous action chunks. Fig. 6 shows a button-pressing task: the evaluated WAM approaches the microwave and reaches toward the start button, but fails to depress it in the illustrated trials. These results alone do not establish whether the failure arises primarily from target localization, contact control, or action generation.

Taken together, these observations answer the first question. These examples show that action competence alone does not guarantee reliable task completion under instruction changes, scene perturbations, or fine-manipulation demands. We therefore investigate an external harness that complements the action model with explicit target grounding, geometric reasoning, and execution checks, while keeping the underlying model fixed.

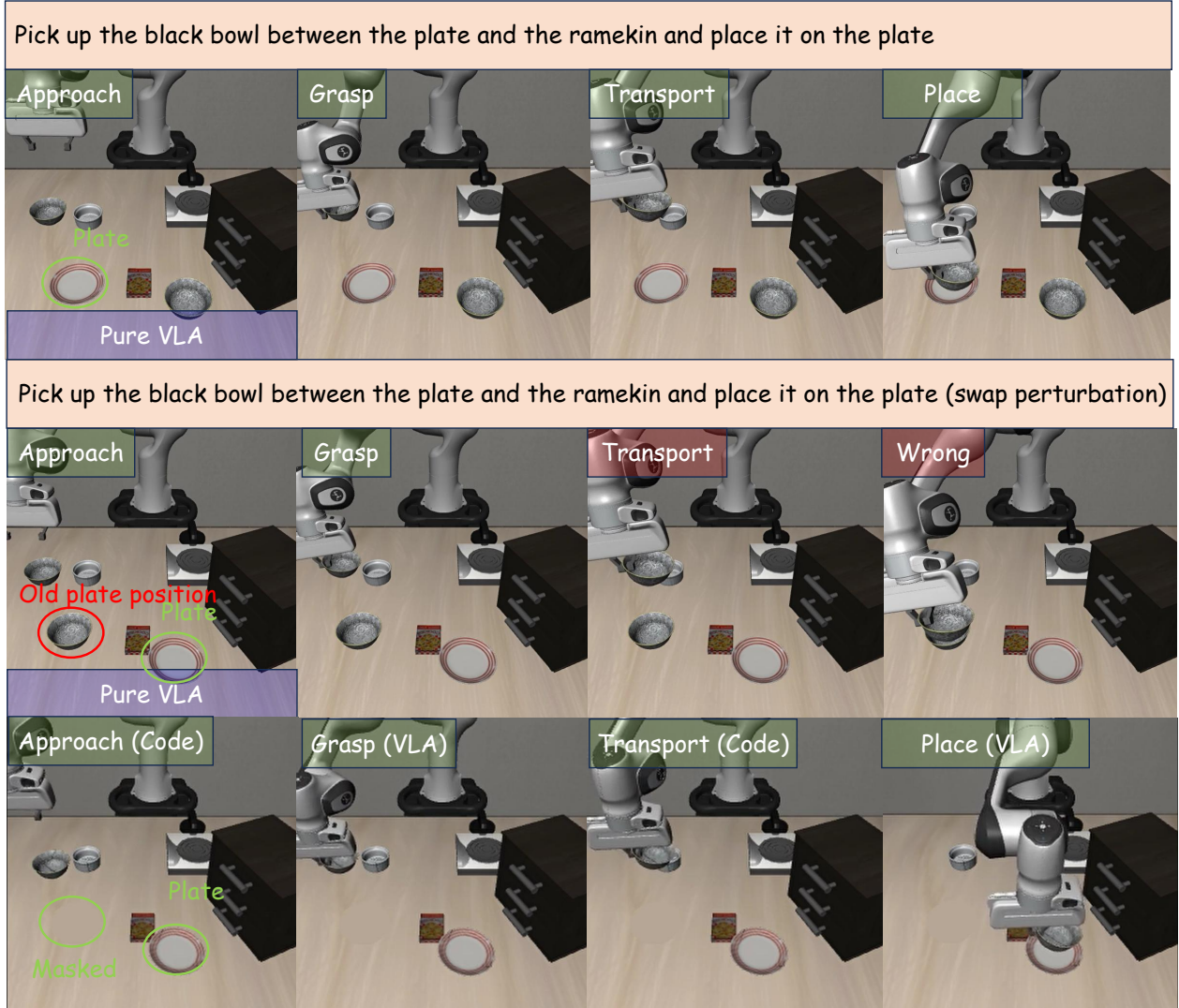


Figure 5 Visual perturbation on LIBERO. After the positions of the plate and ramekin are swapped without changing the instruction, $\pi_{0.5}$ grasps the bowl but attempts to place it at the plate’s original location and fails. **HarnessPAI** re-locates the objects in code and succeeds.

3.2 Why Open-Loop Execution?

A second problem shapes the design of any such harness: physical mistakes are frequently irreversible. Fig. 7 shows a two-arm lift in which the WAM grasps the wrong object and lifts it. In the illustrated rollout, the incorrect grasp changes the scene state, and the subsequent execution fails to recover and complete the task. Unlike cases where execution can be retried after resetting the program state, physical actions can leave persistent changes in the environment: some errors require additional recovery actions, while others may have irreversible consequences. A physical agent should therefore seek to identify potential errors before executing high-risk actions, with the aim of improving first-attempt success and reducing the cost of failure.

These risks motivate organizing task execution within a stable, inspectable program. Here, *open-loop execution* refers specifically to the program level: the harness program remains fixed within each rollout, while its execution can still use current observations and predefined checks to ground targets, carry out actions, and determine whether to proceed to the next step. Program revisions occur between rollouts. We revise the program using execution feedback between rollouts and evaluate the revised program in experimental environments that permit resets. Within a rollout, execution may still use predefined checks and recovery

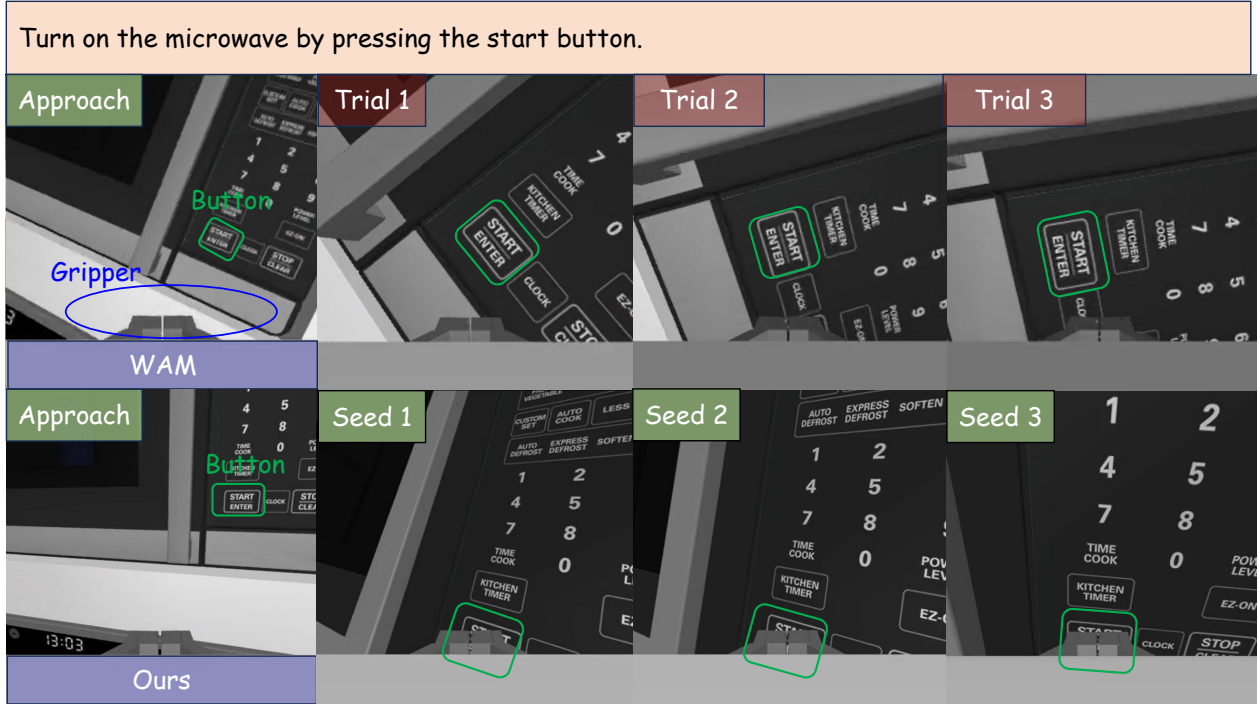


Figure 6 Fine-manipulation error on RoboCasa. The evaluated WAM reaches the microwave but fails to press the small start button in the illustrated trials. **HarnessPAI** successfully completes the button-pressing task.

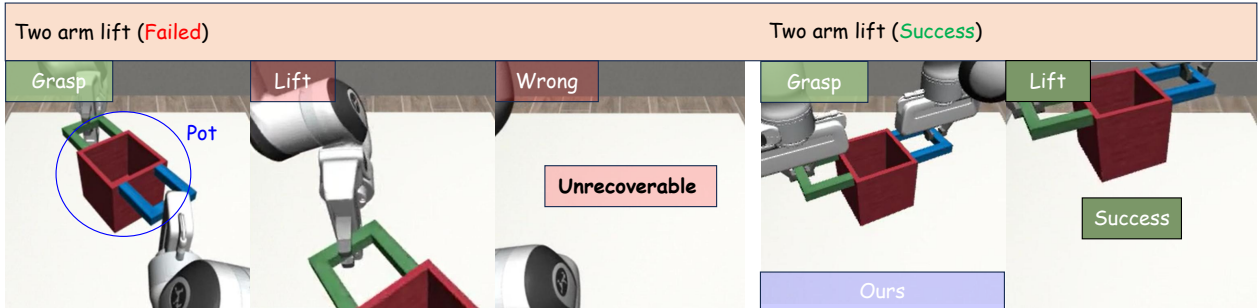


Figure 7 Irreversible mistake on RoboCasa. The robotic arm may knock over the pot, and this mistake cannot be undone, ending the rollout in an unrecoverable state. **HarnessPAI** grounds the grasp in code and succeeds on the first attempt.

mechanisms; the availability of environment resets should not be assumed in real-world deployment. These two mechanisms are complementary: explicit grounding and checks within a rollout aim to reduce the risk of incorrect actions, while evolution across rollouts uses execution feedback to improve the program and the reliability of subsequent executions.

3.3 Why Use Code as a Physical AI Harness?

The first two questions point to a specific locus of improvement. Rather than repeatedly fine-tuning the action model in the hope of improving its visual grounding, language understanding, and manipulation precision, we can support these capabilities through an external harness that executes open-loop and evolves closed-loop. This leaves a final question: why express that harness as *code*?

Code is the natural material for three reasons. First, it is *executable*: a program grounds each task in the current scene, decomposes it into steps, invokes the action model as a primitive, and runs explicit success checks—so open-loop execution is, in effect, running a program that checks task success as it runs. Second, it

is *inspectable*: perception, grounding, and recovery decisions are written as statements that can be read, diagnosed, and edited, rather than hidden inside a monolithic policy’s weights. Third, it is *evolvable*: a program can be revised in response to feedback and then re-executed, which is precisely the closed-loop evolution the first two questions require. Code can wrap different VLA and WAM backbones as interchangeable primitives, and degenerates to direct API control when no learned action model is available.

One might still object that a program only solves the task it was written for: if **HarnessPAI** merely instantiates a hand-written program, it would seem to handle only the tasks whose code already exists. This objection rests on a conflation of two deployment regimes.

Key question

In a real deployment, does the agent face *many tasks, each executed once*, or *one task, executed many times*?

A pure VLA or WAM is built for the first regime, with the goal of generating correct actions at inference time from previously unseen instructions. **HarnessPAI**, by contrast, targets the second regime, in which the same physical task is executed repeatedly. But executing the same task repeatedly does not relax the requirements for each individual run. Each run remains *open-loop* at the program level; because physical errors may be difficult to recover from or even have irreversible consequences, the agent should still aim to complete the task correctly on the first attempt. The many-times regime is therefore not a license to tolerate sloppy attempts; instead, it provides opportunities for closed-loop evolution across rollouts to accumulate feedback and refine the program before the next run, with the aim of improving the reliability of subsequent executions.

Moreover, the repairs discovered for one task do not stay with that task. **HarnessPAI** accumulates repairs distilled from previous tasks into a *how-to-repair* skill library for reuse when a new task arrives. Examples include re-localizing the target after a missed grasp, verifying a press before moving on, and re-grasping after a slip. A novel task is therefore not solved from scratch: its first failure is matched against previously learned repair skills, so the framework generalizes quickly to new scenes and instructions even though it has never seen their code before. Executability, inspectability, and evolvability, together with a how-to-repair skill library that transfers across tasks, are the properties that **HarnessPAI**, described next, formalizes.

4 Framework

4.1 Open-Loop Execution

Because physical actions can be costly or difficult to reverse, **HarnessPAI** uses a fixed program with explicit checks to reduce execution errors within each rollout. To support this goal, the program coordinates perception of task-relevant objects, geometric reasoning, and action execution. The examples in Section 3.1 show that successful action generation does not guarantee reliable grounding or instruction following under perturbations, motivating explicit support for these capabilities. **HarnessPAI**’s response is structural rather than parametric: it keeps the three capabilities *separate* and *explicit*, and organizes them into a program that checks itself as it runs. Verification here is not a retry loop that paws at the world until it finally sticks; it is a *success test*— has the grasp actually closed on the object, is the object where it should be— that *gates the next step*, so the agent commits to *transport* only once the grasp has been confirmed.

Fig. 8 shows this organization. The framework stores what it has learned in three memories and what it can do in a primitive library, with code at the center tying the two together. The *Task Memory* holds the tasks the robot has been asked to carry out, such as “turn on the stove” or “pick up A and place it in B.” When a new task arrives, the agent searches the *Task Memory*: on a hit it directly runs the code already stored in the *Executable Code Memory*, and on a miss it draws on the *Structured Skill Memory* and enters the closed-loop evolution described next (Section 4.2).

What that code invokes lives in the *Primitive Library*, organized into three interchangeable kinds. Perception primitives such as SAM3 ground the scene and answer *what is where*. Control primitives such as `goto_pose` and `close/open_gripper` supply the geometric motion, reaching a computed pose or closing a gripper, that a plan needs to execute. Model primitives such as a generalist VLA ($\pi_{0.5}$) (Intelligence et al., 2025b) or a predictive WAM (WorldDreamer) (Wang et al., 2024) supply learned, contact-rich manipulation. None of

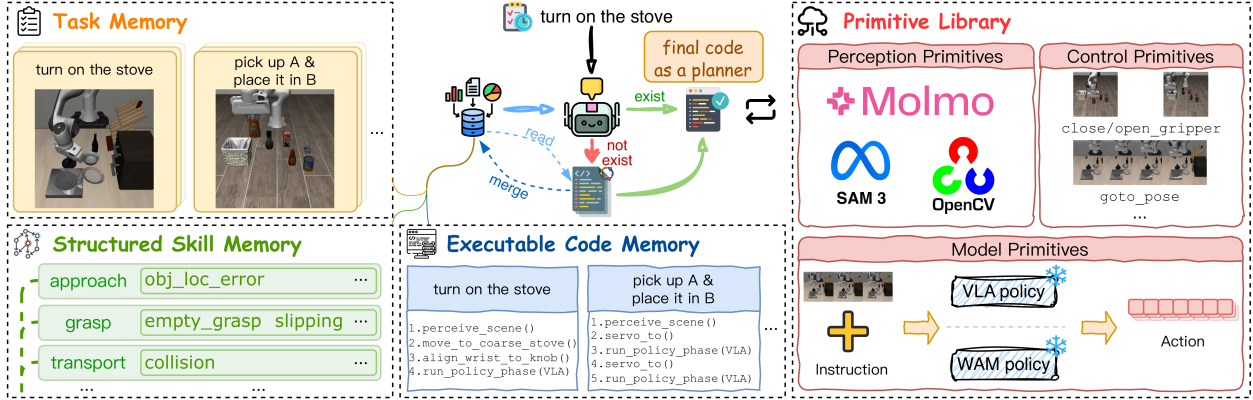


Figure 8 The code framework. Knowledge is organized into three memories, a *Task Memory* of instructions, a *Structured Skill Memory* of failure-to-repair skills, and an *Executable Code Memory* of evolved code, around a central planner. An instruction is matched to code that draws on a *Primitive Library* of perception primitives (SAM3), control primitives (raw control APIs, e.g., grasp, motion), and model primitives (VLA, WAM, PPO), and that verifies each step before committing an action to the environment.

the three kinds is privileged over the others; the action model is one primitive among several, not the agent itself.

For an existing task, the code the coding agent has written serves as the planner. It follows the plan step by step: it calls the perception primitives to ground the scene, reasons geometrically over what it perceives, decides from that which action to take, and only then invokes the action model to complete the task. In this way the perception module compensates for the action model’s weak perception, and code improves its planning ability.

Because the plan is written in this way, every decision the agent makes—which object, where, how, and whether the outcome was right—is a statement that can be read, tested, and edited, rather than a number hidden inside a policy’s weights. The central claim of this work follows: a physical AI is not an action model; it is perception, geometry, and action held together by an executable plan, of which the action model is only one, replaceable part. This is the within-rollout half of the framework; the next subsection turns to the across-rollout half—how a plan is generated, diagnosed, and revised until it succeeds.

4.2 Closed-Loop Evolution

Section 4.1 described the within-rollout half of the framework: how a plan, once written, executes open-loop against the scene. It does not yet say where that plan comes from, or how it becomes correct. That is the role of the closed loop. **HarnessPAI** treats a task’s code not as something a model writes once, but as a lineage that is generated, executed, diagnosed, and rewritten until it succeeds—and whose failures never leak into the code that already works. Fig. 9 shows the full pipeline.

HarnessPAI targets Physical AI embodiments that differ sharply from one another, from desktop robotic arms and household robots to robot vacuums, and each needs a different operating environment and simulation setup. Some provide an expert-trajectory video, some must first have their action model’s strengths assessed, and some lean heavily on perception. Because no single pipeline accommodates all of these at once, the framework divides its evolution into *required* and *optional* steps: it keeps one core loop that every embodiment must pass through, and turns the embodiment-specific parts into steps that attach only when needed.

Three optional steps exist, each matching one kind of on-demand capability. The first is workflow-graph generation: a VLM reads the expert-trajectory video and produces the task’s workflow graph, which fixes the structure of the code and the skills. The second is capability assessment: the VLA or WAM is rolled out on the task several times, and a VLM judges at which workflow nodes the model errs and needs code takeover or repair, producing a capability-assessment report. The third is perception-module initialization: for perception-dependent tasks, a coding agent calls SAM or Molmo to evolve perception code, which a

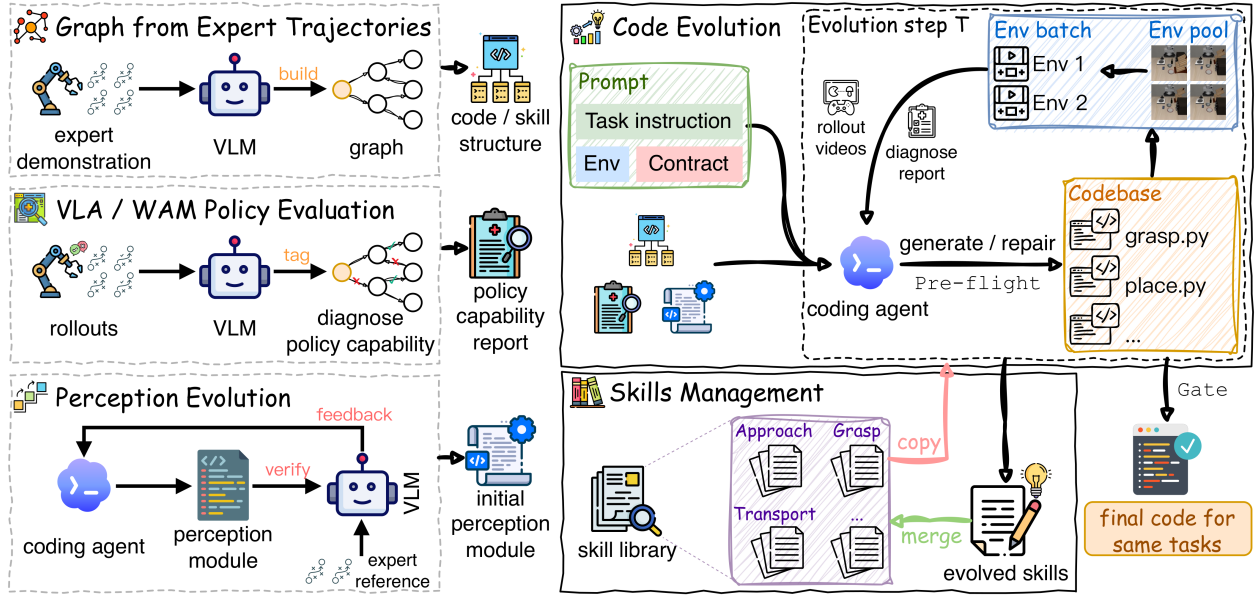


Figure 9 The overall pipeline of **HarnessPAI**. (1) *Optional*: a VLM reads the expert-trajectory video and generates a workflow graph for the task, which fixes the structure of the code and the skills. (2) *Optional*: the VLA/WAM is rolled out on the task several times and a VLM judges at which workflow nodes the model errs and needs code takeover or repair, producing a capability assessment report. (3) *Optional*: for perception-dependent tasks, a coding agent calls SAM/Molmo to evolve perception code, verified against the expert trajectories by a VLM, yielding the initial perception module. (4) The main evolution loop: given the task instruction, environment information, and the success contract, the coding agent builds the code package from the previously built code structure, runs it on multiple environments, and uses the diagnostics and rollout videos to decide the next edit; on errors it retrieves the matching skill from the skill library to repair the code, maintaining temporary in-progress skills, and on success it stores the evolved code in the code library and merges the evolved skills into the skill library.

VLM verifies against the expert trajectories, yielding the initial perception module. None of the three is mandatory; each turns on only when its input is present, namely expert trajectories, a VLA or WAM, or a perception-heavy task.

Perception code is evolved with the expert video doing more than verification: the VLM must also watch the trajectory to confirm which object each name refers to. Consider the LIBERO-Object suite, where *Alphabet soup* is a can-shaped object. Without prior knowledge, even a strong VLM cannot recognize it reliably on every frame, but with the expert trajectory as reference the coding agent can write code that distinguishes *Alphabet soup* from the other objects on the workbench, so recognition no longer depends on the VLM re-recognizing it correctly at each step.

There is a single required step, the main evolution loop, through which every embodiment and every task must pass. Given the task instruction, environment information, and the success contract, the coding agent builds the code package on top of the previously built code structure, runs it across multiple environments, and uses the diagnostics and rollout videos to decide the next edit. On errors it retrieves the matching skill from the skill library to repair the code; on success it stores the evolved code in the code library and merges the new skills into the skill library. Whether or not the optional steps are enabled, every run converges onto this main loop.

4.3 Evolution Mechanism

A few terms are worth fixing up front. Evolution usually targets a single task instantiated across many environment seeds: in reality the same task is executed repeatedly, but the environment shifts a little each time. We evolve programs on development seeds and evaluate them using the benchmark-specific splits described in Section 5.1.

Algorithm 1 spells out the loop. Two design choices are worth flagging. First, the *pre-flight* and *probe* stages form a fast feedback loop that catches regressions without paying for a full batch. Pre-flight runs as soon as the coding agent finishes writing code: it checks that the code runs at all, so a crash from an environment or driver interface is caught before it can be mistaken for a policy failure, and only code that passes pre-flight is evaluated in the environment. Probe is a targeted repair check: the coding agent selects the three seeds that failed the previous round, fixes them, and re-verifies the fix on those three seeds alone; once they pass, the full training set is rolled out again to confirm the repair has not broken the seeds that were already correct. Second, the self-review and the synthesis are a single call to the same writer, so the model that saw the evidence is the model that decides the next move.

Algorithm 1: Closed-loop evolution of a task.

Input: task description, environment information, success contract C , round budget T

Output: promoted code package P , distilled skills S

```

1 Define success contract  $C$  (scope, goal, terminal state, non-success signals);
2  $P \leftarrow$  build the code skeleton (from the task graph if available, else a single file);
3  $I \leftarrow$  task description;
4 for  $r \leftarrow 1$  to  $T$  do
5    $P \leftarrow$  WRITE(Codex edits the candidate copy using  $I$ );
6   if pre-flight or probe fails then
7     | fix the bug;
8   end
9   execute  $P$  open-loop on every episode  $\times$  every suite;
10   $(d, v) \leftarrow$  DIAGNOSE(VLM node-wise verdict and video reward);
11   $g \leftarrow$  GATE(deterministic milestone-prefix reward);
12  if all rollouts succeed under  $C$  then
13    | promote  $P$  into the code library;
14    |  $S \leftarrow$  distill failure-to-repair pairs into the skill library;
15    | return  $P, S$ ;
16  else
17    |  $I \leftarrow$  SYNTHESIZE(Codex self-review of  $d$  and  $g$ );
18  end
19 end

```

During evolution, defining what success is and how it is judged is critical. Before evolution starts, on the first round, the framework writes a *success contract*—the full evaluation scope (every suite, every episode), the authoritative goal condition, the graph’s terminal state, and an explicit list of signals that do *not* by themselves constitute success: a clean exit, a reached terminal state, a high dense reward, or the writer’s own claim of convergence. The same contract is injected into the writer, the diagnoser, and the reviewer on every round, so that what counts as success is never left to the model that is trying to succeed.

Beyond the final gating judgment and the phase-level checks inside the code itself, the loop also deploys a VLM (Gemini) to watch each task’s rollout video and suggest plausible causes of its errors. Two models split the loop (Table 2) because the diagnosis signal must be independent of the model that writes: if a single model both wrote an edit and judged whether it worked, it would be grading its own exam. The video diagnosis is a signal from a different modality and a different model, and the gate reward is a fallback that needs no model at all—a numeric signal that survives even if the video model misfires.

Table 2 Division of labor between the two models.

Model	Role	Responsibility	Edits code
GPT-5.6-sol	writer + synthesizer	edits phase code; self-review; next instruction	✓
Gemini-3.5-Flash	diagnoser	watches rollout video; node-by-node verdict vs. graph	✗

After each round, the evolved code is rolled out across the entire training set to assess how it now performs. The loop then reads the rollout output, both the code trace and the video feedback, and distills the cause of every remaining failure. The video feedback concentrates on three questions: first, which failures were expected to have been fixed but recurred in this rollout; second, whether any regression appeared, where an environment seed that was previously correct now fails, and why; and third, what causes the errors that remain. These answers are turned into the revision plan that drives the next round of evolution.

The product of a finished round is a code package. Its perception code grounds the scene through SAM and Molmo; its planning code carries the geometric reasoning and motion control with NumPy, SciPy, Operational Space Control (OSC), and inverse-kinematics (IK) control; and for fine manipulation it delegates to the VLA. Table 3 lays out this division: the deterministic phases run as fixed code, while the semantically hard phases such as grasp and place are handed to the VLA.

Evolution leaves behind more than the code package. Throughout the run the framework records each problem it encountered and how it was fixed, and distills these into skills that are stored in the skill library for reuse across tasks.

Table 3 The hybrid rollout. Deterministic phases run as fixed code; semantic phases are delegated to the VLA.

Phase	Executor
grounding	fixed code (segmentation + geometry)
approach	fixed code (Cartesian servo, Pyroki IK rescue)
grasp	VLA
transport	fixed code
place	VLA

What the loop leaves behind is the subject of the next subsection (Section 4.4): the promoted code and the *skills* distilled from its failures, and how the two are organized for reuse across tasks.

The writer, the pre-flight, and the rollouts all operate on an isolated candidate copy; the verified code library is never touched while a candidate is in flight. Only after a full episode batch passes across every suite is the candidate promoted into the library atomically; on failure, or on exhaustion of the round budget, the library is left exactly as it was and the candidate remains in the run directory for diagnosis. Candidate isolation prevents unverified edits from overwriting the stored program.

4.4 Skill Library

What the closed loop leaves behind is more than working code. Each converged run also distills, from its failures, a record of what went wrong and what fixed it. **HarnessPAI** accumulates these records into a *skill library*. A skill is a *how-to-repair* note with two parts: the *problem* it addresses—what failure to expect when attempting a given stage of a task—and the *repair*—how to get past that failure once it happens. The code library stores the executable plan; the skill library stores the experience of making that plan correct.

Storage. Skills are anchored on the task graph as a four-level structure (Fig. 10a): a *graph library* (L1) holds the workflow graph of each task; each graph’s nodes (L2)—grasp, transport, place, and so on—carry their own *skill library* (L3), aggregating every problem–repair pair learned at that node; and the *code library* (L4) sits alongside, holding the concrete repair code that each skill’s *repair* directly references. A single node gathers several skills because the same stage recurs across many tasks and fails in many ways: a grasp node may accumulate one skill for a slipping grip, another for misalignment, another for an empty grasp.

A skill may include explanatory code fragments, while the maintained executable implementation resides in the code library. Prior skill libraries often store each skill as an independent snippet of code, so that the library is itself a repository of executable procedures. The repair note references the maintained implementation in the code library through the “refers to” edge in Fig. 10a. The skill stays a small, readable note, and the code remains in exactly one place, maintained by the same evolution loop that promoted it. When a coding agent retrieves a skill, the reference leads it straight to the surrounding implementation it must edit, instead of asking it to regenerate that implementation from a copied fragment.

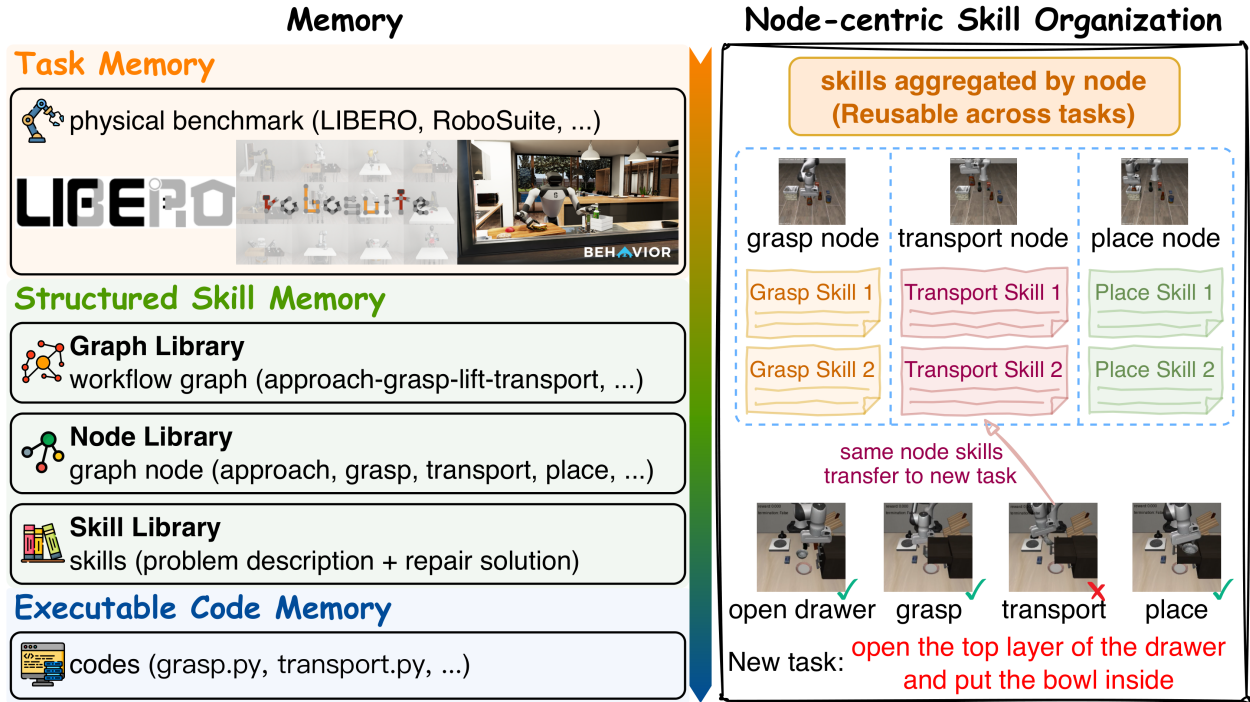


Figure 10 How skills are stored and queried. (a) *Storage*: at the top a *task memory* holds the descriptions of the tasks already evolved; under each task sits its *structured skill memory*, and each task owns a workflow graph whose nodes each carry multiple skills describing how to repair the difficulties encountered at that node, with each skill pointing to the more detailed repair resources in the code memory. (b) *Query*: skills aggregate across tasks, so all skills under the same node are pooled together; when a new task hits a problem at an existing node (e.g., *transport*), it directly queries the skills stored under that node.

Query. Retrieval is symmetric to storage: as Fig. 10b shows, a new task is not searched as a whole but first decomposed into the nodes its graph will traverse, and each node is then the key into the skill library. In this sense the whole skill library is a *node-centric* skill organization: skills are indexed not by task or by name but by the workflow node at which a given failure arises, so the node—not the task—is the unit that gathers and serves repairs.

These two choices—mounting skills on nodes and pointing them at shared code—buy the library its advantages.

The first is **retrievability**. The alternative, on the left of Fig. 11, is to write each problem–repair pair to its own markdown file. That yields a heap of *.md files—*obj_loc_error.md*, *empty_grasp.md*, *slipping.md*, *path_collision.md*—with a weak association to the task stage at which each failure occurs, so a coding agent must search an unordered pile to find the one note relevant to the step it is debugging. The right side shows the alternative **HarnessPAI** adopts: each problem–repair pair is mounted onto the node where it occurred—*approach*, *grasp*, *transport*, *place*. An agent that has reached the *grasp* stage queries the *grasp* node and finds exactly the failures that happen during grasping and their repairs. Node-based indexing groups repair notes by workflow stage to support targeted retrieval.

The second is **transfer**. A novel task such as *open the drawer* shares its grasp, transport, and place nodes with tasks the framework has already solved, so the skills mounted at those nodes transfer to it even though its own code has never been written. This is the mechanism behind the how-to-repair transfer promised in Section 3.3: a repair learned while grasping a bowl is re-offered whenever a later task requires grasping, and Section 5 shows the effect concretely—with the skill library, convergence arrives within the first few rounds.

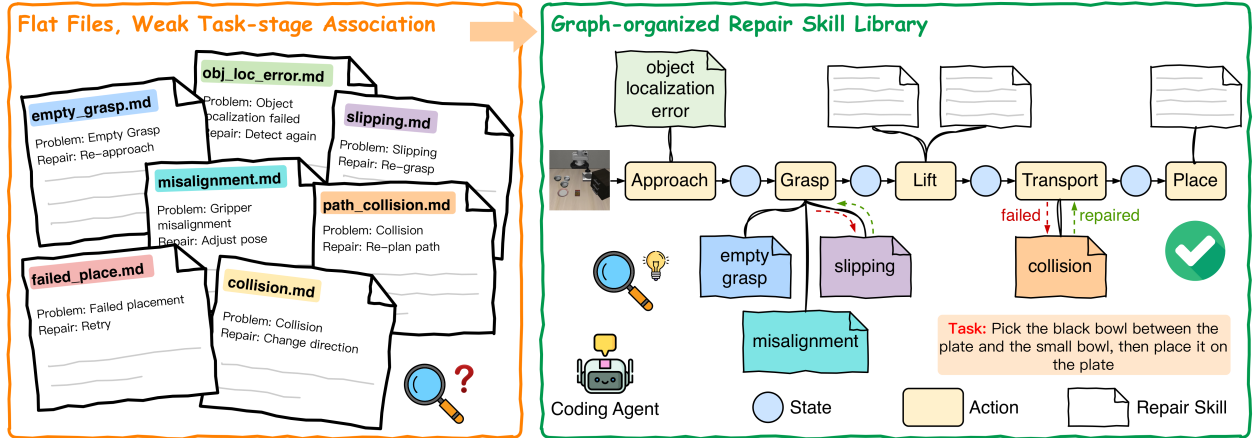


Figure 11 Flat versus graph-organized skill storage. Left: writing each “problem → repair” pair to its own markdown file yields a heap of flat files with a weak association to the task stage. Right: mounting each pair onto the graph node where it occurs (approach, grasp, transport, place) makes every skill retrievable by the stage of the task it belongs to.

Together with the code library, the skill library is what makes the closed loop cumulative. The loop does not merely make one task correct; it converts the cost of each failure into a reusable, node-keyed repair, so that the next task—and every task after it—starts from what the framework has already learned.

4.5 Model Feedback

The loop generates, executes, diagnoses, and revises the code *around* the VLA or WAM. During the main harness-evolution loop, the underlying action model remains frozen. That assumption buys a clean separation—the framework can be judged against a fixed, published checkpoint—but the loop remains limited by the capabilities and observability of its available perception, control, and learned-action primitives. This subsection considers improving the action model through post-training. Successful trajectories from a converged harness provide success-filtered demonstrations, which we use as expert data for fine-tuning on perturbed scenes.

The need is sharpest on benchmarks that offer no expert supervision. LIBERO ships roughly fifty human-teleoperated demonstrations per task, the data a VLA is trained or fine-tuned on (Liu et al., 2023). LIBERO-PRO is different: it perturbs those tasks—swapping object positions, changing the target—to test whether a model has merely memorized its training scenes, and it provides *no demonstrations* for the perturbed variants (Zhou et al., 2025). There is, by construction, no expert trajectory to imitate for “pick up the bowl after the plate and the ramekin have been swapped.” These variants therefore lack benchmark-provided demonstrations for direct supervised adaptation; Section 3.1 illustrates failures under such perturbations.

HarnessPAI changes this picture because a converged code policy is itself a trajectory generator. The loop of Section 4.2 produces, for each perturbed task, a program that grounds, plans, and checks its way to success. Run open-loop, that program emits precisely the data a policy learner needs: a sequence of observations and actions that reaches the goal in the perturbed scene, together with the hard success signal of the loop’s gate. The harness thus manufactures its own demonstrations where none exist—it needs no human to show how to re-grasp after a swap; the converged code already knows, and its rollouts record it.

Those rollouts are the expert data we feed back to the action model, and the route this work takes is *supervised fine-tuning*: successful code rollouts are used directly as fine-tuning targets, distilling the code’s explicit re-grounding and recovery behavior into the weights of the VLA or WAM. The same trajectories could also drive *reinforcement learning*, with the code policy’s success gate as the reward and the model optimized to reach the states the code reaches. Both routes aim to improve task performance under perturbations, but fine-tuning is the simpler, offline route, and the one we evaluate in Section 5.6.

The benefit flows one way, and that is what distinguishes model feedback from a co-evolution of harness and model. The harness is the data source, the model is the learner: the harness supplies demonstrations

for perturbed scenes from its own successful rollouts, enabling model adaptation without new human labels for those trajectories. Nothing flows back: the harness is not re-optimized against the improved checkpoint, and its code library, skill library, and converged programs are exactly what they were before fine-tuning. Wrapping the harness back around the improved model and iterating is a natural extension, but not one this work undertakes.

5 Experiment

5.1 Experimental Setup

We evaluate **HarnessPAI** across seven benchmarks spanning four broad embodiment categories: robot arms (single-arm and bimanual manipulation), a humanoid upper body on a mobile base, a differential-drive vacuum, and a legged walking agent. The settings also deliberately vary the action backend, a VLA ($\pi_{0.5}$), a WAM (WorldDreamer and DreamZero), a grasp detector (GraspNet), a pure control API, and a PPO policy, so that the experiments test the code framework itself rather than any single action model. Throughout, the backends are frozen primitives and only the harness is evolved. Evolution terminates when all development rollouts satisfy the success contract or the round budget is exhausted; only successful candidates are promoted, and crashes or timeouts count as failures. Beyond the main results, which compare **HarnessPAI** against pure VLAs and WAMs, RL-based methods, and prior harness baselines, we further probe its zero-shot transfer to a new action model, its evolution dynamics, its runtime and monetary cost, and the evolved program serving as an expert-data collector whose successful trajectories are reflowed into post-training to improve the action model it wraps.

The evaluated subsets and backends are as follows:

robosuite (Zhu et al., 2020) (desktop robot arms, single-arm and bimanual). We report seven tasks, namely 2A-Hand, 2A-Lift, Insert, Lift, Stack, Restack, and Wipe, with GraspNet (Fang et al., 2020) providing the grasp action primitive. We evolve one program per task on the held-out seeds 101–125 and report its success rate averaged over the 100 evaluation seeds 1–100.

LIBERO (Liu et al., 2023) (desktop 7-DoF Franka Emika Panda arm). We evaluate the Spatial, Object, and Goal suites, with ten tasks per suite, using the official LIBERO-finetuned $\pi_{0.5}$ -LIBERO. We evolve one program per task on seeds 0–14 and evaluate over 50 trials.

LIBERO-PRO (Zhou et al., 2025) (desktop robot arm under controlled perturbations). We evaluate the same three suites under the Swap and Task perturbations, for 60 perturbed tasks in total, again with $\pi_{0.5}$ -LIBERO (abbreviated as $\pi_{0.5}$). Each perturbation is scored over 50 layout seeds and macro-averaged over the ten tasks in each suite; we evolve one program per task on seeds 0–14 and run that same program unchanged on all 50. For LIBERO and LIBERO-PRO, the 50-seed evaluation includes the 15 seeds used for program evolution and 35 additional evaluation seeds, we follow the same setting in previous works. The corresponding zero-shot transfer experiment replaces $\pi_{0.5}$ -LIBERO with DreamZero, without re-evolving the harness.

RoboCasa (Nasiriany et al., 2024) (mobile manipulator). We use WorldDreamer (worldAgents-c, 2026), a video-conditioned WAM that maintains temporal world representations to drive a robot action policy. The main comparison is the Atomic-Seen setting reported as RoboCasa Target50 in the results table. We report 20 seeds in a fixed-layout scene, with obstacles and distractors varying across seeds.

BEHAVIOR-1K (B1K) (Li et al., 2023) (a humanoid household robot on a mobile base). We use GraspNet for two household tasks, picking up a radio and picking up a soda can. We evolve one program on seeds 26–35 and test it zero-shot on seeds 1–25, reporting navigation reachability separately from end-to-end task completion. The adapted variant then re-evolves that program on the previously failing test seeds for eight rounds to repair manipulation errors.

VacuSim (Mmints, 2026) (differential-drive mobile robot vacuum). The robot is controlled through a pure API of velocity commands and read-only sensor feedback, with cleaning progress over time as the outcome and roughly 80% coverage as the attainable ceiling.

MicroDuck (Pollen-robotics, 2026) (legged walking agent). We use a PPO-trained Walker policy for the straight-line task. The harness measures lateral drift and steers the same policy without retraining, and we report lateral error and endpoint distance rather than a binary success rate.

5.2 Main Results

5.2.1 robosuite

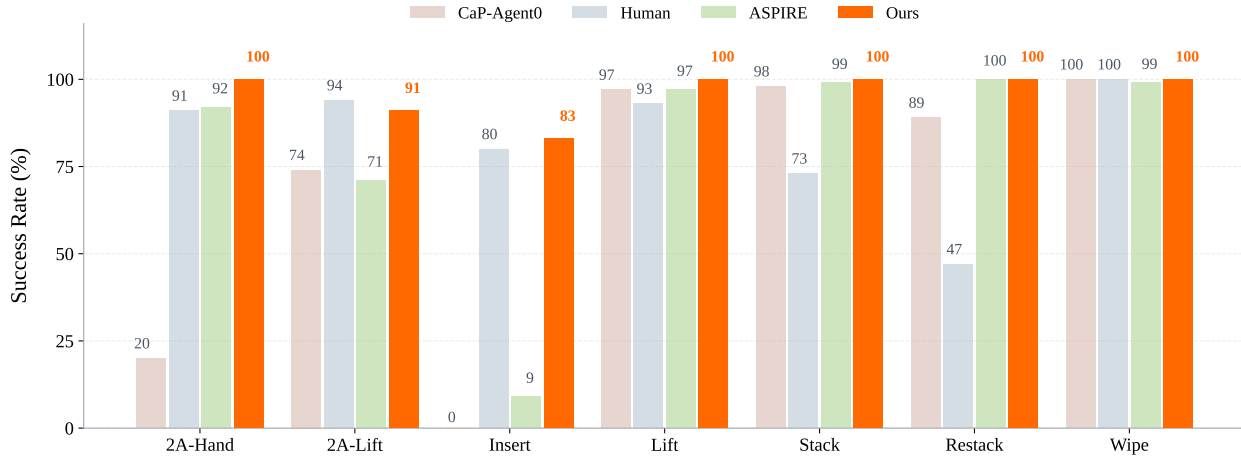


Figure 12 Per-task success rate on robosuite. We report seven tasks: 2A-Hand, 2A-Lift, Insert, Lift, Stack, Restack, and Wipe, where 2A-Hand and 2A-Lift are bimanual (two-arm) tasks. For each task we evolve one program on the held-out seeds 101–125 and report its success rate averaged over the 100 evaluation seeds 1–100. CaP-Agent0, Human, and ASPIRE are taken from the ASPIRE paper (Lu et al., 2026).

Fig. 12 reports per-task success on robosuite, with the underlying numbers tabulated in Table 6. **HarnessPAI** attains the highest average success rate of 96.3%, clearing the strongest prior harness ASPIRE (81.0%) by 15.3 points, the human baseline (82.6%) by 13.7 points, and CaP-Agent0 (68.3%) by 28.0 points. The framework reaches a perfect 100% on five of the seven tasks, namely 2A-Hand, Lift, Stack, Restack, and Wipe, and 91% on the bimanual 2A-Lift, leaving Insert as its only task below 90%. The decisive gains concentrate on the two tasks that break fixed code scaffolds. On Insert, a fine insertion that demands precise geometric reasoning, ASPIRE collapses to 9% and CaP-Agent0 to 0%, yet **HarnessPAI** holds 83%, above even the human baseline of 80%. On 2A-Lift, **HarnessPAI** improves over ASPIRE by 20 points (91% vs. 71%). Both tasks demand substantial perceptual alignment and geometric reasoning. **HarnessPAI** organizes these functions into modular code packages to support inspection and revision.

5.2.2 LIBERO

Table 4 reports per-suite and overall success rates on LIBERO, with methods grouped by family and ordered by increasing unweighted three-suite mean within each family. We organize the comparison around three families of approaches. *Pure VLA* models, such as π_0 (Black et al., 2024), OpenVLA (Kim et al., 2024), SmolVLA (Shukor et al., 2025), and GR00T N1 (NVIDIA et al., 2025), are end-to-end vision-language-action policies that map raw observations and language instructions directly to action tokens. *RL-based* models refine a VLA backbone with reinforcement learning or trajectory search, represented here by $\pi_{0.5}$ RLinf (Chen et al., 2025a) and AtomVLA (Sun et al., 2026). *Harness* methods wrap a policy, or replace it entirely, with an outer scaffold of code, memory, or self-correction. Our harness baselines are VLA-SCT (Zhang et al., 2026a) (over OpenVLA), Harness VLA (Zhang et al., 2026b) (a memory-guided agent that treats a frozen $\pi_{0.5}$ as a low-level primitive), OpenETA (Chen et al., 2026c) (a pure code-as-policy method with no learned action model), and **HarnessPAI**, which, unlike the other three, retains a learned action model yet evolves and re-executes the surrounding code in a closed loop.

Table 4 Comparison of selected methods on the LIBERO benchmark (success rate, %). We report success on the Spatial, Object, and Goal suites; Overall is the unweighted mean of the three displayed suite scores. Note that different papers adopt evaluation settings that are not entirely identical; our method uses the official $\pi_{0.5}$ -LIBERO model (abbreviated as $\pi_{0.5}$), and the $\pi_{0.5}$ and DreamZero-LIBERO results are our own evaluation, with each task in every suite run over 50 seeds.

Method	Spatial	Object	Goal	Overall
Pure-Model				
DreamZero-LIBERO (RLinf, 2026)	81.6	90.2	41.2	71.0
TraceVLA (Zheng et al., 2025)	84.6	85.2	75.1	81.6
OpenVLA (Kim et al., 2024)	84.7	88.4	79.2	84.1
WorldVLA (Cen et al., 2025)	85.6	89.0	82.6	85.7
CoT-VLA (Zhao et al., 2025)	87.5	91.6	87.6	88.9
MolmoAct (Lee et al., 2025)	87.0	95.4	87.6	90.0
NORA (Hung et al., 2025)	92.2	95.4	89.4	92.3
SmolVLA (Shukor et al., 2025)	93.0	94.0	91.0	92.7
GR00T N1 (NVIDIA et al., 2025)	94.4	97.6	93.0	95.0
$\pi_{0.5}$ (Intelligence et al., 2025b)	95.6	100.0	95.2	96.9
π_0 (Black et al., 2024)	96.8	98.8	95.8	97.1
RL-based				
World-Env (Xiao et al., 2025)	87.6	86.6	86.4	86.9
TGRPO (Chen et al., 2025b)	90.4	92.2	81.0	87.9
GRAPE (Zhang et al., 2024)	88.5	92.1	83.1	87.9
VLA-RL (Lu et al., 2025)	90.2	91.8	82.2	88.1
ThinkAct (Huang et al., 2025)	88.3	91.4	87.1	88.9
$\pi_{0.5}$ RLinf (Chen et al., 2025a)	99.0	96.0	97.0	97.3
AtomVLA (Sun et al., 2026)	96.4	99.6	97.6	97.9
Harness				
OpenETA (Chen et al., 2026c) (Code as Policy)	8.0	26.0	21.0	18.3
VLA-SCT (Zhang et al., 2026a) (OpenVLA)	91.2	92.8	82.0	88.7
Harness VLA (Zhang et al., 2026b) ($\pi_{0.5}$)	97.0	100.0	94.0	97.0
HarnessPAI ($\pi_{0.5}$)	97.2	100.0	97.2	98.1
Δ vs. $\pi_{0.5}$	+1.6	0.0	+2.0	+1.2

In Table 4, most results are taken from the original papers, whereas DreamZero-LIBERO, $\pi_{0.5}$ -LIBERO (abbreviated as $\pi_{0.5}$), and **HarnessPAI** are from our own evaluation. In the appendix B.1, we provide more detailed per-suite and per-task results.

Among the three families, **HarnessPAI** achieves the best overall success rate reported in Table 4: 98.1%, which is 0.2 points above AtomVLA (97.9%) and 1.1 points above Harness VLA (97.0%). It reaches a perfect 100.0% on LIBERO-Object, tying $\pi_{0.5}$ and Harness VLA for the highest reported score, ranks second on LIBERO-Spatial at 97.2% (within 1.8 points of $\pi_{0.5}$ RLinf at 99.0%), and ranks second on LIBERO-Goal at 97.2% (within 0.4 points of AtomVLA at 97.6%). It is consequently the only method in the comparison to place in the top two across all three suites.

A further comparison is provided by OpenETA, a pure code-as-policy baseline without a learned action model. OpenETA obtains 18.3% overall success (8.0%/26.0%/21.0% on Spatial/Object/Goal), whereas **HarnessPAI**, which combines evolved code with a learned $\pi_{0.5}$ action backend, obtains 98.1%, a difference of 79.8 points. This complete-system comparison is consistent with the value of the hybrid design, but it does not isolate the contribution of program evolution because the two systems use different action backends.

Both **HarnessPAI** and Harness VLA use a $\pi_{0.5}$ action primitive. Under the evaluation settings reported in the table, **HarnessPAI** scores 1.1 points higher overall than Harness VLA (98.1% vs. 97.0%) and 3.2 points higher on LIBERO-Goal (97.2% vs. 94.0%). Because the two results come from different evaluation protocols, this comparison should be interpreted as evidence of a performance advantage under the reported settings, rather than as a strictly controlled ablation.

5.2.3 LIBERO-PRO

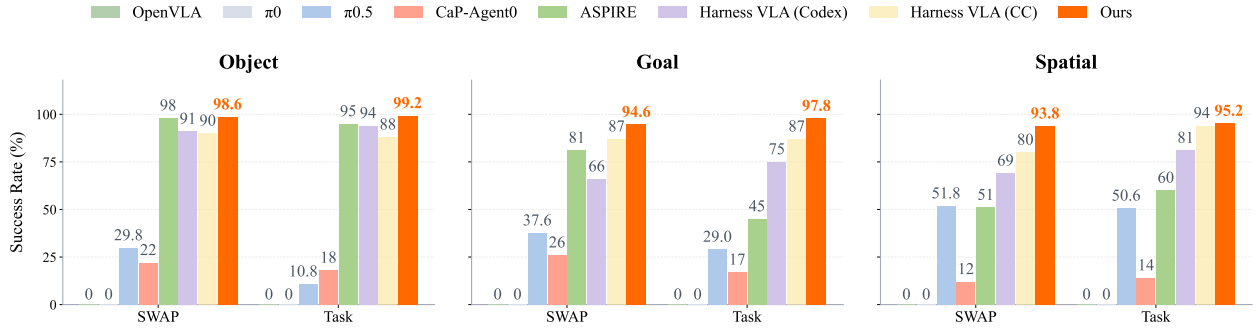


Figure 13 Macro-averaged success rate reproduction on LIBERO-PRO. We evaluate the Object, Goal, and Spatial suites, each under the two perturbations Swap and Task, with 10 tasks per suite, 60 tasks in total. Success is macro-averaged: for each perturbation, the reported rate is the mean over its 10 tasks, each scored over 50 layout seeds (env seeds 0–49). For **HarnessPAI** we evolve one program per task on the first 15 of the 50 layout seeds (seeds 0–14), and then run that same program unchanged on all 50 seeds. The same program serves the base suite and both perturbations, so Swap and Task exercise it unchanged as the unperturbed task. OpenVLA, π_0 , CaP-Agent0, and ASPIRE are taken from the ASPIRE paper (Lu et al., 2026), where ASPIRE is likewise evaluated over 50 seeds per task; $\pi_{0.5}$ is evaluated by ourself; and the Harness VLA (Codex and CC) results are taken from the Harness VLA paper (Zhang et al., 2026b), where each result is evaluated over 10 seeds per task.

Standard LIBERO success can be misleading, however: a model can reach a high score by memorizing fixed action sequences and scene layouts rather than by understanding the task. LIBERO-PRO (Zhou et al., 2025) probes exactly this. It applies controlled perturbations to the LIBERO tasks, and we evaluate two of them, Swap and Task, to test whether a policy still succeeds when the memorized configuration no longer holds. Swap is a position perturbation that exchanges the initial positions of two objects, forcing the model to re-anchor its actions to a changed layout; Task is a task perturbation that changes the task instruction given to the VLA model to test its ability to understand language.

The difficulty of these shifts is stark. Models that exceed 90% on standard LIBERO collapse under perturbation: in the original benchmark, OpenVLA and π_0 drop to 0.0% on both Swap and Task (Zhou et al., 2025), and even the strong $\pi_{0.5}$, which reaches 96.9% on standard LIBERO, falls to 39.7% on Swap and 30.1% on Task in our evaluation. Robustness here cannot be bought with a stronger backbone alone, which has motivated a family of harness-based remedies.

Two recent methods have attacked this gap with harness scaffolds. ASPIRE (Lu et al., 2026), a code-as-policy agent that writes, executes, and self-corrects robot programs against an explicit skill library, lifts LIBERO-PRO success to 72% overall (77% on Swap and 67% on Task). Harness VLA (Zhang et al., 2026b), which treats a frozen VLA as a primitive and drives it with a memory-guided agent, reaches 87.7% with its CC variant and 79.3% with its Codex variant. **HarnessPAI** extends this line of work and attains the best result, **96.5%** overall (95.7% on Swap and 97.4% on Task), surpassing ASPIRE and the stronger Harness VLA CC variant by 24.8 and 8.8 points, respectively: because its closed-loop evolution corrects the code that frames each task, it recovers from the perturbations that break fixed scaffolds as thoroughly as it does on standard LIBERO.

Table 5 Results on the Atomic-Seen subset of RoboCasa Target50 (18 tasks; success rate, %). Type distinguishes pure VLAs, World Action Models (WAM), and Harness methods. Note that different RoboCasa papers adopt different evaluation settings: for instance, Harness VLA runs 10 seeds per task on Atomic-Seen, whereas our method runs 20 seeds in a fixed-layout scene, with obstacles and distractors varying across seeds; to ensure a fair comparison, we evaluate both WorldDreamer and our method under a unified experimental setting.

Method	Type	Atomic-Seen
π_0 (Black et al., 2024)	VLA	34.6
$\pi_{0.5}$ (Intelligence et al., 2025b)	VLA	39.6
RLDX-1 (RLWRLD, 2026)	VLA	60.0
WorldDreamer (worldAgents-c, 2026)	WAM	65.0
Harness VLA (Codex) (Zhang et al., 2026b)	Harness	91.6
Harness VLA (CC) (Zhang et al., 2026b)	Harness	79.4
HarnessPAI(WorldDreamer)	Harness	92.2
Δ vs. WorldDreamer		+27.2

5.2.4 RoboCasa

Table 5 reports results on RoboCasa, a distinct embodiment, the Franka Panda Emika robot, equipped with an Omron mobile base (Haviland et al., 2022), paired with a different underlying action model, WorldDreamer. HarnessPAI achieves 92.2% on Atomic-Seen, compared with the reported 91.6% for Harness VLA (Codex) under a different evaluation protocol. These results are most informative when read against HarnessPAI’s own base policy: wrapping WorldDreamer with the closed-loop code framework lifts Atomic-Seen from 65.0% to 92.2% (+27.2 points).

Across the evaluated arm and mobile-manipulation settings, HarnessPAI improves task execution with both $\pi_{0.5}$ and WorldDreamer. These results support the portability of the harness design across the tested action backends and control interfaces.

5.2.5 BEHAVIOR-1K

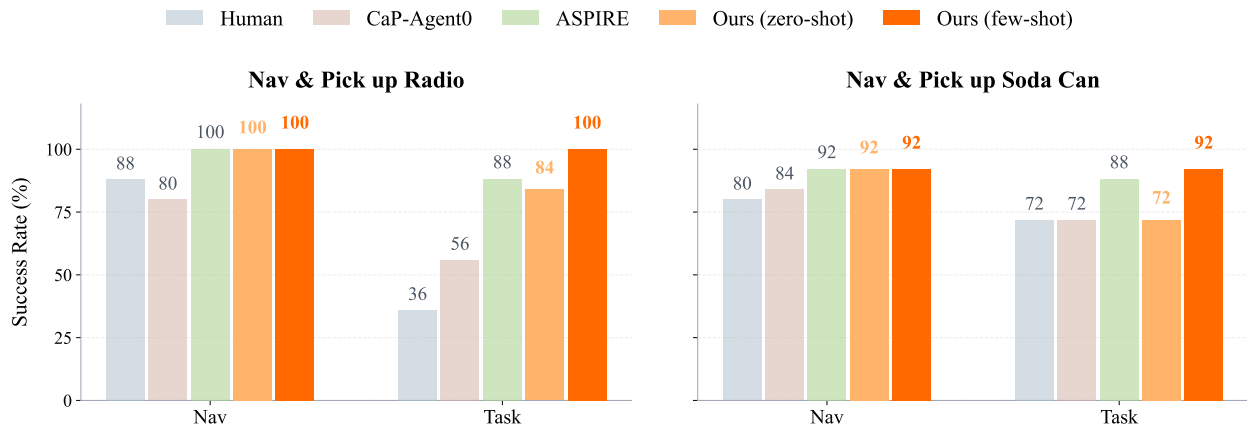


Figure 14 A long-horizon humanoid household robot on a mobile base on two BEHAVIOR-1K household tasks. ASPIRE evolves one program per seed on seeds 26–35, distills “how-to-repair” skills from those runs, and then uses the skills to evolve one program per seed on seeds 1–25. HarnessPAI evolves one program from scratch on seeds 26–35 and distills “how-to-repair” skills; *zero-shot* directly tests that program on seeds 1–25, while *few-shot* re-evolves the zero-shot program on the failing seeds, where few-shot denotes eight iterations. Human, CaP-Agent0, and ASPIRE are taken from the ASPIRE paper (Lu et al., 2026).

Fig. 14 reports results on two BEHAVIOR-1K household tasks using a humanoid upper body on a mobile base, separating navigation from end-to-end completion. On picking up the radio, HarnessPAI’s *zero-shot* program achieves 100% navigation success, matching ASPIRE and beating CaP-Agent0 (80%) and the human baseline

(88%), and its end-to-end task success of 84% clears CaP-Agent0 (56%) and human performance (36%) while closing to within four points of ASPIRE (88%). The adapted variant then re-evolves the zero-shot program on the failing seeds for eight rounds and reaches 100%, so that every navigable episode in the evaluated set is completed. The soda-can task follows the same arc: navigation success is 92% (above the human baseline of 80% and CaP-Agent0 at 84%), zero-shot task success ties CaP-Agent0 at 72%, and adaptation lifts it to 92%, four points above ASPIRE (88%). Across both tasks the navigation-to-completion gap, the episodes the base can reach but not finish, is eliminated entirely by the closed loop, which repairs the manipulation errors on whatever the mobile base can actually approach.

5.2.6 VacuSim

VacuSim is a model-free test of **HarnessPAI**: a differential-drive vacuum robot uses bumper and distance sensors together with a ROS 2 control API to maximize painted floor coverage within a time budget. We evaluate a furnished 25×25 apartment, where recovery around furniture is the main challenge, and an unfurnished 40×40 apartment, where efficient long-range coverage dominates. The score is computed by the ground supervisor rather than by planner-side tile counts.

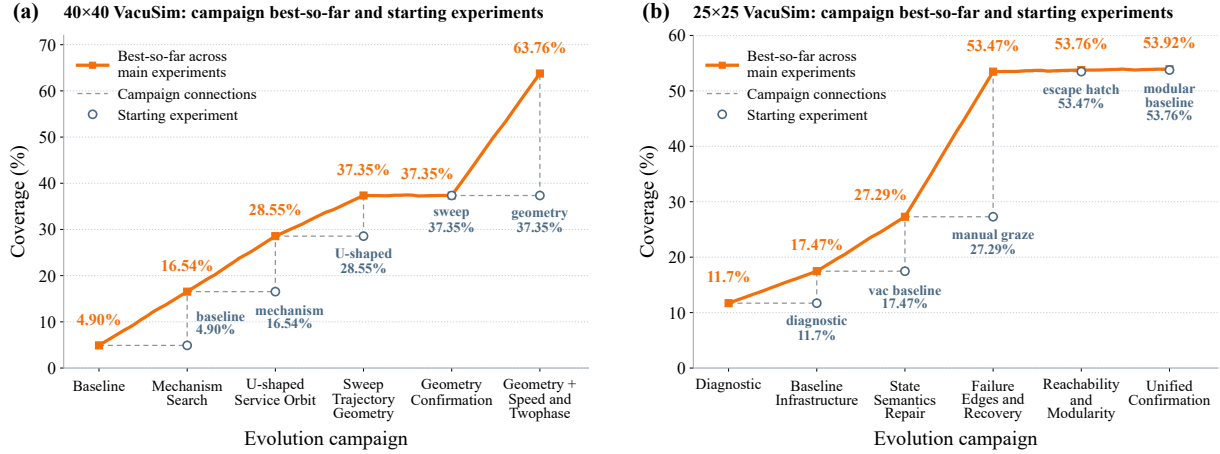


Figure 15 Campaign-level evolution on VacuSim. Coverage improves from 4.90% to 63.76% on the unfurnished 40×40 map and from 11.7% to 53.92% on the furnished 25×25 map.

Across the campaign summarized in Fig. 15, the evolution is hierarchical: each outer round selects one technical direction, which is explored through smaller measurable directions from isolated code candidates. Every direction maintains its own Skill and Code memory; validated improvements are retained with their diagnostics, while failed candidates remain as negative evidence without overwriting the champion. After three valid attempts without improvement, a direction is retired and validated skills are tested in bounded combinations. If these combinations also plateau, the next outer round moves to another direction while preserving the previous libraries for retrieval.

Fig. 16 reports the resulting cleaning progress over time, comparing the final policy against four baselines on the unfurnished 40×40 map and the initial against the evolved policy on the furnished 25×25 map. VacuSim therefore instantiates the main evolution loop without the optional VLA, workflow-graph, or perception stages: diagnostics and painted coverage guide code revision, while validated direction libraries provide reusable repair knowledge. The final policy executes without an LLM call on the critical path.

5.2.7 MicroDuck

MicroDuck is a walking agent whose objective is simply to walk forward in a straight line. The task is more demanding than it appears: a PPO-trained Walker model advances toward the goal, but nothing in its training objective penalizes sideways drift, so it veers off course as it walks. Fig. 17 contrasts the pure PPO model against the same model wrapped by the **HarnessPAI** harness across three views of the rollout.

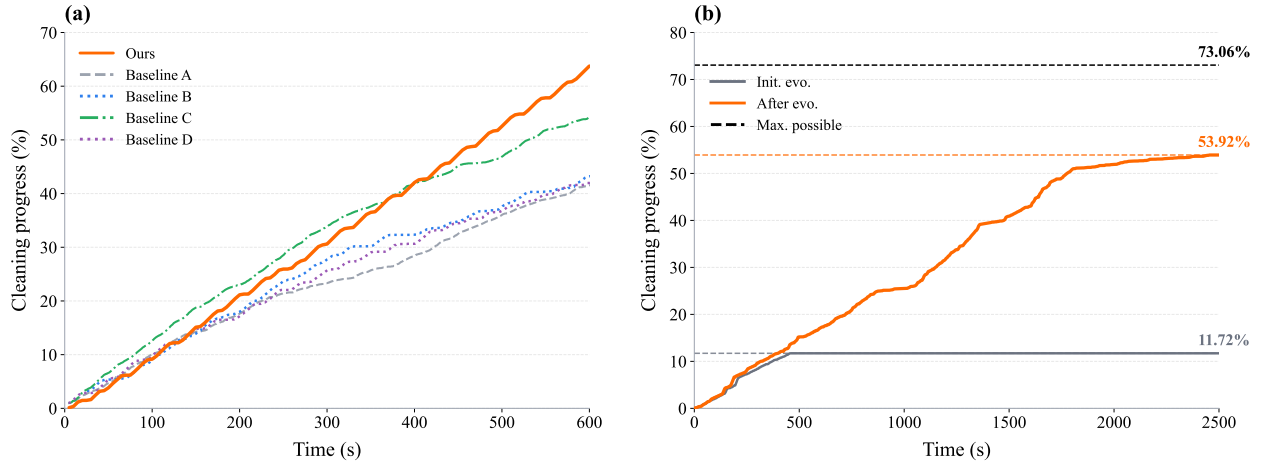


Figure 16 Cleaning progress over time on VacuSim. Panel (a) compares the final **HarnessPAI** policy with four baselines (Mints and Neubert, 2023) on the unfurnished 40×40 apartment, while panel (b) compares the initial and evolved policies on the furnished 25×25 apartment. The final **HarnessPAI** policy reaches 63.76% coverage on the 40×40 map and improves from 11.72% to 53.92% on the 25×25 map; the 73.06% dashed line denotes the maximum physically cleanable coverage of the furnished map.

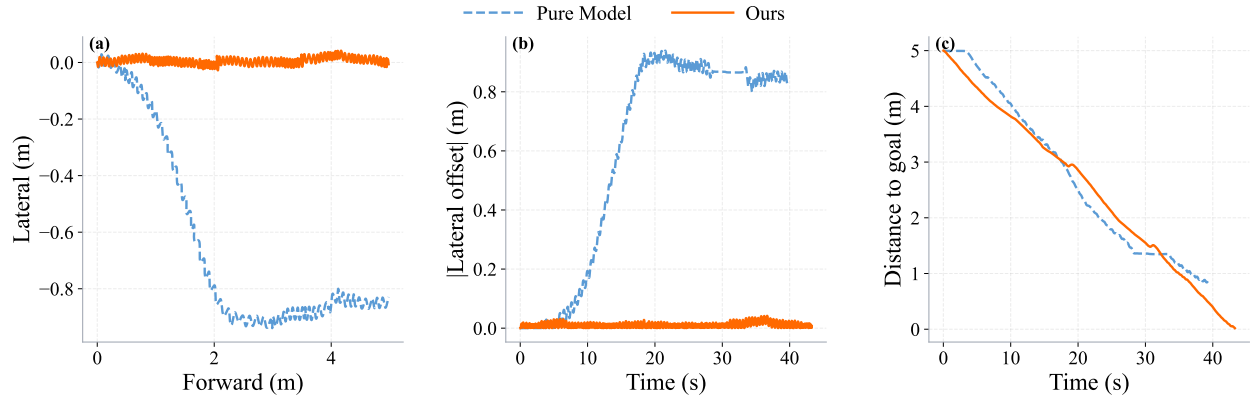


Figure 17 Performance of the pure PPO model and the same model with the **HarnessPAI** harness on MicroDuck’s straight-line walking task: (a) the planar trajectory, showing how the lateral error evolves as forward progress increases; (b) the absolute lateral error over time; and (c) the distance between MicroDuck and the goal over time.

Panel (a) plots the planar trajectory: over 4 m of forward progress the pure model drifts up to 0.8 m off the ideal line, whereas the harness holds its lateral error near zero. Panel (b) shows the absolute lateral error over time, which accumulates for the pure model but remains flat for the harness. Panel (c) tracks the distance to the goal, which the pure model, carried off course by the accumulated drift, fails to close, while the harness drives it to zero and reaches the endpoint. The correction comes from code rather than retraining: the harness measures the heading error and steers the same underlying Walker policy back onto the line, so, as elsewhere in this work, the gain is obtained without retraining the Walker policy.

5.3 Harness Zero-shot

The results so far pair each benchmark with the action model against which its harness was evolved. Here we ask a stricter question: does the harness transfer to a different action model without any retraining? To test this, we take the harness evolved on LIBERO-Object with $\pi_{0.5}$ as the underlying policy and apply it *zero-shot* to DreamZero, a World Action Model (WAM) that jointly predicts future scene evolution and action chunks rather than acting on the current observation alone. Nothing in the harness is re-evolved or re-tuned for the new model; the only change is that every action-model call now invokes DreamZero instead of $\pi_{0.5}$.

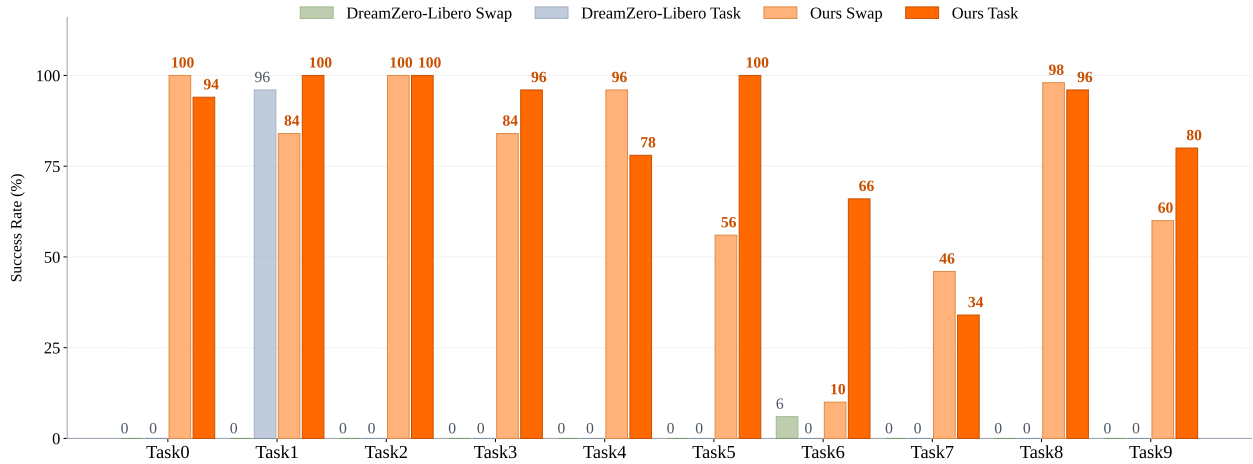


Figure 18 Zero-shot transfer of the **HarnessPAI** harness to DreamZero, a World Action Model. Per-task success rate (%) on the LIBERO-Object suite under the Swap and Task perturbations of LIBERO-PRO, comparing DreamZero-Libero against the **HarnessPAI** harness applied *zero-shot*.

Fig. 18 reports per-task success under the Swap and Task perturbations of LIBERO-PRO (Zhou et al., 2025). The original DreamZero WAM barely generalizes: it scores 0.6% overall on Swap, with its only successes being 3 of 50 on the butter task, and 9.6% on Task, a total carried almost entirely by a single task (cream cheese→alphabet soup, at 96%), a memorization artifact in which the model recognizes the alphabet-soup object it has already seen and replays the corresponding behavior. Applying the $\pi_{0.5}$ -evolved harness *zero-shot* lifts the same model to 73.4% on Swap and 84.4% on Task. The gains come from the code around the action model rather than the action model itself: the harness supplies the perception, geometric reasoning, and verification that DreamZero’s weights do not, and because those components are written as executable code rather than trained parameters, they carry over to a new action model untouched. The residual gap relative to the 96.5% that the same harness attains over $\pi_{0.5}$ on LIBERO-PRO reflects the weaker underlying WAM, but the direction of transfer is the point: code generalizes where weights do not.

5.4 Evolution Process

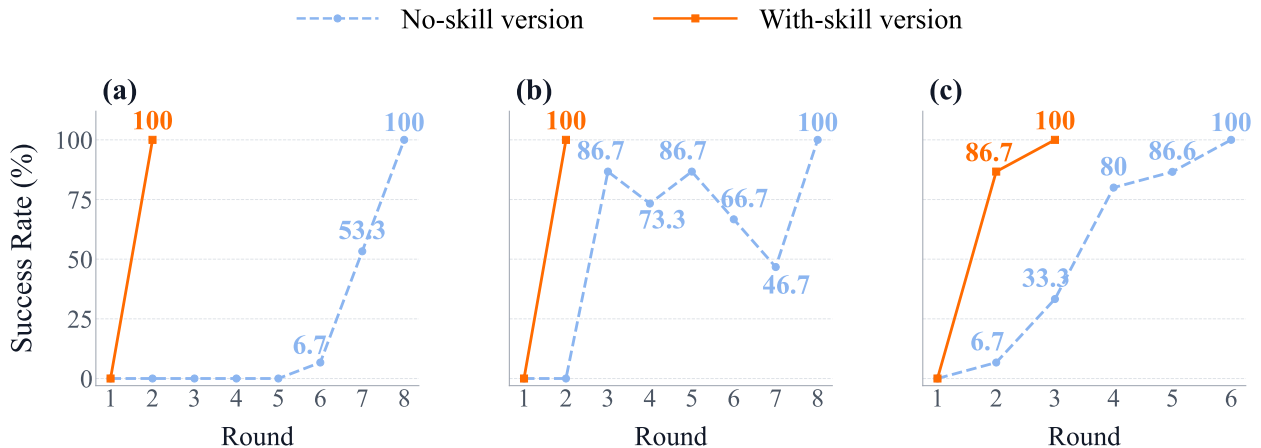


Figure 19 The average results of closed-loop evolution of **HarnessPAI** on LIBERO Object, LIBERO-PRO Object Task and Swap. We evolve the task code under two conditions: with no access to the skill library (*no-skill*) and with the evolved skill library (*with-skill*), and plot success rate per round for three representative tasks: (a) Task 0, (b) Task 1, and (c) Task 5. Success rate is the number of successful rollouts summed over seeds 0–14 across the Swap and Task perturbations, 30 seeds in total.

Fig. 19 traces how the closed loop converges in practice. On LIBERO-PRO we evolve one program per task over the 15 development scenes, and then run that same program unchanged against the base task, the Swap position perturbation, and the Task perturbation (Zhou et al., 2025). Evaluating each candidate across multiple development conditions checks whether an edit preserves task performance across the tested layouts and perturbations. The figure isolates the LIBERO-Object suite and plots three representative tasks, namely task 0, task 1, and task 5, each with two curves: a *no-skill* baseline, in which the code is evolved without access to the skill library, and a *with-skill* variant that draws on it.

Without skills, the three tasks follow three qualitatively different trajectories. Task 0 (Fig. 19a) exhibits an “Aha” moment: success stays pinned near zero for several rounds while no single edit resolves the failure, and then the round that removes the blocking error sends it to 53.3% and on to 100%. Task 1 (Fig. 19b) meanders instead: its success oscillates between 86.7%, 73.3%, 86.7%, 66.7%, and 46.7% before finally landing on the correct program, indicating a rugged search landscape in which the code wanders before converging. Task 5 (Fig. 19c) climbs monotonically, accumulating partial corrections (6.7%→33.3%→80%→86.6%) until it reaches 100%. With the skill library, by contrast, all three jump to 100% within the first two rounds, which is the clearest evidence of the skills’ value: they supply the correct manipulation primitives up front, so the closed loop spends its budget refining rather than re-discovering elementary behaviors.

Not every task exhibits an “Aha” moment. The “Aha” moment indicates a bottleneck: once it is resolved, success rises rapidly. Some tasks instead undergo a meandering exploration in the absence of skills before they eventually find the correct answer, and some tasks are limited by several interacting factors, so that reaching 100% still requires two or three rounds even with the skill library.

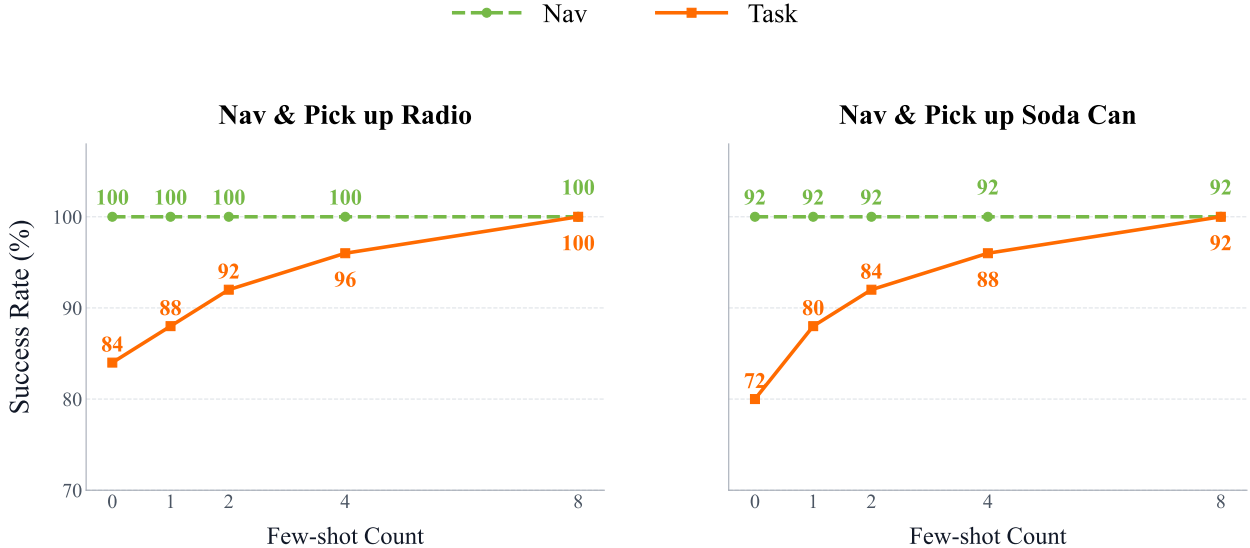


Figure 20 Few-shot evolution on BEHAVIOR-1K. On the two household tasks, picking up a radio and picking up a soda can, HarnessPAI re-evolves the zero-shot program on the failing seeds 1–25, and we separate the navigation stage from the full task: *Nav* measures how often the mobile base reaches the target object at all, while *Task* measures end-to-end success. Navigation is flat (100% and 92%), marking the set of episodes the agent can actually reach, and the task success rate climbs with each few-shot round (84%→100% and 72%→92%) until it converges to that ceiling.

Fig. 20 shows the same closed loop on the mobile-manipulation benchmark. On BEHAVIOR-1K we re-evolve the program on the seeds that failed the zero-shot pass (seeds 1–25), and here we isolate navigation from the full task to make the effect legible. Navigation success remains at 100% for the radio task and 92% for the soda-can task throughout the reported adaptation rounds. The *Task* curve starts below the corresponding navigation success rate (84% and 72%) and then climbs monotonically with the number of adaptation rounds, reaching 100% and 92% after eight rounds. In the reported final evaluation, every episode in which navigation succeeds also reaches task completion.

5.5 Cost

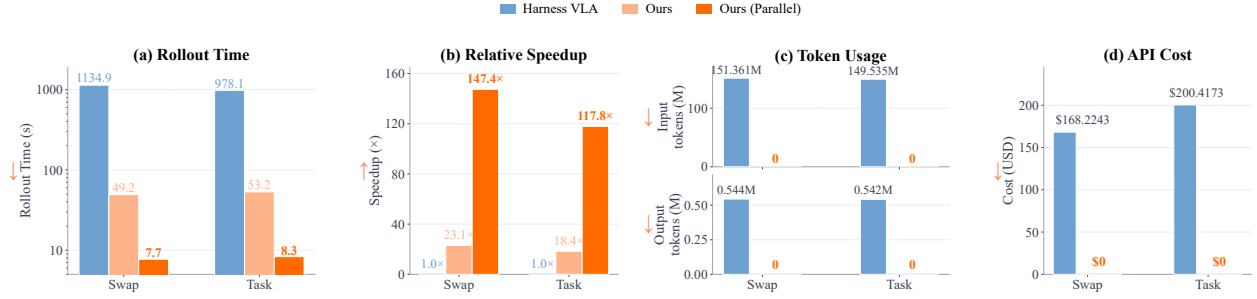


Figure 21 Cost on LIBERO-Goal Task 1 under the Swap and Task perturbations, each evaluated over seeds 0–49 (100 seeds in total). (a) Rollout time. (b) Relative speedup. (c) Online high-level LLM token usage; Harness VLA uses GPT-5.6-sol with reasoning effort *high*. (d) Online high-level LLM API charges. Because Harness VLA queries GPT for an instruction at every action step, GPT’s input (the image and current state) dominates the token count and its deliberation dominates the time. Since **HarnessPAI** runs many tasks in parallel across GPUs, its parallel variant is the result of 7 GPUs rolling out simultaneously.

We measure runtime cost in two regimes. The serial timing is recorded over a complete run of 50 rollouts per task for both Harness VLA and **HarnessPAI**, and it covers the entire execution loop, including environment startup, perception, and control, rather than only the forward pass of the policy. On top of this, **HarnessPAI** reports a parallel variant obtained through our own infrastructure, which runs perception, action, and environment simulation concurrently on 7 GPUs and overlaps their execution. Fig. 21 therefore separates three configurations per task: Harness VLA (serial), **HarnessPAI** serial, and **HarnessPAI** parallel.

Fig. 21a reports rollout time. Harness VLA requires 1134.9 s on Swap and 978.1 s on Task per rollout, whereas **HarnessPAI** only need 49.2 s and 53.2 s per rollout on Swap and Task suite, and the parallel variant cuts this further to 7.7 s and 8.3 s. Relative to serial Harness VLA, **HarnessPAI** achieves speedups of 23.1× on Swap and 18.4× on Task in serial execution, and 147.4× and 117.8× in the separate seven-GPU configuration (Fig. 21b).

The gap is architectural. During execution Harness VLA must invoke an agent at every step, and in our reproduction that agent is Codex, a large reasoning model that reads the observation and memory and deliberates before issuing each action. This per-step reasoning is an in-the-loop dependency: it serializes the rollout and adds the latency of a large model to every step. **HarnessPAI**, in contrast, compiles a task into code once and then executes that code open-loop, without online high-level LLM deliberation on the critical path. The same dependency drives the token usage in Fig. 21c. Across its 50 rollouts, Harness VLA consumes 151.36M input tokens and 0.544M output tokens on Swap, and 149.54M input tokens and 0.542M output tokens on Task. According to the official pricing of GPT-5.6, Harness VLA consumes a total of \$168.2243 on Swap (input + output + cache hits), and a total of \$200.4173 on Task. Since our code does not need to spend money calling a coding agent for reasoning during execution, both the tokens we consume and the amount we spend are 0.

Fixed-program **HarnessPAI** execution incurs no online high-level LLM API charge. This does not imply zero cost for perception, action-model inference, simulation, or hardware, and the execution-stage comparison does not establish total program-development or lifecycle costs.

5.6 Harness Trajectories for Post-Training

The closed loop of Section 4.2 produces something the action model never sees during its own training: successful trajectories under exactly the perturbations that break it. The evolved program is thoroughly validated on the training seeds and runs reliably, though on held-out seeds it does not reach a perfect success rate. This does not weaken the collected data. A hard success gate retains rollouts that satisfy the environment’s task-success condition and discards failed attempts, yielding success-filtered demonstrations for fine-tuning. The collection is cheap, for the reason Section 5.5 quantifies: a converged program executes open-loop, calling no paid model during rollout, so gathering expert trajectories costs roughly one second

per episode and \$0 of API cost. The alternatives are more expensive or absent altogether: a harness that deliberates at every step is priced at \$168-\$200 per fifty rollouts, and LIBERO’s fifty human-teleoperated demonstrations per task have no counterpart at all for the perturbed variants (Zhou et al., 2025). The harness thus doubles as an expert-data collector, and we ask here whether the robustness it achieves at execution time can be distilled back into the model’s weights.

We begin with a per-suite study. On $\pi_{0.5}$ -LIBERO, for each suite (Object, Goal, and Spatial) we collect the trajectories that the **HarnessPAI** harness produces on that suite alone, fine-tune the model on those trajectories, and evaluate the fine-tuned model on the same suite under the Swap and Task perturbations. Within a suite, the training data mixes base, Swap, and Task trajectories in equal proportion (1:1:1), and all of it is harvested by re-running the same converged program under the three conditions. On each task, seeds 0–39 form the training set and seeds 40–49 the test set.

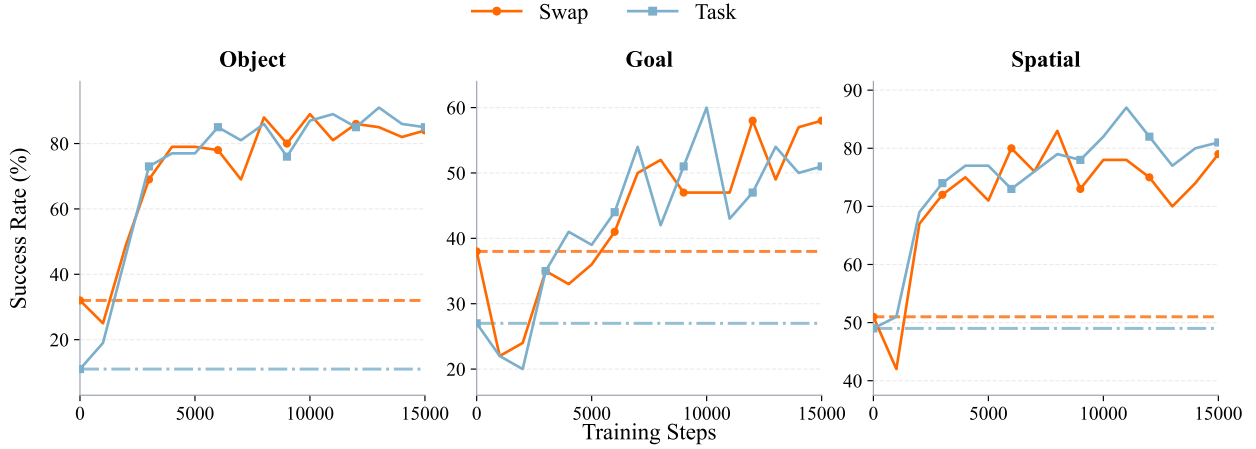


Figure 22 Per-suite post-training performance on test set. For each suite (Object, Goal, and Spatial) we fine-tune $\pi_{0.5}$ -LIBERO on the trajectories the **HarnessPAI** harness produces on that suite, with base, Swap, and Task trajectories in equal proportion (1:1:1), and plot success rate (%) on test set against training steps under the Swap and Task perturbations. On each task, seeds 0–39 form the training set and seeds 40–49 the test set.

Fig. 22 reports the per-suite result. On every suite, fine-tuning on its own harness trajectories lifts LIBERO-PRO success far above the baseline before fine-tuning, and each suite is reported at its best checkpoint within the 15,000-step training budget: Object from 21.5% to 88.0%, Spatial from 50.0% to 82.5%, and Goal from 32.5% to 54.5%. The closed loop thus produces a useful teaching signal on every suite, not just one.

We next compare the two checkpoints of interest: $\pi_{0.5}$ -Base, the pretrained model before any LIBERO fine-tuning, and $\pi_{0.5}$ -LIBERO, the official LIBERO-fine-tuned model (Intelligence et al., 2025b). For each checkpoint we fine-tune on the full mixture of harness trajectories: all three suites (Spatial, Object, and Goal) under all three conditions (base, Swap, and Task), with the nine categories sampled in equal proportion. The fine-tuned model is then evaluated on LIBERO-PRO (Zhou et al., 2025) uniformly across the three suites and the two perturbations (Swap and Task), 60 tasks in total. On each task we use seeds 0–39 for training and hold out seeds 40–49 for evaluation, so that the evaluation seeds are excluded from the additional harness-generated fine-tuning data.

Fig. 23 reports success rate against training steps. Fine-tuning on the full mixture lifts LIBERO-PRO success from 34.9% to 73.7% for $\pi_{0.5}$ -LIBERO and from near zero to 53.3% for $\pi_{0.5}$ -Base, each taken at its best checkpoint within the 30,000-step training budget. Both checkpoints improve across all three suites and both perturbations, so the harness trajectories carry a training signal the original demonstrations do not. The gain is most striking for $\pi_{0.5}$ -Base, which never saw LIBERO during pretraining yet acquires LIBERO-PRO ability from the harness data alone. This shows that the closed loop’s corrections encode task structure that standard behavior cloning omits, such as perception, geometry, and goal grounding. The converged code therefore plays two roles at once, a cheap collector of expert trajectories and, through them, a direct way to improve the action model it wraps.

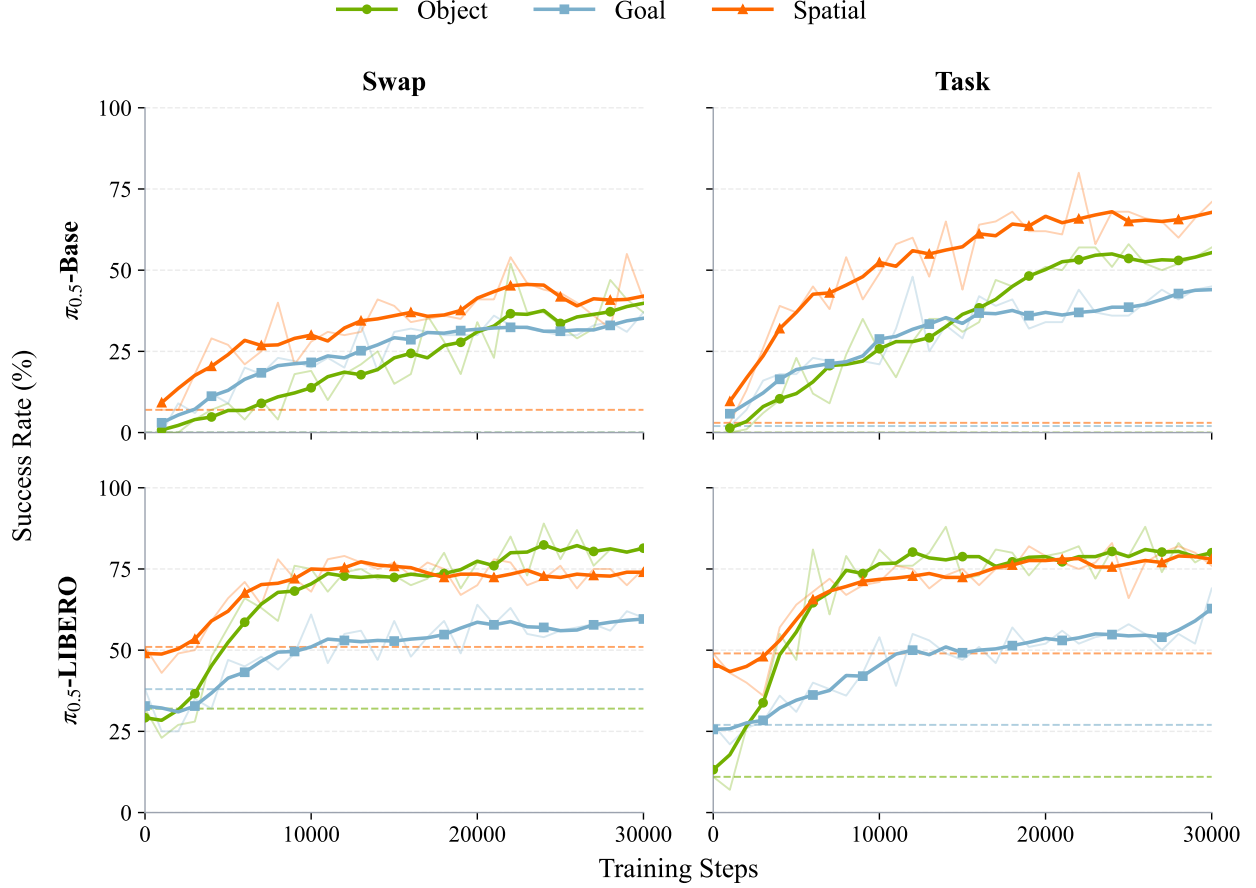


Figure 23 Post-training performance on test set. $\pi_{0.5}$ -Base and $\pi_{0.5}$ -LIBERO are fine-tuned on trajectories collected by the [HarnessPAI](#) harness from all three suites under the base, Swap, and Task conditions in equal proportion, and we plot success rate (%) on test set against training steps, evaluated on LIBERO-PRO across the Spatial, Object, and Goal suites under the Swap and Task perturbations. On each task, seeds 0–39 form the training set and seeds 40–49 the test set.

6 Discussion

Three questions frame what [HarnessPAI](#) does and does not do: why a hybrid of code and a frozen action model outperforms either of its ingredients alone, under what conditions that advantage holds, and where the demonstrations that seed the loop can come from. We take each in turn.

6.1 Why the Framework Works

[HarnessPAI](#)’s advantage is best read against the two extremes it refuses to occupy. A pure VLA is built around a single component, the action model, and the recipe that trains it quietly erodes the other two. The vision–language backbone that supplies semantic perception is overwritten by an action-prediction objective, so the resulting policy keeps its physical competence (the *action*) while its grounding (the *vision*) and its instruction understanding (the *language*) degrade; and its generative action head is reliable at coarse, low-frequency motion but unstable at the fine displacements that delicate manipulation demands. Section 3.1 isolated each failure: $\pi_{0.5}$ grasps the memorized bowl despite a negated instruction, reaches for the old plate position after two objects are swapped, and a WAM’s sampled press misses a small button. A pure code-as-policy system occupies the opposite extreme and fails for the mirror-image reason. Code supplies grounding, geometry, and verification, but it cannot manufacture contact-rich dexterity: the fine, high-frequency motions

a learned action head produces cheaply are precisely the ones hardest to write as explicit statements, which is why a code-only agent collapses to an 18.3% success rate on LIBERO (Table 4).

HarnessPAI sits at the intersection. It asks code to do what code does well, perceive *what is where*, re-ground a referring expression as an explicit predicate, reason geometrically, and verify each step before committing it, and it asks the frozen action model to do what only it does well: the atomic physical acts, delegated only at the phases the model is competent at. Neither ingredient is asked to substitute for the other. Code restores the vision and language the VLA lost and checks the action it retained, without asking code to reproduce that action from scratch; the model supplies the physical competence that no program can express, without being trusted to re-derive the scene or the instruction. The gain over each extreme is therefore the same gap, viewed from both sides: the framework replaces the degraded V and L of the VLA with code, and replaces the missing A of code with the VLA.

6.2 When the Framework Works

The framework is not a universal repair. Its value is conditional on a property of the underlying model that the previous subsection took for granted: a VLA is not purely end-to-end, and retains *atomic* competence, the ability to grasp a localized object or execute a short reach, even when it can no longer chain those primitives into a long-horizon plan or re-ground them under a perturbation. This is the property **HarnessPAI** exploits. The capability attribution of Section 4.2 labels each phase *good*, *needs code*, or *mixed*, and writes code only where the model is weak, delegating the competent atomic phases straight back to the VLA. The harness therefore organizes, grounds, and verifies, but it does not manufacture physical competence that is absent. When a model’s primitive abilities themselves fail, when it cannot grasp at all, rather than merely grasping the wrong object, no amount of code repair helps, because there is no competent action for the code to organize.

This boundary explains the one place the framework’s gains are comparatively small. On RoboCasa’s composite tasks (Table 5) the underlying WorldDreamer executes the middle of a long-horizon trajectory opaquely, and its intermediate atomic steps are both harder for code to inspect and weaker to begin with; **HarnessPAI** corrects the frame of the task but cannot steer the parts it cannot see. The precondition is the same in both directions: the framework improves a model in proportion to the atomic competence that model already has, and leaves untouched whatever the model fundamentally cannot do.

6.3 Where the Expert Data Comes From

The task graph of Section 4.2 is distilled from an expert demonstration, which might seem to chain the framework to human teleoperation. It does not. What the graph actually requires is a *skeleton*, which phases exist, and in what order, not a polished trajectory, and a skeleton can be read off far weaker data. A single successful *video*, one run that happened to reach the goal, even by chance, is enough to recover that skeleton, because the graph keeps only the phase structure and discards the imperfect motion that produced it. Data from a *similar* task serves the same role: the graph and the skill library are organized by node, so a grasp or transport phase learned on one task transfers to a novel task that shares the node (Section 4.4), and a related task’s demonstration can therefore seed a new one it was never written for. Neither source demands a human expert in the loop.

Once the harness has converged, this reliance shrinks further. As Section 4.5 describes, a converged code policy is itself a trajectory generator, and its successful rollouts, recorded in exactly the scenes where human demonstrations are absent, become the training data for the action model. The framework’s demand for expert data is thus weakest precisely where the supply is rarest: it needs a human label only to sketch a graph, and can bootstrap everything after that from occasional successes, related tasks, and its own converged rollouts.

7 Conclusion

This paper argued that Physical AI has been organized around a single component, the action model, and that this action-model-centric bias lets the agent’s perception, grounding, and planning atrophy inside an

end-to-end policy. We proposed **HarnessPAI**, a code framework that treats code not as a replacement for a VLA or WAM but as the executable harness that organizes perception, geometric reasoning, and action-model invocation. The framework separates two timescales: within a rollout it executes *open-loop*, grounding each decision and verifying each step before the world can be harmed; across rollouts it evolves *closed-loop*, feeding execution feedback back to revise the program and distilling each failure into a node-keyed skill that transfers to later tasks. Once the harness converges, its own successful trajectories are fed back to train the action model, closing the gap on tasks for which no expert demonstration exists.

We evaluated **HarnessPAI** across robot arms, a humanoid, and a mobile robot, driven by three generations of action interface, GraspNet, $\pi_{0.5}$, and WorldDreamer, together with a pure API containing no learned model. On LIBERO it reaches 98.1% overall, a new state of the art, and on LIBERO-PRO it attains 96.5% by recovering from the perturbations that collapse fixed policies. On RoboCasa it lifts WorldDreamer from 65.0% to 92.2% on atomic tasks. Because open-loop execution puts no model call on the critical path, these gains come nearly free at rollout time, orders of magnitude faster and without per-run API cost.

Taken together, the results indicate that the frontier of Physical AI lies not only in the action model but in the executable system that organizes, checks, and evolves it. Code, used as an interface rather than a substitute for learned control, is a practical path to more robust and more inspectable physical agents.

8 Future Work

HarnessPAI inherits much of its competence from the action model it wraps, so a natural direction for future work is to make that dependency as thin and uniform as an API. Whether the underlying model is a PPO-trained policy, a VLA, or a WAM of any paradigm, the ideal interface is a single contract: given an instruction, the model reliably completes one atomic action. Standardizing this contract would let the harness compose any action model without reasoning about its internals, and would let **HarnessPAI** benefit directly from progress across all model families.

A second direction is to reduce our reliance on online perception. The code depends heavily on perception to ground each decision, which is both a source of fragility and a recurring cost during evolution. Building a map or a scene model ahead of time, or closing the sim-to-real gap through direct real-to-sim transfer, could offload much of this burden, making perception both cheaper and more stable.

Finally, our method is not the opposite of directly invoking a coding agent; it is complementary to one. A coding agent can supply the open-ended reasoning that a fixed program cannot, while **HarnessPAI** supplies validated, reusable code packages. Combined, an agent completing a long-horizon task need not deliberate or invoke a model at every step; it can instead reuse or lightly adapt a code package of similar functionality, reserving its reasoning for genuinely novel situations.

References

- Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, et al. π_0 : A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.
- Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, et al. RT-1: Robotics Transformer for Real-World Control at Scale. In *Proceedings of Robotics: Science and Systems*, 2023.
- Jun Cen, Chaohui Yu, Hangjie Yuan, Yuming Jiang, Siteng Huang, Jiayan Guo, Xin Li, Yibing Song, Hao Luo, Fan Wang, Deli Zhao, and Hao Chen. WorldVLA: Towards autoregressive action world model. *arXiv preprint arXiv:2506.21539*, 2025.
- Chi-Lam Cheang, Guangzeng Chen, Ya Jing, Tao Kong, Hang Li, Yifeng Li, Yuxiao Liu, Hongtao Wu, Jiafeng Xu, Yichu Yang, Hanbo Zhang, and Minzhao Zhu. GR-2: A generative video-language-action model with web-scale knowledge for robot manipulation. *arXiv preprint arXiv:2410.06158*, 2024.
- Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8973–8979, 2019.
- 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. *arXiv preprint arXiv:2510.25889*, 2025a.
- Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. HarnessX: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026a.
- Yanzhe Chen, Zechen Bai, Zhijun Cao, Wenzheng Zeng, Kevin Qinghong Lin, Yiqi Lin, Guoqiang Liang, Kevin Yuchen Ma, Qiming Huang, and Mike Zheng Shou. Show-Harness: Just a VLM agent can play robots. *arXiv preprint arXiv:2609.10522*, 2026b.
- Yitong Chen, Zezheng Huai, Sixian Li, Yubang Wang, Haozhe Zhang, Yifei Zhang, Hechang Chen, Jingjing Gong, Yutang Jiang, and Xipeng Qiu. ETA: A new agentic paradigm for embodied tasks. *arXiv preprint arXiv:2608.03924*, 2026c.
- Zengjue Chen, Runliang Niu, He Kong, Qi Wang, Qianli Xing, and Zipei Fan. TGRPO: Fine-tuning vision-language-action model via trajectory-wise group relative policy optimization. *arXiv preprint arXiv:2506.08440*, 2025b.
- Xin Ding, Liang Mi, Mingzhe Huang, Zixuan Wang, Chao Zhang, Zixu Hao, Fu Chen, Xiangyu Li, Yikai Zheng, Yaoyu Guo, Weijun Wang, Kun Li, Hao Wu, Yunxin Liu, and Ting Cao. Zetta ζ : An efficient closed-loop embodied harness for self-evolving physical intelligence. *arXiv preprint arXiv:2608.16590*, 2026.
- Hao-Shu Fang, Chenxi Wang, Minghao Gou, and Cewu Lu. GraspNet-1Billion: A large-scale benchmark for general object grasping. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11441–11450, 2020.
- Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Li Fei-Fei, et al. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. In *International Conference on Machine Learning*, 2026.
- Zhaopeng Gu, Bingke Zhu, Tianxi Lin, Guibo Zhu, Yingying Chen, Kai Wang, Tingyu Yuan, Chaoyang Zhao, Zhaowen Li, Peng Su, and Jinqiao Wang. HarnessWAM: Bridging prediction and deliberation in world action models. *arXiv preprint arXiv:2608.09516*, 2026.
- J. Haviland, N. Sünderhauf, and P. Corke. A holistic approach to reactive mobile manipulation. *IEEE Robotics and Automation Letters*, 7:3122–3129, 2022.
- Chi-Pin Huang, Yueh-Hua Wu, Min-Hung Chen, Frank Wang, and Fu-En Yang. ThinkAct: Vision-language-action reasoning via reinforced visual latent planning. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 82782–82802, 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. In *Proceedings of The 7th Conference on Robot Learning*, volume 229, pages 540–562, 2023.

- Chia-Yu Hung, Qi Sun, Pengfei Hong, Amir Zadeh, Chuan Li, U-Xuan Tan, Navonil Majumder, and Soujanya Poria. NORA: A small open-sourced generalist vision language action model for embodied tasks. *arXiv preprint arXiv:2504.19854*, 2025.
- Physical Intelligence, Ali Amin, Raichelle Aniceto, Ashwin Balakrishna, Kevin Black, Ken Conley, Grace Connors, James Darpinian, et al. $\pi_{0.6}^*$: a vla that learns from experience. *arXiv preprint arXiv:2511.14759*, 2025a.
- Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Es-mail, Michael Equi, et al. $\pi_{0.5}$: a vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025b.
- Physical Intelligence, Bo Ai, Ali Amin, Raichelle Aniceto, Ashwin Balakrishna, Greg Balke, Kevin Black, George Bokinsky, et al. $\pi_{0.7}$: a steerable generalist robotic foundation model with emergent capabilities. *arXiv preprint arXiv:2604.154883*, 2026.
- Yunfan Jiang, Chen Wang, Ruohan Zhang, Jiajun Wu, and Li Fei-Fei. TRANSIC: Sim-to-real policy transfer by learning from online correction. In *Proceedings of The 8th Conference on Robot Learning*, 2025.
- Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P Foster, et al. OpenVLA: An open-source vision-language-action model. In *Proceedings of The 8th Conference on Robot Learning*, volume 270, pages 2679–2713, 2024.
- Jason Lee, Jiafei Duan, Haoquan Fang, Yuquan Deng, Shuo Liu, Boyang Li, Bohan Fang, Jieyu Zhang, Yi Ru Wang, Sangho Lee, Winson Han, Wilbert Pumacay, Angelica Wu, Rose Hendrix, Karen Farley, Eli VanderBilt, Ali Farhadi, Dieter Fox, and Ranjay Krishna. MolmoAct: Action reasoning models that can reason in space. *arXiv preprint arXiv:2508.07917*, 2025.
- Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, et al. BEHAVIOR-1K: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Proceedings of The 6th Conference on Robot Learning*, volume 205, pages 80–93, 2023.
- Xuanlin Li, Kyle Hsu, Jiayuan Gu, Oier Mees, Karl Pertsch, Homer Rich Walke, Chuyuan Fu, Ishikaa Lunawat, Isabel Sieh, Sean Kirmani, Sergey Levine, Jiajun Wu, Chelsea Finn, Hao Su, Quan Vuong, and Ted Xiao. Evaluating real-world robot manipulation policies in simulation. In *Proceedings of The 8th Conference on Robot Learning*, 2025.
- Zhuoran Li, Zhiyang Li, Kaijun Zhou, and Jinyu Gu. RoboHarness: A memory-augmented policy harness for vision-language-action model robustness via in-context adaptation. *arXiv preprint arXiv:2603.24060*, 2026.
- 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)*, pages 9493–9500, 2023.
- Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 44776–44791, 2023.
- Xule Liu, Hanlin Teng, Chao Li, Yanan Ni, Shuo Lu, Audrey Wang, Yijun Liu, Yunfei Wang, Xiaofeng Li, Xian Yi, Yuanfa Li, Kang Zhao, Jian Liang, Yuxuan Chen, Jinyuan Chen, Heng Qu, Kun Shao, and Jian Luan. Mi-Memory: A lifecycle memory framework for personal ai. *arXiv preprint arXiv:2607.18975*, 2026a.
- Yang Liu, Weixing Chen, Xinshuai Song, Tao Pu, Siwen Mo, Yongjie Bai, Zihao Chen, Qianran Sun, Liruo Zhong, Ying Shen, and Liang Lin. PhyAgentOS: A self-evolving operating system for embodied agents with decoupled cognitive planning and physical execution. *arXiv preprint arXiv:2607.16636*, 2026b.
- Guanxing Lu, Wenkai Guo, Chubin Zhang, Yuheng Zhou, Haonan Jiang, Zifeng Gao, Yansong Tang, and Ziwei Wang. VLA-RL: Towards masterful and general robotic manipulation with scalable reinforcement learning. *arXiv preprint arXiv:2505.18719*, 2025.
- 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. *arXiv preprint arXiv:2607.00272*, 2026.
- Yecheng Jason Ma, William Liang, Hung-Ju Wang, Sam Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. DrEureka: Language model guided sim-to-real transfer. *arXiv preprint arXiv:2406.01967*, 2024.

- Mark O. Mints and Peer Neubert. Ros 2 und webots in der lehre, 2023. URL https://roscon2023.de/presentations/S2_P6___ROS2_und_Webots_in_der_Lehre.pdf.
- Mmints. VacuSim, 2026. URL <https://github.com/mmints/vacusim>.
- 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.
- NVIDIA, :, Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi "Jim" Fan, Yu Fang, et al. GR00T N1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.
- Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3803–3810, 2018.
- Pollen-robotics. MicroDuck rl, 2026. URL https://github.com/pollen-robotics/microduck_rl.
- Jingyang Qiao, Weicheng Meng, Yu Cheng, Zhihang Lin, Zhizhong Zhang, Xin Tan, Jingyu Gong, Kun Shao, and Yuan Xie. Memory intelligence agent. *arXiv preprint arXiv:2604.04503*, 2026.
- RLinf. DreamZero, 2026. URL <https://huggingface.co/collections/RLinf/dreamzero>.
- RLWRLD. RLDX-1: A dexterity-first robotics foundation model, 2026. URL <https://github.com/RLWRLD/RLDX-1>.
- Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, Simon Alibert, Matthieu Cord, Thomas Wolf, and Remi Cadene. 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. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530, 2023.
- Xiaoquan Sun, Zetian Xu, Chen Cao, Zonghe Liu, Yihan Sun, Jingrui Pang, Ruijian Zhang, Zhen Yang, Kang Pang, Dingxin He, Mingqi Yuan, and Jiayu Chen. AtomVLA: Scalable post-training for robotic manipulation via predictive latent world models. *arXiv preprint arXiv:2603.08519*, 2026.
- Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, et al. Octo: An open-source generalist robot policy. *arXiv preprint arXiv:2405.12213*, 2024.
- Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017.
- Marcel Torne, Anthony Simeonov, Zechu Li, April Chan, Tao Chen, Abhishek Gupta, and Pulkit Agrawal. Reconciling reality through simulation: A real-to-sim-to-real approach for robust manipulation. *arXiv preprint arXiv:2403.03949*, 2024.
- Xiaofeng Wang, Zheng Zhu, Guan Huang, Boyuan Wang, Xinze Chen, and Jiwen Lu. WorldDreamer: Towards general world models for video generation via predicting masked tokens. *arXiv preprint arXiv:2401.09985*, 2024.
- worldAgents-c. WorldDreamer robocasa365 evaluation, 2026. URL https://github.com/worldAgents-c/world_dreamer_server-robocasa365-multi_task/tree/main.
- Hongtao Wu, Ya Jing, Chilam Cheang, Guangzeng Chen, Jiafeng Xu, Xinghang Li, Minghuan Liu, Hang Li, and Tao Kong. Unleashing large-scale video generative pre-training for visual robot manipulation. In *International Conference on Learning Representations*, volume 2024, pages 10641–10662, 2024.
- Wenhao Wu, Menghao Zhang, Xin Wang, Zhi Wang, Kun Shao, and Jian Luan. TRACE: A self-evolving skill bank for consistent, limit-aware LLM agents. *arXiv preprint arXiv:2608.22793*, 2026.
- Junjin Xiao, Yandan Yang, Xinyuan Chang, Ronghan Chen, Feng Xiong, Mu Xu, Wei-Shi Zheng, and Qing Zhang. World-Env: Leveraging world model as a virtual environment for VLA post-training. *arXiv preprint arXiv:2509.24948*, 2025.

- 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*, 2026.
- 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.
- Seonghyeon Ye, Yunhao Ge, Kaiyuan Zheng, Shenyuan Gao, Sihyun Yu, George Kurian, Suneel Indupuru, You Liang Tan, et al. World action models are zero-shot policies. *arXiv preprint arXiv:2602.15922*, 2026.
- Tianyuan Yuan, Zibin Dong, Yicheng Liu, and Hang Zhao. Fast-wam: Do world action models need test-time future imagination? *arXiv preprint arXiv:2603.16666*, 2026.
- Wentao Zhang, Aolan Sun, Wentao Mo, Xiaoyang Qu, Yuxin Zheng, and Jianzong Wang. From knowing to doing precisely: A general self-correction and termination framework for vla models. In *ICASSP 2026 - 2026 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 20581–20585, 2026a.
- Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, Chunyang Zhu, Jiaxing Qiu, Yuchen Yan, Jiyuan Liu, Wenhao Tang, Zhengru Fang, Yi Nie, Changxu Wei, Yu Wang, Wenbo Ding, and Chao Yu. Harness VLA: Steering frozen vlases into reliable manipulation primitives via memory-guided agents. *arXiv preprint arXiv:2607.08448*, 2026b.
- Zijian Zhang, Kaiyuan Zheng, Zhaorun Chen, Joel Jang, Yi Li, Siwei Han, Chaoqi Wang, Mingyu Ding, Dieter Fox, and Huaxiu Yao. GRAPE: Generalizing robot policy via preference alignment. *arXiv preprint arXiv:2411.19309*, 2024.
- Qingqing Zhao, Yao Lu, Moo Jin Kim, Zipeng Fu, Zhuoyang Zhang, Yecheng Wu, Zhaoshuo Li, Qianli Ma, Song Han, Chelsea Finn, Ankur Handa, Tsung-Yi Lin, Gordon Wetzstein, Ming-Yu Liu, and Donglai Xiang. CoT-VLA: Visual chain-of-thought reasoning for vision-language-action models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1702–1713, 2025.
- Ruijie Zheng, Yongyuan Liang, Shuaiyi Huang, Jianfeng Gao, Hal Daumé III, Andrey Kolobov, Furong Huang, and Jianwei Yang. TraceVLA: Visual trace prompting enhances spatial-temporal awareness for generalist robotic policies. In *The Thirteenth International Conference on Learning Representations*, 2025.
- 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.
- Chuning Zhu, Raymond Yu, Siyuan Feng, Benjamin Burchfiel, Paarth Shah, and Abhishek Gupta. Unified world models: Coupling video and action diffusion for pretraining on large robotic datasets. *arXiv preprint arXiv:2504.02792*, 2025.
- Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. *arXiv preprint arXiv:2009.12293*, 2020.
- Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Proceedings of The 7th Conference on Robot Learning*, volume 229, pages 2165–2183, 2023.

Contributions and Acknowledgments

Core Contributors.

- Xin Wang^{1,2,*}
- Wenhao Wu^{1,3,*}
- Menghao Zhang^{1,*}
- Zhi Wang^{3,†}
- Kun Shao^{1,†,‡}
- Jian Luan^{1,†}

Contributors.

- Yang Li⁴
- Qing Li¹
- Shangding Gu⁴
- Huichi Zhou⁶
- Shuqing Shi⁷
- Fei Ni⁸
- Shuo Lu⁹
- Weicheng Meng¹⁰
- Kang Li¹¹
- Jin Wu⁵
- Kang Zhao¹
- Shangmin Guo¹²
- Gen Li¹³
- Yongqiang Tang⁹
- Zhizhong Zhang⁵
- Yuan Xie⁵
- Heng Qu¹

* Equal contribution. † Corresponding author. ‡ Project lead.

¹Xiaomi Inc., ²Tsinghua University, ³Nanjing University, ⁴Shanghai Jiao Tong University, ⁵East China Normal University, ⁶University College London, ⁷King's College London, ⁸Imperial College London, ⁹Institute of Automation, Chinese Academy of Sciences, ¹⁰Shanghai Innovation Institute, ¹¹University of Oxford, ¹²University of Edinburgh, ¹³Nanyang Technological University.

Appendix

A Main Benchmark Results Table

Table 6 reports the per-task success rate on robosuite underlying Fig. 12. The final row gives the improvement of **HarnessPAI** over ASPIRE.

Table 6 Per-task success rate on robosuite (%).

Method	2A-Hand	2A-Lift	Insert	Lift	Stack	Restack	Wipe	Avg
CaP-Agent	20	74	0	97	98	89	100	68.3
Human	91	94	80	93	73	47	100	82.6
ASPIRE	92	71	9	97	99	100	99	81.0
HarnessPAI	100	91	83	100	100	100	100	96.3
Δ vs. ASPIRE	+8.0	+20.0	+74.0	+3.0	+1.0	0.0	+1.0	+15.3

Table 7 reports the full LIBERO-PRO comparison underlying Fig. 13, aggregated by suite and perturbation across all eight baselines. The final row gives the improvement of **HarnessPAI** over the strongest pure VLA, $\pi_{0.5}$.

Table 7 LIBERO-PRO success rate by suite and perturbation (%).

Method	Object		Goal		Spatial		Avg
	Swap	Task	Swap	Task	Swap	Task	
OpenVLA	0	0	0	0	0	0	0.0
π_0	0	0	0	0	0	0	0.0
$\pi_{0.5}$	29.8	10.8	37.6	29.0	51.8	50.6	34.9
CaP-Agent	22	18	26	17	12	14	18.2
ASPIRE	98	95	81	45	51	60	71.7
Harness VLA (Codex)	91	94	66	75	69	81	79.3
Harness VLA (CC)	90	88	87	87	80	94	87.7
HarnessPAI	98.6	99.2	94.6	97.8	93.8	95.2	96.5
Δ vs. $\pi_{0.5}$	+68.8	+88.4	+57.0	+68.8	+42.0	+44.6	+61.6

B Detailed Results

B.1 LIBERO

Table 8 reports the per-task results of our method on the three LIBERO suites, which underlie the Spatial, Object, and Goal columns in Table 4.

Table 9 reports the per-task results of the underlying $\pi_{0.5}$ -LIBERO model on the three LIBERO suites, which serve as the baseline against which the **HarnessPAI** results in Table 8 are measured.

Table 10 reports the per-task results of DreamZero-LIBERO, our own evaluation of the World Action Model DreamZero on standard LIBERO, which underlie the DreamZero-LIBERO row in Table 4.

B.2 LIBERO-PRO Perturbations

Table 11 reports the per-task breakdown of **HarnessPAI** under the two LIBERO-PRO perturbation axes we evaluate. Table 12 reports the per-task results of the underlying $\pi_{0.5}$ model on LIBERO-PRO under the Swap and Task perturbations, the baseline against which the **HarnessPAI** results in Table 11 are measured.

Table 8 Per-task results of **HarnessPAI** on the three LIBERO suites (success rate, %). All results are our own evaluation over 50 seeds per task.

Task	Description	Success / Total	Success Rate
LIBERO-Object			
Task 0	Pick up the alphabet soup and place it in the basket	50/50	100.0
Task 1	Pick up the cream cheese and place it in the basket	50/50	100.0
Task 2	Pick up the salad dressing and place it in the basket	50/50	100.0
Task 3	Pick up the BBQ sauce and place it in the basket	50/50	100.0
Task 4	Pick up the ketchup and place it in the basket	50/50	100.0
Task 5	Pick up the tomato sauce and place it in the basket	50/50	100.0
Task 6	Pick up the butter and place it in the basket	50/50	100.0
Task 7	Pick up the milk and place it in the basket	50/50	100.0
Task 8	Pick up the chocolate pudding and place it in the basket	50/50	100.0
Task 9	Pick up the orange juice and place it in the basket	50/50	100.0
Object subtotal		500/500	100.0
LIBERO-Spatial			
Task 0	Pick up the black bowl between the plate and the ramekin and place it on the plate	50/50	100.0
Task 1	Pick up the black bowl next to the ramekin and place it on the plate	50/50	100.0
Task 2	Pick up the black bowl from table center and place it on the plate	50/50	100.0
Task 3	Pick up the black bowl on the cookie box and place it on the plate	48/50	96.0
Task 4	Pick up the black bowl in the top drawer of the wooden cabinet and place it on the plate	44/50	88.0
Task 5	Pick up the black bowl on the ramekin and place it on the plate	48/50	96.0
Task 6	Pick up the black bowl next to the cookie box and place it on the plate	50/50	100.0
Task 7	Pick up the black bowl on the stove and place it on the plate	47/50	94.0
Task 8	Pick up the black bowl next to the plate and place it on the plate	50/50	100.0
Task 9	Pick up the black bowl on the wooden cabinet and place it on the plate	49/50	98.0
Spatial subtotal		486/500	97.2
LIBERO-Goal			
Task 0	Open the middle drawer of the cabinet	46/50	92.0
Task 1	Put the bowl on the stove	50/50	100.0
Task 2	Put the wine bottle on top of the cabinet	49/50	98.0
Task 3	Open the top drawer and put the bowl inside	44/50	88.0
Task 4	Put the bowl on top of the cabinet	48/50	96.0
Task 5	Push the plate to the front of the stove	50/50	100.0
Task 6	Put the cream cheese in the bowl	50/50	100.0
Task 7	Turn on the stove	50/50	100.0
Task 8	Put the bowl on the plate	49/50	98.0
Task 9	Put the wine bottle on the rack	50/50	100.0
Goal subtotal		486/500	97.2
Overall		1472/1500	98.1

Table 9 Per-task results of $\pi_{0.5}$ on the three LIBERO suites (success rate, %). All results are our own evaluation over 50 seeds per task.

Task	Description	Success / Total	Success Rate
LIBERO-Object			
Task 0	Pick up the alphabet soup and place it in the basket	50/50	100.0
Task 1	Pick up the cream cheese and place it in the basket	50/50	100.0
Task 2	Pick up the salad dressing and place it in the basket	50/50	100.0
Task 3	Pick up the BBQ sauce and place it in the basket	50/50	100.0
Task 4	Pick up the ketchup and place it in the basket	50/50	100.0
Task 5	Pick up the tomato sauce and place it in the basket	50/50	100.0
Task 6	Pick up the butter and place it in the basket	50/50	100.0
Task 7	Pick up the milk and place it in the basket	50/50	100.0
Task 8	Pick up the chocolate pudding and place it in the basket	50/50	100.0
Task 9	Pick up the orange juice and place it in the basket	50/50	100.0
Object subtotal		500/500	100.0
LIBERO-Spatial			
Task 0	Pick up the black bowl between the plate and the ramekin and place it on the plate	50/50	100.0
Task 1	Pick up the black bowl next to the ramekin and place it on the plate	50/50	100.0
Task 2	Pick up the black bowl from table center and place it on the plate	49/50	98.0
Task 3	Pick up the black bowl on the cookie box and place it on the plate	48/50	96.0
Task 4	Pick up the black bowl in the top drawer of the wooden cabinet and place it on the plate	34/50	68.0
Task 5	Pick up the black bowl on the ramekin and place it on the plate	50/50	100.0
Task 6	Pick up the black bowl next to the cookie box and place it on the plate	50/50	100.0
Task 7	Pick up the black bowl on the stove and place it on the plate	48/50	96.0
Task 8	Pick up the black bowl next to the plate and place it on the plate	50/50	100.0
Task 9	Pick up the black bowl on the wooden cabinet and place it on the plate	49/50	98.0
Spatial subtotal		478/500	95.6
LIBERO-Goal			
Task 0	Open the middle drawer of the cabinet	36/50	72.0
Task 1	Put the bowl on the stove	50/50	100.0
Task 2	Put the wine bottle on top of the cabinet	50/50	100.0
Task 3	Open the top drawer and put the bowl inside	46/50	92.0
Task 4	Put the bowl on top of the cabinet	47/50	94.0
Task 5	Push the plate to the front of the stove	50/50	100.0
Task 6	Put the cream cheese in the bowl	50/50	100.0
Task 7	Turn on the stove	50/50	100.0
Task 8	Put the bowl on the plate	48/50	96.0
Task 9	Put the wine bottle on the rack	49/50	98.0
Goal subtotal		476/500	95.2
Overall		1454/1500	96.9

Table 10 Per-task results of DreamZero-LIBERO on the three LIBERO suites (success rate, %). All results are our own evaluation over 50 seeds per task.

Task	Description	Success / Total	Success Rate
LIBERO-Object			
Task 0	Pick up the alphabet soup and place it in the basket	45/50	90.0
Task 1	Pick up the cream cheese and place it in the basket	50/50	100.0
Task 2	Pick up the salad dressing and place it in the basket	50/50	100.0
Task 3	Pick up the BBQ sauce and place it in the basket	46/50	92.0
Task 4	Pick up the ketchup and place it in the basket	41/50	82.0
Task 5	Pick up the tomato sauce and place it in the basket	50/50	100.0
Task 6	Pick up the butter and place it in the basket	46/50	92.0
Task 7	Pick up the milk and place it in the basket	28/50	56.0
Task 8	Pick up the chocolate pudding and place it in the basket	48/50	96.0
Task 9	Pick up the orange juice and place it in the basket	47/50	94.0
Object subtotal		451/500	90.2
LIBERO-Spatial			
Task 0	Pick up the black bowl between the plate and the ramekin and place it on the plate	29/50	58.0
Task 1	Pick up the black bowl next to the ramekin and place it on the plate	43/50	86.0
Task 2	Pick up the black bowl from table center and place it on the plate	47/50	94.0
Task 3	Pick up the black bowl on the cookie box and place it on the plate	48/50	96.0
Task 4	Pick up the black bowl in the top drawer of the wooden cabinet and place it on the plate	35/50	70.0
Task 5	Pick up the black bowl on the ramekin and place it on the plate	34/50	68.0
Task 6	Pick up the black bowl next to the cookie box and place it on the plate	48/50	96.0
Task 7	Pick up the black bowl on the stove and place it on the plate	33/50	66.0
Task 8	Pick up the black bowl next to the plate and place it on the plate	42/50	84.0
Task 9	Pick up the black bowl on the wooden cabinet and place it on the plate	49/50	98.0
Spatial subtotal		408/500	81.6
LIBERO-Goal			
Task 0	Open the middle drawer of the cabinet	24/50	48.0
Task 1	Put the bowl on the stove	44/50	88.0
Task 2	Put the wine bottle on top of the cabinet	16/50	32.0
Task 3	Open the top drawer and put the bowl inside	7/50	14.0
Task 4	Put the bowl on top of the cabinet	0/50	0.0
Task 5	Push the plate to the front of the stove	13/50	26.0
Task 6	Put the cream cheese in the bowl	12/50	24.0
Task 7	Turn on the stove	45/50	90.0
Task 8	Put the bowl on the plate	43/50	86.0
Task 9	Put the wine bottle on the rack	2/50	4.0
Goal subtotal		206/500	41.2
Overall		1065/1500	71.0

Table 11 Per-task results of **HarnessPAI** on LIBERO-PRO under Swap and Task perturbations (success rate, %).

Task	Swap			Task		
	Succ./Total	Fail	Rate	Succ./Total	Fail	Rate
LIBERO-Object						
Task 0	50/50	0	100.0	50/50	0	100.0
Task 1	50/50	0	100.0	50/50	0	100.0
Task 2	50/50	0	100.0	47/50	3	94.0
Task 3	50/50	0	100.0	50/50	0	100.0
Task 4	50/50	0	100.0	49/50	1	98.0
Task 5	45/50	5	90.0	50/50	0	100.0
Task 6	50/50	0	100.0	50/50	0	100.0
Task 7	50/50	0	100.0	50/50	0	100.0
Task 8	50/50	0	100.0	50/50	0	100.0
Task 9	48/50	2	96.0	50/50	0	100.0
Object subtotal	493/500	7	98.6	496/500	4	99.2
LIBERO-Goal						
Task 0	44/50	6	88.0	49/50	1	98.0
Task 1	49/50	1	98.0	45/50	5	90.0
Task 2	43/50	7	86.0	47/50	3	94.0
Task 3	48/50	2	96.0	49/50	1	98.0
Task 4	49/50	1	98.0	49/50	1	98.0
Task 5	48/50	2	96.0	50/50	0	100.0
Task 6	47/50	3	94.0	50/50	0	100.0
Task 7	50/50	0	100.0	50/50	0	100.0
Task 8	50/50	0	100.0	50/50	0	100.0
Task 9	45/50	5	90.0	50/50	0	100.0
Goal subtotal	473/500	27	94.6	489/500	11	97.8
LIBERO-Spatial						
Task 0	50/50	0	100.0	46/50	4	92.0
Task 1	50/50	0	100.0	50/50	0	100.0
Task 2	49/50	1	98.0	45/50	5	90.0
Task 3	45/50	5	90.0	45/50	5	90.0
Task 4	46/50	4	92.0	44/50	6	88.0
Task 5	50/50	0	100.0	50/50	0	100.0
Task 6	45/50	5	90.0	49/50	1	98.0
Task 7	39/50	11	78.0	50/50	0	100.0
Task 8	50/50	0	100.0	48/50	2	96.0
Task 9	45/50	5	90.0	49/50	1	98.0
Spatial subtotal	469/500	31	93.8	476/500	24	95.2
Overall	1435/1500	65	95.7	1461/1500	39	97.4

Table 12 Per-task results of $\pi_{0.5}$ on LIBERO-PRO under Swap and Task perturbations (success rate, %).

Task	Swap			Task		
	Succ./Total	Fail	Rate	Succ./Total	Fail	Rate
LIBERO-Object						
Task 0	41/50	9	82.0	0/50	50	0.0
Task 1	50/50	0	100.0	50/50	0	100.0
Task 2	0/50	50	0.0	0/50	50	0.0
Task 3	5/50	45	10.0	0/50	50	0.0
Task 4	0/50	50	0.0	0/50	50	0.0
Task 5	0/50	50	0.0	0/50	50	0.0
Task 6	44/50	6	88.0	0/50	50	0.0
Task 7	0/50	50	0.0	4/50	46	8.0
Task 8	0/50	50	0.0	0/50	50	0.0
Task 9	9/50	41	18.0	0/50	50	0.0
Object subtotal	149/500	351	29.8	54/500	446	10.8
LIBERO-Goal						
Task 0	0/50	50	0.0	0/50	50	0.0
Task 1	50/50	0	100.0	8/50	42	16.0
Task 2	0/50	50	0.0	15/50	35	30.0
Task 3	50/50	0	100.0	17/50	33	34.0
Task 4	0/50	50	0.0	12/50	38	24.0
Task 5	0/50	50	0.0	6/50	44	12.0
Task 6	0/50	50	0.0	9/50	41	18.0
Task 7	49/50	1	98.0	46/50	4	92.0
Task 8	37/50	13	74.0	32/50	18	64.0
Task 9	2/50	48	4.0	0/50	50	0.0
Goal subtotal	188/500	312	37.6	145/500	355	29.0
LIBERO-Spatial						
Task 0	49/50	1	98.0	0/50	50	0.0
Task 1	25/50	25	50.0	44/50	6	88.0
Task 2	50/50	0	100.0	0/50	50	0.0
Task 3	0/50	50	0.0	3/50	47	6.0
Task 4	41/50	9	82.0	0/50	50	0.0
Task 5	44/50	6	88.0	46/50	4	92.0
Task 6	0/50	50	0.0	48/50	2	96.0
Task 7	0/50	50	0.0	49/50	1	98.0
Task 8	50/50	0	100.0	13/50	37	26.0
Task 9	0/50	50	0.0	50/50	0	100.0
Spatial subtotal	259/500	241	51.8	253/500	247	50.6
Overall	596/1500	904	39.7	452/1500	1048	30.1

B.3 RoboCasa

Table 13 reports the per-task results on the RoboCasa atomic tasks that underlie the aggregate in Table 5. Table 14 reports the per-task results of the underlying WorldDreamer WAM on the same RoboCasa atomic tasks, the baseline against which the HarnessPAI results in Table 13 are measured.

Table 13 Per-task results on RoboCasa atomic tasks (success rate, %).

Task	Name	Layout / Style	Success / Total	Success Rate
Task 0	CloseBlenderLid	8 / 8	19/20	95.0
Task 1	CloseFridge	2 / 2	20/20	100.0
Task 2	CloseToasterOvenDoor	7 / 7	19/20	95.0
Task 3	CoffeeSetupMug	4 / 4	16/20	80.0
Task 4	NavigateKitchen	5 / 5	20/20	100.0
Task 5	OpenCabinet	7 / 7	18/20	90.0
Task 6	OpenDrawer	7 / 7	20/20	100.0
Task 7	OpenStandMixerHead	2 / 2	19/20	95.0
Task 8	PickPlaceCounterToCabinet	2 / 2	19/20	95.0
Task 9	PickPlaceCounterToStove	6 / 6	16/20	80.0
Task 10	PickPlaceDrawerToCounter	10 / 10	15/20	75.0
Task 11	PickPlaceSinkToCounter	5 / 5	17/20	85.0
Task 12	PickPlaceToasterToCounter	5 / 5	20/20	100.0
Task 13	SlideDishwasherRack	2 / 2	19/20	95.0
Task 14	TurnOffStove	2 / 2	18/20	90.0
Task 15	TurnOnElectricKettle	3 / 3	18/20	90.0
Task 16	TurnOnMicrowave	2 / 2	20/20	100.0
Task 17	TurnOnSinkFaucet	2 / 2	19/20	95.0
Total			332/360	92.2

Table 14 Per-task results of the underlying WorldDreamer WAM on RoboCasa atomic tasks (success rate, %).

Task	Name	Layout / Style	Success / Total	Success Rate
Task 0	CloseBlenderLid	8 / 8	6/20	30.0
Task 1	CloseFridge	2 / 2	20/20	100.0
Task 2	CloseToasterOvenDoor	7 / 7	8/20	40.0
Task 3	CoffeeSetupMug	4 / 4	8/20	40.0
Task 4	NavigateKitchen	5 / 5	8/20	40.0
Task 5	OpenCabinet	7 / 7	18/20	90.0
Task 6	OpenDrawer	7 / 7	14/20	70.0
Task 7	OpenStandMixerHead	2 / 2	20/20	100.0
Task 8	PickPlaceCounterToCabinet	2 / 2	20/20	100.0
Task 9	PickPlaceCounterToStove	6 / 6	16/20	80.0
Task 10	PickPlaceDrawerToCounter	10 / 10	15/20	75.0
Task 11	PickPlaceSinkToCounter	5 / 5	20/20	100.0
Task 12	PickPlaceToasterToCounter	5 / 5	18/20	90.0
Task 13	SlideDishwasherRack	2 / 2	13/20	65.0
Task 14	TurnOffStove	2 / 2	0/20	0.0
Task 15	TurnOnElectricKettle	3 / 3	20/20	100.0
Task 16	TurnOnMicrowave	2 / 2	0/20	0.0
Task 17	TurnOnSinkFaucet	2 / 2	10/20	50.0
Total			234/360	65.0

B.4 Harness Zero-shot

Table 15 reports the per-task success rates underlying Fig. 18: the zero-shot transfer of the **HarnessPAI** harness to DreamZero, a World Action Model. The harness evolved on $\pi_{0.5}$ over LIBERO-Object is applied unchanged to DreamZero, and we report per-task success rate (%) under the Swap and Task perturbations of LIBERO-PRO.

Table 15 Per-task results of DreamZero and DreamZero with **HarnessPAI** on LIBERO-PRO Object Suite under Swap and Task perturbation (success rate, %)

Task	DreamZero (WAM)		HarnessPAI (<i>zero-shot</i>)	
	Swap	Task	Swap	Task
task0 (alphabet soup)	0.0	0.0	100.0	94.0
task1 (cream cheese)	0.0	96.0	84.0	100.0
task2 (salad dressing)	0.0	0.0	100.0	100.0
task3 (BBQ sauce)	0.0	0.0	84.0	96.0
task4 (ketchup)	0.0	0.0	96.0	78.0
task5 (tomato sauce)	0.0	0.0	56.0	100.0
task6 (butter)	6.0	0.0	10.0	66.0
task7 (milk)	0.0	0.0	46.0	34.0
task8 (chocolate pudding)	0.0	0.0	98.0	96.0
task9 (orange juice)	0.0	0.0	60.0	80.0
Overall	0.6	9.6	73.4	84.4

C Skills Case

The closed loop leaves behind more than working code: each failure is distilled into a problem–repair note mounted on the task-graph node where it occurred (Section 4.4). This appendix reproduces one representative place-phase skill, `align-object-to-basket-mouth`, verbatim as it is stored in the skill library.

```
align-object-to-basket-mouth.md
---
name: align-object-to-basket-mouth
description: Diagnose and repair place-phase 3D alignment in LIBERO-object pick-and-place. Use
  when the object is 3D-aligned to the basket body instead of the mouth -- the place target
  is biased toward the camera in depth, the rim Z undershoots the opening, or the object
  clips the rim / misses the contain_region despite correct 3D object-center alignment. This
  bias masquerades as a transport drop or a release failure.
phase: place
source_tasks: ["pick-and-place-in-basket"]
---

# align-object-to-basket-mouth

## Problem Encountered
The place phase 3D-servos the object center to `basket + object_to_eef` (see
`align-object-3d-over-basket`). That only drops the object through the opening
when `basket` is the mouth. `basket_center_world` returned the median XY of
all back-projected mask points plus an 82nd-percentile Z. From the angled
agentview, the visible near wall folds into that XY median, biasing the
target toward the camera, and the percentile Z undershoots the true rim.

Measured on `libero_object_task` task2 ep51: the body estimate was
`[0.0585, 0.2653, 0.1321]` while the mouth is `[0.0056, 0.2708, 0.1386]` -- a
```

****5.3 cm X (depth) error****. Place "succeeded" (the object center reached its target), yet the object was 5 cm off the opening, clipped the rim, and never entered the ``contain_region``. This was misdiagnosed as a transport drop (ep51/ep53) and a release timing bug (ep54); all three were one place-target bias.

How to Repair

Workflow

1. Make the place 3D-align target the basket ****mouth****. In ``basket_center_world``, keep only the upper-Z rim points -- the same top-slice ``target_grasp_world`` uses for the object:
``rim = points[points[:, 2] >= percentile(points[:, 2], BASKET_RIM_QUANTILE)]``.
2. Set the target XY to the ****rim median**** (mouth center, unbiased in depth) and Z to the ****rim median**** (mouth height). Fall back to all points if fewer than 5 rim points survive.
3. Leave the place servo (``PlaceMixin.place``) unchanged -- it already aligns the object center to ``basket + object_to_eef``. With ``basket`` now the mouth, the object drops straight through the opening in one 3D move, no VLA.

Guardrails

- Do NOT take the basket XY from the all-points median -- the near wall biases it toward the camera in depth (the 5 cm X error above). This is the failure mode this skill exists to repair.
- Do NOT pair a body-centroid XY with a rim-percentile Z -- compute both from the same top-rim point set, or the target is inconsistent in Z vs XY.
- Do NOT chase the single highest-Z pixel (raise ``BASKET_RIM_QUANTILE`` toward the max) -- the median of the top slice is steadier than the max; keep it symmetric with ``TARGET_TOP_QUANTILE``.
- This skill owns the place ****target**** (the mouth); pair it with ``align-object-3d-over-basket`` (the object-center servo) and ``record-3d-grasp-offset`` (the 3D EEf->object offset). Do not fold it into either -- the target is a distinct failure surface.

Code Links

``basket_center_world`` in ``grounding.py`` (the target the place servo consumes) of the task package, e.g.

``capx/eva/code_library/libero/object/task2/grounding.py``. The place servo itself (``PlaceMixin.place`` in ``place.py``) needs no change -- it already aligns the object center to ``basket + object_to_eef``; it only succeeds once ``basket`` is the mouth.

Read ``references/basket-mouth.md`` for the angled-camera geometry, the measured bias, and why the fix mirrors ``target_grasp_world``.

- ``basket_center_world``
- ``grounding.py``
- `[grounding.py](code_library/libero/object/task2/grounding.py)`
- ``PlaceMixin.place``
- ``place.py``
- ``basket + object_to_eef``
- ``basket``
- ``references/basket-mouth.md``
- ``target_grasp_world``