

SEAR: Self-Evolving Agentic Robotics

Xinyi Wang^{*,†} Bangzheng Li^{*,†} Hanyu Wang[‡] Lin Zhang
 Andrea Yaoyun Cui Zhenggang Tang Shyamal Buch[§] Dejia Xu^{†,§}

 Luma AI, OPAL project

^{*}Co-first authors [†]Project leads [‡]Real robot lead [§]Co-last authors

Frontier language agents can now improve a robot from its own experience: they write control code, maintain skills and memories, and train action models on their own rollouts. When such an agent’s success rises with more interaction, how much of that rise comes from trying more often, from a longer session, or from experience carried over from earlier tasks, and how far does that experience reach past the tasks it practiced on? We introduce **SEAR** (Self-Evolving Agentic Robotics), a framework and benchmark for studying these questions. The framework connects language agents with perception and control tools through a single harness that supports code-as-policy, full-loop agent-as-policy, and auto-research across both simulated and physical robots. The benchmark answers each question with a matched control: budget-matched restarts, a no-library baseline, a five-tier generalization ladder from held-out initial states to physical hardware, a sealed judge, and replicated lineages. Across 1.9 years of agent runtime, five frontier models, 332 simulated tasks in two physics engines, and three models on five physical tasks, we find three results. First, the base model’s capability sets both the floor and the ceiling of self-improvement, and whether the model can see is not what separates them. Second, longer sessions and inherited experience both raise performance, but only inherited experience transfers past the practiced task, and its value grows with distance from practice. Third, the same library that carries skills forward carries mistakes forward: identically specified lineages diverge, and practice success on hardware does not predict held-out success. **SEAR** is the first benchmark that evaluates self-evolving agents on robot tasks under rigorous controls, and we hope it gives future work on this direction a common playground and provide actionable insights.

1 Introduction

A robot that leaves the lab meets objects, conditions, and tasks its designers did not anticipate. Every interaction is a chance to improve: a failed grasp reveals a missing constraint, a successful one can become a reusable skill, and a set of rollouts can train an action model. Language-model agents are the natural foundation for such self-evolving systems: they can inspect a failure, rewrite the program that caused it, save a skill for later, and implement their own training pipeline to distill successful rollouts into action models for faster inference. Recent systems demonstrate pieces of this loop, in simulation (Fu et al., 2026; Zhang et al., 2026; Lu et al., 2026) and on hardware (Xiao et al., 2026). This paper asks questions those systems cannot answer individually:

1. When a robotic agent’s success rises with more interaction, how much of that rise comes from one sustained session of iteration?
2. How much would it get from several independent attempts at the same total budget?
3. How much of the gain comes from experience carried over from earlier tasks?
4. And when the agent does carry experience forward, how far does it generalize beyond the task it practiced on?

Of these four questions, only the third points to something the agent actually retains: skills, memories, or weights it keeps from one task and uses on the next. This is what we mean by self-evolution. But retained experience only matters if it helps beyond the task that produced it, which is what the fourth question tests. Together, the four questions define two axes: where a gain comes from, and how far it reaches.

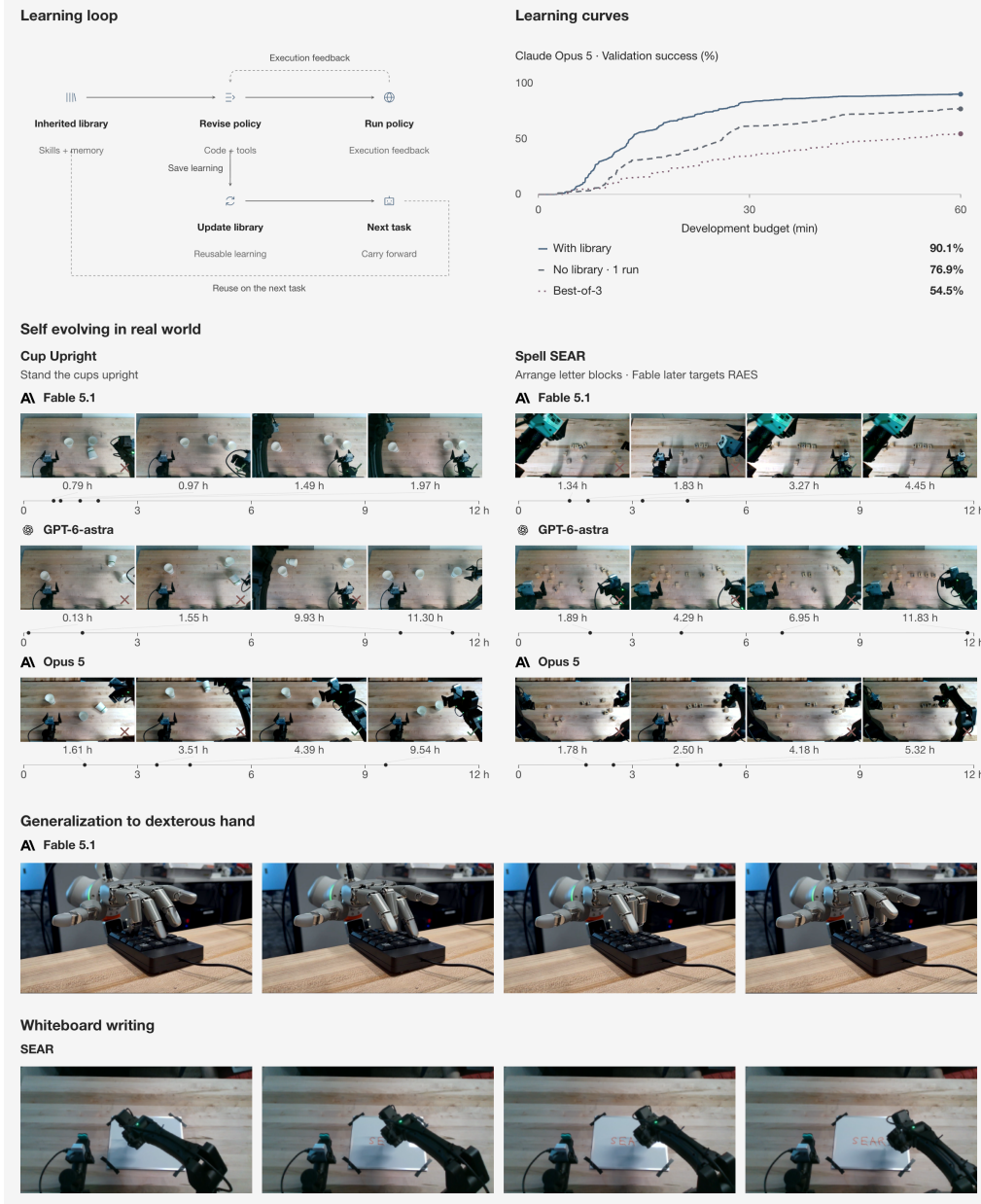


Figure 1. SEAR is a framework and benchmark for **self-evolving agentic robotics**. Top: the agent revises its policy from execution feedback and keeps skills and memory for later tasks; the validation curves compare an agent that inherits a library with one that starts empty and with three restarts on the same budget (Section 6.1). Bottom: selected Cup Upright and Spell “SEAR” attempts by Fable 5.1 and GPT-6-astra on the physical platform. Labels give elapsed hours and the judge’s verdict; the frames illustrate trajectories, not success frequencies (Section 7.3).

Answering these questions is hard. A rising success rate does not mean the agent is learning: it may just be trying more often (Brown et al., 2024), or memorizing the initial states it practices on, or reading simulator state that a physical robot never sees. One lucky run looks like a reliable method, and more compute looks like a better algorithm. Generalization is hard to measure too: a held-out initial state of the same task is the shortest possible distance from practice, and most systems stop there. Each prior system also defines its own protocol, so their results cannot be compared.

We introduce **SEAR (Self-Evolving Agentic Robotics)**, an agentic framework and benchmark that addresses each of these problems (Section 3). Budget-matched best-of- N controls and a no-library baseline separate learning from resampling and sustained effort. A five-tier ladder from held-

out initial states to physical hardware measures how far a gain reaches. A sealed judge withholds privileged simulator state, replicated lineages expose luck, and a resource ledger alongside wall-clock time keeps code-writing and model-training agents comparable. The framework (Section 2) connects a language-model agent with perception and control tools and persistent skills and memories through one harness that runs across simulated and physical robots. The agent may deliver its policy as a program (code-as-policy), as a trained checkpoint (auto-research, which fine-tunes a VLA), or as itself, kept in the control loop at execution time (agent-as-policy). Figure 1 shows two physical tasks being learned this way.

We then use the benchmark for a controlled study of self-evolving robotic agents: five frontier coding agents on 332 simulated tasks across five task families and two physics engines, a VLA fine-tuned inside the loop, and three agents on five tasks on a two-arm physical platform. Three findings organize the results.

Base-model capability sets the floor and the ceiling, and visual perception is not what separates the models (Section 5). A model that rarely writes a working policy has nothing to save, so self-improvement never gets off the ground. Above that floor, the five models rank in the same order under every protocol, and the gap between best and worst grows with task difficulty, from 12 points to 43. Multimodal input is not the deciding factor: the one text-only model, equipped with segmentation and depth tools, outperforms the nearest multimodal model on the hardest family. Where vision does help is diagnosis: the two models that inspect their own failures most often are the ones that fix the most.

Above the floor, performance increases with session time and with inherited experience, and only inherited experience transfers past the practiced task (Sections 6 and 7). Both are forms of test-time compute, since nothing in the model’s weights changes. A longer session helps once it exceeds the time the model needs to reach its first success, and is wasted below that, so whether one long session beats three short ones depends on the model. An inherited library moves first success earlier, adds a few points to strong models and 7 to 18 to weak ones on the practiced task, and under hardware faults it raises the ceiling for the weaker models and shortens the path to it for the strongest. It is also the only source of gain that survives a change of task or simulator. What transfers is specific: control code rather than perception code, shared routines rather than task-specific solutions.

Inherited experience carries mistakes alongside skills, and the best model depends on the policy form (Sections 7.2, 7.3 and 8). Runs starting from the same configuration diverge by more than the gap between different models. In two cases we traced the divergence to a single cause: an agent built its own measurement tool, got a wrong reading, saved it to memory as fact, and every later task inherited the error with no way to detect it. A code policy can fix a hardware fault it can observe but cannot adapt to a shifted camera, while a VLA trained on that code policy’s rollouts inherits both its successes and its blind spots. On hardware, ten consecutive practice successes do not predict held-out success, because the agent’s own reset behavior determines which states it practices on. The model that dominates in code-as-policy performs worse than the other in agent-as-policy.

Our contributions are:

- **An empirical study** of five frontier models improving robots through their own experience, across three policy forms, 332 simulation tasks, two physics engines, and physical hardware.
- **A benchmark** with matched controls that separate what an agent retains from what it gets by trying more, practicing longer, memorizing states, reading privileged information, or spending more compute, and a sealed judge that grades code and trained checkpoints the same way. Beyond the existing 332 tasks, our benchmark is easily extendable to any simulation tasks based on MuJoCo or Issac. We release our code base at <https://github.com/Self-Evolving-Agentic-Robotics/SEAR>.
- **An agentic framework and harness** that runs code-as-policy, auto-research, and agent-as-policy agents against the same environments, tools, and libraries in simulation and on physical robots, and a proposal for co-evolving the foundation model with each component of the self-evolving agentic harness (Section 10).

2 SEAR: Self-Evolving Agentic Robotics

Language models have demonstrated strong long-horizon reasoning, task decomposition, and the ability to write, debug, and revise code. Recent work brings these capabilities to robotics, using language models as the core agent that observes the environment, plans what to do, and drives the robot through perception and control tools (Fu et al., 2026; Lu et al., 2026; Zhang et al., 2026; Xiao et al., 2026; Zhu et al., 2026; ang Gao et al., 2026). **SEAR** organizes these systems into three modes, defined by what the agent produces and what executes at runtime.

- **Code-as-policy (CaP).** The agent writes a program that becomes the policy. No language model runs at execution time, so latency depends only on the code and its primitives, and the resulting policy can be replayed and re-graded without the agent. This is the dominant mode in self-evolving robotics (Lu et al., 2026; Xiao et al., 2026; Zhang et al., 2026; Fu et al., 2026), as low latency and replayability are practical requirements for robot policies.
- **Auto-research (AR).** The agent designs training experiments to improve a learned model (Rank et al., 2026). In **SEAR**, that model is a VLA. The agent writes code policies as teachers, collects and curates rollout data, trains the VLA, evaluates it, and decides what to try next. It may also refine the teacher to produce better training data.
- **Agent-as-policy (AaP).** The language model stays in the control loop at execution time and chooses actions from live observations. Self-evolution updates its context, tools, memory, and skills rather than its weights. The model runs during every rollout, so evaluation cost scales with the number of rollouts.

Every agent starts with the same toolbox: SAM3 (Carion et al., 2026) for perception, a vanilla $\pi_{0.5}$ (Black et al., 2025) as the action model, and inverse-kinematics and servo interfaces for control. During self-evolution, a CaP agent reads the codebase and writes new tools to produce a code policy; an AR agent collects data and trains a VLA checkpoint; an AaP agent accumulates skills and memories that shape its next rollout. Simulation experiments focus on CaP and AR, whose frozen policies a sealed judge can re-grade offline. Physical experiments evaluate CaP and AaP.

Library. Each session inherits and extends a *library* of versioned code skills, textual skills, and memories from previous tasks. Even with a frozen language model, an agent improves by building up better skills and memories over time. The auto-research agent can also fine-tune the action model ($\pi_{0.5}$) on each task, but trained checkpoints do not carry to the next task; only skills do. We always measure improvement by what the agent does next, not by what it reports.

Settings. **SEAR** runs the loop in four settings, detailed in Section 3.

- *Vanilla.* The agent receives a task instruction and a judge, and starts every task with an empty library.
- *Lineage.* Same as Vanilla, but the agent maintains its library from all previous tasks, so skills and knowledge accumulate in a continuous learning way.
- *Self-exploration.* The agent receives neither the instruction nor the judge. It proposes its own practice tasks and receives the real instruction only after exploration ends. item *Real world.* The loop runs on a physical robot. Scenes do not reset between attempts, and there is no simulator state for the judge to read.

These four settings map to the five generalization tiers of Section 3.5: Vanilla and Lineage cover the first two, self-exploration the third and fourth, and the real world the fifth.

3 Benchmark Design

Evaluating a fixed policy is a solved problem: roll it out on held-out states and count. Evaluating a system that changes itself is not, because five things raise its success rate without any experience being retained, or make retained experience look larger than it is. **SEAR** answers each with one mechanism.

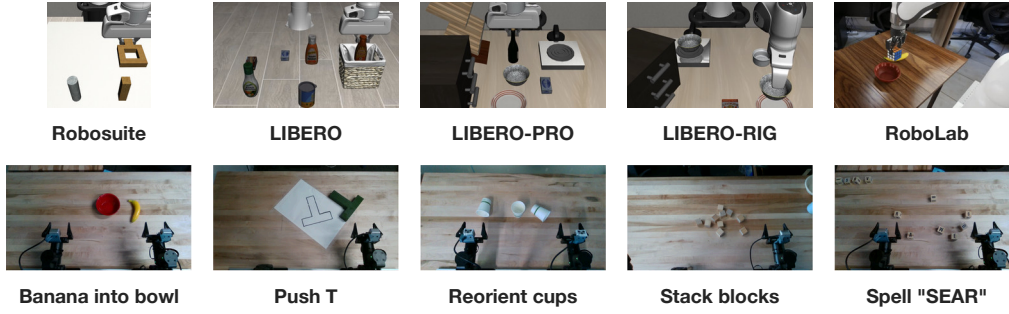


Figure 2. The SEAR testbed. Five simulated task families (top) and five physical tasks (bottom): Banana into Bowl, Push-T, Cup Upright, Block Stacking, and Spell “SEAR”.

1. **More attempts look like improvement.** The fraction of problems solved by at least one sample grows with the number of samples (Brown et al., 2024). We compare one sustained session against best-of- N independent restarts at the same total development time, and every lineage against a matched run that starts with no library (Section 3.3).
2. **A gain on the practiced task says nothing about reach.** Held-out initial states are the shortest distance from practice, and a gain there may be a memorized solution. Five tiers measure how far a gain holds, at new initial states, perturbed conditions, undisclosed tasks, a different simulator, and physical hardware, each with its own adaptation rule (Section 3.5). This is the second axis of the question.
3. **Privileged information looks like capability.** A simulator exposes ground-truth state and success predicates a physical robot never has. A separate judge grades frozen policies with logic the agent cannot reach (Section 3.2).
4. **One lucky run looks like reliability.** Recursive improvement is path-dependent. We repeat the whole process, policy development and library evolution, several times from the same start, not only the evaluation episodes of one policy (Section 3.3).
5. **A larger budget looks like a better algorithm.** Code search and model training spend different resources. We match development time in every controlled comparison and record environment steps, submissions, tokens, and GPU time beside it (Section 3.3).

3.1 Task Families

SEAR uses five simulated families and one physical platform, ordered from easiest to hardest (Figure 2). **robosuite** (Zhu et al., 2020), on MuJoCo (Todorov et al., 2012), contributes seven tasks from lifting to bimanual handover. **LIBERO** (Liu et al., 2023) adds 40 language-conditioned tasks on the same engine. **LIBERO-PRO** (Zhou et al., 2025) perturbs those 40 tasks along four axes (wording, objects, layout, goal) for 160 tasks, and doubles as a contamination control because it moves tasks away from instances a frontier model may have seen in pretraining. **LIBERO-RIG** is a hardware-robustness benchmark (Appendix J): it holds every LIBERO task fixed and perturbs only the rig, from camera pose and calibration to actuation, proprioception, and gripper pads, so that a drop in success isolates a policy’s sensitivity to the machine rather than to the scene. **RoboLab** (Yang et al., 2026) runs on Isaac Sim 5.0 (NVIDIA), a different physics engine and primitive API, with cluttered randomized scenes; we use the 105 tasks for which we can generate 50 initial states. Because the harness, the sealed judge, and the library already run on both MuJoCo and Isaac Sim, SEAR extends easily to other MuJoCo- and Isaac-based simulation benchmarks that expose a programmable primitive API and a success predicate the judge can evaluate offline. One word recurs below: a *cell* is one task under one condition, the unit every score averages over.

Lineages walk the families in difficulty order. The *full lineage* runs robosuite, LIBERO, LIBERO-PRO, and RoboLab, 312 tasks; a second lineage branches off during LIBERO-PRO onto the 20 LIBERO-RIG cells (three lanes per model, each starting from the finished LIBERO-PRO library of one full-lineage lane), because LIBERO-PRO and LIBERO-RIG are perturbations of comparable difficulty. Together the two lineages cover all 332 simulated tasks.

Physical tasks. The platform has two YAM-Ultra arms with parallel-jaw grippers, a fixed overview camera, and two wrist cameras (Appendix E). Its five tabletop tasks cover placement (*Banana into Bowl*), non-prehensile manipulation (*Push-T* (Xiao et al., 2026)), reorientation (*Cup Upright*), stacking (*Block Stacking*, a free-standing tower as high as possible), and language-conditioned arrangement (*Spell SEAR*, letter blocks among distractors, in the instructed direction). Initial poses vary, so policies must ground their actions in the observed scene; exact objects and rules are in Appendix H.

3.2 Sealed Judge and Frozen Policies

The agent works inside its own environment: it can inspect failures, run local simulations, install packages, write programs, update its memory, and train checkpoints. When its budget expires or it submits, its policy is frozen, a program together with the skills it imports, or for AR a checkpoint, and only that frozen policy crosses into the judge. The judge holds the held-out initial states, the privileged simulator state, and the grading logic; none of it is installed where the agent works. The environment’s own success predicate stays callable where the agent works, as a stop signal, but its authority rests with the judge, which rolls the frozen policy out on its own states and reports success under that predicate, ignoring any success claim the policy returns. The same boundary is what makes a program and a checkpoint comparable: both face the same judge.

The seal is not absolute, and we report its limits beside each result. Work environments have internet access, so public LIBERO initial states remain obtainable even though we remove them from the work image. RoboLab’s validation feedback describes layouts, which is a reconstruction route. Static task assets and task files are readable, and the success predicate is callable as a stop signal; Section 8.2 reports what agents did with each. Audits found no reconstruction of test layouts in the inspected logs, but cannot establish that it was impossible.

3.3 Controls: Seed Bands, Libraries, Budgets, and Repetitions

Three seed bands. Every randomized task partitions its initial states into **DEV** seeds the agent may use freely, **VAL** seeds the judge grades during development and reports back, and **TEST** seeds graded once after development ends and never shown to the agent. Because VAL feedback is adaptive, every reported score is TEST. Policy selection takes the highest VAL score among the agent’s submissions and the host’s periodic snapshots; TEST never guides selection or library harvesting. Allocations per family are in Appendix B.

Library and its control. A Lineage run is one agent–model configuration walking the curriculum with its own evolving library. Each task starts in a fresh workspace; the only thing that persists is the library of versioned skills and textual memories. Entries are added from the best policy of each task and recorded with their origin, and nothing written on the current task can be inherited by it. Vanilla is the matched no-library control: the same model, task, work image, judge, and per-task budget, with an empty library.

Budgets. A CaP session lasts one hour per robosuite, LIBERO, LIBERO-PRO, or LIBERO-RIG task and two hours per RoboLab task, with up to eight agent-initiated submissions plus periodic host grades. AR receives the same total development time as CaP on LIBERO, LIBERO-PRO, and RoboLab, spent as one session per suite; robosuite’s AR budget differs and is given in Appendix C with the submission and early-stopping rules. Matching wall clock does not match compute, so we record environment steps, rollouts, submissions, API and token spend, and GPU time separately.

Best-of-3. On every family we compare one full-budget Vanilla session with three independent Vanilla attempts at one third of the budget each, keeping the attempt with the best VAL score. Neither condition has a library, so the comparison isolates budget allocation from inheritance: does sustained iteration within a session beat the diversity of fresh starts?

Repetitions. Every repeated campaign reports its count. Three repetitions of a Lineage develop three independent libraries; three repetitions of Vanilla develop three independent policies. Means and sample standard deviations are over repetitions, never over evaluation episodes. Where a campaign has one trial (the budget-matched VLA analyses on LIBERO, LIBERO-PRO, and RoboLab; every physical lineage), we say so beside the result.

3.4 Self-Exploration and Auto-Research

Self-exploration withholds the instruction and the judge, so the agent practices without being told the task. It inherits a library, proposes its own practice on the target scenes, and judges outcomes with its own tools. When exploration ends the instruction is revealed and the agent gets one chance to write a task-specific policy with no further simulator access; the judge then grades it. We call that policy *authored*. A *frozen* variant instead grades the task-general program the agent left behind at the end of exploration, with the instruction supplied only as an input, so the two together measure what one task-specific rewrite is worth on top of task-blind preparation. When the target simulator differs from the donor library’s, a library-adaptation phase first rewrites inherited code against the new interface.

Auto-research evaluates improvement of a learned action model. Per-task CaP policies serve as teachers; an agent trains one $\pi_{0.5}$ (Black et al., 2025) checkpoint per suite from their rollouts, chooses what data to collect and how to train, and may refine a teacher. Its *Vanilla* condition starts every suite from the teachers alone; its *Lineage* condition also inherits the previous suite’s skills and memory, but never its checkpoints or data. The comparison therefore measures inheritance of the research process, not continued training of one policy. AR runs separately from the CaP campaigns so that gains from changed weights stay distinguishable from gains from inherited skills.

3.5 Generalization Tiers

Five tiers say how evaluation conditions differ from practice, and what adaptation each permits. They are not successive zero-shot transfers of one frozen policy. Figure 3 maps the practice and grading families onto them.

- **T1. New initial states.** Same family, held-out TEST seeds. Frozen policies, no adaptation.
- **T2. New conditions.** Same tasks, perturbed: LIBERO-PRO changes the scene and wording, LIBERO-RIG changes the hardware. The frozen policies of the row family are re-graded on the column family with no further practice.
- **T3. New tasks.** Unseen tasks on the same simulator, with the library inherited. A task-specific program cannot solve an unseen task with no adaptation, so T3 uses self-exploration: task-blind practice, then one rewrite after disclosure. For AR, the VLA is trained during exploration. The frozen exploration program can also be graded on scenes it never reached (Section 7.1).
- **T4. New simulator.** RoboLab’s contact dynamics and primitive API differ from MuJoCo’s, which asks whether a loop that tuned timings against one physics build overfits it. T4 is T3 preceded by library adaptation.
- **T5. The real world.** The loop runs on hardware, where scenes do not reset themselves and there is no simulator state to judge from.

3.6 Real-World Protocol

On hardware two things change. First, there is no automatic reset: objects stay where the last attempt left them. We therefore pair every task with a reset task the agent must also implement: taking the banana out of the bowl, pushing the T away from its outline, laying a cup down, toppling the tower, scrambling the letters. A reset is an objective, not a reversed motion; it only has to produce another valid start. At each attempt the agent looks at the scene, picks a direction, revises or reuses its policy, and runs it; both directions are developed through the same feedback loop. This has a consequence we return to in Section 7.3: the reset policy shapes the states the forward policy practices on.

Second, there is no privileged state. A fixed external evaluator judges before-and-after images and the robot recording, independently of the agent’s own perception, and marks ambiguous evidence unknown. Its verdicts drive policy revision and early stopping: a non-stacking task stops after ten consecutive scored forward successes; Block Stacking has no early stop. Each task has a 12-hour budget and inherits the previous task’s library. Final CaP scores come from the frozen policy run once on each of ten operator-staged scenes per task (ten tower episodes for stacking), with the evaluator’s verdict confirmed by the operator. AaP keeps the model in the loop and uses five scenes per task. Harness, budgets, and scoring rules are in Appendices E to G.

Table 1. Glossary. Terms used in a specific sense in this paper, and where each is introduced.

Term	Meaning	Where
<i>Policy forms</i>		
Code-as-policy (CaP)	The agent delivers a program; no language model runs at execution time.	Section 2
Auto-research (AR)	The agent runs training experiments that fine-tune a VLA from rollouts of its own code policies.	Section 2
Agent-as-policy (AaP)	The language model stays in the control loop and chooses each action from live observations.	Section 2
<i>What persists</i>		
Library	The code skills and textual memories that persist between tasks; each task inherits and extends it.	Section 2
<i>Settings and conditions</i>		
Vanilla	The no-library control: instruction and judge are given, and every task (in AR, every suite) starts from an empty library.	Section 2
Lineage	Vanilla with the library inherited from all earlier tasks; also one such run, with its own evolving library.	Section 2
Self-exploration	Neither instruction nor judge is given; the agent practices on its own and learns the task only afterwards.	Section 3.4
Donor	The library a self-exploration lineage starts from, named after the family that produced it.	Section 7.1
Best-of-3	Three independent no-library attempts at a third of the budget each; the best by validation score is kept.	Section 3.3
<i>Evaluation</i>		
Judge	The sealed grader that rolls a frozen policy out on held-out states and scores it with the environment’s own predicate.	Section 3.2
DEV / VAL / TEST	Initial-state bands: free for development; graded during development with feedback; graded once at the end and never shown.	Section 3.3
Cell	One task under one condition; the unit every score averages over.	Section 3.1
Tiers T1–T5	How far the graded conditions are from practice: new initial states, perturbed conditions, unseen tasks, a new simulator, hardware.	Section 3.5
Authored / frozen	After self-exploration, the policy rewritten once the instruction is disclosed, or the task-general program left at the end of exploration.	Section 3.4

3.7 Terms

Table 1 collects the terms this paper uses in a specific sense, with the section that introduces each. The definitions in the text remain the authoritative ones; the table is a reference for readers who enter the results out of order.

4 Experimental Setup and Overview

Agents. Every agent is a frontier coding agent driving the **SEAR** harness: Claude Opus 5 through Claude Code, GPT-5.6-sol through Codex, and Gemini-3.6-flash, Kimi-k3, and DeepSeek-v4-flash through OpenCode. Each model runs through its vendor’s or a third-party coding tool, so a comparison between models is a comparison between model-and-tool pairs. DeepSeek-v4-flash is the only text-only model: it receives the same RGB-D tools and image files as the others but cannot read pixels. A sixth model, Inkling-small, was run and withdrawn (Section 5.1). The tiered results of Figure 3 use Opus throughout; the physical campaign runs Claude Fable 5.1 and Opus through Claude Code and GPT-6-astra through Codex.

Campaigns and metrics. The *Lineage* campaign runs the full lineage and the LIBERO-RIG lineage with the instruction given, three repetitions per model, against matched *Vanilla* (no library, three repetitions of every task, the 20 LIBERO-RIG cells included) and *best-of-3* controls. The *self-exploration* campaign seeds task-blind lineages from donor libraries of earlier families, three replicates per donor and model. The *AR* campaign compares *Vanilla* and *Lineage* VLA training, with one

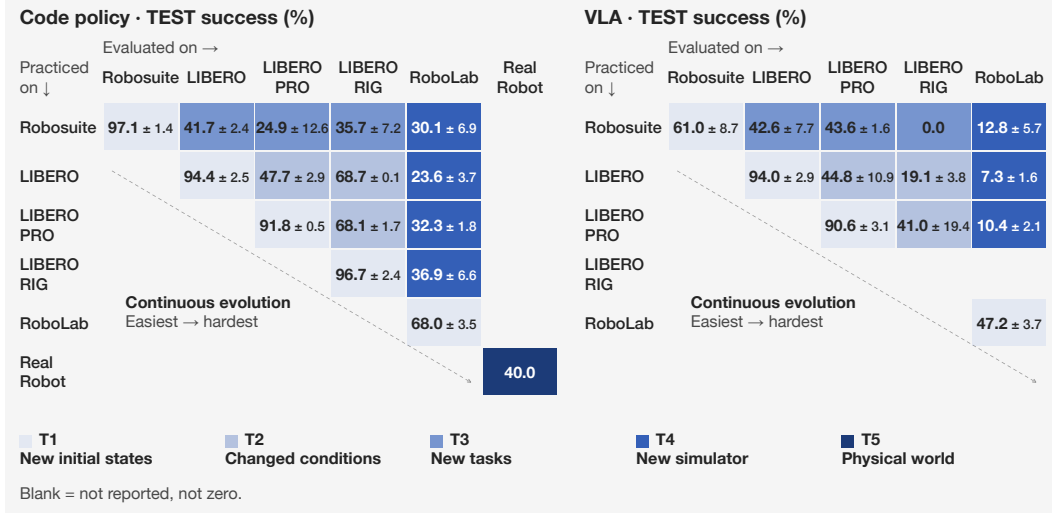


Figure 3. Where a gain comes from and how far it reaches, for Opus. Tiered TEST success (%) of code-as-policy (CaP) and auto-research (AR) agents, all driven by Claude Opus 5. Rows are the family an agent practiced in, columns the family its frozen policy was graded on, both ordered easiest to hardest. The matrix is the two axes of the paper’s question in one picture: the diagonal is where a gain comes from, the curriculum a continuously evolving agent walks with the controls of Table 4 beside each cell, and every cell to its right is how far that gain reaches, graded at increasing distance from practice. T1–T4 are the tiers of Section 3.5; Table 10 gives the numbers.

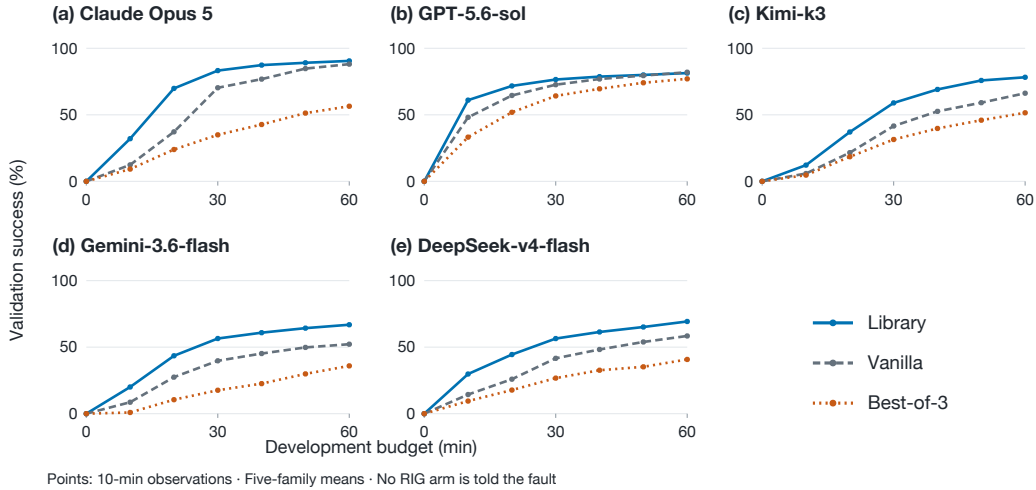


Figure 4. Five models over session time. Validation success averaged over all 332 simulation tasks against self-evolution time, one panel per model on shared axes: solid with an inherited library, dashed for the matched no-library run, dotted for the no-library best-of-3 control, which spends the same total budget as three parallel attempts at a third each. Points every 10 minutes; RoboLab’s 120-minute budget is rescaled onto the shared 60-minute axis.

trial per suite and condition except robosuite’s three. The *physical* campaign runs one lineage per agent and execution mode. Simulation scores are TEST success over ten rollouts per task, averaged over tasks, with standard deviations over repetitions. Learning curves plot best-so-far VAL success against wall clock, because TEST is graded once, after development ends.

Reading order. Figure 3 gives every tier for Opus and Figure 4 the learning curves of all five models. Sections 5 to 8 read them, one research question each, in the order of the introduction: what sets the floor and the ceiling, where a gain comes from, how far it reaches, and what the loop cannot do.

Table 2. The five models rank in the same order under every protocol. TEST success (%) of every model. Lineage columns: mean $\pm$ SD over three repeated runs. LIBERO-RIG: 18 measurable cells, 60 minutes each; the lineage carries one library across cells (three lanes, no-policy cells as zero), Vanilla has no library and is not told the fault. RoboLab Vanilla: the full-budget no-library session over all 105 tasks, the control every RoboLab gain in this section is measured against (Table 3). [¶]Two lanes. [§]deepseek-v4.1-flash; other columns v4-flash.

Model	Lineage (T1)		LIBERO-RIG		RoboLab
	robosuite & LIBERO	LIBERO-PRO	library (lineage)	no library (Vanilla)	Vanilla
Claude Opus 5	94.8 $\pm$ 2.0	91.8 $\pm$ 0.5	96.7 $\pm$ 2.4	95.0 $\pm$ 1.7	61.9 $\pm$ 1.0
GPT-5.6-sol	87.8 $\pm$ 2.8	88.4 $\pm$ 2.8	91.3 $\pm$ 5.9	88.0 $\pm$ 6.6	46.9
Gemini-3.6-flash	81.7 $\pm$ 4.2	76.2 $\pm$ 1.2	73.9 $\pm$ 33.0 [¶]	36.5 $\pm$ 4.1	18.3
Kimi-k3	79.3 $\pm$ 1.3	80.5 $\pm$ 3.5	89.6 $\pm$ 6.1	70.2 $\pm$ 4.2	30.1
DeepSeek-v4-flash	67.6 $\pm$ 8.6	75.5 $\pm$ 4.5	86.1 $\pm$ 4.7 [§]	73.9 $\pm$ 4.3 [§]	23.1

5 Base-Model Capability Sets the Floor and the Ceiling

Finding How capable the underlying language model is decides how much a robot can improve on its own. A weak model rarely produces a working controller in the first place, so it has nothing to build on and never gets better. Among models that clear that bar, the stronger model stays stronger on every task family we tested, and the gap widens as the tasks get harder. Being able to look at images is not what makes the difference: a text-only model given good perception tools keeps up with, and on the hardest tasks beats, models that can see.

The first axis of the question asks where a gain comes from, and the first answer is that it comes from the model before it comes from anything else. Capability sets a floor, below which self-improvement never starts, and a ceiling that the ranking of the five models tracks on every family (Section 5.1). Above the floor, whether a model can see is not what separates the models; how it uses what it sees is (Section 5.2). And what the resulting policies fail at is the grasp, not the scene (Section 5.3).

5.1 Capability gates self-improvement, and the ranking is stable

Every model received the same tasks, tools, budgets, and library machinery. The first thing the campaign shows is a floor. Inkling-small completed the robosuite and LIBERO tasks at 25.2 ± 3.3 and $6.4 \pm 2.2\%$ TEST and was withdrawn partway through LIBERO-PRO; one of its lineages went 29 consecutive tasks without a valid policy. It wrote memories in about half its runs but almost never a code skill: after 47 steps its three libraries held 0, 6, and 2 skills. A model that rarely produces a working policy has nothing to record, so self-improvement never begins. Every table below excludes it.

Above that floor, Table 2 gives the picture. Opus leads on every family; GPT-5.6-sol is second everywhere except with a library on LIBERO-RIG and RoboLab, where Kimi-k3 passes it; the three OpenCode models trade places by family. Apart from that climb, the ordering is the same under every protocol, with a library, without one, and under best-of-3, and the spread widens with difficulty: 12 points between the best and worst Vanilla on robosuite and LIBERO, 16 on LIBERO-PRO, 44 on RoboLab. Difficulty is largely a property of the task before it is a property of the model. On RoboLab a variance decomposition of cell scores gives 56% to the cell, 22% to the model, and 22% to their interaction, and cell difficulty rank-correlates 0.69 on average between pairs of models, so weaker models fail on a superset of the strong model’s failures: seven cells are solved only by Opus, five by nobody.

Two caveats bound this finding. Each model runs through a different coding tool, so a rank is a rank of model-and-tool pairs. And the ranking with the instruction given is not the ranking without it: in self-exploration (Section 7.1) Kimi-k3 matches or beats Opus on five of seven cells, and Gemini-3.6-flash stays near zero on RoboLab even after a rig defect that hurt it was corrected (Table 11).

5.2 Seeing is not what separates the models; how a model uses what it sees is

The natural guess is that manipulation rewards the model that can see. The campaign does not support it. GPT-5.6-sol, second on every family but one, never reads an image by itself: it saves frames to disk and reasons about them numerically. Gemini-3.6-flash opens an image in 2% of runs. DeepSeek-v4-flash cannot read pixels at all, yet it composes the toolbox into working manipulation: nearly all of its RoboLab policies ask SAM3 for a mask of the named object and use the depth image to turn that mask into a 3-D position, it ships a policy in 85% of runs, more often than Kimi-k3 or Gemini-3.6-flash, and on RoboLab it beats Gemini-3.6-flash, the multimodal model of the same tier, under every protocol (Table 2). When the toolbox already locates objects in 3-D, the model does not need to see to manipulate.

What vision does buy is diagnosis. Opus views six to seven images per run, and the looking is deliberate: an image view leads to a code edit 38% of the time against a 24% baseline, concentrated in the second half of a run once the pipeline exists. Kimi-k3 looks late and edits after 68% of its views, the most corrective use of vision in the study. Both models turn what they saw into geometry and verification code, and both are the models whose libraries carry control skills across simulators (Section 6.2). The price of not seeing shows at the edges: when SAM3 misses, DeepSeek-v4-flash has no fallback, it trails Gemini-3.6-flash on the easier LIBERO and robosuite tasks, and on a miscalibrated rig (Section 8.1) it was the one model that never noticed the robot it had been handed was wrong.

Beyond seeing, the five models differ in strategy as much as in score, and their failure modes follow from their strategies. Opus measures first, builds infrastructure, and verifies every grasp by checking the gripper opening; its policies run mostly without perception at execution time. GPT-5.6-sol is fast and submits often; it succeeds early or not at all, and its validation scores overstate its test scores more than any other model’s. Kimi-k3 explores well and ships rarely, because it probes the simulator in background processes and runs out of time, but what it ships is closed-loop and good. Gemini-3.6-flash writes the shortest policies, relies on SAM3 alone, and adapts by adding synonyms to its prompts. Full portraits with the underlying counts are in Appendix K.

5.3 Where policies fail: the grasp, not the scene

Every graded policy shares one skeleton: capture a colour-and-depth frame, locate the target and destination in it, convert their pixels to world coordinates through the calibrated depth, optionally refine from the wrist camera, then approach, descend, close, verify, lift, carry, and release inside a retry loop. No model writes anything more elaborate. The pretrained $\pi_{0.5}$ action model shipped in the toolbox went almost unused: 2% of no-library RoboLab policies call it, and where GPT-5.6-sol did, it scored 16 against 54 without. Scripted perception and servoing beat the pretrained VLA for every model on every family.

Failure has one dominant physical form. We reconstructed where RoboLab Vanilla episodes fail from the score structure, the policies’ own progress prints, a hand tagging of what each of the 105 cells requires, and the transcripts. The start layout is not what fails: over 21,552 graded episodes, how far, how offset, how rotated, or how cluttered the target is barely predicts whether an episode passes (correlation 0.04 in magnitude). What fails is the grasp. Where a policy logged its gripper opening at the moment it closed, 78% of failed episodes had closed on air against 18% of successful ones. The ordering of task categories inverts a natural guess about 3-D understanding: tasks that need relative placement, such as stacking or spatial relations, are the *easiest*, while tools and utensils, reorientation, and thin or handled objects are the hardest. Every model solves height, table plane, and relative placement; what the shared recipe of descending onto the top of the detected object and closing cannot do is grasp a non-box shape, a rim, a handle, or a thin shaft. Upstream of robotics, more than half of Kimi-k3’s and Gemini-3.6-flash’s zero-score runs never delivered a policy file at all, while every Opus and GPT-5.6-sol zero is a robotics failure. Per-category numbers are in Appendix K.

Table 3. Session and library terms on the practiced task. The two terms of Equation (1), TEST success (%). RoboLab budgets are 120 minutes per session or three attempts of 40; LIBERO-RIG 60 or three of 20. First success is the median, over the Vanilla runs that ever succeed on validation, of the time of the first graded round with a validation success; the LIBERO-RIG control is host-graded every 15 minutes plus agent-initiated submissions, so its times fall near that grid. All RoboLab settings are read on the same 105 tasks; [†]DeepSeek-v4-flash’s library term is read on the first 58 cells, where its lanes closed when the model was retired. Section 6.2 discusses the library terms.

Family	Model	First success (median min)		Session term		Library term
			Best-of-3	Vanilla	Vanilla – BO3	Lineage – Vanilla
RoboLab	Claude Opus 5	81	57.1	61.9	+4.8	+6.1
	GPT-5.6-sol	44	48.5	46.9	−1.6	−4.5
	Kimi-k3	110	6.4	30.1	+23.7	+21.8
	Gemini-3.6-flash	55	21.1	18.3	−2.8	+1.9
	DeepSeek-v4-flash	88	14.8	23.1	+8.3	+4.1 [†]
LIBERO-RIG	Claude Opus 5	28	31.1	95.0	+63.9	+1.7
	GPT-5.6-sol	11	71.1	88.0	+16.9	+3.3
	Kimi-k3	43	57.2	70.2	+13.0	+19.4
	Gemini-3.6-flash	26	17.8	36.5	+18.7	+37.4
	DeepSeek-v4-flash	28	31.1	73.9	+42.8	+12.2

6 Both Session Time and Inherited Experience Raise Performance

Finding When an agent gets better at a task, two things can be responsible: iterating longer within one session, and starting from what it learned on earlier tasks (its saved code skills and written notes, which we call its library). Both help. One long session beats splitting the same time into three fresh attempts, but only once the session is long enough for the model to reach its first success; below that, the extra time is wasted. Starting with a library brings that first success earlier and helps the weaker models the most.

In code-as-policy nothing in the model changes. Every gain a lineage shows is inference compute spent interacting with the environment, and SEAR lets us ask how that compute is best spent. At a fixed per-task budget, a lineage’s margin over the cheapest alternative, three fresh attempts, splits into two terms:

$$\underbrace{\text{Lineage} - \text{BO3}}_{\text{margin}} = \underbrace{(\text{Vanilla} - \text{BO3})}_{\text{one long session over three short}} + \underbrace{(\text{Lineage} - \text{Vanilla})}_{\text{inherited library}}. \quad (1)$$

The first term is the value of sustained iteration within a session; the second is the value of experience carried in from earlier tasks. This section measures both on the practiced task, the first axis of the question: Section 6.1 the session term and Section 6.2 the library term. The two differ in reach as well as size. A session leaves nothing behind, so its term exists only on the practiced task; the library is the one term that can be graded at a distance from practice, and Section 7 follows it there. All three conditions are read after the runs end, on TEST; Appendix D gives the scoring rule and the one asymmetry between them, that some best-of-3 attempts are told their time limit.

6.1 Session time: one long session beats three short ones once the model’s first success is covered

Some of what any lineage gains is available to an agent that simply tries more often (Brown et al., 2024). Best-of-3 is the fair version of trying more often: three fresh attempts at a third of the budget, the best one kept by validation score. Table 3 gives the result for every model on the hardest family and on the faulted rig. On robosuite, one hour beats three 20-minute attempts for every model, by 0.5 to 20.5 points. On RoboLab the sign of the session term differs by model, and it follows one quantity: how long the model takes to reach its first success. GPT-5.6-sol and Gemini-3.6-flash succeed within the first hour at the median, so three 40-minute attempts sample the productive part of their curve three times and beat one two-hour session by a point or two. Opus, DeepSeek-v4-flash, and Kimi-k3 need 81 to 110 minutes; a third of the budget ends before they get going, and splitting costs them 5 to 24 points. Part of that cost is delivery: Kimi-k3 ships a policy in 10% of its 40-minute attempts against 68% of its two-hour runs. For Opus the same pattern holds on the middle families (Table 4).

Table 4. Opus on the practiced task under all three conditions. Claude Opus 5 on held-out seeds of the practiced task (T1), TEST success (%), mean $\pm$ SD over three repeated runs where repeated. *Lineage* carries the library forward; *Vanilla* is the matched no-library session; *Best-of-3* spends the same budget as three independent attempts and keeps the best by validation score. *End-of-run TEST of the validation-selected winners on the 47 robosuite and LIBERO tasks. LIBERO-RIG: 18 measurable cells at severity 3, the lineage over three seed-matched lanes, Vanilla over three 60-minute runs per cell that are not told the fault. ‡All three RoboLab settings on the corrected tool frame over the same 105 tasks (Table 3). ¶Mean over the four non-stacking physical tasks, ten operator-staged scenes each (Table 7).

Family	Code-as-policy			Auto-research VLA	
	Lineage	Vanilla	Best-of-3	Vanilla	Lineage
robosuite	97.1 $\pm$ 1.4	93.8 $\pm$ 7.0	90.0	61.0 $\pm$ 8.7	61.0 $\pm$ 8.7
LIBERO	94.4 $\pm$ 2.5	95.2 $\pm$ 1.1	73.8*	94.0 $\pm$ 3.6	94.0 $\pm$ 2.9
LIBERO-PRO	91.8 $\pm$ 0.5	93.0 $\pm$ 1.8	67.2	92.4 $\pm$ 4.9	90.6 $\pm$ 3.1
LIBERO-RIG	96.7 $\pm$ 2.4	95.0 $\pm$ 1.7	31.1	—	—
RoboLab	68.0 $\pm$ 3.5‡	61.9 $\pm$ 1.0‡	57.1‡	43.1 $\pm$ 9.3	47.2 $\pm$ 3.7
real robot	40.0¶	—	—	—	—

On the faulted rig the session term is the largest in the study, 64 points for Opus: a no-library session is at 6% on validation at 15 minutes and 69 at 30, and its median first success comes at 28 minutes, so 20-minute attempts end before the policy works.

Selecting by validation costs the model that submits most. Best-of-3 keeps the attempt with the highest validation score, and the attempt that scored highest on the seen states is often the one that overfit them. GPT-5.6-sol’s RoboLab winners give back 6.6 points between validation and test; Opus’s give back none; the other three lose 2 points or less. GPT-5.6-sol submits about twice as often as any other model, and it is the one model for which best-of-3 beats the single session.

Long runs are worth resampling; short ones are not. Variance across full-budget runs is large for every model. Choosing the best of three full-budget Vanilla repetitions would score 17 points above a single repetition on RoboLab (15 when the repetition is chosen by validation score) and 5 on robosuite. So more parallel samples pay for every model once each sample is long enough to clear the model’s first success, and the equal-time split pays only for the two models that succeed early. Both ways of spending test-time compute, longer sessions and more of them, raise performance; which pays more is set by the model’s time to first success.

6.2 Inherited experience: what a library buys, and where

Library inheritance is the mechanism SEAR is built around, so we measure its value in every setting rather than assume it. Table 4 gives Opus on every family and Table 3 the other models on RoboLab. The answer depends on how hard the task is and on whether the agent knows what it is practicing for.

On tasks one session can solve, the library buys time, not a higher ceiling. On robosuite, LIBERO, LIBERO-PRO, and LIBERO-RIG, Opus and GPT-5.6-sol finish within noise of their no-library twins (-1.2 to $+5.3$ points); Gemini-3.6-flash and DeepSeek-v4-flash gain 7 to 18 on robosuite, LIBERO, and LIBERO-PRO. The library reaches the same score sooner (Figure 4), and its gain concentrates on the cells a model fails alone: on LIBERO-PRO, cells with a Vanilla score below 50% gain $+33$ and $+39$ points for the two weakest models against $+2$ and $+4$ on cells they already solve, $+25$ against $+3$ for GPT-5.6-sol, and $+14$ against -1 for Kimi-k3.

The gain grows with difficulty, and it changes when success arrives more than whether it does. On RoboLab the library is worth $+6.1$ points to Opus and $+21.8$ to Kimi-k3, $+1.9$ to Gemini-3.6-flash, and -4.5 to GPT-5.6-sol (Table 3); Opus’s gain is $+9.1$ on the first 58 cells and $+2.7$ on the harder last 47. It rarely unlocks a cell a model never solves alone (of the cells at zero in every Vanilla repetition, Opus’s lineage lifts 3 of 8, Kimi-k3’s 20 of 34); it raises the hit rate on borderline cells and moves success earlier, from a median first success at 81 minutes to 67 for Opus, and from 41% to 63% of runs ever succeeding for Kimi-k3. In the terms of Section 6.1, the library moves the model’s curve to the left. Its sign also depends on what the library fills with: Kimi-k3’s gain grows along the task sequence as reusable skills accumulate, from -25 points over its first seven tasks to $+34$ over

the last 47, while GPT-5.6-sol’s turns negative after 48 agent-hours, reaching -10 over the last 47, as its library fills with helpers named after individual tasks. A far larger gain, 40 points for Opus, appeared on a rig that turned out to be miscalibrated (Section 8.1). None of the four weaker models’ lineages clears any of the 16 cells Opus fails without a library, while Opus’s own lineage clears two: the library moves a model along its own capability curve, not past the strongest model’s no-library ceiling (Section 5).

Under hardware faults the library decides how soon, and for the weaker models whether. On LIBERO-RIG every model scores 20 to 66 points above its no-library best-of-3 (Table 3). Against a matched no-library session that is never told the fault, the library adds 12 (DeepSeek-v4.1-flash), 19 (Kimi-k3), and 37 (Gemini-3.6-flash) points at the hour; Opus and GPT-5.6-sol finish within noise of their twins (96.7 ± 2.4 against 95.0 ± 1.7 , and 91.3 ± 5.9 against $88.0 \pm 6.6\%$ TEST) but get there sooner, at their ceiling by 30 minutes (95 and 88% on validation) where the twins arrive only by 45 (69 and 79 at 30). The repair does not travel: cells that were never developed move by -8 to $+9$ points.

What the libraries hold is usable, and it moves across tasks, benchmarks, and simulators. On the first half of the RoboLab tasks Opus reused 202 distinct skills in 660 imports, and skills written on the MuJoCo families reached about half of its RoboLab policies and two fifths of Kimi-k3’s, against almost none for the other three models. What crosses the simulator boundary is control code, not perception code: the servo wrappers and grasp routines that cross are also the skills that help most, in that policies importing them succeed 17 to 36 points more often than policies that do not. Reuse is heavy-tailed, a tenth of the skills taking 40 to 68% of the uses. Memories travel alongside code, and two examples show the effect: a bowl rim-pinch skill written on LIBERO because the bowl is wider than the gripper was imported by 66 later policies, and RoboLab’s four spatial-relation tasks were solved 10-of-10 on first contact, twelve cells in a row across three lineages, on libraries that already held LIBERO’s spatial skills and a memory that relation words name regions of the table. Appendix L gives the counts. The same library carries mistakes, and Section 8.1 traces two collapses to it.

In auto-research, inheriting the research process pays only where the task is hard. With checkpoints and datasets never inherited, a VLA lineage that carries the previous suite’s skills and memory ties the no-library Vanilla condition on LIBERO (94.0 each), trails it on LIBERO-PRO (90.6 against 92.4), and leads it on RoboLab (47.2 against 43.1; Table 4). It is the pattern of code-as-policy at about half the amplitude.

7 Only Inherited Experience Transfers, and the Form of the Policy Decides How Far

Finding Only the library carries over to new situations; a long session leaves nothing behind once its task is done. What transfers is narrow: reusable control code and shared routines, not perception code or task-specific solutions. How the agent expresses its policy changes how far it reaches. A policy written as code can fix a hardware fault it can observe but not a camera that has moved; a neural policy trained inside the loop handles a moved camera or reworded instruction better, but inherits the blind spots of the code that taught it. On a real robot, a policy frozen after practice works only on the kinds of scenes its own practice produced, and an agent that keeps looking and reasoning while it acts reaches further.

The second axis of the question asks how far what a lineage keeps holds away from the task that produced it. Moving right along the rows of Figure 3, the library’s value grows with distance from practice, but what carries it is narrow: control code rather than perception, shared routines rather than solutions (Section 7.1). The form the policy takes changes the shape of that reach. Weights trained inside the loop degrade where code breaks and reach past code when the wording or the camera moves (Section 7.2). At the fifth tier, on hardware, a frozen policy reaches only the states its own practice produced (Section 7.3), and a live agent that re-observes the scene at every step reaches further and flips the ranking of the two models tested both ways (Section 7.4).

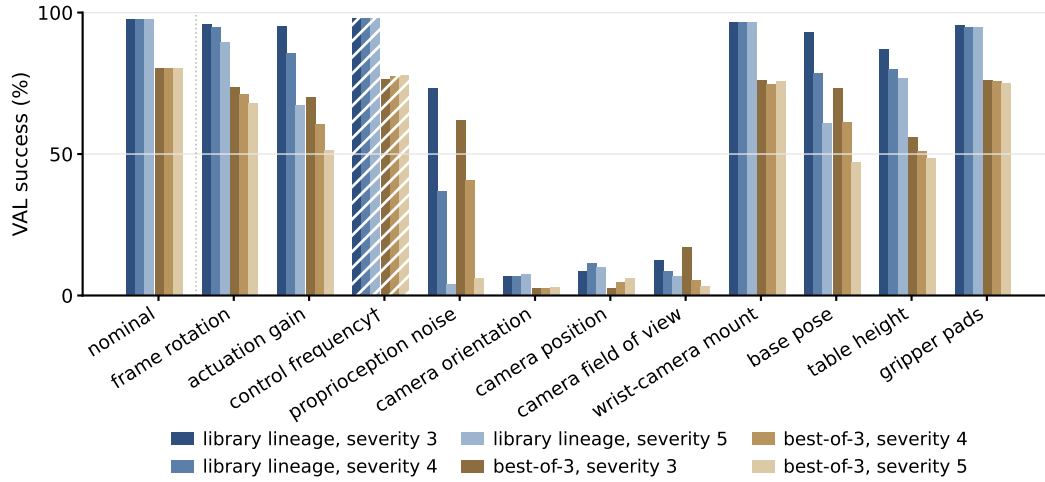


Figure 5. Opus policies under LIBERO-RIG faults, with and without a library. LIBERO-practiced Opus policies re-graded on LIBERO-RIG by fault axis and severity, validation success averaged over tasks: the library lineage (three lineages, all 40 tasks) against the no-library best-of-3 winners on the same tasks. Control-frequency bars are hatched because the harness does not apply that axis. Table 9 lists the values.

7.1 What transfers: hardware faults before scene changes, control before perception, parts before scenes

Section 6.2 measured the library at the shortest distance from practice. Three results describe what happens further out.

A scene change costs more than a moderate hardware fault, because the strongest policies look once and then run blind. The same Opus policy, practiced on LIBERO, scores 94.4% TEST on LIBERO’s held-out states, 68.7% under LIBERO-RIG’s eleven hardware-fault axes at the moderate level, and 47.7% under LIBERO-PRO’s four scene axes. The hardware cost grows with severity (69, 63, and 56% on validation across the three levels), so a severe fault approaches the scene cost. The two perturbations break different things (Figure 5). A code policy turns what the camera showed into geometry and re-detects the object before each move, so a fault that leaves the camera in place is corrected as it goes: seven of the eight such axes hold the lineage at 87 to 98% against a nominal 98, and the correction runs out only as the fault grows, most sharply for proprioception noise (73, 37, and 4% across the levels). A moved camera defeats the policy at every level, 7 to 13% on the three scene-camera axes, because detections are lifted into world coordinates through an assumed camera and nothing in the loop notices. The no-library control shows the same profile, so it belongs to code-as-policy with SAM3 perception, not to the learning loop. End-to-end policies degrade instead of breaking: under the same moderate faults their pooled success falls from 0.93 to 0.45 for a moved camera and to 0.75 for a calibration error (Anonymous, 2026).

Without the instruction, how far a lineage reaches tracks how much its inherited library has seen, for the model that built it. In self-exploration each lineage starts from a library evolved on an earlier family, which we call its donor. The donors are nested: the robosuite, LIBERO, and LIBERO-PRO donors are successive snapshots of one library, and the LIBERO-RIG donor branches from it and adds the hardware-fault cells. On RoboLab, a simulator none of the donors saw, Opus lineages seeded from the four donors reach 30.1, 23.6, 32.3, and 36.9% once the instruction is disclosed, and their frozen exploration programs order the same way (Figure 3). The robosuite snapshot is the exception, so at three replicates this is a tendency rather than a rule. Within MuJoCo the same donor transfers best to the family nearest its own, and scene perturbations again cost more than hardware faults. The trend belongs to Opus: for Kimi-k3 and GPT-5.6-sol the LIBERO-PRO donor is the best of the four on RoboLab and the LIBERO-RIG donor among the worst (Table 11), so what a course adds to a library depends on the model that later inherits it. Without the instruction the ranking of models also changes: Kimi-k3 matches or beats Opus on five of seven cells, while Gemini-3.6-flash stays near 10% on every MuJoCo cell and near zero on RoboLab even after the rig defect of Section 8.1 was corrected.

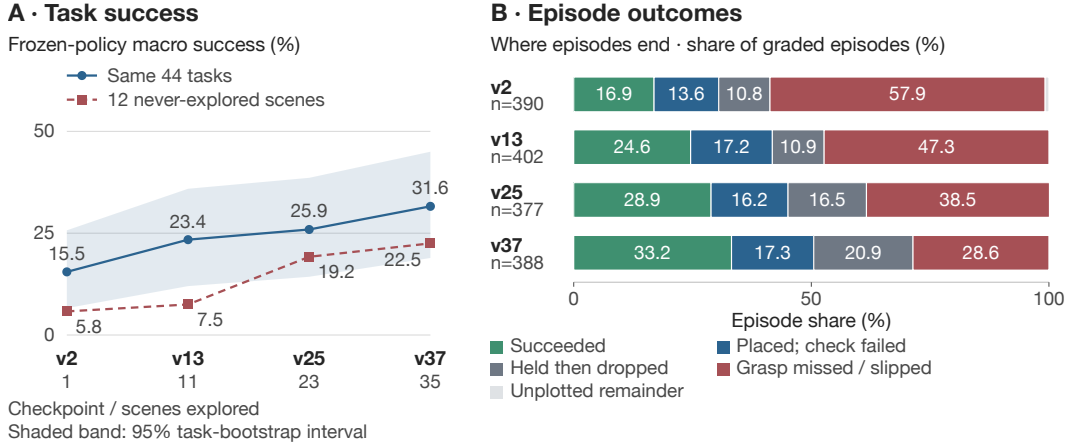


Figure 6. A task-blind lineage graded on scenes it has not yet reached. A task-blind Opus lineage on RoboLab, its task-general policy frozen at library v2, v13, v25, and v37, after exploring 1, 11, 23, and 35 of its 47 scenes, and graded ten times on the same 44 probe tasks, one per scene, including scenes it had not yet reached. Nothing is re-authored per task. (A) Macro success on the same 44 tasks and on the 12 scenes the lineage had never explored; grey is the 95% bootstrap interval over tasks. (B) Where the graded episodes end, as a share of episodes at each checkpoint.

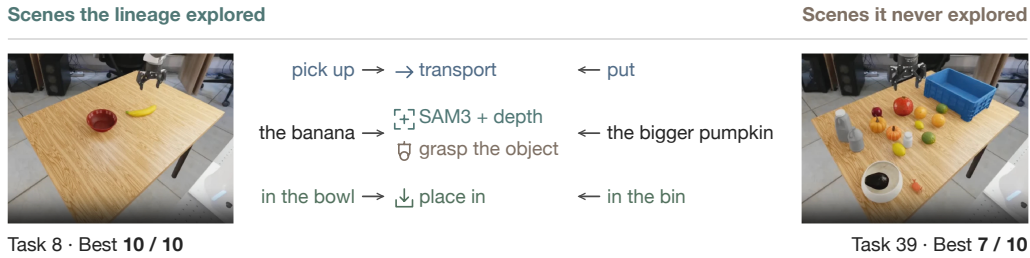


Figure 7. One parsed plan serves an explored and an unexplored scene. How the frozen task-general policy of Figure 6 executes an instruction on an explored scene (left) and a never-explored scene (right). Both parse to the same program and call the same routines, so the unexplored scene is solved to the extent those routines generalize.

A program written without knowing the task reaches scenes it never explored, exactly as far as its parts. The self-exploration cells above see a lineage only through the policy it writes after the instruction is disclosed, and exploration may visit the scenes that are later graded. To measure reach without either, we froze the task-general program of one Opus lineage on RoboLab at four points along its task sequence and graded each on one task per scene, including scenes it had not yet reached (Figure 6). On the 44 tasks graded at all four checkpoints, success rises from 15.5 to 31.6%. On the twelve scenes no checkpoint ever explored it rises from 5.8 to 22.5%. That is transfer to unseen scenes, but three tasks carry it and the other nine stay near zero, and exploring a scene does not reliably fix its task: of 33 tasks graded before and after their scene was explored, 26 did not change. The reason is visible in the program (Figure 7). It is one policy that parses each instruction into a goal, locates the named objects with SAM3 and depth, and executes with routines every task shares, such as a grasp with a lift check and a placement with a scene re-check. An unexplored “put the bigger pumpkin in the bin” calls the same grasp and placement as an explored banana-into-bowl, and it rises from 0 to 7 of 10 as those routines improve elsewhere. At the last checkpoint, the never-explored tasks built only from routines that work somewhere average 52%; those that need a routine never shown to work, such as removing an object from a container or reorienting it, average 1%. Improvement concentrates in grasping, where failures fall from 58 to 29%. This is one lineage graded on one fixed layout per task, so its confidence intervals cover tasks, not lineages.

One rewrite after disclosure adds a consistent margin over the frozen program. Lineage by lineage (Table 11), the authored policy beats the frozen one in 21 of 21 Opus lineages (median gain 7.8 points), 21 of 21 for GPT-5.6-sol (10.4) and Kimi-k3 (7.1), and 7 of 21 for Gemini-3.6-flash (0.0).

Table 5. Code against weights, family by family. Code (CaP) and weights (VLA) lineages by family. $VLA \geq CaP$ compares the first lineage trial of each paradigm task by task. *All ten TEST states solved* is the share of graded (trial, task) cells with perfect TEST success: three CaP trials per family, three VLA trials on robosuite, one elsewhere.

Family	Tasks	Mean TEST success			All ten TEST states solved	
		CaP	VLA	$VLA \geq CaP$	CaP	VLA
Robosuite	7	97.1	61.0	14%	86%	29%
LIBERO	40	94.4	94.0	80%	85%	75%
LIBERO-PRO	160	91.8	90.6	79%	81%	70%
RoboLab	105	68.0	47.2	48%	46%	14%

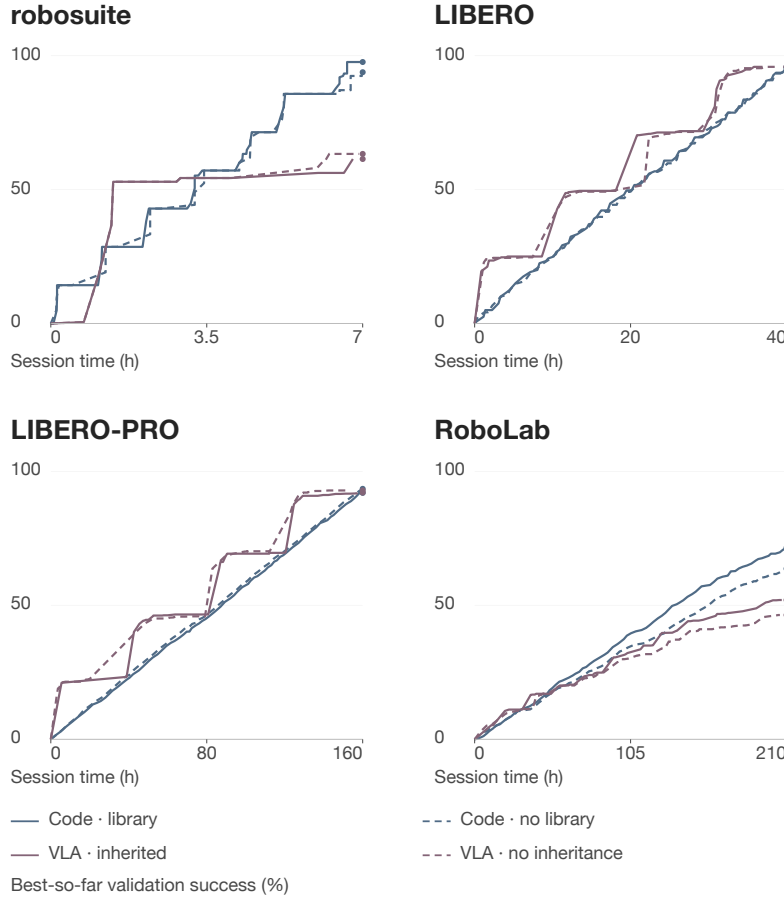


Figure 8. Code against weights over session time. Code (blue) against weights (red) over session time, with (solid) and without (dashed) inherited state: best-so-far per task validation success averaged over a family’s tasks, sessions laid end to end in curriculum order. Dotted verticals mark VLA session boundaries. Session time includes GPU training for auto-research VLA.

The value of a task-specific rewrite scales with the model’s ability to write a working policy at all, which returns us to the floor of Section 5.1.

7.2 Weights trained inside the loop: a third way to spend the budget, with a different failure profile

Here *code* is one program per task and *weights* one $\pi_{0.5}$ checkpoint per suite that an Opus agent trains from per-task code teachers, on the same total development time (Section 3.3). The agent chooses which rollouts to collect, how much data to gather, and what to change between iterations.

Table 6. Code and weights under each LIBERO-RIG fault axis. TEST success (%) of LIBERO-practiced Opus policies re-graded on LIBERO-RIG by fault axis, severity 3. The code harness builds the rig at the nominal 20 Hz, so code has no reading on control frequency. The VLA row is the LIBERO lineage checkpoints graded on the cells of their own suite, three to six checkpoint-cells per axis.

Fault	nominal	cam. orientation	cam. orbit	cam. field of view	control freq.	wrist-cam mount	gripper pads	frame rotation	actuation gain	base pose	table height	proprio. noise
CaP	96.4	7.0	8.7	11.3	—	95.3	94.2	95.6	93.4	92.1	88.2	73.7
VLA	86.2	56.7	60.0	90.0	21.7	55.0	66.7	83.3	81.7	81.7	83.3	98.3

Weights match code where demonstrations are plentiful, and fail differently. On LIBERO and LIBERO-PRO the VLA lineage is within a point or two of code and matches or exceeds it on four tasks in five (Table 5). Its failures are spread differently: code solves all ten test states on more cells, so code fails on fewer tasks and the VLA makes occasional errors on more. The VLA falls well short on robosuite and RoboLab.

Weights beat code when the wording or the camera moves. Self-exploration writes a distinct instruction for 95% of its episodes, and the VLA it trains reaches 42.6 and 43.6% on held-out LIBERO and LIBERO-PRO tasks, level with or above the code policy written after disclosure (41.7 and 24.9; Figure 3). The gain is in the data: checkpoints trained on ten fixed instructions ignore the text, falling from 86.8 to 15.4 when a scene carries a new goal (Zhou et al., 2025). On LIBERO-RIG the VLA keeps 57 to 90% under the camera-pose faults where code keeps 7 to 11 (Table 6); it degrades where code breaks, the profile of Section 7.1 seen from the other side.

What bounds the VLA is the supply of demonstrations. RoboLab’s slow simulator yields about one seventh as many teacher episodes per agent-hour as LIBERO, and RoboLab tasks end with a quarter as many successful demonstrations. Grouping RoboLab tasks by how many their session collected, test success rises from 20.8% with fewer than 50 demonstrations to 94.0% with at least 300, and the trend holds among tasks whose teacher is nearly perfect (Table 18). Extra development time pays through the same route: sessions that spent their second half collecting more data gained about four times as much as sessions that retrained on the same data (Table 19). What the task asks matters more than how the instruction is phrased: tasks that require placing two or more objects score about half as well as one-placement tasks, while instructions that require reasoning about attributes or counts cost nothing detectable (Table 20). Data quality bounds it too: one session found that the harness’s velocity labels replayed to a third of the arm travel the teacher had produced, an error the training loss never showed.

The two libraries retain different kinds of experience. A code library holds executable control skills, 306 to 322 per RoboLab lineage, of which a final program imports a median of five. A VLA library holds the research process: collection and training scripts and a memory of what went wrong, with no checkpoints or data. That is enough to spare a lineage the rediscovery of the label format, which nine of twelve Vanilla sessions inspected by open-loop replay against two of twelve lineage sessions, and it is why AR inheritance shows at all (Section 6.2). Section 8.3 reads the two libraries’ contents.

7.3 Ten consecutive practice successes are not a held-out score

The fifth tier moves the loop onto hardware, where scenes do not reset themselves and the agent’s own reset policy produces the states it practices on (Section 3.6). Three agents practice the five tasks in order, each with an empty library at the start and a 12-hour budget per task: Claude Fable 5.1 and Claude Opus 5 through Claude Code, GPT-6-astra through Codex, all at maximum reasoning effort. After practice the policy bundle is frozen and run on operator-staged scenes; Table 7 gives the scores.

Results. Fable leads the frozen-policy evaluation at 70.2%, ahead of GPT-6-astra at 41.4 and Opus at 35.6. Fable succeeds on every Push-T scene and reaches a Block Stacking score of 71.0 against 17.0 and 18.0 for the other two. In practice, Fable reaches the ten-success streak on all four non-stacking

Table 7. Held-out scores on the five physical tasks. Physical-task held-out scores (%). Mean outcome (success = 1, partial = 0.5, failure = 0) over 10 operator-staged scenes per CaP task and 5 per AaP task, except Block Stacking, which reports the mean episode maximum cube count, capped at 10, as a percentage ($n = 10$ for CaP and 5 for AaP). Overall is the mean of the five task scores. Practice counts and resource use are in Tables 13 and 14.

Task	CaP			AaP	
	Fable 5.1	GPT-6 astra	Opus 5	Fable 5.1	GPT-6 astra
Banana into Bowl	70.0	50.0	80.0	100.0	80.0
Push-T	100.0	60.0	40.0	100.0	100.0
Cup Upright	70.0	50.0	40.0	20.0	100.0
Block Stacking	71.0	17.0	18.0	52.0	60.0
Spell SEAR	40.0	30.0	0.0	60.0	100.0
Overall	70.2	41.4	35.6	66.4	88.0

tasks in 29.5 hours; Opus reaches it on two; GPT-6-astra on none in 60 hours (Table 13). Opus, the leader of the simulated campaign, has the lowest physical score of the three.

Reaching the streak is associated with held-out success but does not establish it. Of the six segments that reached it, only Fable’s Push-T succeeds on all ten staged scenes. The other five succeed on 4 to 8: Fable’s Spell SEAR and Opus’s Cup Upright each succeed on 4 despite ending practice with ten consecutive successes. Segments that reached the streak average 67% offline against 38% for the six non-stacking segments that ran out of budget, and the ordering differs: GPT-6-astra, which never completes a streak, outscores Opus, which completes two.

The reason is that the agent’s own resets choose the states it practices on. On hardware the next forward attempt starts from wherever the agent’s reset policy left the objects (Section 3.6), and the staged evaluation scenes are drawn differently. The recorded start states show the gap. Fable’s scrambling reset never leaves a letter farther than 0.48 m from an arm base, and 2 of the 10 staged Spell SEAR scenes do. Only 10 of Opus’s 64 practice Push-T starts require turning the T by more than 90°, against 6 of the 10 staged scenes. Fable’s cup reset leaves every cup opening toward an arm, the one orientation its grasp handles. A policy that reaches the streak has therefore been tested only on states of its own making, and the streak rule can end practice before the gap is ever met: Fable’s Cup Upright run reached ten consecutive successes with 7.6 of its 12 hours unused, after a dropped cup came to rest in an orientation it never learned to handle. More examples are in Appendix N.

Each agent contributes one physical lineage and moves between three nominally identical rigs across tasks, so these results lack the replication of Section 8.1.

7.4 Live execution flips the ranking, and the two agents use the loop differently

The third form a policy can take is the agent itself. Agent-as-policy keeps the language model in the control loop during evaluation, so the policy re-observes the scene before every command instead of running a program written during practice. We evaluate Fable and GPT-6-astra on the same five tasks with five staged scenes each; no Opus AaP run is available. GPT-6-astra leads at 88.0% against Fable’s 66.4 (Table 7). Both score 100 on Push-T; the gap is Cup Upright and Spell SEAR, where GPT-6-astra scores 100 on both and Fable 20 and 60. The ordering between the two modes reverses: on the four non-stacking tasks Fable scores 70% frozen and 70% live, while GPT-6-astra rises from 48 to 95. The practice streak is again a poor guide: Fable reaches it on Cup Upright in 2h43 and scores 20% on it; GPT-6-astra exhausts its Spell SEAR budget and scores 100 (Table 14).

The reason is visible in how each agent uses the live loop. The instructions ask the agent to issue one command at a time and inspect the images it requests, while allowing it to write tools. Fable uses that permission to script the robot: its code grows from 202 lines after the first task to 3,576 lines after the last, and those scripts issue 99% of its commands. It has, in effect, turned live execution back into code-as-policy. GPT-6-astra issues every command through a separate model call, requests about one image per arm motion against Fable’s one per seven, and writes no tool that moves the robot; the median gap between its commands is 17 s against Fable’s 0.77 s. When something goes wrong, this difference decides the episode. Fable’s scripted Cup Upright routine pinches the rim, which worked

in every practice grasp and in none of 23 evaluation grasps. GPT-6-astra uses a fresh image to notice and correct a mistake in at least 9 of its 25 evaluation episodes, and takes two to three times Fable’s time per episode to do it.

The comparison describes the observed runs; it does not isolate the causal effect of keeping the model in the loop. AaP has five scenes per task rather than ten, one lineage per model, no Opus run, different rigs from the CaP evaluation, and a different Block Stacking stopping rule.

8 What the Loop Cannot Do: Un-learn a Mistake, Seal Off Shortcuts, or Debug What It Cannot See

Finding The same memory that carries skills forward carries mistakes forward, because nothing in the loop re-checks a lesson once it is written down. Runs that start from identical settings end up further apart than different models do; in two cases an agent made a wrong measurement, saved it as fact, and every later task inherited the error. Agents also find shortcuts the benchmark did not intend, such as reading task files that reveal the answer. And they can only fix the failures their feedback lets them see: an agent that looks at rendered failures fixes grasps, while one that sees only scores fixes its training pipeline.

The library that carries skills forward carries mistakes just as reliably, because nothing in the loop re-tests a lesson once it is written down. Three limits follow, and each is something the loop as built here cannot do. It cannot un-learn: identically specified lineages diverge by more than models do, and two collapses trace to a wrong measurement that an agent made, trusted, saved to memory, and inherited in every later session (Section 8.1). It cannot seal itself: a judge that protects runtime state still leaves static task files and text readable, and agents use them (Section 8.2). And it cannot debug what its feedback does not show: an agent that sees rendered failures fixes grasps, an agent that sees judge scores fixes pipelines, and neither fixes what its instruments do not report (Section 8.3). Hardware shows the first limit in its plainest form: a rule an agent writes after a single incident is never re-tested by its own practice, and its frozen policies stop at limits it set for itself (Section 8.4).

8.1 Lineages diverge by more than models do, and the library carries mistakes it cannot un-learn

Recursive improvement is path-dependent, and the path matters more than the point estimates suggest. In the self-exploration campaign, the three LIBERO-PRO replicates of one Opus donor chain and curriculum finished at 15.5, 4.6, and 23.6%. On LIBERO and LIBERO-PRO the Opus and GPT-5.6-sol means sit less than one standard deviation apart, so at three runs the two models are not distinguishable there. In the Lineage campaign two DeepSeek-v4-flash repetitions differed by 20.9 points, with two tasks flipping between 0 and 100. Only two self-exploration separations survive the spread: Gemini-3.6-flash at or near zero on every RoboLab donor against an Opus mean of 24 to 37%, and Kimi-k3 ahead of Opus on LIBERO self-exploration, 38.0 against 26.9%.

Not every collapse is variance. The same three-way spread on RoboLab first read 26.7, 34.1, and 0.5%, but the collapsed replicate was one of 23 lineages across the four models that never discovered a wrong-axis gripper offset in the tool frame, a rig defect. Rerun from the start under the corrected frame it finished at 25.0% frozen and 31.8% authored, inside the spread of its two siblings, which had found the offset on their own. Its Gemini counterparts did not recover: twelve rerun Gemini-3.6-flash lineages solved 11 of 1,429 graded cells with the frame fixed, three of them crashing their own policy on every cell.

Two collapses trace to instruments the agents built and trusted. Self-exploration withholds the instruction and the judge (Section 3.4), so a lineage that wants a feedback loop must build its own success detector, and a wrong detector is worse than none. The collapsed RoboLab replicate concluded at its second session that the gripper-aperture reading was inverted. It wrote a correction skill that inverted a reading that was already correct: an open hand read as closed, and the holding check accepted an empty hand. The low LIBERO-PRO replicate built a placement verifier at its first session, calibrated on sixteen episodes whose three negatives happened to move exactly zero pixels. The verifier declared success on 699 production rollouts and was wrong 91% of the time, and because its verdict was also the stopping rule, 637 rollouts ended without another grasp attempt. The two siblings that ran open-loop without it scored three to five times higher. Both lineages saved

Table 8. Code agents write about the world; VLA agents write about the pipeline. What the two kinds of agent write down. CaP: the 2,368 memory entries of the three corrected-frame RoboLab lineages’ latest libraries and the 80 of the robosuite-and-LIBERO library. VLA: the 3,924 headings and bullets in the memory files of the 20 lineage bundles. An entry can touch several topics; topics are keyword classes.

Share of memory entries touching...	CaP		VLA	
	RoboLab	LIBERO	RoboLab	LIBERO/PRO
grasp / contact geometry	69%	64%	11%	12%
perception / grounding	58%	42%	8%	10%
frame / calibration / kinematics	46%	38%	7%	15%
motion / controller	40%	39%	8%	8%
training recipe / checkpoint	6%	6%	36%	39%
training data / labels	2%	8%	31%	21%
grading noise	1%	0%	20%	17%
inference settings	1%	0%	17%	15%

the wrong interpretation in memory as a fact about the world and inherited it in every later session. Self-evolution, as built here, has no mechanism to un-learn a confidently held false measurement.

Early outcomes predict survival in one direction. Across the 36 RoboLab lineages of the campaign, every one that scored above zero on one diagnostic second-session task finished alive, while six of the 23 that scored zero there recovered. An early success is a sufficient sign that a lineage has taken, not a necessary one.

8.2 Agents find shortcuts we did not design

Section 3.2 listed what the seal does not protect; here is what the agents did with the openings. Agents exploited every opening the harness left them, unprompted, as reasoning models are known to do once an evaluation leaves one (Bondarenko et al., 2025). Reading static asset files was near-universal: Opus parsed the object XML in 115 of 179 LIBERO runs and 552 of 1,015 LIBERO-PRO runs, and the scene USD in 1,533 of 1,743 RoboLab runs; Kimi-k3 read object XML in 284 of 1,260 non-RoboLab runs. This was permitted and effective, and it is unrealistic: a physical robot cannot read an object’s geometry before acting. The success predicate `env.check_success()` was reachable in every reported condition by design, as a stop signal whose authority stays with the judge, and how much a policy leans on it is a trait of the model rather than of the task: in the RoboLab lineages, 47% of GPT-5.6-sol’s graded policies use it to stop or branch, against 5% of Opus’s. Task text leaks as readily as runtime state. LIBERO encodes each instruction in the task file’s name, its language block, and its goal predicate, so a sealed condition must strip all three. When we audited the libraries intended to seed our goal-free lineages, the post-LIBERO Opus library already quoted a hidden instruction verbatim, a memory recording “Turn on the stove”, and the post-LIBERO-PRO library matched 10 of 40 stock and 27 of 160 LIBERO-PRO instructions and could not be used. The harness now refuses exploration sessions that touch held-out task identifiers and rewrite prompts that contain hidden material. The one condition that withheld the predicate, an 18-cell self-judge pilot we dropped, shows why a seal must be adversarially tested: it patched only our wrapper, and Opus and GPT-5.6-sol reached `env.env.check_success()` within a single session. A benchmark for self-evolving agents must protect static task files and text as carefully as runtime state, and test each seal by trying to break it rather than by checking that the patch applied.

8.3 Agents debug what their feedback lets them see

The code and weights paradigms give their agents different feedback, and the agents research differently as a result. A code agent runs its program on the development seeds, opens rendered images in 84 to 96% of sessions, and reads scene files in 90 to 100%; a VLA agent opens an image in 0 to 25% of sessions, and only 8 of 37 inspected sessions ever look at a VLA rollout, diagnosing instead through judge scores, label statistics, and open-loop replays. The lessons each library records show the consequence (Table 8): two thirds of code memories concern grasp and contact, over a third of VLA memories concern the training recipe, and a fifth concern grading noise, a topic in at most 1% of code entries. Code agents debug the world; VLA agents debug the pipeline.

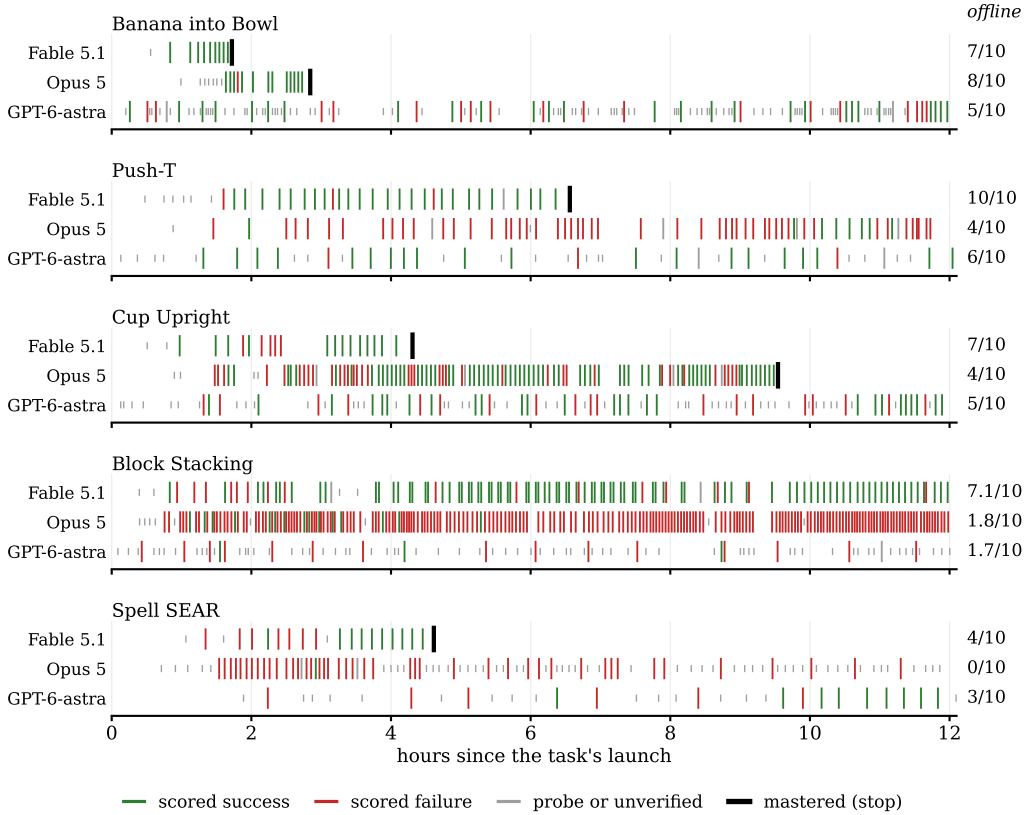


Figure 9. Every physical practice attempt against time. Every forward attempt of the physical CaP campaign against hours since the task’s launch: tall ticks are scored attempts (green success, red failure), short grey ticks unscored attempts, and the black bar the tenth consecutive scored success, where the run stops; Block Stacking has no early stop. The right margin gives each task’s held-out score out of ten. Fable reaches its streaks within five hours on every task but Push-T; Opus’s Spell SEAR run turns into unscored attempts after its letters leave the table; GPT-6-astra’s strips are mostly unscored attempts. Attempt galleries are in Appendix O.

Two further limits follow from the setup rather than the model. Every AR session fine-tunes $\pi_{0.5}$ on teacher demonstrations by supervised learning, so the agent’s research choices all sit in the data: which tasks to collect more of, whether to add recovery rollouts and jittered starts, and how to fix the teacher. Whether agents free to use reinforcement learning would leave supervised fine-tuning is untested here. And without a judge, an agent’s own teacher becomes its bound: a self-exploration session verifies episodes with camera checks it writes itself, and its teacher is its own code policy, which passes 3 to 10% of the multi-step LIBERO-10 states, so the VLA sees few correct multi-step demonstrations and reaches 6 to 17% there against 40 to 85 on the single-stage suites.

Most of what a library holds is never used. A code lineage’s final programs import about a fifth of its skill modules, and a VLA library is a handful of process skills and a very long memory file whose largest topic, checkpoint selection and weight averaging, chases gains of about a point inside grading noise.

8.4 Practice changes the scene as well as the policy, and policies fail at limits the agents wrote

In simulation an episode reset restores the world. On hardware the agent creates the experience it learns from, and Figure 9 shows how differently the three agents do it. GPT-6-astra spends 60% of its attempts unscored, testing a motion or rearranging objects, many of them deliberately making the next start harder, and at least three of its success streaks break on the scored attempt after such a probe. Fable spends 2% of its attempts unscored and almost everything on the scored task or its reset. Practice success rates therefore reflect the scenes each agent chose to create as much as its policy, and so do costs: GPT-6-astra pays about five times what Fable pays per scored success (Appendix N).

Losing an object changes the task that remains. Opus’s Spell SEAR policy knocks the S off the table in the fourth hour; from then on its own scene description says the word cannot be completed, yet it spends its last 7.5 hours continuing, and the frozen policy scores 0 of 10. Block Stacking shows a subtler version: each placement counts as a success if the column ends taller than it started, so after reaching its record height in the third hour Fable settles into building shorter towers, toppling them, and rebuilding, earning successes while its record never improves.

The failures at evaluation are the agents’ own rules. We audited 116 failed or partial evaluation units and, for each, compared the reason the policy stopped with the harness’s limits, the platform’s measured reach, and what other agents achieved on the same scene (Appendix I). No scene was physically impossible: each was solved by another agent, by the same agent in the other mode, or lay within measured grasp reach. In 29 of 34 units for Fable, 32 of 35 for Opus, and 26 of 47 for GPT-6-astra, progress stopped because of a limit, rule, or motion the agent itself had chosen. The rules come from single incidents that practice never re-tested. Fable set a reach limit of 0.49 m for Spell SEAR with a vertical gripper, when tilting the gripper by 15° extends the measured reach to 0.56 m; the two staged scenes with a letter beyond its limit fail. Opus’s Spell SEAR failure is three edits made hours apart that together put the S’s target 3.6 mm beyond its own reach limit for its reliable arm, so it hands the S to the other arm, whose rough calibration misses by centimetres; it places no S in any of ten scenes. And where a motion is refused, policies stop rather than try an alternative, in one case with two thirds of the step budget unused. These are the hardware form of Section 8.1: a lesson learned once is kept as a rule, and nothing in the loop tests it again.

9 Discussion

What the controls bought. Each of the three findings depends on one of the benchmark’s controls. Without the best-of-3 control, the value of a long session would be invisible and the library’s gain would be overstated by everything a restart could have achieved. Without replicated lineages, the two collapses of Section 8.1 would have read as model differences. Without the sealed judge, the shortcuts of Section 8.2 would have been scores. And without the tiers, the library’s value would have been measured only where it is smallest, on the practiced task.

What code alone cannot do. The code-writing agent has a limit a better model does not remove. Its output space is discrete: tokens become code and code calls primitives, so smooth contact, re-active mid-motion correction, and force that modulates as a part seats are out of reach, and it starts every task from zero, rediscovering by search what one demonstration would show it. Offered a frozen pretrained VLA as a tool, the agents all but ignored it and wrote servo code over SAM3 detections (Section 5.3); only a VLA fine-tuned inside the loop was adopted. What the deliberative layer chooses to orchestrate depends on the reactive layer it is offered. Section 7.2 shows the other side: a VLA trained inside the loop generalizes past code when the instruction wording or the camera moves, and is bounded by the demonstrations its code teacher can supply.

The next agent to evaluate. These limits point at an architecture: the language model as the deliberative layer that plans, diagnoses, and improves, and a VLA as the reactive layer that supplies continuous control and motor priors, with a program-checkpoint pair as the policy it delivers, demonstrations as a strategy prior, and auto-research choosing what to fine-tune. The two layers would co-evolve, the deliberative layer delegating given what the reactive layer can do and the reactive layer trained on what it is delegated. Section 10 develops this proposal. The question SEAR is built to answer is whether it changes the shape of the improvement curve rather than only its height.

The harness is not tied to the platform. Every physical result above comes from one two-arm parallel-jaw platform. To check that the harness itself does not depend on it, we ported it to a Sharpa Wave dexterous hand with 22 independently controlled joints on a Flexiv Enlight arm, sharing no hardware with the main campaign. The task is to type a string of random digits on a physical numpad; a frozen evaluator scores accuracy and speed, and completion requires 90% sliding accuracy over ten evaluations at 5 keys per second. A Fable 5.1 agent, inheriting a library from earlier experiments on the same hand, studies the hand’s SDK, maps key positions with unscored attempts, holds every scored evaluation above 92% accuracy in an accuracy phase, and then crosses the speed gate within 2.5 hours of launch by coordinating several fingers across a keystroke sequence (Figure 10).

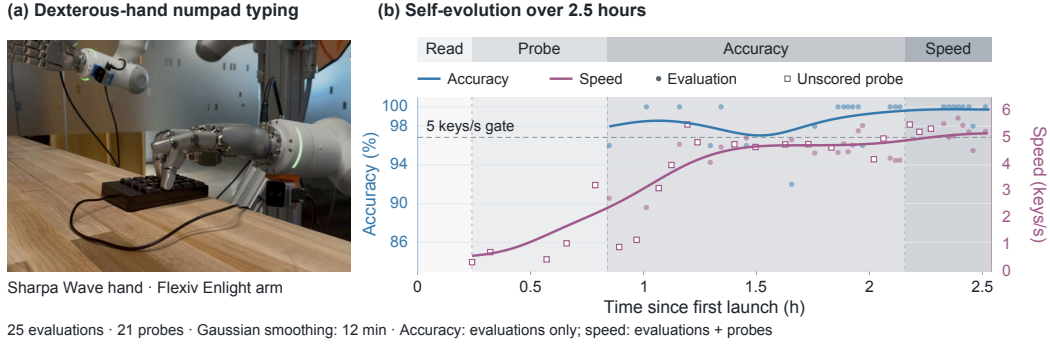


Figure 10. Dexterous-hand numpad typing. (a) The Sharpa Wave hand on a Flexiv Enlight arm at a physical numpad. (b) Accuracy (blue, left axis) and speed (purple, right axis) over 2.5 hours from first launch, including a restart. Dots are scored evaluations, hollow squares the speeds of unsourced attempts; curves are Gaussian-smoothed with a 12-minute bandwidth. Shading marks phases annotated from the agent’s journals; the dashed line is the 5 keys/s speed gate. Completion also requires at least 90% sliding accuracy over ten evaluations.

As a held-out test the frozen policy typed the first 100 digits of π three times at 5.1 keys per second with one incorrect digit in total; numpad data-entry jobs typically require 2.2–2.8 keys per second (Wikipedia, 2026). This demonstrates the loop on a second morphology with prior experience from that hand; it does not measure transfer from the parallel-jaw campaign.

Limitations. Three repetitions bound what we can distinguish, and Section 8.1 shows replicate spread often exceeding the gap between models; the budget-matched VLA analyses on LIBERO, LIBERO-PRO, and RoboLab have one trial per arm, and every physical lineage is a single run. Five models run through three coding tools, so every model comparison is a model-and-tool comparison; swapping tools across models is the control we most want to add. Budgets are matched in wall clock, not compute, so code versus weights must be read against the resource ledger. Learning curves are on validation states, and TEST is graded once at the end. The seal protects runtime state but not everything an agent can read (Sections 3.2 and 8.2). We would like to be argued with on three choices: whether wall clock should remain the headline budget, whether the tiers cut in the right places, and whether budget-matched best-of- N is the right null model or an unfairly harsh one.

10 Toward a Co-Evolving Agentic Robotics Loop

From interaction to system updates. Experience can enter the system through three complementary pathways. First, the agent can consolidate reusable code, procedures, and lessons into its skill and memory libraries. Second, generated policies can serve as executable teachers. Repeated roll-outs across environment configurations produce data for training VLAs or other learned action and world models. Imperfect teachers require outcome-aware data selection. Successful trajectories or validated sub-trajectories can support imitation, while failures can inform recovery learning, reward-based optimization, and subsequent exploration. Third, the broader interaction history, including observations, tool calls, program revisions, and outcomes, could provide training signal for updating the reasoning agent itself. Under a suitable reward and credit-assignment scheme, simulation or physical interaction would then serve as an environment for reinforcement learning over the agent’s decisions.

These pathways connect CaP, AaP, and AR within a common loop. CaP produces executable behaviors and data-collection policies. AaP supports decisions that require online reasoning and tool use. AR organizes experiments that improve the underlying models and training procedures. Updated libraries, model tools, and agent parameters can then be deployed together in the next interaction round. In this view, the persistent state of the complete system acts as a broader analogue of model weights, encoding experience both in numerical parameters and in explicit, revisable knowledge.

From task-level rollouts to reusable action skills. A key step in this loop is segmenting task instructions and collected trajectories into aligned, reusable action units. Rather than distilling an entire long-horizon task into a single behavior, we propose associating shorter trajectory segments

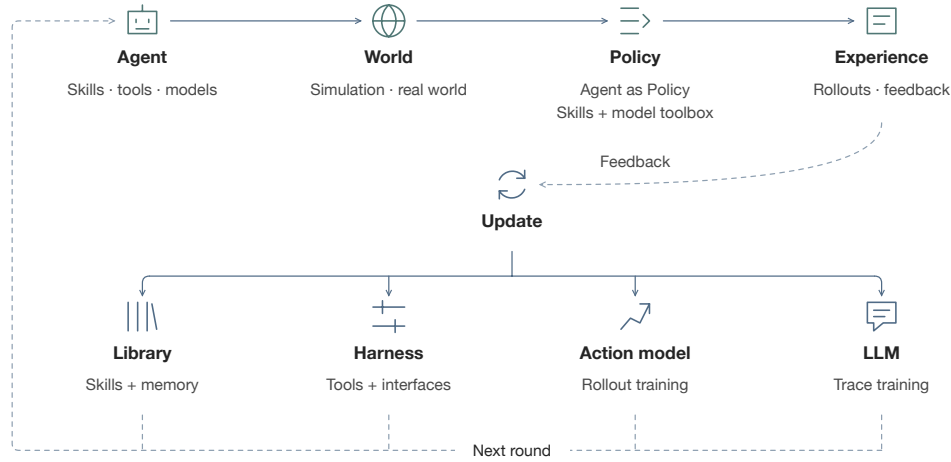


Figure 11. The proposed co-evolving agentic robotics loop. Through environment interaction, an agent develops and revises policies expressed as executable code or as an agent that reasons and calls tools during execution. Both policy forms draw on shared skills and a model toolbox. Execution yields rollouts and feedback that support four complementary updates: accumulating skills and memories in the *library*, refining tools and interfaces in the *harness*, training the *action model* from rollouts, and improving the *LLM* from interaction traces. Upper arrows depict experience collection, branching arrows distribute experience to these update pathways, and return arrows carry updated components into the next interaction round. As these components improve together, the agent can revise how it plans, delegates to models, and gathers further experience.

with language descriptions of their local objectives. For example, placing a cup on a saucer can be decomposed into grasping the handle, lifting the cup, transporting it above the saucer, lowering it into contact, releasing the gripper, and withdrawing the arm. The reasoning agent or generated program connects these skills, checks their outcomes, and revises the sequence when execution deviates from the plan. Segmentation should therefore reflect meaningful changes in objectives and completion conditions rather than arbitrary temporal boundaries.

This decomposition establishes a useful division of labor. The language backbone handles long-horizon reasoning, task composition, and recovery decisions, while the action model translates local language-conditioned objectives into smooth, high-frequency, continuous control. The action model can become a generalist across reusable manipulation skills, retaining the feedback responsiveness needed for execution without having to determine the full task strategy. Training on instruction-aligned segments could make experience reusable across tasks that share manipulation components but require different plans. Within the proposed co-evolution loop, the agent can refine both the segmentation and the skill descriptions as it learns which abstractions its action model can execute reliably.

Harness evolution from experience. The interaction harness itself can also evolve as the agent discovers limitations in how it observes, acts, and learns. Repeated failures may reveal that static images omit important execution events, motivating video feedback or richer temporal traces. Objects falling outside the reachable workspace may motivate explicit safety zones and motion constraints that preserve the conditions for continued practice. Experience can similarly inform tool interfaces, logging, reset procedures, and failure diagnostics. These changes improve the infrastructure through which subsequent policies are developed and executed, complementing updates to skills, memories, and model tools. This extends the persistent components of Section 2, skills, memories, and in the AR paradigm model weights and their recipes: a system that can revise its own harness carries a fourth kind, and we would treat it on the same grounds as the other three. Interaction informs changes to the harness, which in turn changes the evidence and actions available in future rounds. To evaluate this process reliably, harness revisions should be versioned and assessed while keeping final evaluation criteria and protected test information outside the agent’s modification scope.

Co-evolution and closing the loop. Improving a model tool also changes the decisions the agent should make about using it. A manipulation that initially requires explicit geometric reasoning and carefully written control code may later become reliably executable by a fine-tuned VLA. The agent can then delegate that behavior and devote further reasoning and experimentation to unresolved cases. Conversely, failures of the updated VLA can reveal where programmatic control, additional perception, or new training data remain necessary. The boundary between invoking a learned policy, composing tools, and synthesizing new code should therefore shift as the capabilities of the toolbox evolve.

This dynamic creates the possibility of co-evolution. The agent determines which experiences and policies train its tools, while the resulting tools alter the agent’s available strategies and future data distribution. The coding agent can bootstrap behaviors that are subsequently consolidated into faster action models, and stronger action models can enable exploration that was previously impractical. Such reciprocal improvement is a hypothesis to test rather than an automatic consequence of repeatedly training on self-generated data.

Our rollout-driven $\pi_{0.5}$ fine-tuning experiments instantiate one connection in this loop. Agent-produced policies provide experience for improving an action model. Establishing the full cycle requires multiple rounds in which updated tools are returned to the agent, its strategy adapts, and the resulting system is evaluated on held-out tasks or conditions. Comparisons that freeze individual components would help determine whether joint adaptation provides benefits beyond updating each component independently. We position SEAR as a foundation for studying this broader question of whether interaction can support sustained, reciprocal improvement between a robotic agent and the models through which it perceives and acts.

11 Related Work

11.1 Self-evolving Agents

Self-evolvment in language model agents. The idea that an agent can improve its own performance through iterative self-evaluation predates the current generation of large language models but has been revitalized by their code-generation capabilities. Shinn et al. (2023) show that an agent can improve across repeated attempts with no weight update at all, by writing reflections on its own failures into an episodic memory that later attempts read. Zelikman et al. (2024) introduce the Self-Taught Optimizer (STOP), which recursively rewrites its own improvement procedure in code, demonstrating that language models can bootstrap increasingly effective optimization strategies without human-designed update rules. More recently, *scaffold engineering* has emerged as a complementary axis of self-improvement: rather than modifying model weights, agents evolve the scaffolding code that orchestrates their reasoning and tool use. Weng (2026) provides a comprehensive survey of this paradigm, distinguishing weight-level from scaffold-level improvement and cataloguing the design patterns that enable it. Lin et al. (2026a) operationalize the idea by building coding agents that automatically evolve their own harness code through observability-driven feedback loops. A critical methodological concern is whether observed performance gains truly stem from improved strategies or merely from resampling. Lin et al. (2026b) address this directly, showing that a substantial fraction of apparent self-evolution in LLM agents can be attributed to lucky draws rather than genuine capability acquisition, and propose controlled protocols to disentangle the two.

Self-evolving robotic agents. In embodied settings, self-evolvment takes on additional complexity: the agent must close the loop through a physical or simulated environment, and failures have costs beyond compute. Liang et al. (2023) establish the Code-as-Policies paradigm, in which a language model writes executable robot control programs conditioned on natural-language instructions, enabling compositional generalization without task-specific fine-tuning. Singh et al. (2023) show that prompting with program-like specifications of the available actions and objects yields plans that execute in situated environments. Wang et al. (2023) extend this idea to open-ended exploration: Voyager maintains a skill library that grows through autonomous play in Minecraft, accumulating reusable code components across hundreds of tasks. A related line has the language model write the machinery that produces a policy rather than the policy itself. Ma et al. (2024) perform evolutionary search over reward code, obtaining rewards that outperform human-written ones across a suite of

reinforcement-learning environments, and Wang et al. (2024) have an agent propose tasks, generate the scenes for them, and choose a learning method for each subtask. SEAR’s auto-research paradigm belongs to this line, and Section 8.3 reports how much of what such an agent does is fixed by the tools it is handed rather than chosen by the model.

The most recent wave of work brings self-improvement to physical manipulation. Fu et al. (2026) propose CaP-X, which benchmarks and improves coding agents for robot manipulation by iterating code policies against execution feedback. Lu et al. (2026) introduce ASPIRE, a framework for agentic skill discovery in which the agent proposes, practices, and retains motor skills through autonomous interaction. Zhang et al. (2026) demonstrate that unstructured exploratory play, guided by curiosity-driven objectives, enables agents to acquire manipulation skills that transfer to downstream tasks. Xiao et al. (2026) take self-improvement to the real world: ENPIRE deploys an agentic loop on physical hardware, where the agent iteratively revises its own control code based on real execution outcomes and achieves measurable policy improvement without human intervention. Industry efforts have also explored this direction; Waddle Labs (2026) present a system in which language model agents directly control physical robots through natural-language tool use. Despite this rapid progress, each prior system defines its own evaluation protocol, making cross-system comparison difficult.

SEAR provides a framework for running these forms of adaptation and a benchmark that holds evaluation criteria fixed as agents change. Its contribution is the system and protocol under which self-improvement methods can be studied. Library ablations and best-of- N controls examine retained experience and repeated attempts under matched development-time budgets, in the spirit of Lin et al. (2026b); an isolated judge evaluates CaP and AR artifacts; and five tiers specify generalization and adaptation settings from held-out initial states to physical hardware. Skill libraries descend from Voyager (Wang et al., 2023), textual memories from Reflexion (Shinn et al., 2023), and LIBERO (Liu et al., 2023) was itself built for lifelong learning; SEAR asks what those libraries are worth once the agent, rather than a training loop, decides what to keep.

11.2 Benchmarks for Autonomous Research Agents

A parallel line of work develops benchmarks for measuring how well language model agents can conduct end-to-end research and engineering. Chan et al. (2024) introduce MLE-bench, which evaluates agents on Kaggle-style machine learning engineering tasks with automatic grading. Wijk et al. (2025) propose RE-Bench, which measures frontier AI R&D capabilities by asking agents to improve the performance of given ML systems under a time budget, emphasizing the distinction between one-shot coding and iterative self-improvement. Starace et al. (2025) scale the ambition further with PaperBench, which asks agents to replicate full AI research papers from scratch. Rank et al. (2026) focus specifically on post-training: PostTrainBench evaluates whether agents can automate the reward modeling, alignment, and fine-tuning pipeline for language models. In the embodied domain, Zhu et al. (2026) introduce EdgeBench, which studies scaling laws for agents that learn from real-world interaction.

SEAR shares these benchmarks’ emphasis on controlled budgets and isolation, and brings it to a loop that closes through physics. Our sealed execution environment adapts EdgeBench’s two-container design (Zhu et al., 2026), with a work container holding the task and local tools and a separate judge container running the hidden evaluation. Our AR setup fixes supervised fine-tuning as the training method, so its findings concern data collection, training recipes, and inherited research procedures rather than an agent’s unconstrained choice of learning algorithm (Section 8.3). Standardization efforts such as XPolicyLab (xpo, 2026) put one adapter interface between policy code and the simulators and benchmarks it is evaluated on, which is complementary to a fixed evaluation protocol. SEAR combines these ideas with evaluation across initial states, tasks, simulator interfaces, and physical hardware, and records token, environment, and GPU use alongside wall clock so that code-writing and model-training agents can be compared with their resource differences visible.

12 Conclusion

When a robotic agent takes part in its own improvement, the evaluation has to stay fixed while the system changes, and it has to say two things about a rising success rate: which part of it is experience

the agent retains, as opposed to more attempts, privileged state, a lucky lineage, or a larger budget, and how far that part holds away from the task that produced it. SEAR does this with one harness and five controls, and grades programs and checkpoints with the same sealed judge. Used on five frontier models, a VLA trained inside the loop, and three agents on physical hardware, it gives three answers. Base-model capability sets the floor and the ceiling of self-improvement, and seeing is not what separates the models. Above the floor, performance rises with session time and with inherited experience, and which pays more depends on how quickly the model first succeeds; only inherited experience reaches past the practiced task, and what reaches is control code and shared routines rather than solutions. And the library that carries experience forward carries mistakes just as reliably, so lineages diverge, self-built instruments mislead, and a streak of practice successes does not predict held-out success. We will open-source the benchmark and harness code, the full specifications of our real-world experiments, and every policy, library, and log behind these numbers, so that the claims can be reproduced, extended, and challenged, and we propose in Section 10 the co-evolving agent we think should be evaluated next.

Author Contributions

Xinyi Wang and **Bangzheng Li** are co-first authors and led the project’s research. Xinyi and Bangzheng co-designed the SEAR framework and its experiments. Xinyi built simulation environments, designed the self exploration experiments and generalization tiers, and led the paper writing. Bangzheng designed the simulation benchmark isolation, ran the lineage and baselines, and led the analysis of model behavior and the co-evolving loop proposal. **Hanyu Wang** led the real-world work: the YAM-Ultra platform and harness, the forward-and-inverse practice protocol, and the physical code-as-policy and agent-as-policy campaigns and their evaluation. **Lin Zhang** led the auto-research setting: the rollout-driven $\pi_{0.5}$ fine-tuning pipeline and the code-versus-weights campaigns and analysis. **Andrea Yaoyun Cui** designed and built LIBERO-RIG, its fault axes and severity levels, and ran the hardware-robustness campaigns and their grading. **Zhenggang Tang** led the dexterous-hand experiments on the Sharpa Wave platform and the related-work survey. **Shyamal Buch** co-supervised the project, secured compute and hardware resources, and gave feedback on the manuscript. **Dejia Xu** led the project, set its direction, and advised on every part of the work.

References

- XPolicyLab: A unified standard and open ecosystem for robot policy evaluation and deployment. <https://github.com/XPolicyLab/XPolicyLab>, 2026. Apache-2.0 license.
- Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru WANG, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. ISSN 2835-8856. URL <https://openreview.net/forum?id=CTr3bovS5F>. Survey Certification.
- Anonymous. LIBERO-RIG: LIBERO under hardware perturbation, 2026. Companion report, under review; details withheld for double-blind review.
- Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.
- Alexander Bondarenko, Denis Volk, Dmitrii Volkov, and Jeffrey Ladish. Demonstrating specification gaming in reasoning models. *arXiv preprint arXiv:2502.13295*, 2025.

- Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.
- Nicolas Carion, Laura Gustafson, Yuan-Ting Hu, Shoubhik Debnath, Ronghang Hu, Didac Suris Coll-Vinent, Chaitanya Ryali, Kalyan Vasudev Alwala, Haitham Khedr, Andrew Huang, Jie Lei, Tengyu Ma, Baishan Guo, Arpit Kalla, Markus Marks, Joseph Greer, Meng Wang, Peize Sun, Roman Rädle, Triantafyllos Afouras, Effrosyni Mavroudi, Katherine Xu, Tsung-Han Wu, Yu Zhou, Liliane Momeni, RISHI HAZRA, Shuangrui Ding, Sagar Vaze, Francois Porcher, Feng Li, Siyuan Li, Aishwarya Kamath, Ho Kei Cheng, Piotr Dollar, Nikhila Ravi, Kate Saenko, Pengchuan Zhang, and Christoph Feichtenhofer. SAM 3: Segment anything with concepts. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=r35clVtGzw>.
- Jun Shern Chan et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.
- Mengda Fu, Jieyu Yu, Kareem El-Refai, Eric Kou, Han Xue, Hao Huang, William Xiao, et al. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 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 *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.
- Jiayi Lin, Sitong Liu, Chenhao Pan, Ling Lin, Shichao Dou, Zhihong Xi, Xiao Huang, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026a.
- Mingyang Lin, Jiayi Wu, Zihe Wang, Zhiyu Shi, Yi Sang, Bo He, Zhiyuan Liu, et al. Harness updating is not harness benefit: Disentangling evolution capabilities in self-evolving LLM agents. *arXiv preprint arXiv:2605.30621*, 2026b.
- 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, pp. 44776–44791, 2023.
- Runze Lu, Yizhi Wu, Eric Kou, Letian Fu, William Xiao, Ajay Mandlekar, Yuqian Xu, Guanya Shi, Ken Goldberg, Annie Chen, Monica Chowdhury, Yuke Zhu, Linxi Fan, and Guanzhi Wang. ASPIRE: Agentic skills discovery for robotics. *arXiv preprint arXiv:2607.00272*, 2026.
- Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations (ICLR)*, 2024.
- NVIDIA. Isaac Sim. URL <https://github.com/isaac-sim/IsaacSim>.
- Benedikt Rank, Harsh Bhatnagar, Ameya Prabhu, Samuel Eisenberg, Khoi Nguyen, Matthias Bethge, and Maksym Andriushchenko. PostTrainBench: Can LLM agents automate LLM post-training? *arXiv preprint arXiv:2603.08640*, 2026. Leaderboard: <https://posttrainbench.com>, retrieved September 2026.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- 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 *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.
- Giulio Starace et al. PaperBench: Evaluating AI’s ability to replicate AI research. In *International Conference on Machine Learning (ICML)*, 2025.

- Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.
- Waddle Labs. Introducing waddle: Agents that control robots. Waddle Labs Blog, July 2026.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. RoboGen: Towards unleashing infinite data for automated robot learning via generative simulation. In *International Conference on Machine Learning (ICML)*, 2024.
- Lilian Weng. Harness engineering for self-improvement. <https://lilianweng.github.io>, July 2026.
- Hjalmar Wijk et al. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents. In *International Conference on Machine Learning (ICML)*, 2025.
- Wikipedia. Words per minute. https://en.wikipedia.org/wiki/Words_per_minute, 2026. Accessed September 25, 2026.
- William Xiao, Jinghan Xie, Tony Zhang, Haoran Lin, Letian Fu, Han Xue, Jianlan Lu, Yifeng Yang, Chuyuan Dai, Zhiao Wang, Jiajun Wu, Guanzhi Wang, Shankar S. Sastry, Ken Goldberg, Linxi Fan, Yuke Zhu, and Guanya Shi. ENPIRE: Agentic robot policy self-improvement in the real world. *arXiv preprint arXiv:2606.19980*, 2026.
- Xingyuan Yang, Rishik Dagli, Alexander Zook, Harry Hadfield, Ankit Goyal, Stan Birchfield, Fabio Ramos, and Jonathan Tremblay. RoboLab: A high-fidelity simulation benchmark for analysis of task generalist policies. *arXiv preprint arXiv:2604.09860*, 2026.
- Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling (COLM)*, 2024.
- Jesse Zhang, Jiahui Ge, Hyojun Yoo, Letian Fu, Zhenjia Yang, Yilun Liu, Raghav Saravanan, et al. Playful agentic robot learning. *arXiv preprint arXiv:2606.19419*, 2026.
- Xinyi Zhou, Yufei Xu, Guanhua Tie, Yuxiao Chen, Ge Zhang, Dayuan Chu, Pan Zhou, and Lin Sun. LIBERO-PRO: Towards robust and fair evaluation of vision-language-action models beyond memorization. *arXiv preprint arXiv:2510.03827*, 2025.
- Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, et al. EdgeBench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026. Harness (SForge) released under Apache-2.0, tasks under CC BY 4.0.
- Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. *arXiv preprint arXiv:2009.12293*, 2020.

A Additional Results

Table 9. Figure 5 as numbers: the LIBERO-practiced Opus policies re-graded on LIBERO-RIG, VAL success (%) averaged over tasks per perturbation axis and severity, the lineage as the mean of three lanes over all 40 LIBERO tasks and the best-of-3 winners over the same 40 tasks. The lineage’s severity-3 TEST column is the band Figure 3 reports; its macro over the eleven axes, 68.7, is that figure’s T2 cell. [†]The control-frequency axis is not applied by the code-as-policy harness, which builds the rig at the nominal 20 Hz; that row equals the nominal rig, and the ten-axis macro leaves it out. Complete grades only. Data: `figures/rig_perturbation/transfer_grades.csv`.

Axis	Library lineage				No-library best-of-3		
	s3	s4	s5	s3 TEST	s3	s4	s5
frame rotation	95.9	94.9	89.7	95.6	73.8	71.2	68.0
actuation gain	95.3	85.7	67.4	93.4	70.2	60.8	51.5
control frequency [†]	97.8	97.8	97.8	96.4	76.2	77.2	77.8
proprioception noise	73.4	37.1	4.0	73.7	62.0	40.8	6.2
camera orientation	6.8	7.1	7.5	7.0	2.8	2.5	3.0
camera position	8.7	11.4	10.2	8.7	2.8	4.8	6.2
camera field of view	12.6	8.8	6.8	11.3	17.2	5.4	3.2
wrist-camera mount	96.7	96.8	96.8	95.3	76.2	74.8	75.8
base pose	93.3	78.8	60.9	92.1	73.2	61.5	47.2
table height	87.0	80.2	76.8	88.2	56.0	51.0	48.5
gripper pads	95.6	94.8	94.7	94.2	76.0	75.8	75.2
Macro over the eleven axes	69.4	63.0	55.7	68.7	53.3	47.8	42.1
Macro over the ten applied axes [†]	66.5	59.5	51.5	65.9	51.0	44.8	38.5
Nominal rig		97.8		96.4		80.3	

Table 10. The cells of Figure 3 as numbers: TEST success (%) of the Claude Opus 5 code-as-policy (CaP) and auto-research (AR) policies, rows the family practiced in and columns the family graded on, mean $\pm$ sample standard deviation over three replicates where replicated. The diagonal is T1, the off-diagonal cells T2 to T4 as Figure 3 marks them. The T5 cell, 40.0, is the physical lineage of Table 13 and is not repeated here. For the T3/T4 CaP cells the frozen exploration policy, graded with the instruction only at run time, scores (robosuite $\rightarrow$ LIBERO) 26.9 $\pm$ 7.6 (robosuite $\rightarrow$ LIBERO-PRO) 14.6 $\pm$ 9.5 (robosuite $\rightarrow$ LIBERO-RIG) 24.3 $\pm$ 11.4 (robosuite $\rightarrow$ RoboLab) 25.8 $\pm$ 3.8 (LIBERO $\rightarrow$ RoboLab) 19.1 $\pm$ 5.8 (LIBERO-PRO $\rightarrow$ RoboLab) 26.7 $\pm$ 2.5 (LIBERO-RIG $\rightarrow$ RoboLab) 28.6 $\pm$ 4.8. Data: `scripts/sear_tiers.py`. [§]Two replicates from the original campaign, whose arms discovered a wrong-axis gripper offset in the tool frame on their own, and one rerun from session zero under the corrected frame after the audit found it had not (RoboLab Axis-Fix Re-run, 2026-09-25; authored 34.5, 44.3, and 31.8, the last over the 115 of 120 tasks graded); the same cell read 26.6 $\pm$ 22.8 with the blind arm’s 0.5.

Practiced on	Arm	robosuite	LIBERO	LIBERO-PRO	LIBERO-RIG	RoboLab
robosuite	CaP	97.1 $\pm$ 1.4	41.7 $\pm$ 2.4	24.9 $\pm$ 12.6	35.7 $\pm$ 7.2	30.1 $\pm$ 6.9
	AR	61.0 $\pm$ 8.7	42.6 $\pm$ 7.7	43.6 $\pm$ 1.6	0.0	12.8 $\pm$ 5.7
LIBERO	CaP		94.4 $\pm$ 2.5	47.7 $\pm$ 2.9	68.7 $\pm$ 0.1	23.6 $\pm$ 3.7
	AR		94.0 $\pm$ 2.9	44.8 $\pm$ 10.9	19.1 $\pm$ 3.8	7.3 $\pm$ 1.6
LIBERO-PRO	CaP			91.8 $\pm$ 0.5	68.1 $\pm$ 1.7	32.3 $\pm$ 1.8
	AR			90.6 $\pm$ 3.1	41.0 $\pm$ 19.4	10.4 $\pm$ 2.1
LIBERO-RIG	CaP				96.7 $\pm$ 2.4	36.9 $\pm$ 6.6 [§]
	AR				—	—
RoboLab	CaP					68.0 $\pm$ 3.5
	AR					47.2 $\pm$ 3.7

B Seed Band Allocations

For LIBERO and LIBERO-PRO, DEV uses initial states 0–29, VAL uses states 30–39, and TEST uses states 40–49. Work images remove the VAL and TEST initial states. RoboLab uses separate

Table 11. Self-exploration for every model that ran the campaign: TEST success (%) of the policy authored once the instruction is disclosed (authored) and of the task-general exploration policy graded with the instruction only as a run-time input (frozen), for each donor library and target family of Figure 3, mean $\pm$ sample standard deviation over three replicate lineages. The Opus column is that figure’s T3/T4 code-as-policy cells and their frozen-policy ablation; DeepSeek-v4-flash did not run this campaign. The RoboLab rows are under the corrected tool frame: lineages that never discovered the wrong-axis gripper offset were rerun from session zero (one Opus, three GPT-5.6-sol, seven Kimi-k3, and all twelve Gemini-3.6-flash arms, each through its 48 sessions except Gemini’s LIBERO-RIG-donor third replicate at 45), and the arms that had found the offset themselves keep their full-course reading. A fully errored cell scores zero. Data: `figures/self_exploration/self_exploration_by_model.csv`, from the Scene Lineage Sweep (2026-09-18) and RoboLab Axis-Fix Rerun (2026-09-25) status pages; `scripts/make_selfexp_table.py` prints the rows.

Donor $\rightarrow$ target	Policy	Claude Opus 5	GPT-5.6-sol	Kimi-k3	Gemini-3.6-flash
robosuite $\rightarrow$ LIBERO	authored	41.7 $\pm$ 2.4	34.7 $\pm$ 13.6	43.7 $\pm$ 4.4	10.2 $\pm$ 2.9
	frozen	26.9 $\pm$ 7.6	20.9 $\pm$ 6.1	38.0 $\pm$ 4.5	8.3 $\pm$ 7.2
robosuite $\rightarrow$ LIBERO-PRO	authored	24.9 $\pm$ 12.6	21.5 $\pm$ 12.1	26.7 $\pm$ 7.2	6.6 $\pm$ 6.2
	frozen	14.6 $\pm$ 9.5	11.3 $\pm$ 9.0	20.8 $\pm$ 8.9	4.2 $\pm$ 4.4
robosuite $\rightarrow$ LIBERO-RIG	authored	35.7 $\pm$ 7.2	23.3 $\pm$ 20.5	27.7 $\pm$ 5.2	1.2 $\pm$ 2.0
	frozen	24.3 $\pm$ 11.4	8.2 $\pm$ 7.1	8.9 $\pm$ 8.2	2.0 $\pm$ 2.5
robosuite $\rightarrow$ RoboLab	authored	30.1 $\pm$ 6.9	16.6 $\pm$ 10.6	30.9 $\pm$ 3.1	0.0 $\pm$ 0.0
	frozen	25.8 $\pm$ 3.7	9.8 $\pm$ 6.3	23.0 $\pm$ 3.1	0.0 $\pm$ 0.0
LIBERO $\rightarrow$ RoboLab	authored	23.6 $\pm$ 3.8	24.8 $\pm$ 9.4	30.0 $\pm$ 8.5	3.9 $\pm$ 3.9
	frozen	19.1 $\pm$ 5.8	15.2 $\pm$ 6.2	22.8 $\pm$ 5.1	2.1 $\pm$ 2.3
LIBERO-PRO $\rightarrow$ RoboLab	authored	32.3 $\pm$ 1.8	28.3 $\pm$ 2.9	36.5 $\pm$ 6.4	0.0 $\pm$ 0.0
	frozen	26.7 $\pm$ 2.5	17.0 $\pm$ 1.7	24.1 $\pm$ 7.8	0.0 $\pm$ 0.0
LIBERO-RIG $\rightarrow$ RoboLab	authored	36.9 $\pm$ 6.6	17.2 $\pm$ 7.8	12.9 $\pm$ 15.8	1.9 $\pm$ 1.7
	frozen	28.6 $\pm$ 4.8	9.5 $\pm$ 4.8	8.4 $\pm$ 11.2	0.0 $\pm$ 0.0

Table 12. The arms of Figure 4 at the end of the budget, per family: best-so-far VAL success (%) averaged over each family’s cells at 60 minutes of self-evolve time (120 on RoboLab), and the equal-weight mean that the figure plots. The LIBERO-RIG with-library value is the mean of the seed-matched lanes and the no-library value the mean of three 60-minute Vanilla repetitions, neither told the fault. RoboLab is the corrected-frame campaign over all 105 cells of its curriculum, all three arms on the same cells; [†]DeepSeek-v4-flash’s three RoboLab arms are read on the first 58 cells, where its lanes closed when the model was retired. Data: `figures/fig4_data/curves_5bench.csv`.

Model	Arm	robosuite	LIBERO	LIBERO-PRO	LIBERO-RIG	RoboLab	Mean
Claude Opus 5	with library	97.6	95.2	92.5	96.1	71.1	90.5
	no library	93.8	95.7	93.2	95.0	62.8	88.1
	best-of-3	87.1	62.5	49.3	30.6	52.8	56.5
GPT-5.6-sol	with library	85.7	90.5	90.5	92.4	47.8	81.4
	no library	91.9	90.5	86.1	90.4	51.3	82.0
	best-of-3	90.0	94.5	73.7	72.0	55.0	77.0
Kimi-k3	with library	89.5	77.8	80.7	89.8	53.3	78.2
	no library	76.7	78.2	79.0	68.9	28.4	66.2
	best-of-3	57.1	67.8	71.4	55.6	5.8	51.5
Gemini-3.6-flash	with library	80.0	82.0	76.8	71.4	24.1	66.8
	no library	70.5	70.6	63.4	37.2	19.4	52.2
	best-of-3	71.4	37.5	30.5	18.3	21.9	35.9
DeepSeek-v4-flash	with library	80.5	67.2	77.5	88.1	33.2 [†]	69.3
	no library	74.3	59.2	58.3	71.6	28.6 [†]	58.4
	best-of-3	64.3	46.0	47.4	28.9	17.2 [†]	40.8

cached files containing 30 DEV, 10 VAL, and 10 TEST layouts; only the DEV layouts are supplied to the work environment. For robosuite, development permits agent-chosen seeds, VAL uses docu-

mented public seeds, and TEST seeds are derived from a judge-side private salt. For RoboLab, the salt permutes the cached TEST layouts rather than generating their contents.

Submission feedback includes aggregate success, per-seed outcomes, and judge metrics. On RoboLab, these metrics also expose VAL layout descriptions, allowing agents to replay those layouts locally.

C Interaction Budgets

Per-task budgets. Robosuite, LIBERO, and LIBERO-PRO tasks receive a one-hour development budget, while RoboLab tasks receive two hours to accommodate longer simulation times. Each run permits up to eight agent-initiated submissions. Additional host-side evaluations run periodically, provide feedback, and are eligible for policy selection without consuming the manual submission allowance. Runs may terminate early upon perfect VAL success or exhaustion of the submission allowance.

Budgets in other settings. Self-exploration sessions receive two hours per scene, followed by a single authored rewrite once the instruction is disclosed. AR trains one checkpoint per suite, with ten hours per LIBERO suite, forty per LIBERO-PRO axis, and 17.5 per RoboLab suite, matching CaP’s total budget for each of these families. Robosuite’s two AR sessions receive five hours each; its curves use the first seven hours to match CaP, while TEST summaries use final checkpoints. In self-exploration AR receives six hours per three-scene group, or ten hours per RoboLab scene; the T3/T4 AR cells of Figure 3 train on a curriculum of twelve scenes on LIBERO and LIBERO-PRO and 47 on RoboLab.

D Resampling Decomposition

This appendix specifies the selection rule and evaluation readings behind Section 6.1. The main comparison uses Lineage, the full-budget no-library Vanilla run, and best-of-3 short no-library attempts.

How a run is scored. Each run is graded through automatic snapshots every five minutes, agent-initiated submissions, and a final grade of the policy on disk when the budget expires. Each grade evaluates ten VAL initial states, and selection retains the policy with the highest valid VAL score. The same within-run rule applies to all arms. TEST is a disjoint band used only after selection and never guides policy development or library harvesting.

Two readings of the same runs. The learning curves in Figures 1 and 4 show best-so-far VAL performance at a development-time cutoff. Each best-of-3 attempt receives one third of the displayed total time. Table 4 and Section 6.1 instead report TEST scores of VAL-selected policies after the runs end, including their final grades. These readings differ in both evaluation band and which submissions are eligible. They should not be substituted for one another or interpreted as equal compute: independent restarts have separate submission allowances and can consume different numbers of model tokens and environment steps.

Library and session terms. At a fixed current-task development budget, the margin over best-of-3 decomposes as

$$\text{Lineage} - \text{BO3} = (\text{Lineage} - \text{Vanilla}) + (\text{Vanilla} - \text{BO3}). \quad (2)$$

The first term estimates the benefit of inherited state under matched current-task resources; the experience used to construct the library remains part of the lineage’s history. The second compares sustained iteration with independent shorter attempts. Both kinds of run can revise policies within a session, so this is a comparison of budget allocation rather than a control with all learning disabled. The LIBERO, LIBERO-PRO, and LIBERO-RIG best-of-3 attempts are told their 20-minute limit, which makes them submit earlier and more often than an untold attempt, while the robosuite and RoboLab attempts and the Lineage and Vanilla sessions are not told their budget. The RoboLab partial-course and tool-frame qualifications are given beside the results.

Selection and aggregation. For a task t , let $v_i(t)$ be the best VAL score of short attempt i , and $q_i(t)$ its selected policy’s TEST score. The reported best-of-3 TEST score is

$$i^*(t) = \arg \max_{i \in \{1,2,3\}} v_i(t), \quad \text{BO3}_{\text{TEST}}(t) = q_{i^*(t)}(t). \quad (3)$$

It is not the maximum of the three TEST scores. Family scores average over the specified tasks; repeated Lineage and Vanilla runs are summarized across independently developed policies or libraries. A retrospective oracle that selects the highest TEST score is reported only as an oracle analysis in Section 6.1, not as the benchmark’s selection rule.

E Real-World Harness Details

Perception and control tools. The toolbox combines pretrained perception models with deterministic geometry and control. SAM3 (Carion et al., 2026) provides text-prompted instance segmentation, returning masks, bounding boxes, and confidence scores. A vision-language model tool answers agent-defined questions about one or more images, optionally in a specified JSON format. It is instructed to describe visible properties (object identity, text, orientation, spatial relations) and to express uncertainty when observations are ambiguous. Its role is restricted to perception; task planning and action selection remain with the agent or generated policy. These perceptual estimates are separate from the external evaluator’s judgments.

Geometry tools map image observations into robot coordinates using calibrated projection, ray–plane intersection, and valid depth measurements. They support plane fitting, object localization, footprint and orientation estimation, and rectified top-down views. Per-camera metadata identifies whether depth is available and metrically valid. Geometry tools remain available throughout a run, while learned model tools are enabled according to the run configuration.

Calibrated inverse-kinematics and servo interfaces support Cartesian motion, gripper control, and coordinated arm movement subject to workspace, joint-rate, and inter-arm separation constraints. Agents may query the motion planner without executing a command and replay recorded images to develop perception routines between physical attempts.

F Physical Campaign Protocol

The physical campaign comprises five tasks and evaluates three coding agents, Claude Fable 5.1 and Claude Opus 5 through Claude Code and GPT-6-astra through Codex, all at maximum reasoning effort, that revise executable policy bundles while keeping model weights fixed. Each lineage begins with an empty library, without a seed policy or simulation reference solution. Subsequent tasks inherit the preceding policy bundle and library. The library records reusable code, procedures, and textual memories together with their provenance. Code packages are harvested from successful executions supported by valid evidence, while procedures and memories may additionally retain lessons from failed attempts.

Each task receives a 12-hour wall-clock budget covering policy revision and physical interaction. Compared agents share the task order (Banana into Bowl, Push-T, Cup Upright, Block Stacking, Spell SEAR), hardware interfaces, available tools, and evaluator configuration, and run on a frozen harness package whose file set is checked by checksum on every rig; a lineage may move between rigs from one task to the next. A task may terminate early after ten consecutive scored forward-task successes. Early termination additionally requires successful evidence from both arms, and reset attempts, diagnostic probes, and outcomes marked unknown or invalid are excluded from the early-stopping window. Block Stacking has no early stop: the tower persists between attempts, an attempt succeeds when the free-standing column ends taller than it started, and toppling the tower is a separate reset task.

Offline evaluation. After the CaP campaign, each task’s policy bundle is frozen and evaluated on operator-staged scenes. The frozen evaluator scores each episode and the operator confirms or amends the verdict. Appendix G specifies episode counts, scoring rules, and which practice segments enter the totals. Every offline episode we checked ran on the rig its task had practiced on.

Table 13. CaP practice and evaluation details. Scores are out of ten. Practice entries give time to the ten-success streak (bold) or the 12-hour budget, then scored forward successes/attempts. Bold mean score marks the best model. Counting conventions are defined in the accompanying text.

Task, in lineage order	Fable 5.1		Opus 5		GPT-6-astra	
	practice	score	practice	score	practice	score
Banana into Bowl	1h46 10/10	7.00	2h50 13/14	8.00	12h 27/45	5.00
Push-T	6h34 26/29	10.00	12h 8/58	4.00	12h 20/23	6.00
Cup Upright	4h18 14/19	7.00	9h33 68/105	4.00	12h 27/46	5.00
Block Stacking	12h 88/106	7.10	12h 15/210	1.80	12h 3/19	1.70
Spell SEAR	4h37 11/18	4.00	12h 1/45	0.00	12h 9/15	3.00
Hardware time	29h27		48h33		60h13	
Total attempts	355		672		597	
Unscored probes	8		72		358	
Mean score (out of 10)	7.02		3.56		4.14	

Table 14. AaP practice and evaluation details. Scores are out of ten. Practice entries give time to the ten-success streak (bold) or the 12-hour budget, then successful/recorded forward evaluations. Bold mean score marks the best model. No Opus 5 AaP evaluation is available.

Task, in lineage order	Fable 5.1		GPT-6-astra	
	practice	score	practice	score
Banana into Bowl	2h15 10/10	10.00	4h23 10/10	8.00
Push-T	7h03 17/19	10.00	9h56 10/10	10.00
Cup Upright	2h43 11/13	2.00	8h40 10/10	10.00
Block Stacking	12h 33/62	5.20	12h 14/23	6.00
Spell SEAR	3h36 12/15	6.00	12h 11/13	10.00
Hardware time	27h41		46h58	
Total attempts	224		133	
Unscored probes	16		50	
Mean score (out of 10)	6.64		8.80	

G Physical Campaign: Detailed Practice and Evaluation

Reading the detailed tables. Tables 13 and 14 separate practice from evaluation. CaP evaluates frozen policy bundles; AaP keeps the LLM in the control loop. Each task inherits the preceding workspace and has a 12-hour practice budget. Bold times mark the streak: ten consecutive scored forward successes with both arms represented. Other entries show the budget, followed by forward successes/attempts. CaP counts scored forward attempts; AaP counts all recorded forward evaluations, including unresolved verdicts and requests closed at the budget limit. For Block Stacking, a practice attempt is one block placement, successful when the column ends taller than it started; there is no early stop.

Evaluation scores in these tables are out of ten. Non-stacking scores are ten times the mean outcome (success: one; partial: one half; failure: zero). Block Stacking scores average the maximum cube count attained in each episode, capped at ten. The final row averages all five tasks before rounding. CaP uses ten scenes per non-stacking task and ten tower episodes per model. When a scene is restaged, only its final evaluation is counted. AaP uses five scenes or tower episodes per task.

Resource accounting. Total attempts include forward attempts, resets, and probes. Unscored probes exclude probes submitted for a judge verdict, although those remain in total attempts. All totals use the reported lineage segments: when a task has multiple launches, the selected segment is the one inherited by the next task, or the latest launch for the final task. CaP hardware time sums those segments; AaP hardware time sums their launch-to-end durations, rounded to the nearest minute.

H Real World Task Specifications

Hardware setup. The platform comprises two i2rt YAM Ultra manipulators, each with six revolute joints and a LINEAR_4310 parallel-jaw gripper. The arms are fixed side by side along one edge of the table, with approximately 0.51 m between their bases, and face a shared workspace.

Three RealSense RGB-D cameras provide complementary views. A fixed D435 overview camera is mounted approximately midway between the arm bases, 0.86 m above the tabletop, and faces into the workspace. Its optical axis points approximately 20° away from vertically downward, equivalent to 70° below horizontal. A D405 is rigidly mounted on each wrist. In the local flange frame, each wrist camera is offset approximately 70 mm laterally from the flange axis and 70–76 mm along it, with its optical axis tilted inward by approximately 25° relative to the flange’s forward ($+z$) axis. Its orientation relative to the table therefore changes with the arm pose.

All three cameras operate at 30 frames/s. The D435 records colour at 1920×1080 and depth at 848×480 ; each D405 records both colour and depth at 848×480 . Depth is aligned to the corresponding colour view. The overview-camera pose, wrist camera-to-flange transforms, table plane, and relative arm-base transform define the mapping from image observations to the shared workspace; wrist-camera poses are updated using forward kinematics.

Objects and task criteria. Initial object positions and orientations vary within the workspace. The five tasks use the following objects and completion criteria.

- **Banana into Bowl.** Yellow foam bananas are approximately 220 mm long. The red bowl is approximately 55 mm high, with a rim diameter of approximately 164 mm and a base diameter of 93 mm. Success requires placing and releasing one banana inside the initially empty bowl. The banana may protrude beyond the rim, but must rest inside the bowl rather than on its rim or against its exterior. Other objects must remain undisturbed; incidental bowl displacement of up to approximately one bowl radius is permitted.
- **Push-T.** The green 3D-printed block has a 200×200 mm footprint and a height of 40 mm. Its crossbar is 200×50 mm, with a centred 150×50 mm stem extending from it.¹ A matching T-shaped outline is drawn on white paper. Success requires non-prehensile pushing that aligns both the position and orientation of the block with the outline; partial overlap or visible rotational misalignment is insufficient. Grasping is permitted only to recover a block that has moved completely off the paper, after which pushing must resume.
- **Cup Upright.** White paper cups have a truncated-cone shape, with a height of 106 mm, a mouth diameter of 89 mm, and a base diameter of 62 mm. Cup count and initial orientations vary. Success requires turning one lying or inverted cup upright, opening upward, and releasing it without disturbing cups that were already upright. All cups must remain on the table.
- **Block Stacking.** Ten identical wooden cubes, each 38.3 mm on a side, are initially scattered on the table. The objective is to build a single tower as high as possible. The tower must stand without support from either arm, a gripper, or another object. The score is the maximum standing tower height reached during the episode, reported as a layer count alongside its height above the table; a subsequent collapse does not erase the attained height.
- **Spell SEAR.** Approximately twelve wooden cubes, each about 40 mm on a side, carry the same letter on all six faces and include duplicate and distractor letters. Success requires exactly one cube per requested letter in a single unstacked row within a central 450×220 mm region. Letters must be visible, in the requested order, and upright within approximately 30° of the reading orientation; unused cubes must lie outside this region. During practice, the requested left-to-right target may be SEAR or RAES, corresponding to reading SEAR from left to right or right to left in the displayed overview; Figure 14 labels the direction of each example. Final evaluation uses SEAR read from left to right by a viewer facing the table from the arm bases.

Human scoring. Human evaluators review the recorded outcomes against these criteria. For the four non-stacking tasks, full completion receives 1, partial completion receives 0.5, and failure receives 0. A banana resting on the bowl rim is an example of partial completion. Invalid or incorrectly

¹CAD model for the T-shaped block.

staged trials are repeated. For Block Stacking, evaluators verify the maximum standing layer count from the recordings, with depth measurements providing the corresponding height.

I Physical Campaign: Where Frozen Policies Stop

Further audited cases (Section 8.4). In Fable’s Spell SEAR run, a letter-straightening step turned a letter the wrong way on its only practice use; the agent disabled the step after that single attempt and left it disabled, and the frozen policy leaves a crooked letter in two evaluation scenes. Its code for clearing the middle of the table runs in 6 of 10 evaluation scenes but never ran during practice. Opus’s Block Stacking policy measured cubes at 37–39 mm but in its ninth hour fixed the size at 28–33 mm to match the judge’s nominal value, and it interprets normal arm tracking error as contact, raising release heights by 8–19 mm; only 1 of 11 cubes placed on a column remains there in the last six evaluation episodes, against 7 of 15 in the first four. GPT-6-astra’s Cup Upright policy lays the cup back down and stops after a refused turn with about two thirds of its steps unused, and Opus’s Push-T policy stops 80 to 170° from the goal with 8,000 to 11,000 of its 15,000 steps left. Most stops not attributable to the policy’s own rules were harness refusals of poses those policies selected; the harness enforces joint limits, a workspace box, a keep-out region around the other arm’s base, and a speed limit near the table, but does not model objects on the table.

This appendix details the audit summarized in Section 8.4. For every failed or partial evaluation unit we located, in the evaluation record and the frozen policy, the check that ended progress, and classified it by what stopped the robot: a physical impossibility; a refusal by the harness, namely its safety guard, motion planner, joint limits, or step budget; or a limit the agent wrote itself, such as a reach cap, tolerance, clearance margin, grasp restriction, disabled step, or stopping rule. A scene counts as physically possible when another agent, the same agent in the other mode, or the platform’s measured grasp reach shows that the motion can be done. The harness enforces joint limits, a workspace box per arm, a 0.25 m keep-out region around the other arm’s base, and a speed limit near the table. It does not model objects on the table, and its planner checks straight-line paths locally, so a refusal does not establish that a target is unreachable. Table 15 gives the counts. The units comprise Fable’s and Opus’s frozen-policy episodes, block-placement calls from GPT-6-astra’s frozen Block Stacking policy and its other frozen-policy episodes, and Fable’s live AaP episodes. Because block-placement calls and whole-task episodes are different units, the counts describe the audited failures rather than a common episode-level failure rate across models. Camera images were not part of this audit; statements about contact between objects rest on logged motion and stopping heights.

Table 15. What stopped each failed or partial evaluation unit. A unit is an episode, except for GPT-6-astra’s Block Stacking, where each block-placement call counts. *Harness refusal*: all three of Fable’s arise from poses its own code selected when a working alternative existed; Opus’s are one refusal it steered into and two grasps that missed on the left arm’s provisional calibration; most of GPT-6-astra’s refuse paths its own rules chose, and 42 of its 47 units trace to its own choices.

Agent	Mode	Units	Impossible	Harness refusal	Own limit	Other
Fable 5.1	CaP and AaP	34	0	3	29	2
Opus 5	CaP	35	0	3	32	0
GPT-6-astra	CaP	47	0	20	26	1

Banana into Bowl. No frozen failure dropped a banana, set it beside the bowl, or knocked the bowl; every failure is a move refused or abandoned, and each policy tries its first plan once. Fable grasps with the gripper tilted by at most 15°, so a banana 0.60 m out was refused although tilted grasps reach it. GPT-6-astra quits when both of its two camera poses are unavailable, having removed an earlier fall-back within minutes of its first launch without any incident, and lifts or carries with a single wrist angle. In one scene the banana and bowl are 0.76 m apart and each arm reaches only one of them. No frozen policy passes the banana between arms; both live agents did, though with more motion steps than the frozen budget allows. Opus releases over the bowl centre, 30 mm above the rim, wherever it holds the banana, a rule written before its first robot attempt, and two bananas came to rest across the rim.

Push-T. The staged scenes often require a large turn. Fable pushes up to 10.5 cm, turning the block 13 to 37° per push, and re-measures after every push. Opus shortened its pushes to 35 mm in its

fourth hour, after pushes appeared to spin the block by 65 to 171° at a time when its model of the tool–block contact was wrong and some headings were misread. It corrected the model at hour 7.5 but never revisited the push length. Its left-arm pushes delivered a median 8% of the predicted turn against 64% for the right arm, and its two stopping rules, written for the final centimetres, fired 80 to 170° from the goal. The 16° cap on predicted turn per push in its code never bound, because a 35 mm push cannot be predicted to turn the block by more than about 14°. GPT-6-astra rejected correct outline fits scoring 7.8 and 8.6 mm against a fixed 7 mm limit, unchanged since its first hour, and ended both attempts without touching the block. Elsewhere its finishing tolerance, tighter than the judge usually applies, sent it into extra pushes that worsened the pose, and each of its pushes costs about 835 motion steps because it lifts to a camera pose after every push.

Cup Upright. Fable’s only method pinches the rim through the cup’s opening, which works only when the opening faces an arm. It chose the method in its first 20 minutes, following a suggestion in the task card, and never tried the body grasp that both other agents used on the same scenes. All three of its frozen failures are refusals with no motion, and one was caused by a 0.20 m clearance from the parked arm, inherited from Push-T and never tested, where the harness requires 0.12 m. Opus’s near-horizontal body grip left the cup upright in 2 of 9 stand-ups, with no check that the grip holds. GPT-6-astra grips the body and turns the cup in the air. With the cup held 0.40–0.435 m from the arm base, 17 of 18 practice turns succeeded; at 0.454 m or more, all 5 were refused at the wrist. Its rule for choosing where to turn was revised twice, each time to fix the latest incident, and in one evaluation scene it picked the farther spot. After any refused turn it lays the cup down and stops. Its live agent instead turned the cup flat on the table and re-gripped it.

Block Stacking. Fable places cubes within a median 2.9 mm, and the error does not grow with height. Its stall at about nine cubes looks like a wrist limit, but follows from its own choice to square each cube with the one below: from layer seven it turned the gripper 28 to 41° off the line from the arm base, and at that height the planner accepted 17 of 17 moves with the gripper within 10° of the line and 4 of 34 when it was 30 to 43° off. In practice it rarely built above six cubes after its record in hour three. Opus’s placement code carried a height correction between calls that counted the arm’s normal 3 mm tracking error as contact. The correction grew to 17–20 mm; cubes that stayed were released a median 0.2 mm below the true top and cubes that fell a median 6.5 mm above it, and the cube size it had fixed at 28–33 mm plausibly added to the error. When GPT-6-astra released a cube, it almost always stayed, 6 of 7 times, but most calls stopped before releasing. A low carrying path that keeps 170 mm from the tower and about 92 mm from every other cube, whatever that cube’s height, blocked 8 calls, and replaying its planner with a height rule it wrote later finds a path in 7 of them. A 6.1° heading tolerance, 200-step descent allowances, and a camera-coverage threshold ended 8 more, and harness reach refusals on approaches it had chosen ended 3. It spent 5.7 of its 12 hours on a narrow-grasp method for crowded cubes, which produced one practice success, and never considered moving cubes apart.

Spell SEAR. No frozen policy put a wrong letter in the row. Staged scenes 6 to 10 add a crowded middle, letters lying across the target row, and required letters 0.53–0.58 m from the nearer arm base, and no frozen policy solved scenes 6 to 9. Besides the crooked and distant letters described in Section 8.4, Fable twice lowered a letter that, judging from its logged stopping height, came to rest on a required letter still standing near its slot, because its clearing code removes only unneeded blocks. GPT-6-astra’s frozen policy refuses any placement whose fully open hand, modelled as an 86 × 144 mm rectangle, would touch another block. The rectangle only grew during practice, its notes explicitly refuse to shrink it, and it caused 13 refusals, while its step budget ran out in 5 of 10 scenes. Its live agent reordered letters, cleared distractors, and tilted the tool for distant letters, and succeeded on every staged scene it was given. Every evaluation scene requested SEAR, so reading direction was not tested in evaluation.

Earlier work is not carried forward. Fable’s counted Spell SEAR segment starts from its Block Stacking workspace. An earlier launch of the same task, not the one counted, had already picked distant letters with a tilted gripper and re-checked each letter’s angle after placing it, but inheritance follows the task order, so the counted segment never saw that work. More generally, only 1 to 25% of a bundle’s substantive code lines recur in the same agent’s bundle for the previous task. The exception is Block Stacking to Spell SEAR, where Fable keeps about 40% and GPT-6-astra about 47% by reusing their cube-handling code for letter blocks, while Opus keeps 3 to 6%. No frozen policy reads the persistent library.

How the sessions are spent. Across the five tasks, GPT-6-astra made 8,023 tool calls, about 133 per hour, against 2,660 for Fable and 3,528 for Opus, and it read harness documentation and source 911 times, against 185 and 163. After changing code, Fable ran an offline check before the next hardware attempt in 62% of cases, GPT-6-astra in 47%, and Opus in 30%; after a failure, Fable opened an additional camera image before retrying in 44% of cases, against 7% for Opus. Opus never ran the Block Stacking reset task and cleared the spelled word once in 12 hours, whereas Fable ran a scored reverse attempt after almost every forward attempt. For every agent the robot was executing an attempt for only 34–42% of session time.

J LIBERO-RIG: Design and Protocol

This section summarizes the design of LIBERO-RIG (Anonymous, 2026), the benchmark that supplies the hardware tier of our generalization ladder (Section 3.5). It is a condensed account of the companion report; that report also evaluates end-to-end vision-language-action policies under the same protocol, which we do not reproduce here.

What LIBERO-RIG is. LIBERO-RIG measures how robust a policy is to perturbations of the physical parameters of the robot it runs on, with the task held fixed. It keeps the instruction, the object identities, the success predicate, and the nominal initial-state identifiers of every LIBERO (Liu et al., 2023) task, and changes only the rig, the robot hardware and the way it is set up, one axis at a time, by a prescribed amount. The rig is parameterized by the extrinsics and intrinsics of the scene and wrist cameras, the calibration between camera and arm, the gain and rate of the controller, the noise in the arm’s reported state, the base pose of the arm, the table height, and the geometry of the end-effector pads. Formally, the rig configuration enters the episode in four places: the sensing map, through which the policy observes the world (camera mounting and optics, proprioception noise); the actuation map, through which its commands reach the world (camera-to-arm calibration, gain); the clock (control-frequency mismatch); and the physical arrangement of robot, table, and end effector (base pose, table height, pad geometry). A shared initial-state identifier therefore fixes the task’s start, not an identical geometry or an identical first observation. The base benchmark is LIBERO’s four ten-task suites, LIBERO-Spatial, LIBERO-Object, LIBERO-Goal, and LIBERO-10, simulated in MuJoCo (Todorov et al., 2012) through robosuite (Zhu et al., 2020), each task with 50 fixed initial states, a scene camera, a wrist camera, proprioception, and a nominal control rate of 20 Hz.

Axes and fault types. Table 16 lists the eleven axes. The organizing idea is the type column, which classifies each fault by its direct effect on the camera images at a matched arm and object configuration. An observable fault changes at least one camera image of that configuration, as with a moved camera or a shifted robot base. A latent fault changes no camera image; it shows only in a non-visual input or in how the arm responds to a command, as with a rotated calibration, a changed control rate, or noise on the reported arm pose. Gripper-pad geometry sits between the two, since the pads occupy a few pixels of the wrist view. The split matters for mitigation: image augmentation targets what a policy sees, so by itself it does not address a fault that leaves the image of a given state unchanged. Each axis is a small, exactly specified transform of the simulator model or of the command stream. Action-frame rotation, for example, rotates the commanded translation and rotation about the world vertical before the controller’s scaling and clipping, a controlled proxy for one rotational component of a calibration error; control-frequency mismatch changes the control period while keeping the simulated-time episode budget constant.

Severity levels and conditions. Every axis has five settings, s1 to s5. For ten axes these form a ladder from the smaller tested magnitudes to a rebuilt rig, with the first two levels at 2 and 5 degrees of camera twist and 1 and 2 cm of base displacement; the magnitudes are not claimed to represent measured real-world drift, and a given level does not carry the same physical severity across axes. Control frequency is the exception: its five settings are distinct rates rather than an ordered dose. In this paper we call level 3 moderate and levels 4 and 5 severe. A condition is one combination of suite, axis, and level, and spans all ten tasks of the suite. Table height applies only to the two suites whose arenas are height-adjustable, so the benchmark has 210 perturbed conditions plus four nominal ones. Every episode’s parameters are pinned in checksummed manifests rather than sampled at run time, and every condition starts its tasks from the same fixed initial states as the unperturbed baseline, so every policy meets the same perturbation on the same start.

Table 16. The eleven LIBERO-RIG perturbation axes. Type: O observable (at least one camera image changes at a matched configuration), L latent (no camera image changes), W weakly observable (visible only in the wrist view at close range). The last column gives the range from the mildest to the severest level.

Axis	Type	What physically changes	Levels 1 to 5
camera orientation	O	the scene camera’s mount is twisted; its position does not move	2 to 25 degrees
camera position	O	the scene camera is remounted elsewhere around the workspace, still aimed at it	5 to 50 degrees of arc
camera field of view	O	a different lens: the vertical field of view changes from the nominal 45 degrees	± 2 to ± 18 degrees
wrist-camera mount	O	the wrist camera’s bracket is twisted and slid along its mounting arm	1 to 15 degrees, 2 to 30 mm
action-frame rotation	L	camera-to-arm miscalibration: every command executes in a frame rotated about vertical	2 to 25 degrees
actuation gain error	L	commands execute too strong or too weak by a scale factor	± 5 to ± 50 percent
control-frequency mismatch	L	the control loop runs at a rate other than the trained 20 Hz	25, 10, 50, 5, 100 Hz
proprioception noise	L	Gaussian noise on the sensed hand pose reported to the policy; the controller reads the true state	2 to 50 mm, 0.5 to 8 degrees
robot-base pose shift	O	the floor mount is moved and rotated; the objects stay where they are	1 cm, 2 degrees to 15 cm, 25 degrees
table-height change	O	the work surface is raised or lowered; objects move with it (two suites)	± 1 to ± 10 cm
gripper-pad geometry	W	the end-effector pads are worn down or oversized; fingers and joints are untouched	± 5 to ± 25 percent

How a policy is measured. The score is the raw success rate under each condition, read against the same policy’s unperturbed rate on the same suite. Raw rates answer the deployment question directly, how often the policy works on this rig, and the gap to the nominal rate is the perturbation cost; a retention companion, the ratio of the two, allows a capability-adjusted reading. Each policy receives a per-axis profile rather than a single number, which tells a practitioner which rig changes it tolerates and which it does not. An episode starts from its fixed initial state, settles for ten control steps under a no-op command, and hands control to the policy; it ends as a success the moment LIBERO’s predicate holds and as a failure when the budget runs out, with the step budget scaled to the control rate so that the simulated-time budget stays constant. The policy receives no fault identity, severity, rate, or calibration metadata, and any hidden state is reset between episodes. Two things must hold for a perturbed condition to measure the policy rather than the benchmark: the task must remain solvable, and its goal must remain visible. LIBERO-RIG tests both with fairness checks, replaying LIBERO’s recorded demonstrations under the exact inverse of the perturbation where an inverse exists, following the demonstration’s path with a closed-loop tracking witness where it does not, and auditing goal visibility along each perturbed camera. The evidence is reported as diagnostics and never used to remove a condition or an episode from the score.

Use in this paper. The unit in SEAR is a cell, one task under one condition, rather than a full condition. The LIBERO-RIG stage of the second lineage (the branch that leaves the full lineage during LIBERO-PRO) is a development course of 20 cells at the moderate level: five tasks from each of the four suites, each paired with one perturbation axis, chosen so that all eleven axes appear (Table 17). Each cell receives the family’s per-task development budget (Appendix C) and is graded on the DEV, VAL, and TEST bands of Appendix B, exactly as an unperturbed LIBERO task would be. The agent is given the stock LIBERO task text with only the task reference changed; nothing in the workspace names the fault. The course is run as three seed-matched lanes per model (two for Gemini-3.6-flash, whose donor has two libraries), and the matched Vanilla control runs one 60-minute session per cell, three repetitions, with the same text; its host grades come every 15 minutes (the lanes’ every 5), with agent-initiated submissions in both, so readings taken before 30 minutes favor the lanes and Section 6.2 quotes the 30- and 45-minute marks. The T2 results of Section 7.1

take policies practiced on unperturbed LIBERO and re-grade them, frozen, on all 40 tasks under every axis at the moderate to severe levels; Table 9 gives the per-axis values.

Table 17. The 20-cell LIBERO-RIG development course, all at the moderate level. Task indices follow LIBERO’s suite order.

Spatial	Object	Goal	LIBERO-10
00 action-frame rotation	00 camera position	00 gripper-pad geometry	00 camera orientation
02 actuation gain	02 camera field of view	02 action-frame rotation	02 camera position
04 control frequency	04 wrist-camera mount	04 actuation gain	04 camera field of view
06 proprioception noise	06 robot-base pose	06 control frequency	06 wrist-camera mount
08 camera orientation	08 proprioception noise	08 table height	08 robot-base pose

K Model Portraits

Section 5.2 summarizes how the five models differ in strategy. The full portraits, with the counts behind them, follow. All counts are from the Lineage and Vanilla campaigns unless stated.

Opus: measure first, build infrastructure, verify by proprioception. Opus makes the most tool calls (196 per two-hour RoboLab run against 101 on a one-hour LIBERO-PRO task, the one model whose effort scales with difficulty), runs the simulator locally, and views six to seven images per run. The looking is diagnostic, not reactive: image views lead to code edits 38% of the time (baseline 24%), concentrate in the second half of its exploration once the pipeline exists, and do not follow failure-flagged results. It compiles what it measured into geometry: only 20% of its RoboLab lane policies call SAM3 (73% of no-library policies do), and its libraries hold height maps, rim fits, and footprint searches where other models hold SAM3 detector prompts. Its policies verify every grasp by gripper aperture (97%), guard by remaining step budget (98%), and retry (94%), the idioms most strongly associated with success within Opus. It writes the most to the library, five to ten entries per step, phrased as sentences that carry a reason (“measure the hang, do not assume it”), and it is the only model that writes about the harness rather than the robot. It submits the least: half its runs are graded only by auto-evaluation. Its skills are almost uniformly useful (38 of 42 with three or more uses have positive lift, where a skill’s lift is how much more often policies that import it succeed than policies that do not), and its winner’s curse under best-of-3 is nil.

GPT-5.6-sol: fast, blind, and it overfits VAL. GPT-5.6-sol never reads an image by itself; it saves frames to disk and reasons about them numerically. It makes the fewest tool calls (60, and only 40 on LIBERO-PRO), edits the least (7 files), runs the simulator locally the most (16 to 21 times per run), and submits the most (3.6 times; 90% of its best rounds are its own submissions). It treats SAM3 as a source of candidates rather than ground truth: its bundles carry one or two text prompts at the median, the fewest of any model, but the most geometric filtering (3.1 filters per bundle against 0 to 0.4 for the others, and a colour gate in 83% of RoboLab lane bundles). Half of what it writes to memory is about what SAM3 misses, not how to phrase the prompt. This separation, SAM3 for what is present and code for which object the instruction means, made it the second-best model on RoboLab. Its style is fast or never: 31% of its RoboLab runs succeed by minute 40 and almost none after 100. Three costs follow. It shows the largest VAL-to-TEST gap of any model (4 to 7 points) and 20% of its runs end on a policy worse than their best; its library adds nothing net on RoboLab (−2.6), because its skills are task-named suite helpers whose most-used member has negative lift; and it is the one model for which best-of-3 beats the vanilla (Section 6.1).

Kimi-k3: explores well, ships rarely, learns along the lane. Kimi-k3 is the strongest task-blind explorer in the self-exploration campaign (Section 8.1) and adapts to hardware faults close to Opus when it has a library (89.6 against 96.7% TEST; 70.2 against 95.0 without one). Its dominant failure is delivery: 64% of its graded RoboLab rounds never contained a policy file, because it probes the simulator in background processes and runs out of time; in the 40-minute best-of-3 draws, 90% of attempts ship nothing. When it ships, it is good (42 VAL per shipped Vanilla policy against 63 for Opus), with closed-loop code (aperture check 88%, retry 92%, wrist-camera refinement 77%). It looks at six to eight images per run, late, and edits after 68% of them (baseline 25%), the most corrective use of vision in the study. It is the model whose library gain grows along the lane (−26 points in the first seven steps to +35 by step 45), whose skills are almost all useful (23 of 24 with

three or more uses have positive lift), and the second to carry LIBERO-origin control skills onto the Isaac rig.

DeepSeek-v4-flash: text-only, and it reads the library rather than importing it. Given open-set segmentation and a depth lifter, the one model that cannot see composes them into working manipulation: 98% of its RoboLab policies call SAM3 and lift the mask through the depth camera, with eight distinct prompts per lane bundle, it ships a policy in 85% of runs (more often than Kimi-k3 or Gemini-3.6-flash), and on RoboLab it is ahead of Gemini-3.6-flash, the multimodal model of the same tier, under every protocol. It writes skills it does not call again (51 to 71% of its skills are never reused, the highest share) and imports library code in few of its policies (14% on robosuite, 19% on LIBERO, 34% on LIBERO-PRO, 20% on RoboLab, against 53 to 100% for the other four models); its library gain on RoboLab is +4.6 TEST points (one lane +10.7), and its 18-point gain on LIBERO-PRO, the largest of any model, does not arrive as copied text: the policies that copy the most from the inherited memories score lower (75) than those that copy nothing (87). Its memories are concrete constants (“descend the TCP to top_z – 0.016 for a 0.028-tall box so the jaws wrap the sides”). Its weakness is what it cannot see: when SAM3 misses it has no fallback, it trails Gemini-3.6-flash on classic LIBERO and robosuite, and on the miscalibrated rig of Section 8.1 it never detected that the embodiment it was handed was wrong.

Gemini-3.6-flash: shortest policies, one perception route, skills it never reuses. Gemini-3.6-flash writes the shortest policies in the study (median 22 lines on the RoboLab lanes; 153 without a library), edits four times per run, opens an image in 2% of runs, and never runs the simulator locally. SAM3 is its entire perception (39 VAL with it against 2 without), and its adaptation is the prompt: 11 distinct prompts per lane bundle with synonym lists in 95% of them (“grey bin”, “gray bin”, “plastic bin”, “storage bin”, “container”). Its one strong control habit is polling the success predicate to the end (86% of lane bundles): 68% of its logged failures end with the motion complete and the jaws closed on air. It authors skills it mostly calls once, and on RoboLab it has called rig methods that do not exist. Its library gain on RoboLab is –1.1; its gains on the cheaper families (+10 on the course, +13 on LIBERO-PRO) sit on the cells it fails alone. Its no-policy rounds (38%) are delivery failures upstream of robotics.

Failure by task category (RoboLab Vanilla, validation episodes). Relative placement (stacking, spatial relations) scores 1.7 times each model’s mean; tools and utensils score 28% for Opus, 24 for GPT-5.6-sol, and 3 on average for the other three; reorientation scores 0.29 of each model’s mean and thin or handled objects 0.48. Over 21,552 graded episodes the within-cell rank correlation between a layout’s pass rate and the target’s reach, lateral offset, yaw, or clutter averages 0.04 in magnitude. Where a policy logged its closing aperture, 78% of failed episodes closed on air (under 5 mm) against 18% of successful ones. A variance decomposition of cell scores gives 56% to the cell, 22% to the model, and 22% to their interaction. For Kimi-k3 and Gemini-3.6-flash, 54 and 55% of zero-score runs never delivered a policy file.

Verification schools and policy shape. Two verification schools never mix within a model: Opus, Kimi-k3, and DeepSeek-v4-flash test the grasp by gripper aperture after closing (97, 88, and 74%); GPT-5.6-sol and Gemini-3.6-flash test by the environment’s holding flag (92 and 45%). Opus refines from the wrist camera at pre-grasp in 75% of RoboLab lane bundles and Kimi-k3 in 77%, the other three in 22% on average. The library halves the policy and moves logic into imports: an Opus lane policy is 293 lines importing 524 lines of skills, a GPT-5.6-sol one 124 plus 256, a Gemini-3.6-flash one 22 plus 137. A second failure root scales with model strength: open-category selection and multi-object cells sit at 0.83 of the mean for Opus and 0.66 for the others. Success falls monotonically from blocks (81% for Opus) through bananas and bottles to mugs and bowls (55) to tools.

Accumulation mattered more than perturbation. Every model handled LIBERO-PRO’s reworded instructions comparatively well; the harder problem was accumulating useful experience. On the hardest LIBERO-Goal task, Opus scored 0 of 5 with the stock instruction and 2 of 5 after it was rephrased seventeen steps into the LIBERO-PRO curriculum. The drawer, not the wording, was the difficulty, and what differed was the library: 171 entries at the stock run against 262 at the PRO run, including two self-authored skills that did not exist earlier. The comparison is not strictly like for like, since the two scores come from different bands and both runs timed out mid-refactor.

L Library Contents and Reuse

Counts behind Section 6.2. On the first half of the RoboLab curriculum Opus reused 202 distinct skills in 660 imports; the other models reused 26 to 126 skills. Across simulators (MuJoCo Panda to Isaac Sim Franka), 86 Opus imports of 12 skills reached 52% of its RoboLab policies and 73 Kimi-k3 imports of 14 skills reached 42%; GPT-5.6-sol made three such imports and the other two models none. The skills that cross are servo wrappers and grasp idioms, with per-skill lift of +17 to +36 points. Reuse is heavy-tailed: the top tenth of skills takes 40 to 68% of uses, and independent lineages converge on the same concepts under different names. Memories propagate alongside code: 83 to 93% of Opus’s lineage policies carry identifiers from inherited memories that are absent from the matched no-library policies. A code library holds 306 to 322 executable skills per RoboLab lineage, of which a final program imports a median of five; final programs import 16 to 21% of a lineage’s skill modules, and 96 to 133 skills per lineage have been shadowed by newer versions. A VLA library is a handful of process skills and a 180,000-character memory file. Library gains concentrate on cells a model fails alone: on LIBERO-PRO, cells with a Vanilla score below 50% gain +33 and +39 points for the two weakest models against +2 and +4 on cells they already solve, +25 against +3 for GPT-5.6-sol, and +14 against −1 for Kimi-k3. Kimi-k3’s RoboLab library gain grows along the lane from −26 points in its first seven steps to +24 to +35 from step 28 on. The frozen transfer of the Opus lineage and best-of-3 policies to LIBERO-PRO reads 47.7 against 40.7% TEST.

M Auto-Research: Supply, Time, and Task Structure

These tables support Section 7.2. The groups are observational, not randomized interventions. A RoboLab session records 66 to 72 teacher episodes per agent-hour against about 490 on LIBERO, and ends with a median 109 to 122 successful demonstrations per task against 430 to 486. Among the 52 well-supplied RoboLab tasks, the reasoning gap within a placement count (1.2 and 9.4 percentage points) is within the task-to-task spread (permutation $p = 0.57$), while multi-placement demonstrations are roughly three times as long. Robosuite’s precise multi-stage tasks stay hard with ample data: nut assembly scores 3% and two-arm handover 7%, against 100% for cube lifting. One RoboLab session found that the harness’s velocity labels, differenced from teacher commands, replayed to 4.37 rad of arm travel where the teacher had produced 14.80.

Table 18. More demonstrations, higher success. The 105 RoboLab tasks grouped by the number of successful demonstrations recorded by their VLA lineage session, with the VLA’s mean TEST success (%) per bin; the right half keeps only tasks whose teacher passes at least 90% of TEST states.

Successful demonstrations per task	All tasks		Tasks with teacher $\geq 90\%$	
	Tasks	VLA success	Tasks	VLA success
none	9	0.0	—	—
1 to 49	12	20.8	4	40.0
50 to 99	20	41.0	10	60.0
100 to 299	59	58.0	46	63.0
300 or more	5	94.0	5	94.0
all	105	47.2	65	63.5

N Physical Campaign: Practice Behavior in Detail

This section expands Section 8.4.

Probes and scene variation. GPT-6-astra spends 358 of its 597 attempts, 60%, on unscored probes, compared with 8 of 355 for Fable. Opus uses 72 probes in total, 51 of them on Spell SEAR. On that task, GPT-6-astra spends five hours practicing pick and place through the clearing task before attempting the Spell SEAR task; its last seven scored spellings then succeed, three short of the streak. Many of GPT-6-astra’s probes deliberately change the next starting state: with its jaws closed it pushes the empty bowl 20–40 mm to a new position; it sets up a T rotated by 180°; and it performs 48 of its 52 cup resets as probes to place cups farther out. These actions follow an instruction to

Table 19. What additional development time changes in AR. For each session with submissions before and after its first 5 hours on LIBERO or 10 hours on LIBERO-PRO and RoboLab, the best later submission is compared with the best earlier one. Entries give session counts and median in-session VAL gains in percentage points, grouped by the changes recorded in the 882 submissions. On LIBERO-PRO, retraining on the same data yields a median 7.4-point gain, close to the 8.0 points associated with collecting more data; returns are smallest on LIBERO, where scores are near the ceiling.

What changed	LIBERO		LIBERO-PRO		RoboLab	
	Sessions	Median	Sessions	Median	Sessions	Median
Collect more data	2	0.5	2	8.0	11	8.9
Improve inference settings	0	—	0	—	4	4.4
Average multiple model weights	0	—	2	4.2	0	—
Re-train on the same data	4	2.5	4	7.4	6	2.3
Regrade same weights	2	0.0	0	—	3	1.1
All	8	1.5	8	7.1	24	5.0

Table 20. TEST success (%) of the VLA on RoboLab by task structure. (a) The 52 tasks with a teacher at 90% or more and at least 100 successful demonstrations per session, grouped by placements and instruction type; demo length is the median teacher demonstration in simulator steps; a *reasoning* instruction requires inferring the target through an attribute, category, count, order, spatial relation, logical connective, or memory. (b) Five groups that share objects and scenes but differ in the number of placements required; scores average the Vanilla and Lineage AR sessions.

(a) By placements and instruction type

Placements	Instruction	Tasks	Demo length (steps)	Teacher	VLA
one	plain	16	204	96.2	76.9
one	reasoning	22	201	99.1	75.7
two or more	plain	6	585	98.3	40.0
two or more	reasoning	8	551	98.8	30.6

(b) Matched tasks

Matched tasks	One placement			Two or more placements		
	Task	Teacher	VLA	Task	Teacher	VLA
cube, banana, bowl	cube or banana	100.0	75.0	cube and banana	80.0	35.0
				banana then cube	100.0	30.0
				cube then banana	100.0	40.0
white bottles, pail	any white bottle	100.0	85.0	all three small bottles	40.0	0.0
canned food, bin	one can	100.0	100.0	two cans	90.0	20.0
				three cans	90.0	0.0
pumpkins, bin	small pumpkin	100.0	70.0	both pumpkins	100.0	5.0
	big pumpkin	100.0	40.0			
apple, yogurt, pail	throw away the apple	100.0	95.0	put away the snacks	100.0	65.0

succeed across layouts. At least three of its success streaks break on the scored attempt following such a probe.

Reset states versus staged states. In every one of Fable’s practice spellings, its scrambling reset leaves at most one block in the middle of the table and no required letter farther than 0.48 m from an arm base; in evaluation, 8 of 10 scenes have blocks in the middle and 2 have a required letter 0.53–0.58 m away. Only 10 of Opus’s 64 practice Push-T starts require a turn of more than 90°, compared with 6 of the 10 staged scenes. GPT-6-astra’s reset always leaves the T within 175 mm of its outline, whereas the staged starts are 158–269 mm away. Fable’s cup reset leaves every cup with its opening toward an arm. In Fable’s Cup Upright run a dropped cup came to rest facing away from both arms; the agent continued without resolving that configuration, 14 refused resets did not count toward the streak, and it reached the streak with 7.6 of its 12 hours unused.

Live execution. Under agent-as-policy, Fable’s code grows from 202 lines after Banana into Bowl to 3,576 lines in 30 files after Spell SEAR, with at least 15 files issuing robot commands; these scripts account for 99% of its 33,532 practice commands and 98% of its 5,916 evaluation commands, and on Block Stacking and Spell SEAR background loops open and close most practice attempts without a separate model decision. GPT-6-astra issues each of its 7,537 commands through a separate model call, requests about one image per arm motion against Fable’s 0.14, and writes ten small helpers, none of which moves the robot. Fable’s scripted Cup Upright routine closes on the cup’s paper wall in all 18 visible practice grasps and in none of 23 evaluation grasps; its single evaluation success comes after switching to a body-squeeze routine written during practice.

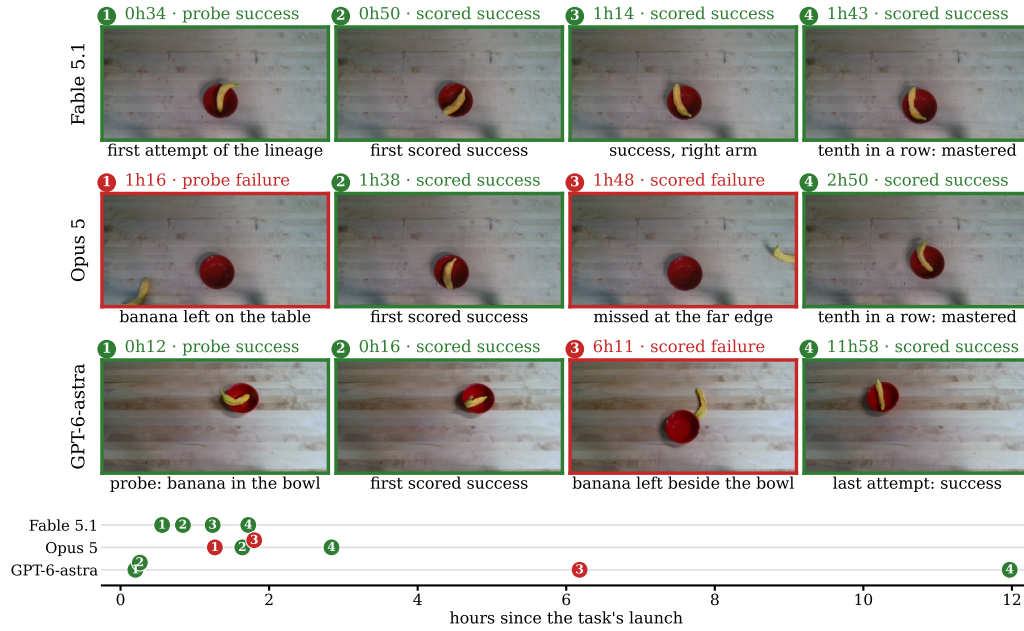
Cost. GPT-6-astra issues its first robot attempt 3 to 8 minutes after each launch, while Fable and Opus first spend 23 to 53 minutes reading and writing code. Per scored forward success, practice costs about \$3.7 in API spend for Fable, \$11 for Opus, and \$18 for GPT-6-astra. For the Cup Upright segment the recorded API costs are about \$96, \$254, and \$368, respectively.

Losing objects. Opus’s Spell SEAR policy knocks the S off the table in the fourth hour, the E in the sixth, and the R in the ninth. From hour 4.4 its own scene description reports that the word cannot be completed because the S is missing. Nevertheless it spends its last 7.5 hours, and 62 of its 101 attempts, continuing the task; its memory records the loss of the S and E, but further code revision cannot replace the missing letters. Practice ends with 1 success in 45 scored spelling attempts, and the frozen policy scores 0 of 10 on the staged scenes. In the third hour of Opus’s Cup Upright run an attempt knocks a lying cup off the table; the next 169 attempts, including the ten-success streak, operate on the remaining two-cup scene.

Block Stacking. The task asks for the tallest tower, and the run’s score is the maximum height reached, but the feedback for each placement is a binary success whenever the column ends taller than it started. Fable reaches 310 mm, about eight cubes, in its third hour, then cycles through building towers of five or six cubes, toppling them, and rebuilding (Figure 13). Placements that increase the height of these shorter towers continue to earn successful verdicts even though the record does not improve. Placement success and progress toward a higher tower must therefore be reported separately; the pattern does not by itself establish that the feedback caused the rebuilding.

O Physical Campaign: Attempt Galleries

Banana into Bowl



Push-T

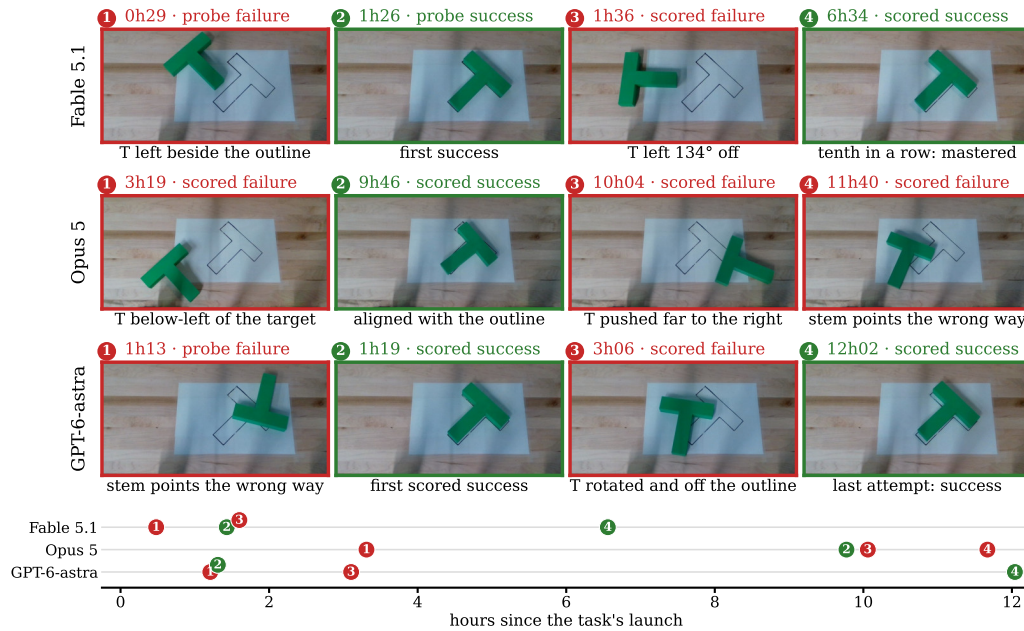
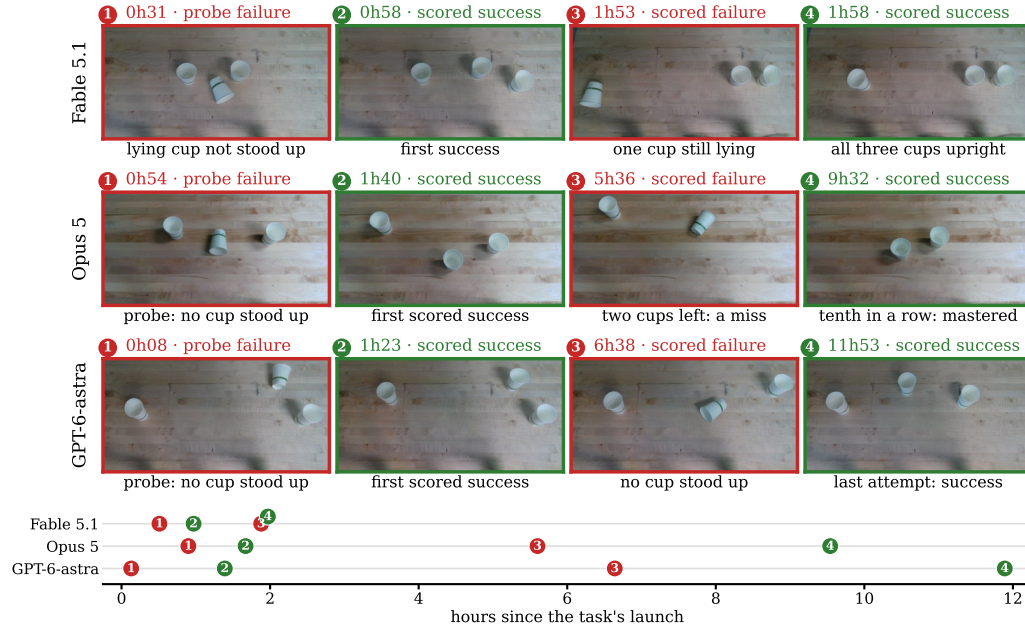


Figure 12. Self-evolution on hardware, every agent on every task (continued in Figures 13 and 14). Each block is one task, with one row per agent and four attempts per row, numbered in session order: the overview camera at the end of the attempt, with the elapsed session time, the attempt kind, and the judge’s verdict (green success, red failure). The axis under each block places the four numbered attempts of each agent at their session times, so the rows read as time lapses of 12-hour sessions. For Opus 5 on Banana into Bowl and Fable 5.1 on the other tasks the attempts are picked to show the story; the other rows generally show the first recorded attempt, the first scored success, a later scored failure, and the last recorded scored attempt. The displayed Opus and Astra Push-T failures, Astra Spell SEAR failure, and Fable Cup Upright examples are selected for visually unambiguous outcomes rather than proximity to a fixed session time. Each lineage starts from an empty library on Banana into Bowl and inherits its own earlier tasks’ workspace thereafter.

Cup Upright



Block Stacking

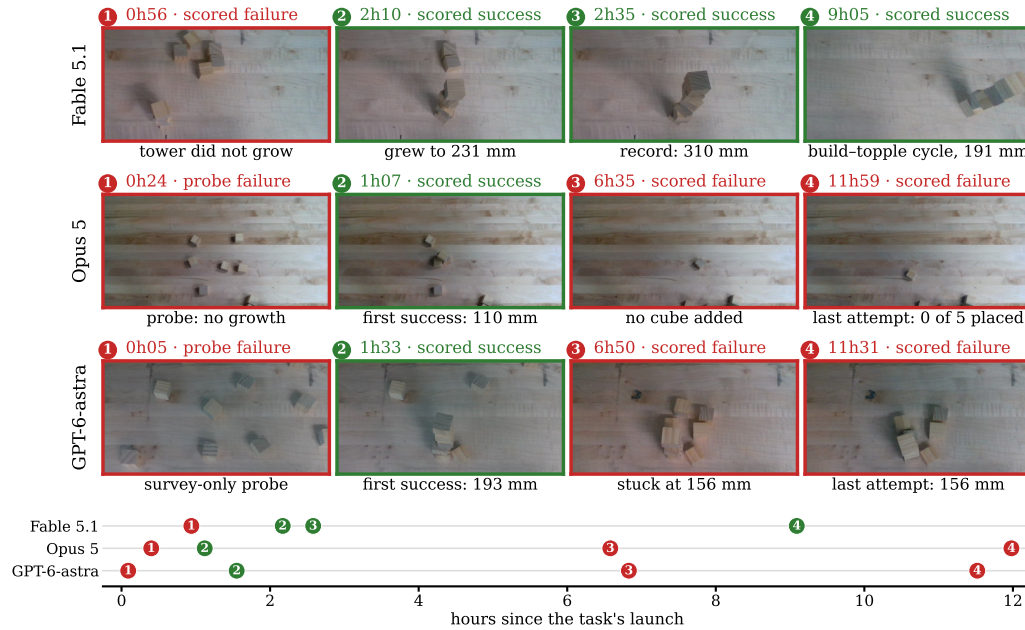


Figure 13. Self-evolution on hardware, continued from Figure 12: Cup Upright and Block Stacking. Opus’s Cup Upright attempts after hour 2.9 practice a two-cup scene, because one cup left the table. On Block Stacking, Fable’s last tile is the build-and-topple cycle it settled into after its 310 mm record; Opus’s last stacking success comes at hour 5.3 and it has only single cubes after hour 6.3, and GPT-6-astra’s tower sits at 156 mm.

Spell SEAR

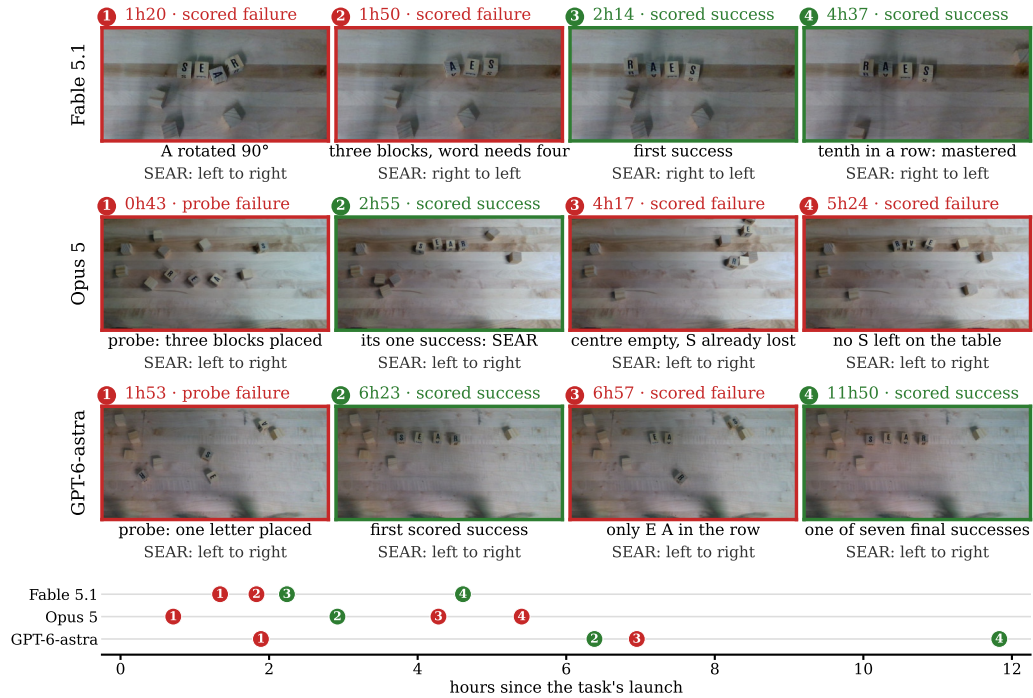


Figure 14. Self-evolution on hardware, continued from Figure 13: Spell SEAR. Each tile states the instructed reading direction for **SEAR** in the displayed camera view. Left-to-right spells **SEAR**; right-to-left appears as **RAES** from camera left to right. Fable tiles 2–4 request **RAES**, while its first tile and all displayed **Opus** and **GPT-6-astra** tiles request **SEAR**; **RAES** is a valid forward spelling target, not a failed **SEAR** arrangement. **Opus**’s one success comes at hour 2.9; by hour 4.3 its **S** has left the table, and its last recorded scored attempt, at hour 5.4, reads a table with no **S**. **GPT-6-astra**’s first success comes at hour 6.4, and its last seven scored spellings all succeed.