SEAR: BENCHMARKING SELF-EVOLVING AGENTIC ROBOTICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Robots deployed in the real world need to improve from their own experience. We introduce **SEAR** (Self-Evolving Agentic Robotics), a framework and benchmark for robotic agents that get better with use, whether by rewriting their code, skills, and memories or by training a vision-language-action (VLA) model. Evaluating self-improvement is harder than evaluating a fixed policy, and **SEAR** addresses five specific problems. First, an agent that tries more often may look like it is learning when it is only sampling more, so we compare against best-of-N baselines with the same budget. Second, getting better at a practiced task is not the same as generalizing, so we test on a five-level ladder: held-out initial states, unseen tasks, a second physics engine, and real hardware. Third, agents can exploit privileged simulator state, so success is scored by a judge the agent cannot access. Fourth, results can depend on the path an agent happens to take, so we run several independent copies of each agent from the same starting point. Fifth, code-writing and model-training agents spend very different resources, so we track compute alongside wall-clock time. We evaluate on five families of simulated tasks and five physical tasks. Inherited skills raise success from 61.9% to 72.3% on the hardest simulated family, yet inherited mistakes can persist across later tasks. A distilled VLA succeeds under camera faults that disable its own teacher. On hardware, the strongest simulated agent finishes last, and switching from frozen code to live execution reverses the ranking of the two models tested in both modes. **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 shapes the community's understanding of this area.

1 INTRODUCTION

A robot deployed in the real world needs to keep getting better. It will meet objects, conditions, and tasks its designers did not anticipate, and every interaction is a chance to improve: a failed grasp reveals a missing constraint, a successful manipulation becomes a reusable skill, a batch of rollouts refines an action model. This raises a central question: *how should we evaluate whether a robotic system is genuinely improving from its own experience?*

Foundation models make this question practical. An agent can reason about observations, call perception and control tools, write executable programs, and revise them after failure; it can also use learned action models such as VLAs (Kim et al., 2024; Black et al., 2025) alongside programmatic control. Experience can persist in several forms: a skill, a memory, a revised policy, or an updated weight. We call these persistent components the *adaptive state* of an agentic robotic system. Like model weights, they encode experience and shape future behavior, even when the language model itself is frozen. Self-evolving agentic robotics is the process of updating this state through interaction and judging each update by the behavior that follows. Recent systems show pieces of this loop (Fu et al., 2026; Zhang et al., 2026; Lu et al., 2026; Xiao et al., 2026; Zhu et al., 2026), but each under its own protocol. What is missing is a common account of what is retained, which components are updated, when task information arrives, and how to tell whether an update actually helped.

We introduce **SEAR** (**S**elf-**E**volving **A**gentic **R**obotics) to provide that account (Figure 1). **SEAR** connects agents, environments, persistent skill and memory stores, and model toolboxes through one harness. Within a session, an agent explores or attempts tasks, observes execution

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

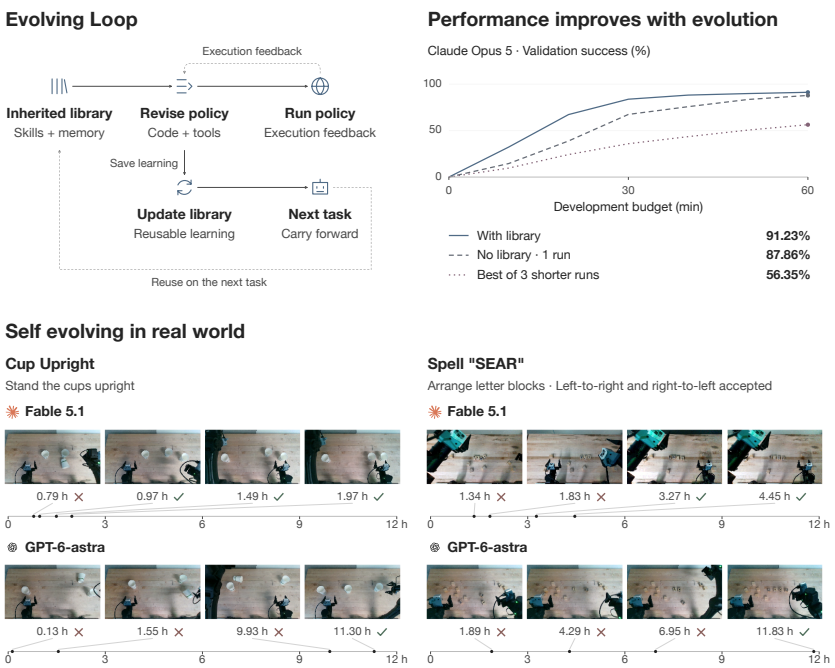

Figure 1: Our goal with **SEAR** is to provide a general framework and benchmark for **self-evolving agentic robotics**. Top: the agent revises its policy using execution feedback and preserves skills and memory for later tasks; validation curves compare inherited experience with strong control baselines (a no-library run and best-of-3 restarts; Section 4.2). Bottom: selected Cup Upright and Spell "SEAR" attempts by Fable 5.1 and GPT-6-astra. Labels report elapsed hours and success or failure; selected frames illustrate trajectories, not success frequencies (Section 4). Experience can improve performance on later tasks, but success in practice does not guarantee success at evaluation.

feedback, and develops policies. Across sessions, it keeps code and knowledge that later agents inherit. The policies it produces also generate rollouts from which an auto-research agent fine-tunes $\pi_{0.5}$ (Black et al., 2025). Evaluating self-improvement is harder than evaluating a fixed policy, and the benchmark (Section 3) addresses five specific problems, each with a dedicated mechanism.

Running frontier coding agents under this protocol shows both the benefits and the limits of getting better through interaction. Given the task instruction, an inherited library improves the Claude Opus 5 code-as-policy agent from 61.9% to 72.3% test success on the hardest simulated family, and it lets every model fix hardware faults that practice without a library does not. A VLA distilled from the agent's own code policies matches them where demonstrations are plentiful and beats them when the instruction wording or the camera changes. Three short attempts recover much of what one long session achieves. Identically configured lineages diverge by more than the gap between models, and in two cases the cause was an agent trusting a measurement instrument it had built itself. On hardware, a mastered practice streak does not predict the held-out score.

Our contributions are as follows.

- **A unified framework** of self-evolving agentic robotics around persistent, improvable components, and a harness that runs code-as-policy, full-loop agent-as-policy, and auto-research agents against the same environments and tools.
- **A benchmark** built around five problems of evaluating self-improvement: budget-matched best-of-N controls, a five-tier ladder from held-out states to hardware, a sealed judge, replicated lineages, and a budget ledger, across five simulated families and five physical tasks.
- **An empirical study** of frontier coding agents under this protocol: a learning-versus-resampling decomposition, code evolution against rollout-driven fine-tuning of $\pi_{0.5}$, and root-caused failures of self-improvement that a mean curve would hide.

2 **SEAR**: SELF-EVOLVING AGENTIC ROBOTICS

In every self-evolving robotics framework, a foundation model is the agent: it sees the environment, decides what to do, and drives the robot through perception and control tools. We call this pattern **agent-as-policy (AaP)** (Zhu et al., 2026; ang Gao et al., 2026) and distinguish three modes by what the agent produces and what runs at test time. In **code-as-policy (CaP)**, the agent writes a program, and that program is the policy. No language model runs at test time, so latency depends only on the program and its primitives; most self-evolving robotics work to date uses CaP (Lu et al., 2026; Xiao et al., 2026; Zhang et al., 2026; Fu et al., 2026). In **auto-research (AR)**, the agent designs and runs training experiments that improve a learned model (Rank et al., 2026); here the learned model is a VLA trained on rollouts from CaP teachers and self-collected data. In **full-loop AaP**, the language model stays in the control loop, and what evolves is the context around it (tools, memory, and skills) rather than its weights. Because the agent itself is the artifact, full-loop AaP leaves no frozen program or checkpoint for an offline judge to re-grade, and its cost grows with every rollout. Our simulation experiments therefore use CaP and AR, whose frozen artifacts a sealed judge can re-grade offline; the physical experiments also evaluate full-loop AaP (Section 4.5). Every agent starts with the same toolbox: SAM3 (Carion et al., 2026) and a Qwen vision-language model for perception, inverse-kinematics and servo interfaces for control, and the ability to write its own tools.

Four settings. **SEAR** runs the loop in four settings (Section 3). *Vanilla* gives the agent an instruction and a judge and starts every task from an empty library. *Lineage* adds the library from all previous tasks. *Self-exploration* withholds both the instruction and the judge: the agent picks its own practice and gets the instruction only afterward. *Real world* runs the same loop on hardware, where scenes do not reset and there is no simulator state to grade from. Vanilla and Lineage fill tiers T1 and T2 of the generalization ladder (Section 3.5), self-exploration T3 and T4, and the real world T5.

3 **SEAR** BENCHMARK DESIGN

We implement the framework as a benchmark spanning simulated and physical robots, decoupling the policy mode (CaP or AR) from the setting in which experience is acquired and evaluated, so that different approaches are measured under one protocol. Evaluating a self-improving agent is harder than evaluating a fixed policy, and we address five specific problems.

- **More attempts can look like learning.** The fraction of problems solved by at least one sample grows with the number of samples (Brown et al., 2024), so every learning curve is paired with a best-of-N baseline at the same budget (Section 3.3).
- **Practice can look like generalization.** Held-out seeds still test the same task, so we grade frozen artifacts across five tiers: from unseen initial conditions to unseen tasks, a second physics engine, and the physical world (Section 3.5).
- **Privileged information can look like capability.** Simulation exposes ground-truth poses, task identifiers, reset logic, and success predicates that no physical robot has, so success is scored by a judge the agent cannot access (Section 3.2).
- **One lucky run can look like reliability.** Self-improvement is path-dependent, so every simulated configuration is run three times independently and we report the spread.
- **A larger budget can look like a better algorithm.** Code search and policy training spend very different resources, so we match wall-clock time and record what each run consumed.

3.1 ENVIRONMENT FAMILIES

SEAR has five simulated families and one physical platform (Figure 2), ordered easiest to hardest. **Robosuite** (Zhu et al., 2020), on MuJoCo (Todorov et al., 2012): seven tasks covering lifting, stacking, assembly, and bimanual coordination. **LIBERO** (Liu et al., 2023): 40 language-conditioned tasks in four suites on the same engine. **LIBERO-PRO** (Zhou et al., 2025): the LIBERO tasks perturbed along four axes (reworded instructions, unseen objects, swapped layouts, and changed goals), giving 160 tasks that double as a contamination control. **LIBERO-RIG** is a hardware-robustness benchmark: it keeps every LIBERO task fixed and perturbs only the rig along eleven axes — camera

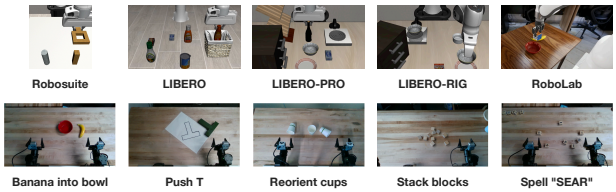

Figure 2: **SEAR** encompasses: (top) five simulated benchmark families, (bottom) five physical tasks.

mounting, calibration, actuation, proprioception, base and table pose, and gripper pads — each at five severities (Section P). **RoboLab** (Yang et al., 2026), on Isaac Sim 5.0 (NVIDIA): a different physics engine and primitive API with 120 cluttered, randomized scenes, 105 of which have 50 generated initial states. The physical platform (Section 3.6) has five tabletop tasks with varied initial poses (Section M): *Banana into Bowl*; *Push-T* (Xiao et al., 2026), pushing a T-shaped block to match a target outline in position and orientation; *Cup Upright*, standing a lying cup without knocking over upright ones; *Block Stacking*, building a free-standing tower of identical cubes as tall as possible; and *Spell SEAR*, arranging letter blocks among distractors to spell **SEAR** in the instructed direction.

3.2 Sealed Judge and Frozen Artifacts

The agent works in its own environment, where it can inspect failures, run simulations, install packages, write programs, update memory, and train checkpoints. When the budget runs out or the agent submits, everything it produced is frozen into an *artifact*: code, skills, memory, configuration, and, for AR, learned weights. Only this artifact crosses into the judge, which holds the held-out initial states, the success predicates, the privileged simulator state, and the grading logic. The judge rolls out the artifact and scores success by the environment's own predicate, regardless of what the policy claims; the same boundary makes programs and checkpoints directly comparable.

3.3 Lineages and Matched Controls

Controls and selection. Vanilla receives the same task instruction, budget, and feedback as Lineage, but no inherited library or instructions for using it. Best-of-3 divides the budget among three independent no-library attempts and keeps the one with the highest VAL score. **DEV** seeds support local development, **VAL** provides judge-graded feedback, and **TEST** is reserved for final evaluation (Section B). For each run, we select the highest-VAL artifact from agent submissions and host snapshots, then evaluate it on TEST. In Lineage, this artifact also supplies new library entries. TEST never guides selection.

Budgets and replication. Development lasts one hour per Robosuite, LIBERO, or LIBERO-PRO task and two per RoboLab task, with up to eight submissions plus periodic host grades. Self-exploration gets two hours per scene, followed by one policy rewrite after the task instruction is revealed. AR receives the same total development time as CaP per family: seven hours across Robosuite, ten per LIBERO suite, forty per LIBERO-PRO axis, and 17.5 per RoboLab suite; AR exploration gets six hours per three-scene group or ten per RoboLab scene. Compute use can differ, particularly when AR trains a model, so we record resource use separately. We repeat each simulated configuration independently three times and report the mean and sample standard deviation. These repetitions are separate from the attempts within best-of-3.

3.4 Self-Exploration and Auto-Research

Self-exploration removes both the instruction and the judge. The agent inherits an eligible library, picks its own practice, and checks outcomes with its own tools; when the simulator interface changes, a library-adaptation phase first rewrites the inherited code. Exploration may use the evaluation configurations, so the generalization axis is what the agent does not know. Afterward, the instruction is revealed and the agent writes one task-specific policy without further simulator access. The instruction-disclosure ablation compares this policy with the exploration policy frozen before disclosure, which gets the instruction only as a run-time input. In the AR setting, learned CaP policies act as teachers that generate rollout data, and the agent runs training experiments on $\pi_{0.5}$, op-

tionally refining the teacher. AR is evaluated separately from the CaP campaigns so that gains from updated weights stay distinguishable from gains from inherited skills and memories (Section 4.3).

3.5 Generalization Tiers

Every frozen artifact is graded at one of five tiers, each testing one way the world can differ from practice. Because families are ordered by difficulty, a row of the tier matrix (Figure 3) never decreases in tier from left to right. **T1, new initial states**: the same family with held-out initial conditions. **T2, new conditions**: the same tasks under perturbation, LIBERO-PRO in scene and phrasing, LIBERO-RIG in hardware, with the row family's frozen policies re-graded without further practice. **T3, new tasks**: unseen tasks on the same simulator with the lineage library inherited. A task-specific code policy cannot solve an unseen task without adaptation, so T3 uses self-exploration; for AR the VLA is trained during that exploration. **T4, new simulator**: a different physics engine and primitive API, following the T3 protocol after library adaptation. **T5, the real world**: the full loop on hardware, where scenes do not reset and there is no simulator state to grade from.

3.6 Real-World Harness and Self-Evolution

The real-world harness runs on two YAM-Ultra arms with parallel-jaw grippers, a fixed overview camera, and two wrist cameras. Agents see images, depth, and proprioception through a toolbox that mirrors the simulated one (Section J), and each attempt produces an immutable policy snapshot, recorded observations, and an evaluator verdict. Three coding agents revise executable policy bundles with fixed model weights across the five tasks. Each lineage starts from an empty library with no seed policy or simulation reference; later tasks inherit the preceding bundle and library. Each task gets 12 hours of wall clock and stops early after ten consecutive forward successes confirmed by evidence. A fixed external evaluator judges before-and-after images and recordings independently of the agent's own perception, marking ambiguous evidence as unknown. Final performance is the frozen bundle run once on each of ten operator-staged scenes per task, with the evaluator's verdicts confirmed by the operator.

Self-evolution through forward and inverse tasks. Objects stay where each attempt leaves them, so the agent must develop reset behaviors alongside the forward task. The pairs are: putting the banana into the bowl and taking it out; aligning the T and moving it away; standing a cup upright and laying it on its side; building the tower higher and toppling it; and spelling "**SEAR**" and scrambling the blocks. The inverse task prepares a valid new starting configuration, not a reversal of the forward motion or a restoration of exact poses.

The agent observes the current scene, chooses the applicable direction, runs its policy, and uses the recorded observations and external verdict to revise its code and persistent knowledge. Both directions get evaluation feedback. The next attempt starts from whatever physical state the last one left, so forward and inverse tasks need not alternate strictly after failures or incomplete resets. Inverse attempts use the same practice budget but do not count toward the forward-only mastery streak; diagnostic probes are also excluded, and Block Stacking has no early stop (Table 14). The agent develops both task execution and the reset behaviors that shape its practice, without updating the language model's weights.

4 Findings

Agents, harness, and metrics. In this section, we investigate how frontier coding agents improve a robot through interaction, not only which model scores highest. We consider the following models in simulations: Claude Opus 5 through Claude Code, GPT-5.6-sol through Codex, and Gemini-3.6-flash, Kimi-k3, and DeepSeek-v4-flash (text-only) through OpenCode. On hardware we run Claude Fable 5.1 and Opus 5 through Claude Code and GPT-6-astra through Codex. To evaluate agents on the five generalization tiers (T1-5), we have four different harness settings to meet the generalization requirements, as described in Section 3. *Lineage* runs each model through the task sequence with its library, next to a Vanilla run with no library and a best-of-3 run, and fills T1 and T2. The *self-exploration* campaign starts task-blind lineages from libraries built on earlier families and fills T3 and T4. The *AR* campaign trains one VLA per family, with and without an inherited library. Every

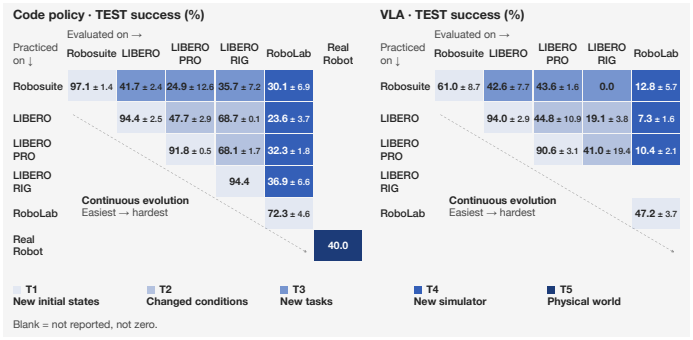

Figure 3: TEST success (%) of Claude Opus 5 by tier for CaP and AR. Rows: practice family; columns: grading family, easiest to hardest. Diagonal is T1; cells to its right test transfer (T2–T4). T1–T2: Lineage (instruction given). T3–T4: self-exploration from the row family's library. T2 AR: zero-shot. T5: physical lineage. Mean ± SD over three runs.

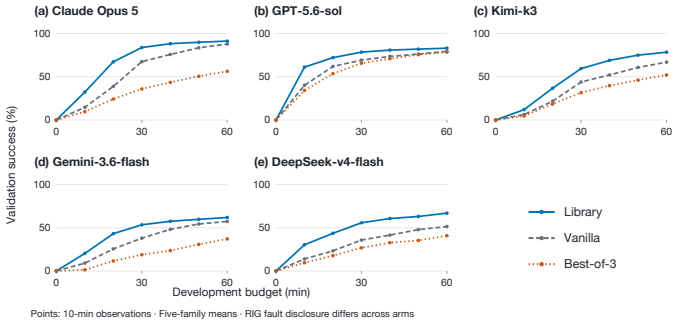

Figure 4: Validation success against self-evolve time for five models averaged over all tasks, one panel per model on shared axes. Solid: with the inherited library. Dashed: the same run with no library. Dotted: best-of-3, three no-library attempts at a third of the budget each, which costs the same total budget. RoboLab's 120-minute budget is rescaled to the 60-minute axis and read on the first 56 cells of its curriculum.

simulated arm is run three times. The primary metric is TEST success rate over ten rollouts per task, reported as mean and sample standard deviation across the three runs.

4.1 Libraries transfer gains on hard tasks but also transfer errors

With the instruction given, the library adds a few points for the strongest models and ten to eighteen for the weakest, with the largest gains on hard tasks. As shown in Table 2, on Robosuite, LIBERO, and LIBERO-PRO, Opus and GPT-5.6-sol lineage only slightly improve their no-library controls, while Gemini-3.6-flash and DeepSeek-v4-flash gain 7 to 18 points, concentrated on cells they cannot solve without a library. When one hour is enough to solve a task from scratch, the library reduces time to solution rather than raising the success ceiling as shown in Figure 4. RoboLab is harder with larger improvements: $+8.7\%$ points for Opus and $+11.9\%$ for Kimi-k3 between lineage and no-library control. The library primarily increases the success rate on borderline cells and reduces time to first success. With the library, every model scores 19 to 64 points above its no-library best-of-3 on LIBERO-RIG (Section O), and 7 points (Opus) to 40 points (DeepSeek-v4-flash) above a matched no-library run that was given the identity of the injected fault (Table 2).

Without the instruction, lineage performance correlates with how much the donor library has seen. In self-exploration (T3/T4), each lineage starts from a donor library evolved on an earlier family. Each library accumulate skills from the previous family so the later family will have a larger library. RoboLab uses a simulator, robot, and primitive API that none of the donors encountered. As shown in Figure 3, Opus lineages reach the highest accuracy with the largest library (LIBERO-RIG), although the smallest library (Robosuite) does not end with the lowest accuracy. A task-blind policy can succeed on never-explored scenes due to the agent learn to parse the task description into shared

routines: the same parsing logic and the same routines solve both the pumpkin the lineage never saw and the banana it practiced on as shown in Figure 6.

Libraries transfer usable knowledge across tasks and simulators, and they transfer errors the same way. A library only helps on a new task if the skills stored in it still apply there. We observe that inherited control skills are always imported into the agents RoboLab policies. When a task matched what the library already knew, the transfer results in 100% success in the new task. However, the same mechanism spreads mistakes. When one session measures a rig property incorrectly and saves it to memory as fact, every later session inherits the wrong value, and nothing in the loop unlearns it.

4.2 Resampling: three short attempts versus one long session

Some of what a lineage gains is available to any agent that simply tries more often (Brown et al., 2024). Every family was therefore run under three arms with the same per-task budget and identical grading: *Lineage*, one full-budget session per task with the library carried forward; *Vanilla*, the same session with no library; and *best-of-3*, three independent no-library attempts at a third of the budget each, keeping the best by VAL (Figure 1, Figure 4; scoring rule and every model's arms in Section D). The lineage's margin over equal-budget resampling then splits into two terms, $\text{Lineage} - \text{BO3} = (\text{Lineage} - \text{Vanilla}) + (\text{Vanilla} - \text{BO3})$: a library term, and a session term that compares one long session against three short ones. **The session term tracks time to first success.** On Robosuite, one hour beats three 20-minute attempts for every model. On RoboLab, the sign depends on the model. GPT-5.6-sol reaches its first VAL success at a median of 44 minutes and Gemini-3.6-flash at 55, so three 40-minute attempts beat one two-hour run, by 1.4 and 2.6 points. Opus needs 81 minutes, DeepSeek-v4-flash 88, and Kimi-k3 110, so splitting the budget costs them 4.8, 8.7, and 23.8 points. Resampling pays only for the models that succeed early.

4.3 VLA performance varies with training data and evaluation conditions

To compare CaP execution with learning from CaP demonstrations, an Opus agent trains one $\pi_{0.5}$ checkpoint per suite using CaP teachers per-task (Section I). The VLA lineage scores 94.0% and 90.6% on LIBERO and LIBERO-PRO, close to CaP's 94.4% and 91.8%, and equals or exceeds CaP success on four-fifths of tasks (Table 8). On RoboLab, it scores 47.2%, compared with 72.3% for CaP. In self-exploration, VLA success on held-out LIBERO and LIBERO-PRO tasks is 42.6% and 43.6%, compared with 41.7% and 24.9% for CaP (Figure 3). Under LIBERO-RIG camera faults, the VLA scores 57% to 90% and CaP 7% to 11%; under a changed control rate, VLA success falls to 21.7%. CaP has no corresponding control-rate measurement because its harness uses the nominal rate (Table 11). To investigate the lower VLA success on RoboLab, we group tasks by demonstration count and structure. Mean success is 20.8% with 1–49 successful demonstrations and 94.0% with at least 300 (Table 9). Among 52 tasks with strong teachers and abundant data, VLA success for one placement is 76.9% with plain instructions and 75.7% with instructions requiring reasoning. For multiple placements, success falls to 40.0% and 30.6%, respectively. These results show that VLA heavily relies on sufficient training data. When inheriting libraries, VLA mostly records how to train the model and prepare its training data, while CaP mostly describes how the robot grasps objects and makes contact with them (Table 13).

4.4 Performance varies across repeated lineages

We compare repeated runs to measure how much scores vary and then inspect why some runs perform poorly. Three repetitions of one Opus self-exploration lineage scored 15.5%, 4.6%, and 23.6% on LIBERO-PRO. By comparison, the Opus and GPT- 5.6-sol mean scores on LIBERO and LIBERO-PRO differ by less than one standard deviation. Differences between runs can therefore exceed differences between models. On RoboLab, three runs initially scored 26.7%, 34.1%, and 0.5%. The lowest-scoring run did not detect that a gripper offset had been applied along the wrong axis of the tool frame. After rerunning with the corrected frame, its score was within the range of the other two runs. In the original run, the agent also misinterpreted the gripper-opening measurement and wrote a grasp check that reported success when the gripper was empty. The low-scoring LIBERO-PRO run used 16 episodes to set up its placement check. In subsequent rollouts, 91% of the outcomes marked as successful were failures. This check also determined when to stop, so 637 rollouts ended without another grasp attempt. The other two runs followed their action plans

	CaP			AaP	
Task	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

Table 1: Physical-task evaluation scores (%). Non-stacking scores are mean outcomes (success $= 1$, partial $= 0.5$, failure $= 0$) over 10 evaluation scenes per CaP task and 5 per AaP task. Block Stacking reports the mean episode maximum cube count, capped at 10, as a percentage ($n = 10$ for CaP and $n = 5$ for AaP). Overall is the unweighted mean of the five tasks.

without these checks and scored three to five times higher. Both low-scoring runs saved incorrect interpretations in memory and continued using them in later sessions.

4.5 Physical performance depends on both practice and execution

Three CaP lineages, Fable 5.1, Opus 5, and GPT-6-astra, each start from an empty library and practice the five tasks in order, 12 hours per task, inheriting the previous task's workspace. Non-stacking tasks stop at *mastery*: ten consecutive scored forward successes with both arms represented. Frozen bundles are then evaluated on operator-staged scenes (Table 1; protocol and practice statistics in Section L). Fable scores 70.2%, GPT-6-astra 41.4%, and Opus 35.6%. Fable masters all four non-stacking tasks in 29.5 hours, Opus two, and GPT-6-astra none. Opus leads every simulated family and is last here.

Mastery does not predict the staged score. Mastered segments average 67% on staged scenes and budget-exhausted segments 38%, but only Fable's Push-T succeeds on all ten scenes; the other five mastered segments succeed on 4 to 8. Practice starts from the states the agent's own inverse-task resets leave behind (Section 3.6): The mastery rule can also stop practice early: Fable's Cup Upright run ends with 7.6 of its 12 hours unused and a cup facing away from both arms never handled; its frozen policy later refuses all three evaluation scenes it fails without attempting a motion.

Practice success depends on the scenes agents create. GPT-6-astra uses 60% of its attempts for unscored probes, often changing layouts and sometimes breaking success streaks. It never achieves ten consecutive practice successes, yet its frozen policies score higher than Opus's on operator-staged scenes (41.4% vs. 35.6%). Object loss can make practice impossible or easier: Opus continues Spell SEAR for 7.5 hours after losing a required letter, but completes its Cup Upright success streak after a cup falls off the table, leaving only two.

Evaluation failures stop at limits the agents wrote. We audited 116 failed or partial evaluation units, mixing episodes and placements (Section N). None involved a physically impossible scene. In 29 of 34 Fable, 32 of 35 Opus, and 26 of 47 GPT-6-astra units, the stop came from a limit, rule, or motion the agent chose; most of the rest were harness refusals of poses those policies selected. Fable rejects distant letter blocks that the robot could reach with a tilted gripper. In Spell SEAR, conflicting rules in Opus's code assign a required letter block to its less accurately calibrated arm, so it never completes the word in evaluation. GPT-6-astra's Cup Upright policy and Opus's Push-T policy stop without trying alternative motions, despite having substantial execution budget left.

Agent-as-policy reverses the ranking. In agent-as-policy (AaP), the model observes and chooses actions during evaluation. Across all five tasks, GPT-6-astra scores 88.0%, up from 41.4% in CaP, while Fable scores 66.4%, compared with 70.2% in CaP (Table 1). Interestingly, Fable still issues 98% of evaluation commands through scripts, whereas GPT-6-astra observes between motions and corrects mistakes in at least 9 of 25 episodes. GPT-6-astra also takes two to three times as long on Push-T, Block Stacking, and Spell SEAR. For example, on Cup Upright, GPT-6-astra's frozen policy stops when a turn is refused; in AaP, it instead turns the cup on the table and re-grips it.

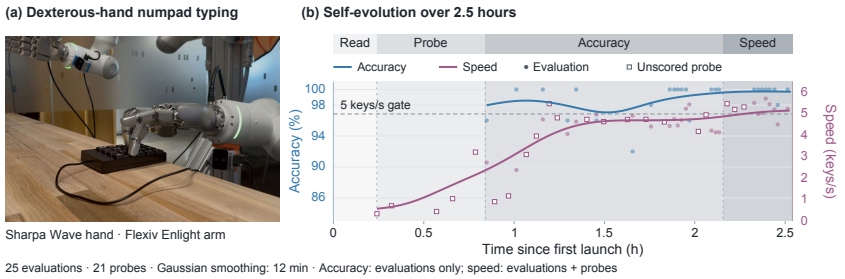

Figure 5: Generalization to Dexterous Hand (Numpad typing task). Curves are 12-minute Gaussian smooths of the plotted points. Shading marks the phases the agent named in its journals.

4.6 The loop generalizes to a dexterous hand: numpad typing

To test the same loop on a different morphology, a Fable 5.1 agent ran on a 22-joint Sharpa Wave dexterous hand on a Flexiv Enlight arm, with a library inherited from earlier experiments on this hand. The task is to type a string of random digits on the number pad of a physical keyboard (Figure 5). A frozen evaluator (Section 3.2) scores each attempt on accuracy and speed, and the run completes when accuracy over the latest ten evaluations is at least 90% and speed is at least 5 keys per second. Figure 5b shows the run. The agent first reads the hand's SDK, then maps key positions with unscored probes, then keeps every scored evaluation at 92% accuracy or higher, and finally raises speed past 5 keys per second within 2.5 hours of launch by moving several fingers in coordination across a keystroke sequence. After the run, the frozen policy typed the first 100 digits of π three times at 5.1 keys per second with one wrong digit in 300. Numpad data-entry jobs typically require 2.2 to 2.8 keys per second.

5 Related Work

Self-evolving LLM agents improve without weight updates by writing reflections into memory (Shinn et al., 2023), rewriting their own improvement procedure (Zelikman et al., 2024), or evolving the harness around a frozen model (Weng, 2026; Lin et al., 2026a), and Lin et al. (2026b) show that much of that apparent self-evolution is resampling. In robotics, Code-as-Policies (Liang et al., 2023) and ProgPrompt (Singh et al., 2023) write control from language, Voyager (Wang et al., 2023) accumulates it in a skill library, Eureka (Ma et al., 2024) and RoboGen (Wang et al., 2024) have the model write rewards and scenes, and a wave of self-improving manipulation systems followed (Fu et al., 2026; Lu et al., 2026; Zhang et al., 2026; Xiao et al., 2026; Waddle Labs, 2026), each evaluated under its own protocol. Benchmarks for autonomous research agents (Chan et al., 2024; Wijk et al., 2025; Starace et al., 2025; Rank et al., 2026; Zhu et al., 2026) grade engineering, replication, and post-training under controlled budgets and isolation. **SEAR** is not a new self-improvement method but a fixed judge and protocol under which such methods can be compared: it brings the budget-matched control of Lin et al. (2026b) and the isolation of these benchmarks to a loop that closes through physics, and adds the generalization ladder, frozen-artifact transfer across simulators, and a ledger that compares an agent that spends tokens with one that spends GPU-hours.

6 Conclusion

As robotic agents begin to participate in their own improvement, the benchmark has to stay fixed while they change. It has to separate learning from sampling, practice from transfer, real progress from luck, and finishing the task from gaming the score. **SEAR** is our attempt to make recursive improvement in robotics measurable across those five confounds. It does so with a sealed judge, a tiered ladder from held-out states to physical hardware, replicated lineages, and budget-matched controls, applied to code-writing and model-training agents alike. Under this protocol, inherited experience raised a code-as-policy agent from 61.9% to 72.3% on the hardest simulated family, a distilled VLA outscored the collapsed teacher it learned from, and identically specified lineages diverged by more than the gap between models. We will release all benchmark artifacts so that the results can be reproduced, extended, and challenged. Section R discusses what the benchmark does not cover, the agent architectures to evaluate next, and the limitations of this study.

AI DISCLOSURE

In this work, we used generative AI tools to edit research paper text to improve readability and conciseness. All AI-assisted edits were reviewed and revised by the authors. We did not use generative AI tools to generate synthetic data, develop theoretical or conceptual frameworks, formulate mathematical claims, design research methodology or experiments, implement methods, or interpret results; these tasks were performed entirely by the authors. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

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.

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.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *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.

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.

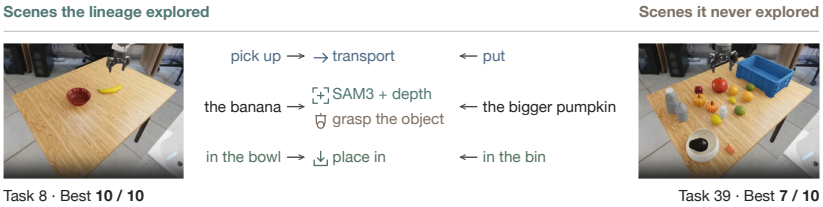

Figure 6: How a frozen task-general policy from a task-blind Opus lineage on RoboLab (Section H) executes an instruction on an explored scene (left) and an unexplored scene (right). Both instructions parse to the same program and invoke the same routines, so the unexplored scene is solved to the extent that those routines generalize.

A Additional Results

	Code-as-policy			Auto-research VLA	
Family	Lineage	Vanilla	Best-of-3	Baseline	Lineage
Robosuite	97.1 ± 1.4	93.8 ± 7.0	90.0	61.0 ± 8.7	61.0 ± 8.7
LIBERO	94.4 ± 2.5	95.2 ± 1.1	73.8*	94.0 ± 3.6	94.0 ± 2.9
LIBERO-PRO	91.8 ± 0.5	93.0 ± 1.8	67.2	92.4 ± 4.9	90.6 ± 3.1
LIBERO-RIG	94.4	87.2 ± 5.5‖	30.8§	—	—
RoboLab	72.3 ± 4.6	61.9 ± 1.0‡	57.1‡	43.1 ± 9.3	47.2 ± 3.7
real robot	40.0¶	—	—	—	—

Table 2: T1 results for Claude Opus 5, TEST success (%) on held-out initial states of the practiced family, mean ± SD over three replicates where replicated. *Lineage* carries the library forward, *Vanilla* is the matched no-library control, *Best-of-3* spends the same budget as three independent attempts and keeps the best by VAL, read after each run ends (Figure 4 reads the same runs at 60 agent-minutes on VAL, 20.6 there against 21.1 here). ‖Told the injected fault, unlike the other arms. §TEST graded for 12 of 18 cells; 21.1 on validation over all 18. *On the 47-task LIBERO course, where the lineage scores 94.8. ‡Ran after a tool-frame fix that the lineage lanes predate. ¶One physical lineage, the four placement tasks on ten staged scenes each (Table 1).

			LIBERO-RIG		
Model	LIBERO course (T1)	LIBERO-PRO (T1)	library, serial	library, parallel	no library
Claude Opus 5	94.8 ± 2.0	91.8 ± 0.5	94.4	99.4	30.8‡
GPT-5.6-sol	87.8 ± 2.8	88.4 ± 2.8	90.0	91.8‡	71.1
Gemini-3.6-flash	81.7 ± 4.2	76.2 ± 1.2	70.0‡	71.8‡	35.6‡
Kimi-k3	79.3 ± 1.3	80.5 ± 3.5	97.1	92.2	57.2
DeepSeek-v4-flash	67.6 ± 8.6	75.5 ± 4.5	74.4	66.1	31.1

Table 3: Lineage campaign TEST success (%) for every model. LIBERO course: 47 Robosuite and LIBERO tasks, mean ± SD over three lanes; LIBERO-PRO: 160 tasks, three lanes (two for Gemini); LIBERO-RIG: 18 measurable fault cells at the moderate level, 60 minutes per cell, either as one serial lineage that carries the library from cell to cell, or in parallel with every cell seeded from a fixed inherited library and nothing carried between cells (the serial lane's starting library for Opus, the Robosuite and LIBERO library for the other models), or as best-of-3 over 20-minute attempts with no library. Each cell is read after its run ends. Opus's serial cell counts one unloadable artifact as zero (100.0 over 17). ‡Partial: 12, 17, 13, 17, and 9 cells, reading across the rows.

Practiced on	Arm	Robosuite	LIBERO	LIBERO-PRO	LIBERO-RIG	RoboLab
Robosuite	CaP	97.1 ± 1.4	41.7 ± 2.4	24.9 ± 12.6	35.7 ± 7.2	30.1 ± 6.9
	AR	61.0 ± 8.7	42.6 ± 7.7	43.6 ± 1.6	0.0	12.8 ± 5.7
LIBERO	CaP		94.4 ± 2.5	47.7 ± 2.9	68.7 ± 0.1	23.6 ± 3.7
	AR		94.0 ± 2.9	44.8 ± 10.9	19.1 ± 3.8	7.3 ± 1.6
LIBERO-PRO	CaP			91.8 ± 0.5	68.1 ± 1.7	32.3 ± 1.8
	AR			90.6 ± 3.1	41.0 ± 19.4	10.4 ± 2.1
LIBERO-RIG	CaP				94.4	36.9 ± 6.6[§]
	AR				—	—
RoboLab	CaP					72.3 ± 4.6
	AR					47.2 ± 3.7

Table 4: The cells of Figure 3 as numbers: TEST success (%) of the Claude Opus 5 code-as-policy (CaP) and auto-research (AR) artifacts, rows the family practiced in and columns the family graded on, mean ± 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 1 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→LIBERO) 26.9±7.6 (Robosuite→LIBERO-PRO) 14.6±9.5 (Robosuite→LIBERO-RIG) 24.3±11.4 (Robosuite→RoboLab) 25.8±3.8 (LIBERO→RoboLab) 19.1±5.8 (LIBERO-PRO→RoboLab) 26.7±2.5 (LIBERO-RIG→RoboLab) 28.6±4.8. [§]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 (authored 34.5, 44.3, and 31.8, the last over the 115 of 120 tasks graded); the same cell read 26.6 ± 22.8 with the blind arm's 0.5.

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

Table 5: 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 ± 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), and the arms that had found the offset themselves keep their full-course reading. A fully errored cell scores zero. Lineage by lineage, the authored policy beats the frozen one on 21 of 21 Opus arms (median gain 7.8 points), 21 of 21 GPT-5.6-sol arms (10.4), 21 of 21 Kimi-k3 arms (7.1), and 7 of 21 Gemini-3.6-flash arms (0.0), so the value of one task-specific rewrite scales with the model's ability to write a working policy at all.

Model	Arm	Robosuite	LIBERO	LIBERO-PRO	LIBERO-RIG	RoboLab	Mean
Claude Opus 5	with library	97.6	95.2	92.5	94.4	76.3	91.2
	no library	93.8	95.7	93.2	90.0	66.6	87.9
	best-of-3	87.1	62.5	49.3	20.6	62.2	56.4
GPT-5.6-sol	with library	85.7	90.5	90.5	92.2	56.3	83.0
	no library	91.9	90.5	86.1	71.1	57.2	79.4
	best-of-3	90.0	94.5	73.7	72.0	61.9	78.4
Kimi-k3	with library	89.5	77.8	80.7	93.9	50.1	78.4
	no library	76.7	78.2	79.0	62.6	37.9	66.9
	best-of-3	57.1	67.8	71.4	55.6	8.6	52.1
Gemini-3.6-flash	with library	80.0	82.0	76.8	45.0	25.6	61.9
	no library	70.5	70.6	63.4	56.1	26.0	57.3
	best-of-3	71.4	37.5	30.5	18.3	27.7	37.1
DeepSeek-v4-flash	with library	80.5	67.2	77.5	73.9	35.4	66.9
	no library	74.3	59.2	58.3	35.7	29.7	51.4
	best-of-3	64.3	46.0	47.4	28.9	17.5	40.8

Table 6: 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 no-library runs were told the injected fault; the other two arms were not. RoboLab is the corrected-frame campaign read on the first 56 cells of its curriculum, the prefix every library lane has finished, all three arms on the same cells.

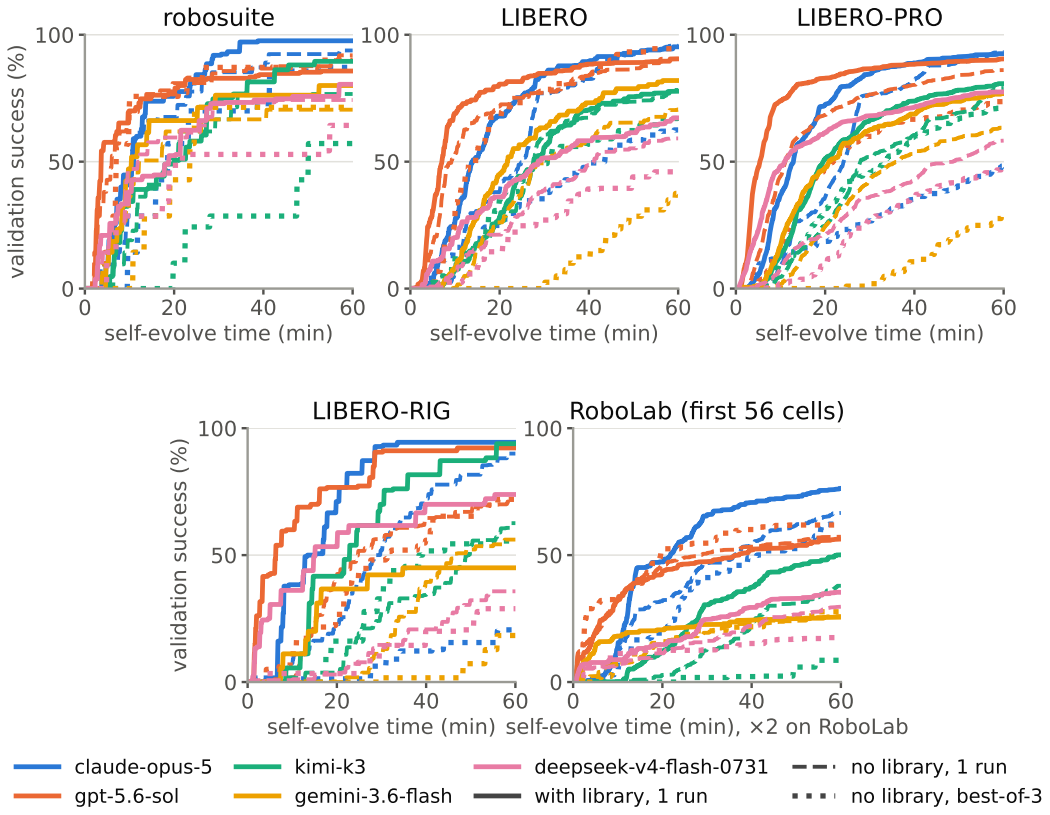

Figure 7: The three arms of Figure 4 broken out by family, one panel each, every model in its colour: solid with the inherited library, dashed the matched no-library single run, dotted the no-library best-of-3. Same axis as Figure 4: time is when the submitted code was last written, RoboLab's 120 minutes are halved, and RoboLab is read on the first 56 cells of its curriculum.

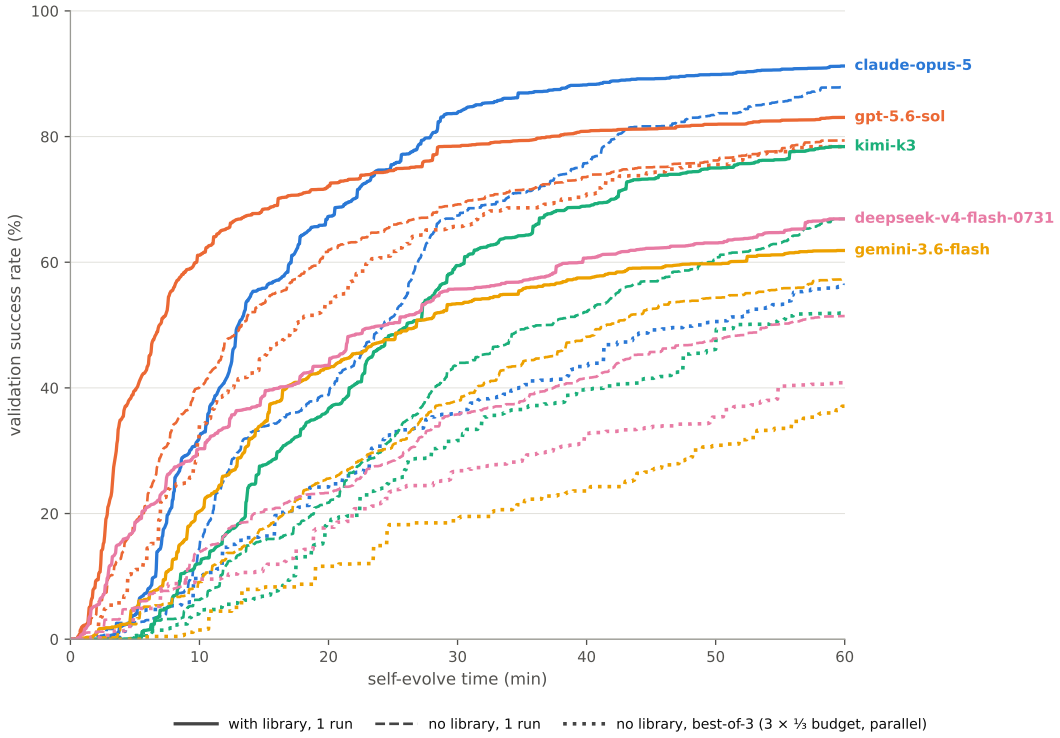

Figure 8: The fifteen curves of Figure 4 in one panel, for direct comparison across models; model names sit at the right end of each library line.

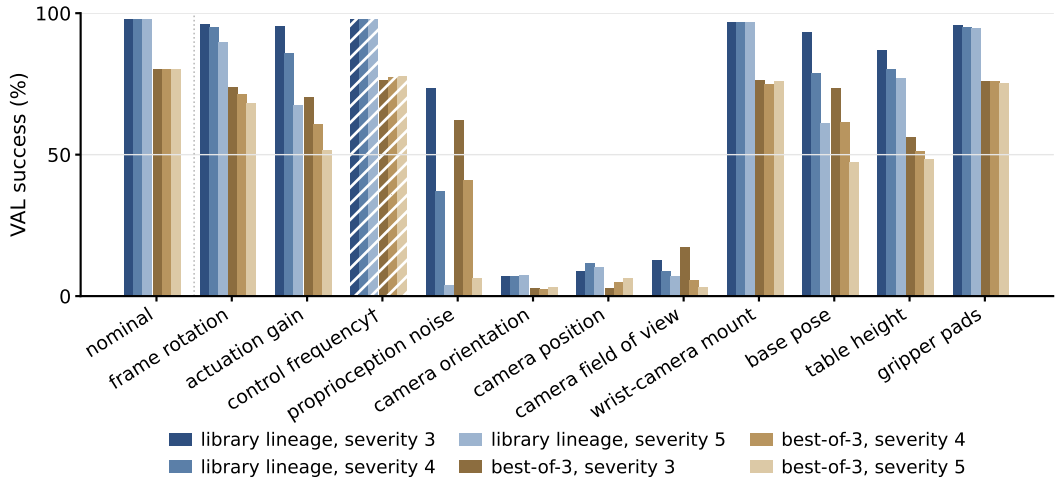

Figure 9: The LIBERO-practiced Opus artifacts re-graded on LIBERO-RIG by perturbation axis and severity, raw VAL success averaged over tasks: the library lineage (60-minute sessions, three lanes, all 40 tasks) against the no-library best-of-3 winners (three 20-minute attempts, re-graded on the same 40 tasks). Complete grades only. Control-frequency bars are hatched: the harness does not apply that axis, so they equal the nominal rig. Table 7 lists the values.

	Library lineage				No-library best-of-3		
Axis	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 7: Figure 9 as numbers: the LIBERO-practiced Opus artifacts 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.

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 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 documented 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 runs receive the same total budget as code-as-policy for each family, one checkpoint per suite: seven hours across Robosuite's two suites, ten hours per LIBERO suite, forty per LIBERO-PRO axis, and 17.5 per RoboLab suite. In self-exploration they receive six hours per three-scene group, or ten hours per RoboLab scene; the T3/T4 AR cells of Figure 3 train on a scene curriculum of twelve scenes on LIBERO and LIBERO-PRO and 47 on RoboLab.

D RESAMPLING DECOMPOSITION

This appendix gives the scoring rule, the per-task regimes in which each gain is earned, and the across-run definitions behind Section 4.2.

How a run is scored. Every run, a session or an attempt, is graded repeatedly as it works: automatic snapshots every 5 minutes, agent-initiated submissions, and a final grade of whatever is on disk when the budget expires. Each grade rolls out the policy from 10 held-out validation initial states that the agent never sees. Because those rounds are developmental snapshots of a single improving policy, a run's score is its *best valid round*, the best policy that run ever produced. This within-run rule is applied identically in every arm. VAL grades are visible to a run and so available for selection; TEST is a disjoint seed band graded once after development ends, which no arm can select on.

Every model's arms. The Opus settings are in Table 2 and the lineage for every model in Table 3. For GPT-5.6-sol, Kimi-k3, Gemini-3.6-flash, and DeepSeek-v4-flash, Vanilla is 89.0, 76.2, 70.5, and 70.0 TEST on Robosuite and 47.1, 30.2, 18.6, and 23.4 on RoboLab, and best-of-3 is 85.7, 55.7, 70.0, and 61.4 on Robosuite and 48.5, 6.4, 21.1, and 14.8 on RoboLab. On RoboLab the lineage enters as Vanilla plus the cell-matched per-step gain (70.6 for Opus), because the lanes cover the first half of the curriculum and the controls all 105 cells. The oracle best of three full-budget Vanilla repetitions scores 5 TEST points above a single repetition on Robosuite and 17 on RoboLab (15 when the repetition is chosen by VAL).

Two readings of the same runs, and the held-out band. The runs can be scored at a fixed compute budget or after they end. The learning-curve panel of Figure 1 takes the first reading: the budget is counted in agent-minutes, each best-of-3 attempt receives a third of it, and only rounds graded before the cutoff count. Its arms are those of Figure 4, whose Robosuite attempts are the export of 2026-09-22 (parity arm, corrected tool frame) and whose LIBERO attempts are the best-of-3 campaign of August; for Opus at 60 minutes these give 0.956, 0.662, and 0.401 on VAL for the learner,

best-of-3, and single attempt. Table 2 and Section 4.2 take the second reading, including the grade taken when each run's budget expires, and report the private TEST band.

What TEST adds is measured on the campaign report's version of the same course, which grades every attempt on both bands but uses an earlier set of Robosuite attempts (August, uncorrected tool frame; the two sets differ by one lucky attempt each on nut assembly and two-arm handover, worth 15.7 points of the Robosuite best-of-3 mean). There, at 60 minutes, the arms read 0.956, 0.636, 0.365 on VAL and 0.941, 0.602, 0.342 on TEST, so selecting the winner on VAL costs it only 3.4 points between the bands against the learner's 1.5. Run to completion the learner's lanes do not move (0.940, 0.960, 0.968 on VAL), but the attempts keep climbing on the band they can see, best-of-3 to 0.766 and the single attempt to 0.552. On the LIBERO-PRO course best-of-3 reaches 0.450 on VAL at the cutoff and 0.698 at completion, the widest gap, because its attempts are furthest from converged at the cutoff.

Three per-task regimes.

1. Where all three attempts succeed, $\max = \mathrm{mean}$, so both gains are zero: easy tasks contribute nothing to either term.
2. The *resampling gain* (best-of-3 minus single attempt) is earned exactly on tasks where attempt outcomes vary, some attempts succeed and others fail, so selection keeps the winner while the expectation is dragged down by the failures.
3. The *learning gain* is earned on tasks where no 20-minute attempt succeeds but the 60-minute, library-carrying session does.

Across-run summaries. Writing $s_i(t)$ for the score of the i-th attempt on task t, and $\mathcal{T}$ for the set of tasks:

$$\mathrm{BO3}(t) = \max\big(s_1(t),\, s_2(t),\, s_3(t)\big), \tag{1}$$

$$\mathrm{SA}(t) = \mathrm{mean}\big(s_1(t),\, s_2(t),\, s_3(t)\big) \qquad \text{(unbiased estimate of } \mathbb{E}[\text{one try}]), \tag{2}$$

$$\mathrm{RL}(t) = \text{3-lane mean of the best valid grade in 60 min}, \tag{3}$$

$$\bar{S}(a) = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} a(t), \qquad a \in \{\mathrm{RL},\, \mathrm{BO3},\, \mathrm{SA}\}. \tag{4}$$

E Model Portraits

This section expands the one-line portraits of **??**. Every model received the same tasks, tools, budgets, and skill-library machinery, each under its own coding harness. The three OpenCode models trade places by family: Gemini-3.6-flash is third on the course, Kimi-k3 third on LIBERO-PRO and RoboLab, and DeepSeek-v4-flash, as a text-only model, is last on the course but ahead of Gemini-3.6-flash on RoboLab under every protocol.

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, 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), 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 4.2).

Kimi-k3: explores well, ships rarely, learns along the lane. Kimi-k3 is the strongest task-blind explorer in the self-exploration campaign and adapts to hardware faults nearly as well as Opus, while trailing on the stock course. 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. Its exploration transcripts also quoted hidden goal text read out of the environment (Section G).

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, and it trails Gemini-3.6-flash on classic LIBERO and Robosuite.

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.

Inkling-small: below the accumulation threshold. A sixth model, Inkling-small, completed the Robosuite and LIBERO lanes at 25.2 ± 3.3 and 6.4 ± 2.2% TEST and was withdrawn partway through LIBERO-PRO; one lane went 29 consecutive tasks without a valid policy. It wrote memories in about half of its runs but almost never a code skill: after 47 steps its three libraries held 0, 6, and 2 skills. This exposes a capability floor: a model that rarely produces a working policy has nothing to record, so self-improvement never begins. It is excluded from every table.

Task-blind, the ranking is not the lineage ranking. Table 5 gives every model's self-exploration cells. Once the instruction is disclosed, Kimi-k3 matches or beats Opus on five of the seven cells, 43.7 against 41.7% on LIBERO, 26.7 against 24.9 on LIBERO-PRO, and 30.9, 30.0, and 36.5 against 30.1, 23.6, and 32.3 on RoboLab from the Robosuite, LIBERO, and LIBERO-PRO donors, and its frozen exploration policy is the strongest on LIBERO, 38.0 against 26.9. Where it falls behind is the hardware-fault course and the donor that carries it: 27.7 against 35.7 on LIBERO-RIG, and 12.9 ± 15.8 against 36.9 from the LIBERO-RIG donor on RoboLab, where two of its three lineages finished under 10. GPT-5.6-sol trails both on every cell but one (24.8 from the LIBERO donor on RoboLab, against Opus's 23.6) and has the widest spreads, 47, 10, and 12% across its three LIBERO-RIG lineages. Gemini-3.6-flash stays near 10% on every MuJoCo cell and at 0 to 4 on RoboLab even with the tool frame corrected: its twelve rerun arms solved 11 of 1,429 graded cells, three of them crashing their own policy on every cell, so its zeros were never only the frame. Two readings follow. The donor-order trend of Section 4.1 is Opus's: 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 the worst or next to worst, so what the hardware-fault course adds to a library depends on the model that later inherits it. And at three replicates most of these gaps sit inside the spread; Section 4.4 says which survive it.

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.

F What the Policies Are, and Where They Fail

Every graded policy shares a skeleton: capture one RGB-D frame from the over-shoulder camera, ground the target and destination (an open-set detector for four models, a table-plane fit with colour or height blobs for Opus), lift pixels to world coordinates through calibrated depth, optionally refine from the wrist camera at pre-grasp (Opus in 75% of RoboLab lane bundles, Kimi-k3 in 77%, the other three in 22% on average), then run approach, descend, close, verify, lift, carry and release as a straight servo sequence inside a retry loop (85% of bundles; no model produces a state machine). 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%). 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. The learned action model shipped in the toolbox (a $\pi_{0.5}$ VLA) went almost unused: 2% of no-library RoboLab policies call it. Scripted perception and servoing beat the VLA for every model on every family.

Failure has one dominant physical form. We reconstructed where failures happen from four sources: score structure across cells, policies' own progress prints in the judge log, a capability tagging of the 105 RoboLab cells, and agent transcripts (all on the RoboLab Vanilla arm, validation episodes). Difficulty is a property of the cell before the model: a variance decomposition 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 solved only by Opus, five by nobody). The start layout is not what fails: 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, and agents never blame held-out layouts. What fails is the grasp. Where a policy logged its closing aperture, 78% of failed episodes closed on air (under 5 mm) against 18% of successful ones. The capability ordering inverts a natural guess about 3-D understanding: families requiring relative placement (stacking, spatial relations) are the *easiest* at 1.7 times each model's mean, while tools and utensils (Opus 28%, GPT-5.6-sol 24, the other three 3 on average), reorientation (0.29 of each model's mean), and thin or handled objects (0.48) are the hardest. Success falls monotonically from blocks (81% for Opus) through bananas and bottles to mugs and bowls (55) to tools. Height, table plane, and relative placement are solved by everyone; grasp synthesis for a non-box shape, a rim, a handle, or a thin shaft is what the descend-to-the-mask-

top-and-close recipe cannot do. 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. Upstream of all this, for Kimi-k3 and Gemini-3.6-flash more than half of their zero-score runs (54 and 55%) never delivered a policy file to the judge, while every Opus and GPT-5.6-sol zero is a robotics failure.

G SHORTCUTS THE AGENTS FOUND

The seal of Section 3.2 protected runtime state but not static task files, and agents found shortcuts we did not design. Opus inspected object assets in 34 runs, extracting the exact geometry of objects it was about to manipulate, legal but impossible for a real robot. Kimi-k3's exploration transcripts quoted hidden goal sentences read out of the environment, and because later rewrite prompts render prior transcripts the leak propagated down the lineage, so we flag those lineages. When a success predicate was left reachable in one arm, 48% of its final policies simply called it (Bondarenko et al., 2025). Withholding the predicate is therefore deliberate, at the cost the collapses of Section 4.4 make visible: every lineage that wants a feedback loop must first build its own success detector.

H UNSEEN-SCENE PROBE

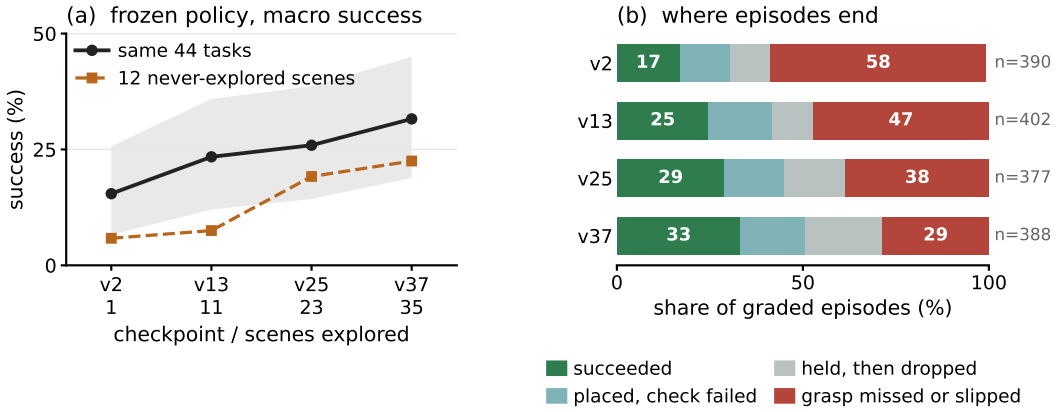

Figure 10: One task-blind Opus lineage on RoboLab (LIBERO-RIG donor, after the tool-frame fix), with its task-general policy frozen at bundles v2, v13, v25, and v37, after 1, 11, 23, and 35 of its 47 scenes, and graded on the same 47 probe tasks, one per scene, ten episodes each on the sealed judge. Nothing is re-authored per task. (a) Macro success on the 44 tasks graded at all four checkpoints, with a 95% bootstrap interval over tasks, and on the twelve scenes no checkpoint explored. (b) The furthest stage each graded episode reached, classified from its own log and pooled over tasks, so the success share differs slightly from the macro. The probe uses the judge's fixed band, one layout per task replayed by all ten episodes, rather than the salted TEST layouts of Section B.

The self-exploration cells of Figure 3 see a task-blind lineage only through the policy it authors once the instruction is disclosed. They cannot separate the donor from the model, and because exploration may visit the scenes it is later graded on, they do not measure unseen scenes either. A single lineage can measure both what exploration adds on top of the donor and how far the lineage's own program reaches past the scenes it explored, with no per-task rewrite in between. One Opus lineage on RoboLab, seeded from the LIBERO-RIG donor and run after the tool-frame fix, explored its 47 scenes goal-free, one per session in curriculum order. Its task-general `policy.py`, which reads the instruction at run time, grounds the named objects with SAM3, and drives the arm, was frozen at four bundles and graded on one task per scene, chosen greedily to cover every success-predicate class (Figure 10). A task counts as unseen at a checkpoint when its scene lies past the checkpoint's curriculum rank.

On the 44 tasks graded at all four checkpoints, success rises from 15.5 to 23.4, 25.9, and 31.6%, a paired gain of 16.1 points (95% bootstrap interval over tasks, 7.0 to 25.9). On the twelve scenes no checkpoint explored it rises from 5.8 to 22.5% (interval of the gain 1.7 to 33.3). That is forward

transfer, but two stacking tasks and one pick-and-place task carry it, and the other nine stay at or near zero. Nor does exploring a scene reliably fix its task: of 33 tasks graded both before and after their scene was explored, 26 did not change, and the gap between explored and unexplored scenes has an interval that includes zero at v13, v25, and v37 alike. At v25, 23 tasks on each side, the policy scores 28.7% on scenes it explored and 21.7% on scenes it did not.

What transfers is a shared program, not a solution per task. The frozen policy is one program per checkpoint (Figure 6). It parses each instruction into a symbolic goal (a verb template, target noun phrases with colour, size, and plural selectors, and a source or destination relation), grounds the phrases with SAM3 and depth, and executes with routines every task shares: body or rim grasp with a lift check, placement into, onto, or stacked on the destination with a scene re-check, a queue for plural targets, and a $\pi_{0.5}$ fallback when nothing grounds. "Put the bigger pumpkin in the bin", in a scene the lineage never explored, invokes the same body grasp and container placement as "Pick up the banana and place it in the bowl" from session 8; it rises from 0 to 7 / 10 as those routines improve on explored scenes. The parser is shared the same way: "take the mugs off the shelf", one noun phrase at v2, is parsed as a source relation by v37, and the same fix reads "take the keyboard out of the bin". But a correct parse buys nothing when the routine behind it has never worked. At v37, the five never-explored tasks built only from parts that succeed on some explored task average 52%; the seven that need a part never shown to work (removal, reorientation, ring grasping, or an unmatched template) average 1%. The split is absent at v13 and rests on five versus seven tasks, so we read it as a tendency. The policy reaches exactly as far as its parts, and exploration widens that reach by improving parts rather than by adding scenes.

Improvements are concentrated in grasping. Episodes ending with the jaws closing on air or the object slipping fall from 58% to 29% between v2 and v37, while failures to ground the named object stay below 1%. As grasping improves, failures shift downstream: placements that fail verification or drop the object rise from 24% to 38%. Several capabilities remain absent throughout: removing objects from containers or shelves, reorienting objects, and placing objects on shelves stay at or near zero regardless of exploration status, and 26 of the 44 tasks never succeed. Two limitations qualify these findings. First, the analysis follows a single lineage, and Section 4.4 shows substantial divergence among identically specified lineages, so the reported intervals capture variation across tasks, not lineages. Second, each probe cell repeats a single fixed layout ten times, measuring reliability on that layout rather than across the TEST layouts of Table 2.

I AUTO-RESEARCH DETAILS

This section gives the setup behind Section 4.3 and the evidence its paragraphs summarize.

Setup. Each AR session trains one $\pi_{0.5}$ checkpoint for a suite from that suite's per-task CaP teachers, within the total budget CaP receives for the same tasks: seven hours across Robosuite's two suites, ten per LIBERO suite, forty per LIBERO-PRO axis, and 17.5 per RoboLab suite. Robosuite has two suites, LIBERO and LIBERO-PRO four each, and RoboLab twelve. The baseline arm starts every suite from the teachers alone. The lineage arm also inherits the previous suite's skills and memory, but never checkpoints or datasets, so only the research process crosses a suite boundary; on Robosuite the two arms share their first session. The agent chooses which rollouts to collect and distill, how much data to gather, and what to change between iterations, and may refine a teacher. Every final checkpoint is graded on TEST (Table 2).

Learning curves. Figure 11 lays each arm's sessions end to end. On LIBERO and LIBERO-PRO both paradigms are paced by the curriculum, since tasks not yet reached count as zero, so the curves climb together and end within a point of each other. On RoboLab the VLA tracks CaP for the first quarter of the budget, then falls behind and ends about 20 points below the CaP lineage on validation. On Robosuite the VLA leads early, because one checkpoint scores the whole suite at once, and then plateaus near 60 while CaP keeps solving tasks.

Task by task comparison between CaP and VLA. Table 8 compares the two lineages task by task. On LIBERO and LIBERO-PRO, VLAs match or beat their teacher CaPs on 80% and 79% of tasks. While VLAs pass all ten TEST states on 75% and 70% tasks, CaPs do in 85% and 81%. Further analysis reveals that CaP's failures concentrate in a few tasks while the VLA's are spread as

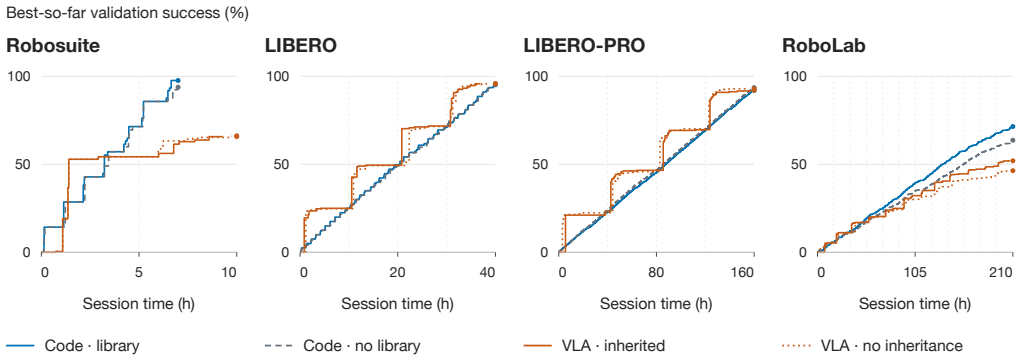

Figure 11: CaP (blue) against VLA self-evolve (red) over session time, with (solid) and without (dashed) inherited state: best-so-far in-session VAL success averaged over a family's tasks, sessions laid end to end in curriculum order. Both paradigms spend the same budget per family. Dotted lines mark VLA session boundaries; CaP and Robosuite VLA curves average three trials. RoboLab's dashed CaP lane is the corrected-frame control of Table 2. Session time includes GPU training for the VLA.

occasional misses across many tasks. On Robosuite and RoboLab, VLAs match CaPs on only 14% and 38% of tasks, and passes all 10 TEST states in 29% and 14% tasks, while CaPs do in 86% and 46%.

		Mean TEST success			All ten TEST states solved	
Family	Tasks	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	72.3	47.2	38%	46%	14%

Table 8: Performance analysis between CaP lineages and VLA lineages. *VLA $\geq$ CaP* is the percentage of tasks on which VLAs' TEST scores are no worse than CaPs'. *All ten TEST states solved* is the percentage of tasks that pass every TEST state. On RoboLab, the mean is the T1 cell of Table 2, while the per-task CaP columns use the lineages run before the tool-frame fix, because the corrected-frame lineages have not yet finished all 105 tasks.

Demonstration supply. Table 9 bins the 105 RoboLab tasks by the number of successful demonstrations their VLA lineage session recorded. Success rises with every bin, and it rises the same way among tasks whose teacher passes at least 90% of TEST states. This shows that the number of successful demonstrations is the game changer for VLA auto research.

How does extra time help. Table 10 compares each VLA auto research session's best checkpoint over the whole session with the best checkpoint within its first 5, 10, or 10 hours on LIBERO, LIBERO-PRO, or RoboLab, respectively, grouped by what changed between the two. On RoboLab, where demonstrations are scarce, collecting more data gained the most, a median 8.9% over 11 sessions, while retraining on the same data improved only 2.3%. On LIBERO-PRO, longer training on the same data gained nearly as much as new data, 7.4% against 8.0%, and on LIBERO, where the VLA is already near its ceiling, extra time bought little either way. Regrading unchanged weights moved the score by at most 1.1%, indicating limited randomness in grading. Improving inference-settings and averaging model weights also help.

Fault tolerance. Table 11 re-grades CaP and VLA checkpoints developed from LIBERO sessions on LIBERO-RIG at severity 3. The two break on opposite faults. CaP lifts detections into world coordinates through an assumed camera, so it collapses when the camera moves (Section O); the VLA maps pixels to actions directly and loses only part of its score. A changed control rate leaves

	All tasks		Tasks with teacher $\geq 90\%$	
Successful demonstrations per task	Num tasks	VLA success	Num 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

Table 9: More demonstrations, higher success. We classify the 105 RoboLab tasks into different bins by the number of successful task rollouts their VLA session recorded, with the number of tasks and the VLA's mean TEST success (%) per bin; the right half keeps only tasks whose teacher passes at least 90% of TEST states.

	LIBERO		LIBERO-PRO		RoboLab	
What changed	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 10: How does extra time help auto research. For each session with submissions both before and after its first 5 (LIBERO), 10 (LIBERO-PRO), or 10 (RoboLab) hours, the in-session VAL gain (%) of its best later submission over its best earlier one (*Median* over sessions), grouped by what changed between the two according to the sealed manifests of all 882 submissions.

CaP's re-grounding servo intact but breaks the VLA, whose actions are tied to the rate it was trained at.

Task structure. We tag each RoboLab task along two axes. *Placements* counts how many objects the success predicate requires placing. *Reasoning* marks instructions that require inferring the target through an attribute, category, count, order, spatial relation, logical connective, or memory. In contrast to *Reasoning*, instructions that name the target directly are *plain*. Table 12(a) restricts to the 52 tasks whose teacher passes at least 90% of TEST states and that received at least 100 successful demonstrations per session, so neither teacher quality nor data supply can explain the performance drop. Adding a second placement roughly triples the demonstration length, from about 200 to 550 or more simulator steps, and VLA success falls accordingly. Within each placement tier, tasks requiring reasoning perform slightly worse than plain instructions: the gap is 1.2% with one placement and 9.4% with two or more. Table 12(b) shows five matched groups that share a scene and objects but differ in placement count; in every group the multi-placement tasks score 30–85% lower, even though its teacher reaches 80–100% in four of the five.

What the agents record. Both CaP and VLA agents maintain a library memory across suites, but the topics they write about differ sharply. Table 13 classifies each library entry by topic using keyword patterns (an entry can match several topics). We compare library entries rather than transcript narration because both paradigms record libraries in the same structured format, whereas CaP transcripts are far more terse. CaP agents focus on the physical world: roughly two thirds of their entries concern grasp and contact mechanics, and about half discuss perception. VLA agents focus on the training pipeline: over a third of entries address the training recipe, a fifth to a third concern the training data and labels, and a sixth to a fifth discuss grading noise or inference settings.

	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

Table 11: Evaluation of CaPs and VLAs developed from LIBERO sessions on LIBERO-RIG TEST states at severity level 3. CaP is the Opus lineage of Table 7. The CaP harness builds the rig at the nominal 20 Hz, so CaP has no reading on control frequency.

(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

	One placement			Two or more placements		
Matched tasks	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

Table 12: VLA on RoboLab by task structure, TEST success (%). (a) The 52 tasks with a teacher at 90% or more and at least 100 successful demonstrations, by objects placed and instruction type; demo length is the median teacher demonstration in simulator steps. Within each placement group, plain and reasoning instructions differ by 1.2% with one placement and 9.4% with two or more. (b) Five matched groups of tasks that use the same objects and scene but differ in how many objects must be placed. Within each group the tasks share the same objects and scene, so the main difference is how many placements the success predicate requires. Placements come from each task's success predicate, instruction types from hand tags.

	CaP		VLA	
Share of memory entries touching…	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%

Table 13: What agents write down in CaP and VLA sessions.

J 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.

K 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 inverse 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. Section L 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.

L Physical Campaign: Detailed Practice and Evaluation

Reading the detailed tables. Tables 14 and 15 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 mastery: 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, inverse 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. Where the run archive prices a segment, Cup Upright cost about $96 in API spend for Fable, $254 for Opus, and $368 for GPT-6-astra; per scored forward success, practice cost about $3.7 for Fable, $11 for Opus, and $18 for GPT-6-astra. GPT-6-astra issued its first robot attempt 3 to 8 minutes after each launch, while Fable and Opus first spent 23 to 53 minutes reading and writing code.

	Fable 5.1		Opus 5		GPT-6-astra	
Task, in lineage order	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: CaP practice and evaluation details. Scores are out of ten. Practice entries give time to mastery (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.

	Fable 5.1		GPT-6-astra	
Task, in lineage order	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**

Table 15: AaP practice and evaluation details. Scores are out of ten. Practice entries give time to mastery (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.

Successful block placements need not produce a taller record tower. Block Stacking exposes a further limitation of practice success rates. The task asks the agent to build the tallest tower it can, and the practice instruction defines the run's score as the maximum height reached, but the feedback for each block-placement attempt is a binary success whenever the column ends taller than it started. Fable reaches 310 mm, about eight cubes, in its third hour. It subsequently 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, so its scored success rate rises while the towers it builds remain below its earlier best. 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 behavior.

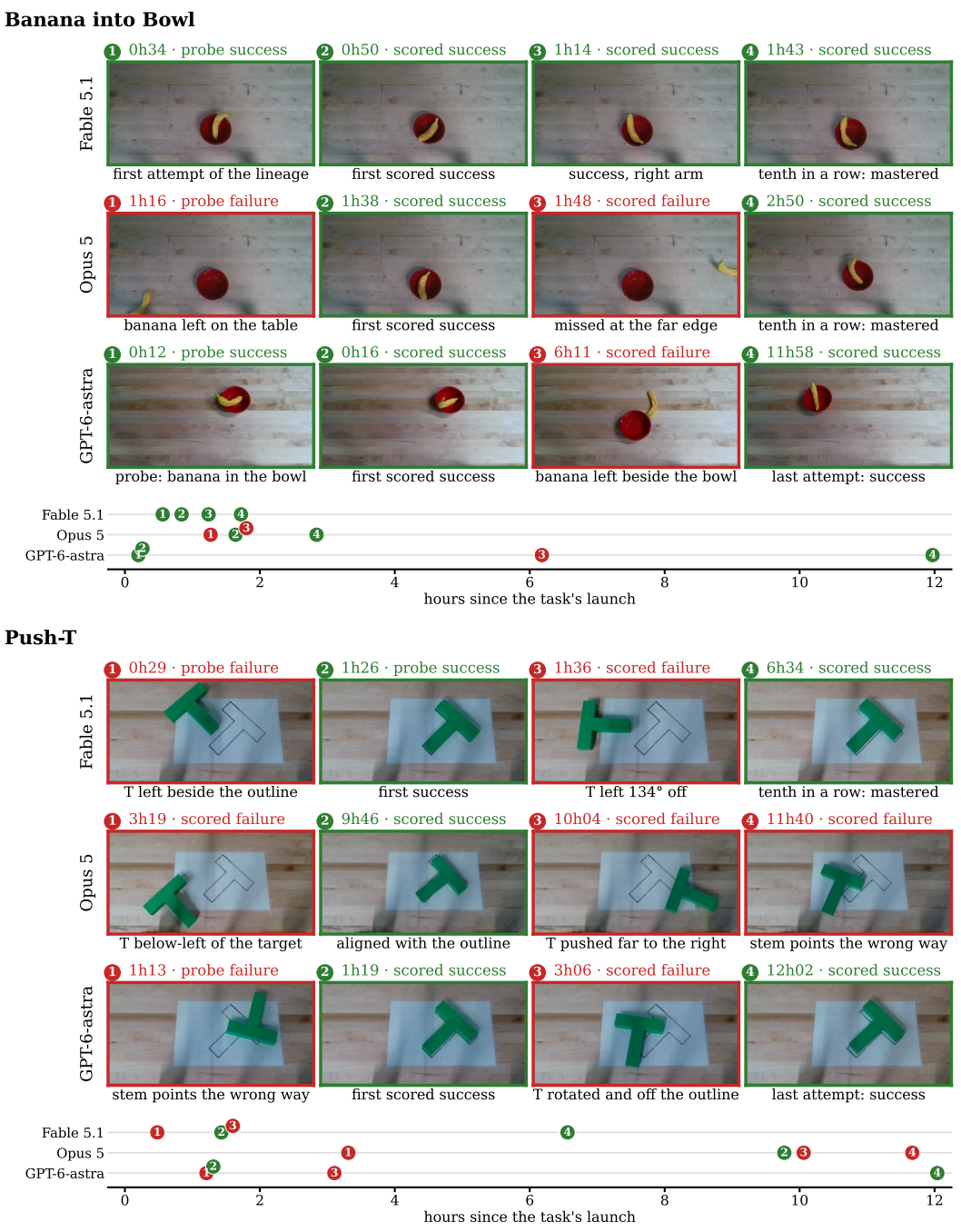

Figure 12: Self-evolution on hardware, every agent on every task (continued in Figures 13 and 14). Each block is one task, with one row per agent and four attempts per row, numbered in session order: the overview camera at the end of the attempt, with the elapsed session time, the attempt kind, and the judge's verdict (green success, red failure); unverified outcomes are excluded. 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.

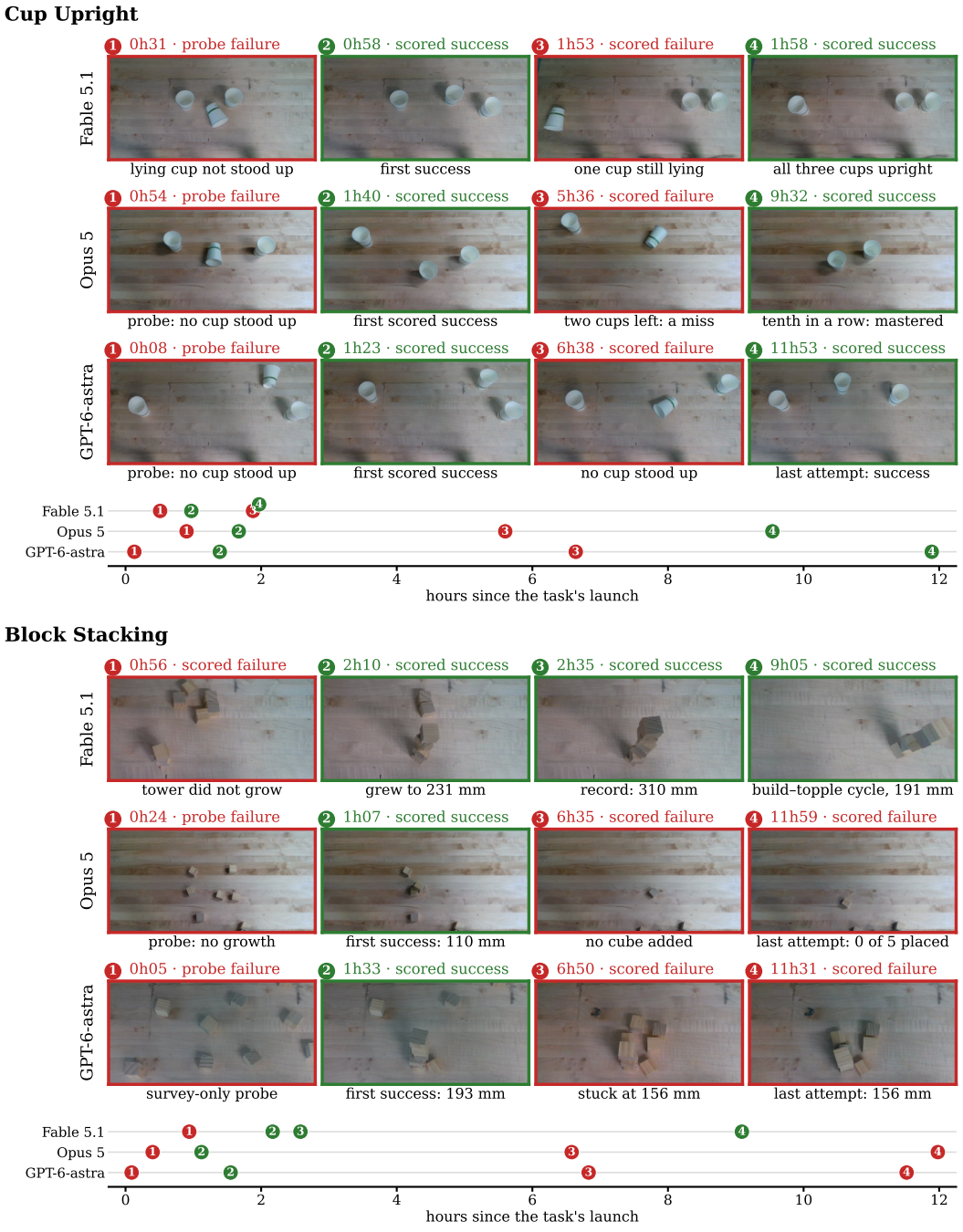

Figure 13: Self-evolution on hardware, continued from Figure 12: Cup Upright and Block Stacking. Opus's Cup Upright attempts after hour 2.9 practice a two-cup scene, because one cup left the table. On Block Stacking, Fable's last tile is the build-and-topple cycle it settled into after its 310 mm record; 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.

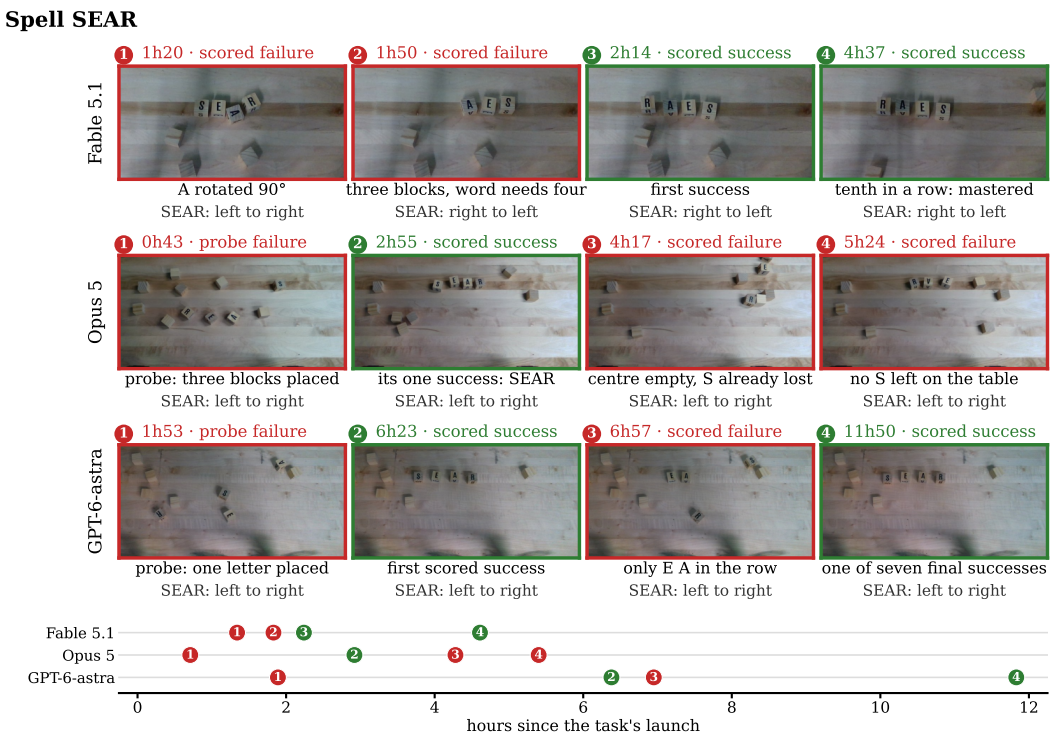

Figure 14: Self-evolution on hardware, continued from Figure 13: Spell SEAR. Each tile states the instructed reading direction for **SEAR** in the displayed camera view. Left-to-right spells SEAR; right-to-left appears as RAES from camera left to right. Fable tiles 2–4 request RAES, while its first tile 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.

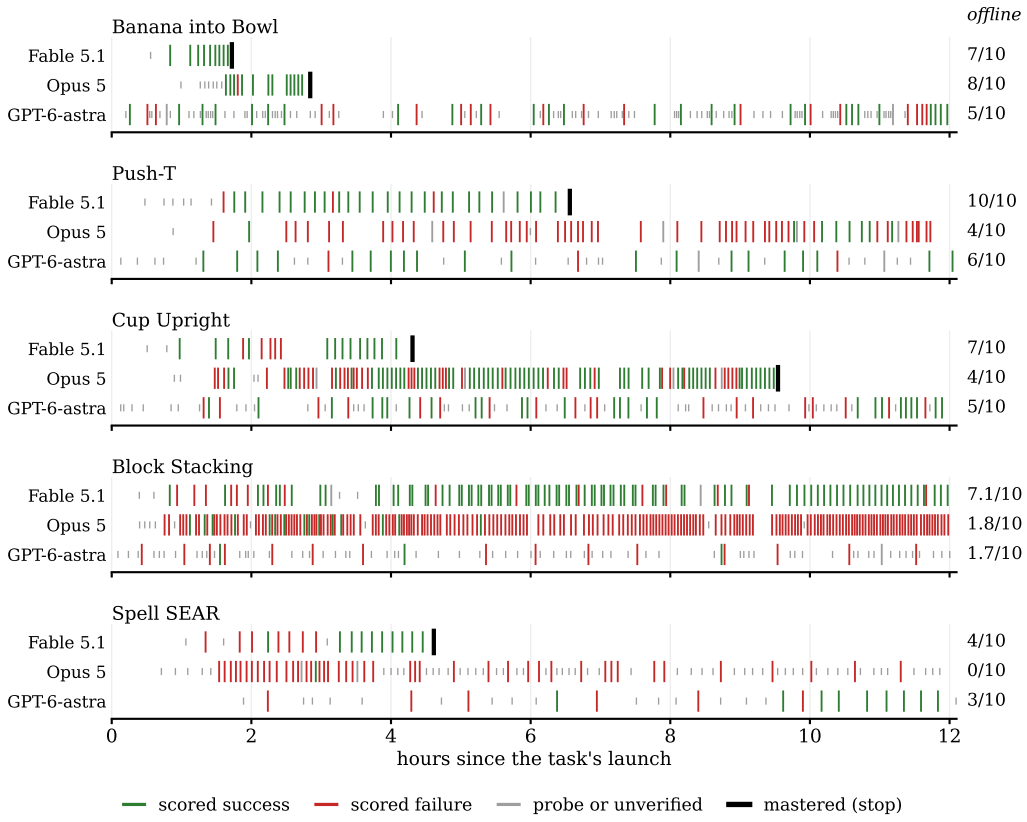

Figure 15: Every forward attempt of the physical campaign against the hours since its task's launch: tall ticks are scored attempts (green success, red failure), short grey ticks unscored probes, and the black bar is the tenth consecutive scored success, where the run stops; Block Stacking has no early stop. Reverse attempts are not drawn. The right margin shows each task's evaluation score out of ten, using the scoring rule in Section L. Fable reaches its streaks within five hours on every task but Push-T; Opus's Spell SEAR run turns into probes after its letters leave the table; GPT-6-astra's strips are mostly probes.

M 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 re-

[1] CAD model for the T-shaped block.

ceives 0. A banana resting on the bowl rim is an example of partial completion. Invalid or incorrectly staged trials are repeated. For Block Stacking, evaluators verify the maximum standing layer count from the recordings, with depth measurements providing the corresponding height.

N PHYSICAL CAMPAIGN: WHERE FROZEN POLICIES STOP

This appendix details the audit summarized in Section 4.5. 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 16 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; block-placement calls and whole-task episodes are different units, so 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.

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

Table 16: 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.

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 4.5, 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 inverse 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.

O SCENE PERTURBATION COSTS MORE THAN HARDWARE PERTURBATION, UNTIL THE FAULT IS SEVERE

Both T2 tests take the same LIBERO-practiced artifact and change the world while holding the tasks fixed. The Opus lineage 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 (Figure 3). The hardware cost grows with severity, to 56% at the severe level. **The two perturbations break different things** (Table 7). The strongest code-as-policy strategy compiles what the camera showed into geometry and then runs without looking again. A scene perturbation is absorbed at planning time. A fault that leaves the camera in place is corrected as each move is re-grounded, and seven of eight such axes hold the lineage at 87 to 98%. A moved camera, however, defeats the policy at every severity, 7 to 13% on the three scene-camera axes, because detections are lifted into world coordinates through an assumed camera pose and nothing in the loop notices. The no-library control shows the same pattern, so it belongs to code-as-policy with SAM3 perception rather than to the learning loop; the end-to-end VLA of Section 4.3 does not share it (Table 11). **Practice repairs the fault, and the inherited library is what makes practice work.** After 60 minutes per cell, the Opus lineage reaches 100% TEST on 17 of the 18 measurable cells, every camera axis included, whereas three 20-minute attempts without a library reach 21.1% on validation. The library lifts GPT-5.6-sol, Kimi-k3, and DeepSeek-v4-flash from 71.1, 57.2, and 31.1 to 90.0, 97.1, and 74.4% (Table 3). Seeding every cell from a fixed library, with nothing carried between cells, lands within two to eight points of the serial lineage. The repair therefore comes with the library, not from the earlier cells.

P LIBERO-RIG: DESIGN AND PROTOCOL

This section describes LIBERO-RIG, the hardware-robustness benchmark to supply the hardware tier of our generalization ladder (Section 3.5), and the protocol under which this paper uses it. The companion report that presents the benchmark in full, including an evaluation of end-to-end vision-language-action policies under the same protocol, is withheld for double-blind review.

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 17 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.

Table 17: 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	$\pm$2 to $\pm$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	$\pm$5 to $\pm$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)	$\pm$1 to $\pm$10 cm
gripper-pad geometry	W	the end-effector pads are worn down or oversized; fingers and joints are untouched	$\pm$5 to $\pm$25 percent

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.

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 18). Each cell receives the family's per-task development budget (Section C) and is graded on the DEV, VAL, and TEST bands of Section B, exactly as an unperturbed LIBERO task would be. The T2 results of Section O take artifacts practiced on unperturbed LIBERO and re-grade them, frozen, on all 40 tasks under every axis at the moderate to severe levels; Table 7 gives the per-axis values.

Table 18: 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

Q Extended Related Work

Self-evolving agents. Reflexion improves across repeated attempts with no weight update, by writing reflections on its own failures into an episodic memory (Shinn et al., 2023); STOP recursively rewrites its own improvement procedure (Zelikman et al., 2024); scaffold engineering evolves the harness around a frozen model (Weng, 2026; Lin et al., 2026a); and Lin et al. (2026b) show that much apparent self-evolution in LLM agents is resampling rather than improved strategy. In robotics, Code-as-Policies (Liang et al., 2023) and ProgPrompt (Singh et al., 2023) write executable control from language, Voyager's skill library (Wang et al., 2023) accumulates it, and Eureka (Ma et al., 2024) and RoboGen (Wang et al., 2024) have the model write reward code and task scenes instead of the policy. These precede a wave of self-improving manipulation systems, CaP-X (Fu et al., 2026), ASPIRE (Lu et al., 2026), RATS (Zhang et al., 2026), ENPIRE (Xiao et al., 2026), and industry agents that drive robots through tool use (Waddle Labs, 2026), each with its own evaluation protocol. **SEAR** is not a new self-improvement method but a fixed judge and protocol under which such methods can be compared, with a budget-matched control under every learning curve in the spirit of Lin et al. (2026b); its skill libraries descend from Voyager and its textual memories from Reflexion.

Benchmarks for autonomous research agents. MLE-bench (Chan et al., 2024), RE-Bench (Wijk et al., 2025), PaperBench (Starace et al., 2025), and PostTrainBench (Rank et al., 2026) grade agents on engineering, R&D under a time budget, paper replication, and language-model post-training, and EdgeBench (Zhu et al., 2026) studies scaling laws for agents learning from real-world interaction. **SEAR** shares their insistence on controlled budgets and isolation and brings it to a loop that closes

through physics: its sealed execution environment adapts EdgeBench's two-container design, and PostTrainBench's finding that agents default to supervised fine-tuning reappears in our AR runs (**??**); XPolicyLab's common policy interface (xpo, 2026) is complementary. What **SEAR** adds is the generalization ladder, frozen-artifact transfer across simulator boundaries, and a ledger that compares an agent that spends tokens with one that spends GPU-hours.

R Discussion

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, reactive correction, and modulated force 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 ignored it and wrote servo code over SAM3 detections; only a VLA fine-tuned inside the loop was adopted, so the orchestration the deliberative layer chooses depends on the reactive layer it is offered. Section 4.3 shows the other side: a VLA fine-tuned inside the loop generalizes past code when the instruction wording or the camera shifts.

The next agent type. These limits point at the architecture to evaluate next: 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 artifact, demonstrations as a strategy prior, and auto-research choosing what to fine-tune. The two layers co-evolve, the deliberative layer delegating given what the reactive layer can do and the reactive layer trained on what it is delegated, a leader-follower division of labor whose boundary should move as the policy improves; the question **SEAR** is built to answer is whether it changes the shape of the improvement curve rather than only its height.

Limitations and open questions. Three repetitions bound what we can distinguish (Section 4.4); budgets are matched across paradigms in wall clock but not in compute, so code versus weights must be read against a ledger; and the seal protects runtime state but not everything an agent can read (Section 3.2). We would like to be argued with on 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.

S Toward a Co-Evolving Agentic Robotics Loop

Running the CaP and AR modes side by side exposed a limit of the code-writing agent that a better language model does not remove. Its output space is discrete: tokens become code, and code calls primitives. Smooth contact, reactive mid-motion correction, and force that modulates as a part seats are not expressible that way, so the primitive API that makes failures legible also caps what any amount of clever code can do. And it starts every task from zero, with no motor prior and no strategy prior, rediscovering by search what one demonstration would show it. Offered a frozen pretrained VLA as a tool, the agents ignored it and wrote servo code over SAM3 detections (Section F); only a VLA fine-tuned inside the loop was adopted, so what the deliberative layer chooses to orchestrate depends on the reactive layer it is offered. Section 4.3 shows the other side of the same coin: a VLA trained from the agent's own data collection reaches states the program never did. These limits point past a system whose evaluated artifact is a program, toward one whose artifact is a program-checkpoint pair.

From interaction to system updates. Experience can enter the system through three complementary pathways (Figure 16). First, the agent consolidates reusable code, procedures, and lessons into its skill and memory libraries. Second, generated policies serve as executable teachers: repeated rollouts across environment configurations produce data for training VLAs or other learned action and world models, with outcome-aware selection so that successful trajectories support imitation while failures inform recovery learning and subsequent exploration. Third, the broader interaction history, 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

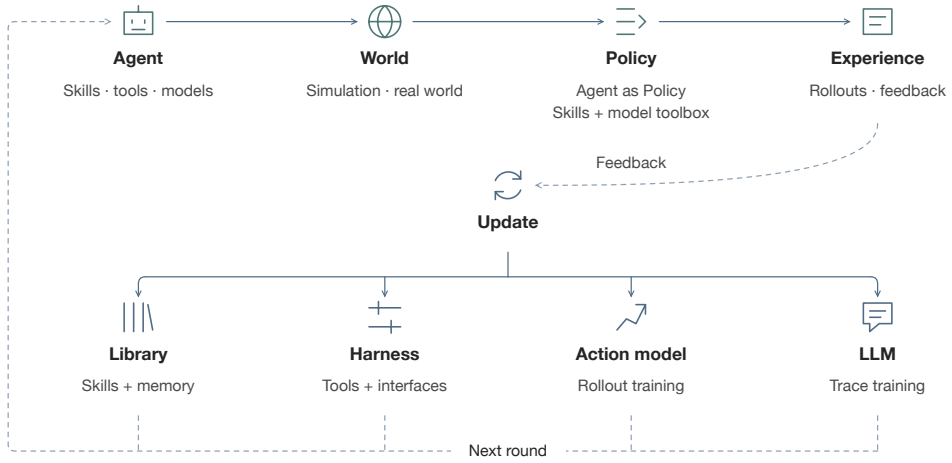

Figure 16: **The proposed co-evolving agentic robotics loop.** Through environment interaction, an agent develops and revises policies expressed as executable code or as an agent that reasons and calls tools during execution. Both policy forms draw on shared skills and a model toolbox. Execution yields 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.

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, and AR organizes experiments that improve the underlying models. 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 with language descriptions of their local objectives: placing a cup on a saucer decomposes into grasping the handle, lifting, transporting, lowering into contact, releasing, and withdrawing. The reasoning agent or generated program connects these skills, checks their outcomes, and revises the sequence when execution deviates from the plan. This establishes a 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 control and can become a generalist across reusable manipulation skills. Within the 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, and objects falling outside the reachable workspace may motivate explicit safety zones. A system that can revise its own harness carries a fourth kind of persistent component beyond the skills, memories, and weights of Section 2, and we would include it in the adaptive state on the same grounds. 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 to unresolved cases, and 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 toolbox evolves. The agent determines which experiences and policies train its tools, while the resulting tools alter the agent's available strategies and future data distribution; 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. 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 determine whether joint adaptation provides benefits beyond updating each component independently. The question **SEAR** is built to answer is whether coupling the two layers changes the shape of the improvement curve rather than only its height.