

ASPIRE: Agentic /Skills Discovery for Robotics

Runyu Lu^{1 2 * †}, Yubo Wu^{1 3 *}, Ethan Kou^{1 4 *}

Letian Fu^{1 4}, Wenli Xiao^{1 5}, Ajay Mandlekar¹, Yinzhen Xu¹
Guanya Shi⁵, Ken Goldberg⁴, Ang Chen², Mosharaf Chowdhury²
Yuke Zhu^{1 †}, Linxi “Jim” Fan^{1 †}, Guanzhi Wang^{1 †}

¹ NVIDIA, ² UMich, ³ UIUC, ⁴ UC Berkeley, ⁵ CMU

* Equal contribution, † Project leads

<https://research.nvidia.com/labs/gear/aspire/>

Abstract:

Traditional robot programming is notoriously challenging: it requires orchestrating multimodal perception, managing complex physical contact dynamics, and handling diverse environment configurations and execution failures. We introduce ASPIRE (**A**gentic **S**kill **P**rogramming through **I**terative **R**obot **E**xploration), a continual learning system for robotics that autonomously writes and refines robot control programs in a code-as-policy paradigm while compounding experience into a reusable skill library. ASPIRE enables automated discovery of reusable skills that persist across multiple tasks, simulation and real-world settings, and different embodiments. Rather than relying on fixed, human-engineered pipelines, ASPIRE operates in an open-ended learning loop, consisting of three key components: (1) a closed-loop robot execution engine that exposes fine-grained multimodal traces (e.g., perception overlays, grasp candidates, motion trajectories, and collision feedback), enabling the agent to autonomously diagnose failures, synthesize repairs, and validate outcomes; (2) a continually expanding skill library that distills validated fixes into reusable, transferable robotic knowledge; and (3) an evolutionary search procedure that generates diverse task sequences and control programs, systematically debugging them to explore beyond single-trajectory refinement. As ASPIRE encounters more tasks, its growing skill library enables increasingly rapid adaptation. Consequently, ASPIRE surpasses prior methods by up to 77% on manipulation tasks under perturbation (LIBERO-Pro), 72% on Robosuite’s bimanual handover task, and up to 32% on long-horizon household tasks (BEHAVIOR-1K). The accumulated skill library further enables strong zero-shot generalization: on representative unseen long-horizon tasks (LIBERO-Pro Long), ASPIRE achieves 31% success, substantially outperforming the 4% success rate of prior methods despite their heavy reliance on test-time reasoning and retries. Finally, skills discovered in simulation provide initial evidence of sim-to-real transfer, substantially reducing real-robot programming effort despite different embodiments and robot APIs.

1. Introduction

Recent progress in software engineering agents demonstrates that language models can autonomously inspect execution traces, localize failures, revise implementations, and improve through repeated interaction with execution environments [Anthropic, 2025, OpenAI, 2025, anomalyco, 2025, Wang et al., 2025, Yang et al., 2024]. In robotics, this paradigm has inspired *code-as-policy* systems that compose perception modules, planning APIs, and control primitives into executable robot programs [Liang et al., 2023, Singh et al., 2023, Ahn et al., 2022, Huang et al., 2023, Mu et al., 2024, Fu et al., 2026]. Because robot behaviors are represented explicitly as programs, they can in principle be inspected, edited, debugged, and refined through interaction feedback.

However, existing robotic coding agents remain fundamentally limited by naive execution environments that provide only coarse task-level feedback. Debugging robot programs is intrinsically challenging because failures can arise from many interacting components, including multimodal perception, motion planning, grasp generation, contact dynamics, and long-horizon task coordination. A failed rollout may indicate that the task did not succeed, but not whether the root cause is incorrect perception, an unstable grasp, a planning error, or a downstream recovery failure. Without

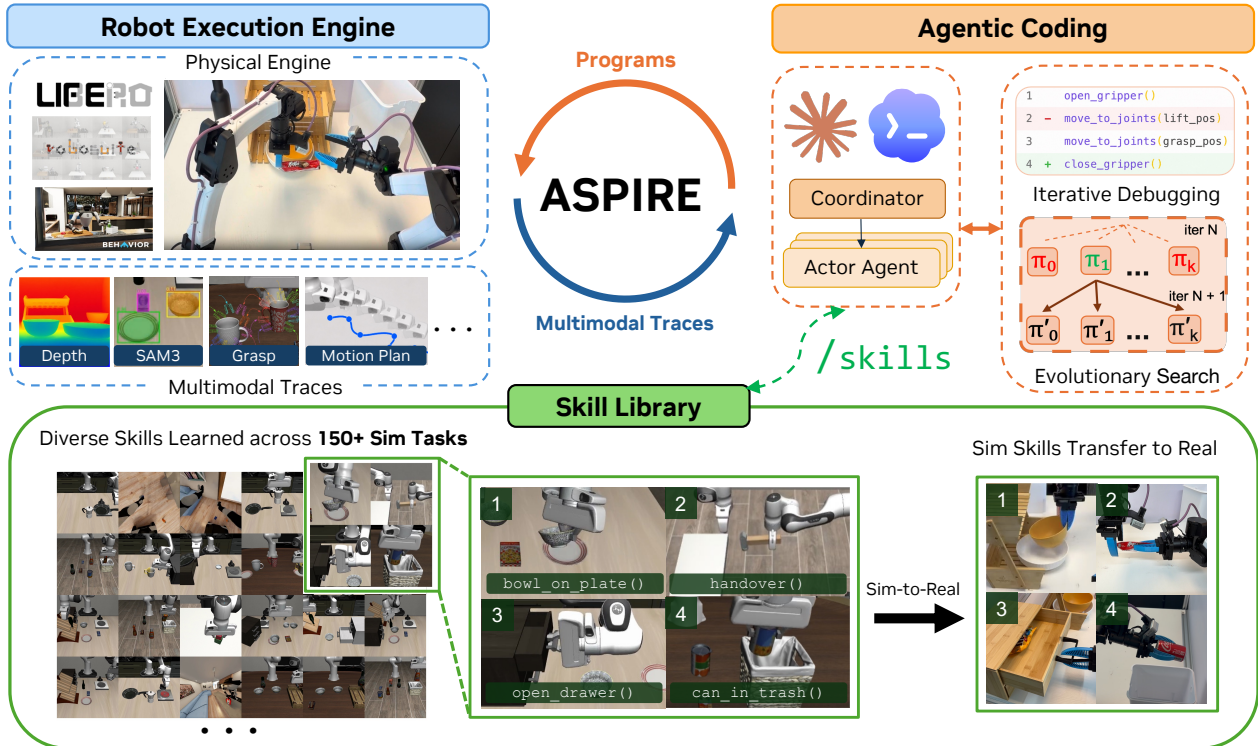


Figure 1: **Aspire system overview.** A *coordinator* spawns an *actor agent* (coding agent) per task, enabling parallel learning across tasks. Each actor refines and validates robot programs through *iterative debugging* with the *robot execution engine*, which exposes per-primitive multimodal traces for failure attribution and repair. *Evolutionary search* samples diverse candidate programs (π_0, \dots, π_k), sends each through the engine, and conditions the next generation (π'_0, \dots, π'_k) on surviving programs and residual failure traces. The *coordinator* writes validated repairs into a shared *skill library*, which future actors retrieve as in-context guidance; skills discovered in sim can also be adapted as cross-embodiment guidance for real-robot programming.

fine-grained diagnostic traces, agents have limited ability to determine what evidence to inspect, how to localize failures, or what repair strategy to attempt.

Moreover, existing systems do not accumulate experience across tasks. Once a task is completed, discovered fixes and recovery strategies are discarded rather than consolidated into reusable skills. As a result, the agent solving its hundredth task is effectively no more experienced than the agent solving its first.

Human robotics engineers take a fundamentally different approach. When a robot program fails, they replay executions, inspect perception outputs and motion trajectories, localize the failing subsystem, revise the implementation, and internalize reusable recovery strategies. Over time, debugging experience compounds into transferable knowledge, including grasp recovery heuristics, navigation strategies, prompting recipes, and procedural fixes that generalize across tasks. This accumulation of reusable knowledge is a key reason why human robot programmers become progressively more effective over time.

In this work, we introduce ASPIRE, a self-improving continual learning robotic system that autonomously writes and refines robot control programs in a code-as-policy paradigm while accumulating experience into a reusable skill library. Rather than operating within a fixed perception-plan-execute

pipeline, ASPIRE runs in an open-ended learning loop, in which the agent self-determines how to inspect execution traces, diagnose failures, synthesize repairs, validate corrected behaviors, and consolidate successful recovery patterns into persistent skills that transfer across tasks.

An overview of ASPIRE is shown in Fig. 1. ASPIRE is built from three components. First, a **closed-loop robot execution engine** replaces coarse rollout-level feedback with per-primitive execution traces. For each perception, planning, grasping, and control call, the execution engine records the observations, inputs, outputs, and visual evidence if possible. These rich multimodal traces allow the agent to selectively inspect salient primitive logs, progressively localize failures, and validate repairs through re-execution. Second, ASPIRE maintains a growing **skill library** that distills validated fixes into reusable, transferable robotic knowledge retrievable as in-context guidance for future tasks. Third, ASPIRE employs an **evolutionary search procedure** that generates diverse task sequences and control programs, exploring beyond single-trajectory self-improvement through iterative debugging and parallel refinement. Together, these components establish a self-improving robotic system whose performance scales with experience: the more tasks ASPIRE sees, the larger its skill library grows, and the more it transfers to novel tasks, longer-horizon behaviors, and real-world robotic settings where similar failure-recovery patterns emerge under different embodiments.

Empirically, ASPIRE demonstrates strong self-improvement across diverse short- and long-horizon robot benchmarks. Against prior coding agents [Fu et al., 2026], ASPIRE improves success rate by up to 77 points on LIBERO-Pro [Zhou et al., 2025] perturbation suites, by up to 72 points on Robosuite [Zhu et al., 2020] contact-rich manipulation tasks, and by up to 32 points on BEHAVIOR-1K [Li et al., 2024] long-horizon household tasks with procedurally generated layouts. Based on the skill library accumulated on LIBERO-90 [Liu et al., 2023], ASPIRE transfers zero-shot to LIBERO-Pro Long [Zhou et al., 2025], and reaches 31% success, while prior methods saturate at 4% despite their reliance on test-time reasoning and retries. We further evaluate three simulation-discovered skills on a real bimanual robot, showing that retrieving skills discovered in sim as in-context guidance reduces real-world reasoning tokens and enables successful programs where naive debugging without skill transfer fails entirely.

2. Method

ASPIRE consists of three components that together form an open-ended learning loop: (1) a *robot execution engine* (§2.1) that exposes per-primitive multimodal traces for failure attribution and executes agent-written repairs for closed-loop validation; (2) a *skill library* (§2.2) that accumulates validated repairs as reusable skills; and (3) an *evolutionary search* procedure (§2.3) that broadens exploration beyond single-trajectory repair. As ASPIRE encounters more tasks, its skill library grows, allowing future tasks to inherit accumulated repairs and reusable strategies.

As shown in Fig. 1, ASPIRE adopts a coordinator–actor architecture. A central coordinator manages the shared skill library and dispatches actor coding agents to individual tasks, while each actor writes, executes, diagnoses, and repairs robot programs within the robot execution engine. Actors do not exchange full chat histories or raw rollout trajectories. Instead, transferable experience is distilled into the skill library, allowing each actor’s context window to remain focused on the task specification, current program, and structured execution traces associated with the current failure.

2.1. Robot Execution Engine

Embodied coding agents need execution evidence to debug robot programs. Prior methods expose this evidence through fixed, human-designed interfaces, typically as manually curated scene-level summaries or a set of pre-defined observations. This creates a trade-off: too little evidence can hide

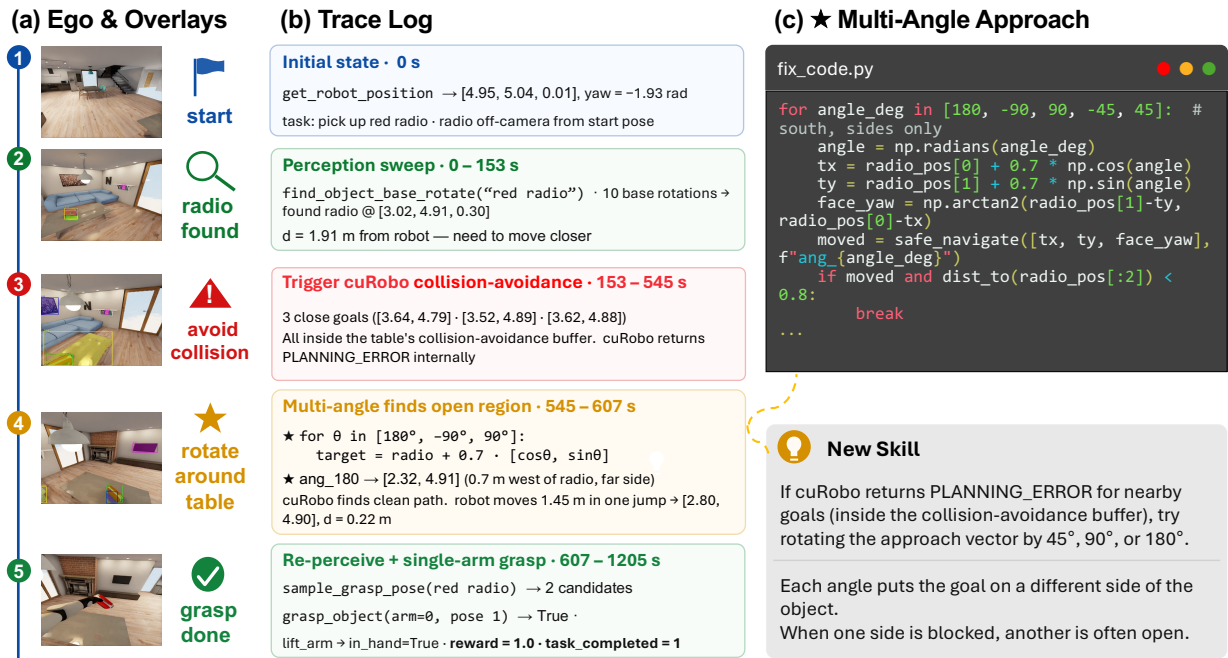


Figure 2: **Robot execution engine.** Trace-guided debugging on a BEHAVIOR-1K NAVIGATE-AND-PICK-UP-RADIO task. (a) Ego-view keyframes and overlays show the robot locating the radio but failing to approach it. (b) The primitive trace localizes the failure to repeated `PLANNING_ERRORS`: candidate navigation goals fall inside the table’s collision-avoidance buffer. (c) The agent patches the program with a multi-angle approach routine, re-perceives the radio from a reachable side, and completes the grasp. The validated repair is admitted as a reusable *Multi-Angle Approach* skill.

the failing primitive, while too much raw visual context can distract the agent from the causal chain that produced the failure.

ASPIRE turns this fixed feedback channel into an open-ended debugging environment. The robot execution engine records per-primitive multimodal traces for perception, planning, and control calls, exposes the trace to the coding agent, and executes agent-written repairs for closed-loop validation. For each primitive call, the trace stores the invoked API, inputs and outputs, return status, and relevant multimodal evidence such as RGB keyframes, overlays, grasp candidates, object poses, and motion-planning results. The agent does not receive full video frames; the engine keeps frames immediately before and after each primitive call together with the corresponding overlays and return values, so the agent can focus on evidence around calls implicated by the failure.

Fig. 2 zooms in on one BEHAVIOR-1K NAVIGATE-AND-PICK-UP-RADIO debugging episode. The ego-view keyframes show that the robot finds the radio, repeatedly fails to approach it, and then succeeds after changing its approach direction. The primitive trace localizes the failure: perception succeeds and returns a radio pose, but repeated `navigate_to_pose` calls return `PLANNING_ERROR`. By checking the navigation return values and associated logs, the agent finds that the generated navigation target lies too close to the table boundary, approximately within 20 centimeters of the table edge. This triggers collision avoidance and causes the planner to fail. Thus, the failure is not due to detecting or grasping the radio, but rather to the infeasibility of the target pose under the table’s collision constraints.



Figure 3: **Skill library.** ASPIRE stores validated, agent-discovered repair knowledge as reusable in-context skills rather than a fixed set of human-written primitives. Top: representative entries show learned skills about localization disambiguation, motion-primitive construction, and navigation recovery. Middle: the library grows across heterogeneous categories, including localization, navigation, motion primitives, object-level grasping, scene understanding, and debugging workflows. Bottom: selected skills discovered in sim are used as in-context guidance for real-robot programming, providing evidence that skills can transfer across embodiments. Additional skill examples and prompts are shown in Appendix A.

The repair follows directly from this diagnosis. Rather than changing the perception prompt or grasp primitive, the agent writes a multi-angle approach routine that samples alternative navigation targets around the radio and executes an approach direction that clears the collision buffer. The execution engine exposes the evidence, validates the patched program, and enables the agent to analyze the resulting logs, form a hypothesis, and make a targeted repair decision.

2.2. Skill Library

Program failures recur across tasks, but the reusable knowledge is rarely an entire task program. ASPIRE’s skill library stores heterogeneous repair knowledge: localization heuristics, perception prompts, grasping constraints, navigation recovery strategies, motion primitives, scene-understanding routines, and debugging workflows. We do not prescribe this taxonomy in advance. Skills are induced from validated repairs: the coding agent diagnoses a failure from execution traces, patches the program, validates the fix on debugging configurations, and the coordinator admits only reusable patterns into the shared library.

Each skill is stored as compact in-context guidance, including the failure signature, when-to-apply condition, repair strategy, and, when useful, a representative code sketch. Fig. 3 shows representative entries and the resulting library breadth. For the radio task in Fig. 2, the admitted skill is a navigation recovery pattern rather than a complete radio-pickup program: when planner errors recur near an obstacle boundary because sampled target poses fall inside the collision buffer, sample alternative approach directions around the object before retrying perception and grasping. This representation lets future actors reuse validated repairs instead of rediscovering them through test-time reasoning, supports zero-shot transfer to harder simulated tasks, and provides the mechanism for selected simulation-discovered skills to generalize across embodiments and transfer to real robots.

Actors report structured findings that summarize the failure mode, validated fix, and potentially transferable repair pattern. The coordinator audits these findings, verifies compliance with the allowed API policy, and promotes only reusable repairs that have passed debug validation into the shared skill library. Appendix E.1 provides the prompts used for actor reporting and coordinator-guided skill admission.

2.3. Evolutionary Search

Trace-guided debugging alone can collapse into local repair loops, where the agent repeatedly patches the same failed strategy instead of exploring fundamentally different ways to solve the task. ASPIRE uses evolutionary search to broaden exploration of executable robot programs, encouraging diverse repair hypotheses and task strategies.

In each round, based on the skill library, the coding agent proposes a population of K candidate programs conditioned on the top-performing previous programs and failure traces from previous evaluations. Each candidate is executed in the robot execution engine, producing task outcomes together with new diagnostic traces. The next round is then conditioned on the best-performing programs together with their remaining failure modes, allowing the search to explore distinct strategies rather than repeatedly refining the same solution.

The search target is the robot program itself. Candidates are selected through closed-loop execution, and validated repairs are admitted into the skill library after search concludes, provided they generalize across environment variations and tasks. Search terminates when a candidate solves the debugging configurations or when the search budget is exhausted. Algorithm 1 summarizes the evolutionary search procedure. See Appendix E.4 for pipeline details.

Algorithm 1 Evolutionary search over programs

Require: task τ , program P^0 , sets $S_{\text{dbg}}, S_{\text{val}}$, skill library \mathcal{L} , agent M , budget (K, T) , threshold θ

Notation: EXECUTE(P, S) = (r, Z) returns score r and trace bundle Z .

```

1:  $(r^*, Z^0) \leftarrow \text{EXECUTE}(P^0, S_{\text{dbg}})$ ;  $P^* \leftarrow P^0$ 
2:  $\mathcal{H} \leftarrow \{(P^0, r^*, Z^0)\}$ 
3: for  $i = 1, \dots, T$  do
4:    $\{P_i^k\}_{k=1}^K \leftarrow \text{PROPOSERREPAIRS}(M, \tau, \text{Top3}(\mathcal{H}), \mathcal{L}, \mathcal{H})$ 
5:   for  $k = 1, \dots, K$  do
6:      $(r_i^k, Z_i^k) \leftarrow \text{EXECUTE}(P_i^k, S_{\text{dbg}})$ 
7:   end for
8:    $\mathcal{H} \leftarrow \mathcal{H} \cup \{(P_i^k, r_i^k, Z_i^k)\}_{k=1}^K$ ;  $k^* \leftarrow \arg \max_k r_i^k$ 
9:   if  $r_i^{k^*} > r^*$  then
10:     $(P^*, r^*) \leftarrow (P_i^{k^*}, r_i^{k^*})$ 
11:   end if
12:   if  $r^* \geq \theta$  then
13:     break
14:   end if
15: end for
16:  $(r_{\text{val}}, Z_{\text{val}}) \leftarrow \text{EXECUTE}(P^*, S_{\text{val}})$ 
17:  $\mathcal{G} \leftarrow \text{EXTRACTVALIDATEDPATTERNS}(\mathcal{H}, P^*, r_{\text{val}}, Z_{\text{val}})$ 
18: return  $(P^*, r_{\text{val}}, \mathcal{G})$ 

```

3. Experiments

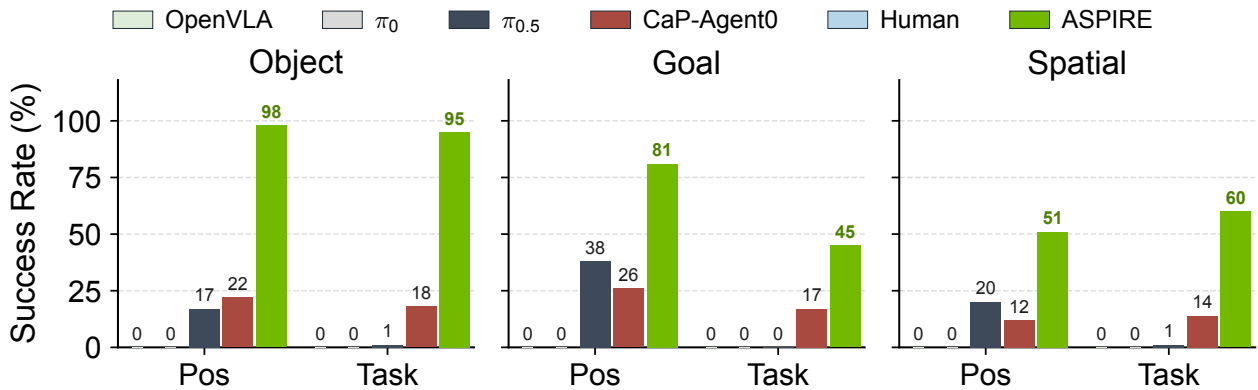
3.1. Experimental Setup

For all the simulation benchmarks, we use Claude Code [Anthropic, 2025] with Claude Opus 4.6 and a 1M-token context window as the coding agent. The agent writes executable Python robot programs in CaP-X [Fu et al., 2026], an open-source code-as-policy framework built on MuJoCo Playground [Zakka et al., 2025], with robot programming APIs for perception, geometry, and motion planning. The agent, environment, and API set are fixed across all experiments.

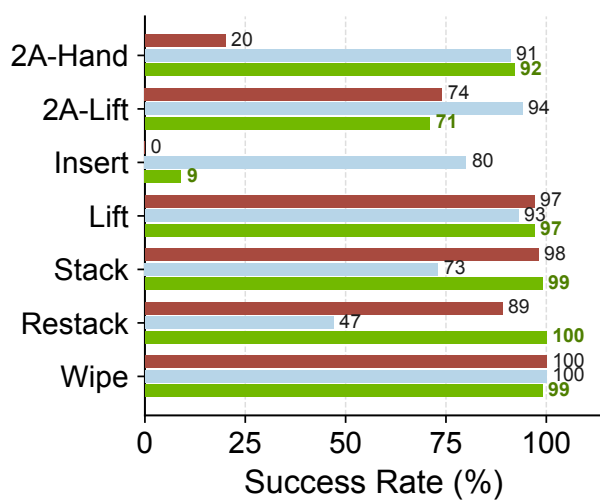
For the real-robot skill transfer study (§3.6), we use OpenAI Codex GPT-5.5 in reasoning-xhigh mode on a bimanual YAM manipulation station. We select three skills compiled by ASPIRE in Franka-based simulation: soda-can pickup, bowl-on-plate placement, and drawer push/pull. We provide these as in-context guidance to the real-robot coding agent. The real-robot setting uses a different embodiment and API from simulation, but exposes ASPIRE multimodal execution traces so the agent can autonomously run, inspect, and debug programs without task-specific human guidance during the debugging loop. We compare runs with and without the corresponding simulation-discovered skill, measuring tokens to first success and held-out real-robot success rate.

3.2. Benchmarks and Baselines

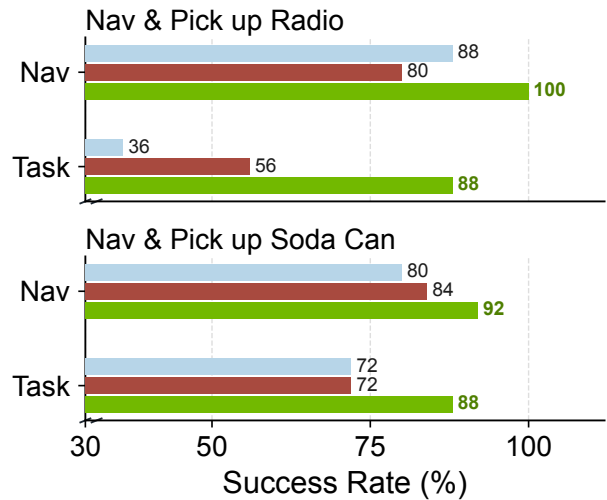
We evaluate ASPIRE on three benchmark families: LIBERO-Pro [Zhou et al., 2025] for short-horizon robustness under object, goal, and spatial perturbations; Robosuite [Zhu et al., 2020] for contact-rich single- and dual-arm manipulation; and BEHAVIOR-1K [Li et al., 2024] for long-horizon household mobile manipulation on NAVIGATE-AND-PICK-UP-SODA-CAN and NAVIGATE-AND-PICK-UP-RADIO. Our primary coding-agent baseline is CaP-Agent0 [Fu et al., 2026], which uses visual differencing, a predefined skill library, and per-episode test-time retries. We also compare with end-to-end VLA policies, including OpenVLA [Kim et al., 2024], π_0 [Black et al., 2025], and $\pi_{0.5}$ [Physical Intelligence et al., 2025]. For zero-shot transfer, we evaluate held-out LIBERO-Pro Long tasks using the skill library accumulated on LIBERO-90 [Liu et al., 2023].



(a) **LIBERO-Pro** [Zhou et al., 2025]: macro-averaged success over 10 tasks \times 50 held-out environment seeds per suite/perturbation. For each task, ASPIRE learns and collects skills on seeds 51–65 and evaluates one generated program on seeds 1–50.



(b) **Robosuite** [Zhu et al., 2020]: success over 100 held-out trials per task. ASPIRE learns on seeds 101–125 and evaluates one generated program per task on seeds 1–100; 2A tasks are bimanual.



(c) **BEHAVIOR-1K** [Li et al., 2024]: long-horizon mobile manipulation on two household tasks. ASPIRE learns on seeds 26–35 and evaluates on seeds 1–25 with incremental block execution; Navigation and Task success are reported separately.

Figure 4: **Aspire improves over prior coding agents and end-to-end VLAs across three benchmark families.** (a) Short-horizon manipulation on LIBERO-Pro; (b) contact-rich manipulation on Robosuite; (c) long-horizon mobile manipulation on BEHAVIOR-1K. ASPIRE evaluates one generated program per task across held-out seeds, while CaP-Agent0 regenerates a separate program per seed with test-time reasoning and retries. ASPIRE improves performance across all benchmarks, with several results surpassing programs written by human experts. Detailed per-task results are in Appendix B and D.1.

3.3. Evaluation Protocol

Across all benchmarks, an environment seed fixes each task instance, including object poses, distractors, and initial robot/object states. We use disjoint debug and evaluation seeds: ASPIRE learns on a small debug split, then reports success on larger held-out evaluation seeds with one generated program per LIBERO-Pro/Robosuite task, while CaP-Agent0 regenerates a separate program for each seed with test-time reasoning and retries. For BEHAVIOR-1K evaluation, ASPIRE uses incremental block execution, generating each next code block from the current multimodal

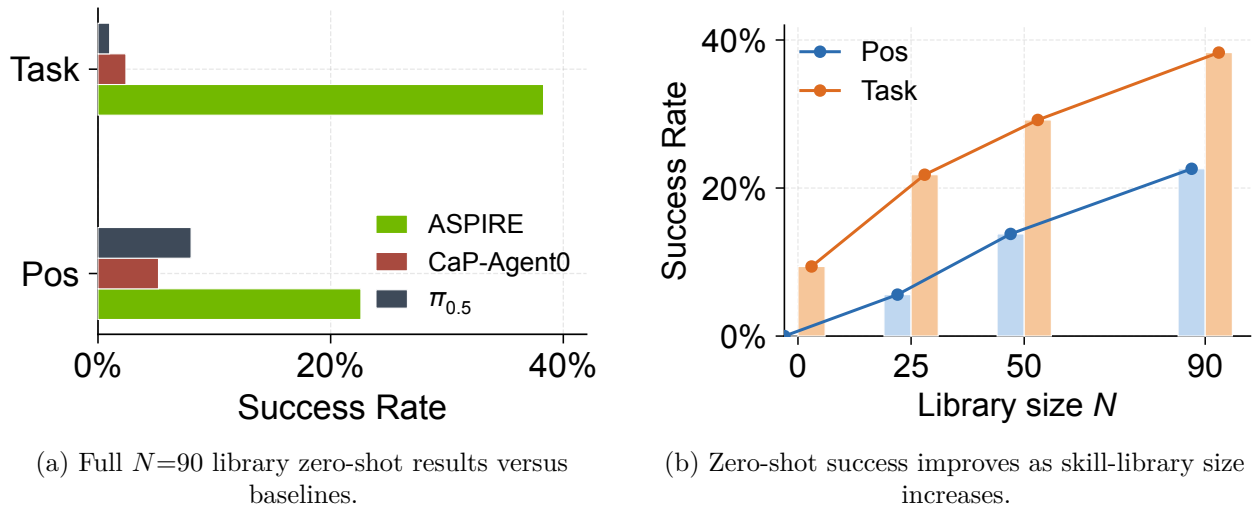


Figure 5: **Cross-task zero-shot transfer on LIBERO-Pro Long.** Skills accumulated on LIBERO-90 improve zero-shot performance on held-out long-horizon tasks. Figure (a) compares the full $N=90$ library with baselines. Figure (b) shows Pos/Task success as the size of the skill library increases. All success rates are macro-averaged over 10 tasks per axis. Per-task results are in Appendix C.

trace.

3.4. Main Evaluation Results

Figure 4 summarizes the main evaluation. “Human” means the programs are written by human experts. On LIBERO-Pro, ASPIRE improves over all the baselines on all three suites: averaging the Pos and Task perturbation axes, ASPIRE gains 77% on Object, 41.5% on Goal, and 42.5% on Spatial over the strongest baseline in each suite. $\pi_{0.5}$ is stronger than OpenVLA and π_0 on some position perturbations, but remains far below ASPIRE and largely collapses under task paraphrases. On Robosuite, ASPIRE preserves near-saturated performance on easier contact-rich tasks and substantially improves bimanual handover, from 20% to 92%. On BEHAVIOR-1K, ASPIRE outperforms both human and CaP-Agent0 on navigation and task success, with the largest task-level gain over CaP-Agent0 on NAVIGATE-AND-PICK-UP-RADIO, from 56% to 88%.

3.5. Zero-Shot Transfer to Unseen Tasks

We evaluate whether repair skills accumulated on LIBERO-90 transfer zero-shot to held-out LIBERO-Pro Long tasks. ASPIRE builds library snapshots from $N \in \{0, 25, 50, 90\}$ source tasks, where $N=0$ is an empty-library setting and $N=90$ is the full suite. For each held-out task, ASPIRE generates one program and evaluates it across seeds with no additional debugging, retries, or task-specific library updates.

Figure 5(a) compares the full $N=90$ skill library with prior baselines on the two LIBERO-Pro Long transfer axes: ASPIRE reaches 23% success on position perturbations and 38% on task perturbations, outperforming CaP-Agent0 and $\pi_{0.5}$ on both axes. Figure 5(b) shows that success increases consistently as the size of the skill library grows, indicating that validated repairs from short-horizon tasks provide reusable robotic knowledge for longer-horizon compositions.

Table 1: **Real-robot cross-embodiment skill transfer.** For each task, we compare real-robot debugging with and without retrieving a corresponding simulation-discovered skill from ASPIRE’s library. Token counts are measured until the first successful real-robot program, or until the debugging budget is exhausted for runs that do not reach success. Success rate reports evaluation trials for the generated programs.

Task	Output Tokens (M) ↓		Total Tokens (M) ↓		Success Rate ↑	
	w/o Skills	w/ Skills	w/o Skills	w/ Skills	w/o Skills	w/ Skills
Put bowl on plate	0.05	0.04	8.65	5.11	20/20	20/20
Lift soda can	0.18	0.03	61.94	6.58	13/20	19/20
Open/push drawer	1.33	0.36	334.917	81.67	0/20	11/20

3.6. Real-Robot Skill Transfer Across Embodiments

We evaluate whether skills learned in simulation can reduce debugging effort on a real robot with a different embodiment. The transfer is not a direct policy deployment: the real robot uses its own perception, calibration, and control stack, and the coding agent must still adapt programs through real-world execution feedback. The question is whether retrieving simulation-discovered skills provides useful in-context guidance that reduces the amount of real-world debugging needed to reach a successful program.

Table 1 shows that transferred skills consistently reduce debugging cost, while their effect on final success is task-dependent. Bowl placement succeeds in both settings but uses fewer tokens with skill retrieval; soda-can lifting improves from 13/20 to 19/20 while reducing total tokens by nearly an order of magnitude; and drawer manipulation reaches 11/20 success with skill guidance, while the no-skill baseline exhausts a larger token budget without producing a successful evaluation program. These results indicate that selected failure-derived skills can guide real-robot program synthesis across embodiment and API changes rather than merely memorizing simulator-specific code.

3.7. Ablation Studies

Robot execution engine and evolutionary search. We ablate two key components of ASPIRE on LIBERO-Pro: the robot execution engine, which exposes dense execution traces and validates task-level repairs, and evolutionary search, which explores additional repair candidates after robot-execution-engine debugging. Figure 6(a) and Figure 6(b) decompose success into a base system without either component, the gain from adding the robot execution engine, and the additional gain from evolutionary search. The robot execution engine provides the largest average improvement, raising macro-average success from 14% to 62%; evolutionary search further improves the remaining hard tasks, reaching 72% with both components.

Evolutionary search iterations. Figure 6(c) tracks performance on low-performing tasks as the evolutionary-search budget increases. Success improves steadily in the first few rounds, suggesting that sampling multiple repair hypotheses quickly recovers alternatives missed by single-iteration debugging. The curve continues to increase more gradually afterward, indicating that additional rounds still help on residual hard cases, but with diminishing returns.

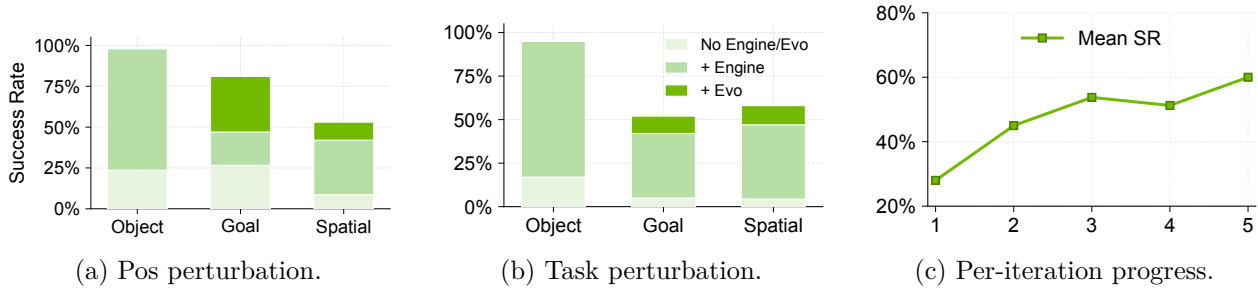


Figure 6: **Robot execution engine and evolutionary search ablations on LIBERO-Pro.** Figures (a) and (b) show stacked bars for position and task perturbations: the base system without the robot execution engine or evolutionary search, the gain from adding the robot execution engine, and the additional gain from evolutionary search. On average, the robot execution engine provides the largest gain, raising macro-average success from 14% to 62%; evolutionary search further improves remaining hard tasks. Figure (c) shows average progress over low-performing tasks across evolutionary-search iterations. Per-task results are in Appendix D.

4. Related Work

Agentic robot control. Robot control has been studied through end-to-end vision-language-action policies and executable programs that compose perception, planning, and control APIs [Ahn et al., 2022, Brohan et al., 2023, Octo Model Team et al., 2024, Kim et al., 2024, Black et al., 2025, Mu et al., 2024, Physical Intelligence et al., 2025, NVIDIA et al., 2025, Sun et al., 2026, Li et al., 2026, Fu et al., 2026]. Software-engineering agents provide a related write-execute-debug loop for code [Jimenez et al., 2024, Wang et al., 2025, Yang et al., 2024, Wang et al., 2024a, Chen et al., 2024, Anthropic, 2025, OpenAI, 2025, anomalyco, 2025]. ASPIRE builds on the executable-program paradigm, but focuses on persistent embodied improvement: the agent freely selects primitive-level multimodal traces, writes repairs, validates, and preserves successful experiences across tasks.

Self-improving agents and skill libraries. LLM agents have been improved through open-ended memory, skill libraries, and self-evolving skill repositories [Wang et al., 2023, Tziafas and Kasaei, 2024, Peng et al., 2024, Fu et al., 2025, Zhang et al., 2026, Xie et al., 2026]. Other systems use LLMs to generate rewards, curricula, environments, or search candidates [Ma et al., 2024a, Yu et al., 2023, Xie et al., 2024, Liang et al., 2024, Ma et al., 2024b, Du et al., 2023, Wang et al., 2024b, Romera-Paredes et al., 2024, Novikov et al., 2025, Guo et al., 2025, 2026, Cao et al., 2026]. Unlike success-only memories, textual reflections, or reward functions for downstream policy training, ASPIRE stores validated repair knowledge extracted from attributed embodied failures. Its skills are open-ended in both content and admission: they emerge from the agent’s own debugging experience, span heterogeneous categories (§2.2), and are reused as in-context guidance for future program repair. Evolutionary search (§2.3) further broadens the space of executable program repairs before reusable patterns are admitted to the library.

5. Limitations

ASPIRE has several important limitations. First, while selected simulation-discovered skills support autonomous real-robot debugging on a different embodiment, our system is not yet a fully autonomous real-world lifelong learner. Unlike simulation, where success checking and scene resets are cheap and programmatic, real-world deployment still requires robust success detection, safe reset, safety monitoring, and calibration maintenance. Future work should close this evaluation-and-reset loop to

scale sim-to-real skill transfer across broader real-world task suites. Second, our method relies on a frozen frontier LLM (Claude Opus 4.6 with a 1M-token context window) to interpret multimodal traces, write program repairs, and propose evolutionary-search candidates; we have not verified that smaller or weaker LLMs can sustain the same debugging loop. Third, ASPIRE writes programs using a predefined API of perception, planning, and control primitives. This API makes debugging tractable and safe, but also bounds the behaviors the agent can express: if a task requires sensing, control, or interaction capabilities outside the exposed primitives, the agent must either approximate them inefficiently or rely on humans to extend the API. Future systems should study how agents can safely propose, validate, and incorporate new robot primitives. Fourth, the skill library currently prioritizes validated reusable repairs but does not fully solve long-term memory management. As the library grows, some entries may become stale, overly specific, redundant, or misleading for a new task, which can explain non-monotonic trends in zero-shot transfer. More robust retrieval, pruning, ranking, and re-validation mechanisms are needed to keep the library useful at scale. Finally, the debug and evolutionary-search loop is compute-intensive, consuming many LLM calls and simulator or robot rollouts per task; scaling to very large task suites will require either cheaper LLM inference, more sample-efficient search, or stronger mechanisms for reusing prior repairs.

6. Conclusion

We present ASPIRE, a continual learning robotic system that autonomously writes and refines robot control programs while compounding experience into a reusable skill library. ASPIRE operates in an open-ended learning loop with three components: a closed-loop robot execution engine that exposes fine-grained multimodal traces, a continually expanding skill library that distills validated fixes into transferable knowledge, and an evolutionary search procedure that explores diverse task sequences and control programs. Across diverse benchmarks, ASPIRE substantially outperforms existing VLA and coding-agent baselines, demonstrates strong zero-shot transfer to unseen long-horizon tasks, and provides initial evidence that the skills discovered in sim can transfer across embodiment to significantly reduce real-robot programming token cost despite different robot embodiments and APIs.

Acknowledgments

We thank Nadun Ranawaka, Jimmy Wu, Matin Furutan, Haotian Lin, Abhi Maddukuri, Yulu Gan, Matin Nikoui, and Yuqi Xie for their help with open-source support, real robot infrastructure, advice and guidance on the paper, website release, and experimental equipment. We are also grateful to the members of NVIDIA GEAR, UMich SymbioticLab, UsesysLab, UC Berkeley AUTOLab, CMU LeCAR Lab for their kind support.

References

- Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Goper, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning (CoRL)*, 2022.
- anomalyco. Opencode: The open source ai coding agent, 2025. URL <https://github.com/anomalyco/opencode>. Accessed May 2026.
- Anthropic. Claude code: Best practices for agentic coding, 2025. URL <https://www.anthropic.com/engineering/claude-code-best-practices>. Accessed April 2026.

- Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0 : A vision-language-action flow model for general robot control. In *Robotics: Science and Systems (RSS)*, 2025.
- Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning (CoRL)*, 2023.
- Shiyi Cao, Ziming Mao, Joseph Gonzalez, and Ion Stoica. K-search: LLM kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.
- Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. In *International Conference on Machine Learning (ICML)*, 2023.
- Honghao Fu, Junlong Ren, Qi Chai, Deheng Ye, Yujun Cai, and Hao Wang. Vistawise: Building cost-effective agent with cross-modal knowledge graph for minecraft. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025.
- Max Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. Cap-x: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026.
- Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. Evoengineer: Mastering automated CUDA kernel code evolution with large language models. *arXiv preprint arXiv:2510.03760*, 2025.
- Ping Guo, Chao Li, Yinglan Feng, and Chaoning Zhang. Code evolution for control: Synthesizing policies via LLM-driven evolutionary search. *arXiv preprint arXiv:2601.06845*, 2026.
- Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3D value maps for robotic manipulation with language models. In *Conference on Robot Learning (CoRL)*, 2023.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.
- 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. In *Conference on Robot Learning (CoRL)*, 2024.
- Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabriel Levine, Wensi Ai, Benjamin Martinez, et al. Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation. In *Conference on Robot Learning (CoRL)*, 2024.

- Ruiying Li, Yunlang Zhou, YuYao Zhu, Kylin Chen, Jingyuan Wang, Sukai Wang, Kongtao Hu, Minhui Yu, Bowen Jiang, Zhan Su, Jiayao Ma, Xin He, Yongjian Shen, Yang Yang, Guanghui Ren, Maoqing Yao, Wenhao Wang, and Yao Mu. Roboclaw: An agentic framework for scalable long-horizon robotic tasks. *arXiv preprint arXiv:2603.11558*, 2026.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.
- William Liang, Sam Wang, Hung-Ju Wang, Osbert Bastani, Dinesh Jayaraman, and Yecheng Jason Ma. Eurekaverse: Environment curriculum generation via large language models. In *Conference on Robot Learning (CoRL)*, 2024.
- 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 (NeurIPS)*, 2023.
- 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)*, 2024a.
- Yecheng Jason Ma, William Liang, Hung-Ju Wang, Sam Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. Dreureka: Language model guided sim-to-real transfer. In *Robotics: Science and Systems (RSS)*, 2024b.
- Yao Mu, Junting Chen, Qinglong Zhang, Shoufa Chen, Qiaojun Yu, Chongjian Ge, Runjian Chen, Zhixuan Liang, Mengkang Hu, Chaofan Tao, Peize Sun, Haibao Yu, Chao Yang, Wenqi Shao, Wenhai Wang, Jifeng Dai, Yu Qiao, Mingyu Ding, and Ping Luo. Robocodex: Multimodal code generation for robotic behavior synthesis. In *International Conference on Machine Learning (ICML)*, 2024.
- Alexander Novikov, Ngãn Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- NVIDIA, Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, Joel Jang, Zhenyu Jiang, Jan Kautz, Kaushil Kundalia, Lawrence Lao, Zhiqi Li, Zongyu Lin, Kevin Lin, Guilin Liu, Edith Llontop, Loic Magne, Ajay Mandlekar, Avnish Narayan, Soroush Nasiriany, Scott Reed, You Liang Tan, Guanzhi Wang, Zu Wang, Jing Wang, Qi Wang, Jiannan Xiang, Yuqi Xie, Yinzhen Xu, Zhenjia Xu, Seonghyeon Ye, Zhiding Yu, Ao Zhang, Hao Zhang, Yizhou Zhao, Ruijie Zheng, and Yuke Zhu. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.
- Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, You Liang Tan, Lawrence Yunliang Chen, Pannag Sanketi, Quan Vuong, Ted Xiao, Dorsa Sadigh, Chelsea Finn, and Sergey Levine. Octo: An open-source generalist robot policy. In *Robotics: Science and Systems (RSS)*, 2024.
- OpenAI. Codex: Ai coding partner from openai, 2025. URL <https://openai.com/codex/>. Accessed May 2026.
- Shaohui Peng, Xing Hu, Qi Yi, Rui Zhang, Jiaming Guo, Di Huang, Zikang Tian, Ruizhi Chen, Zidong Du, Qi Guo, Yunji Chen, and Ling Li. Hypothesis, verification, and induction: grounding

- large language models with self-driven skill learning. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. ISBN 978-1-57735-887-9. doi: 10.1609/aaai.v38i13.29376. URL <https://doi.org/10.1609/aaai.v38i13.29376>.
- Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, et al. $\pi_{0.5}$: a vision-language-action model with open-world generalization. In *Conference on Robot Learning (CoRL)*, 2025.
- Bernardino Romera-Paredes, Mohammadamin Barekatin, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.
- 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.
- Xiaoquan Sun, Zetian Xu, Chen Cao, Zonghe Liu, Yihan Sun, Jingrui Pang, Ruijian Zhang, Zhen Yang, Kang Pang, Dingxin He, Mingqi Yuan, and Jiayu Chen. Atomvla: Scalable post-training for robotic manipulation via predictive latent world models. *arXiv preprint arXiv:2603.08519*, 2026.
- Georgios Tzifas and Hamidreza Kasaei. Lifelong robot library learning: Bootstrapping composable and generalizable skills for embodied control with language models. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2024.
- 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*, 2024a.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025.
- Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, 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)*, 2024b.
- Senwei Xie, Yuntian Zhang, Ruiping Wang, and Xilin Chen. Uni-skill: Building self-evolving skill repository for generalizable robotic manipulation. *arXiv preprint arXiv:2603.02623*, 2026.
- Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2024.

- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
- Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis. In *Conference on Robot Learning (CoRL)*, 2023.
- Kevin Zakka, Baruch Tabanpour, Qiayuan Liao, Mustafa Haiderbhai, Samuel Holt, Jing Yuan Luo, Arthur Allshire, Erik Frey, Koushil Sreenath, Lueder A. Kahrs, Carmelo Sferrazza, Yuval Tassa, and Pieter Abbeel. Mujoco playground. *arXiv preprint arXiv:2502.08844*, 2025.
- Ziao Zhang, Kou Shi, Shiting Huang, Avery Nie, Yu Zeng, Yiming Zhao, Zhen Fang, Qishen Su, Haibo Qiu, Wei Yang, Qingnan Ren, Shun Zou, Wenxuan Huang, Lin Chen, Zehui Chen, and Feng Zhao. Skillflow: benchmarking lifelong skill discovery and evolution for autonomous agents, 2026. URL <https://arxiv.org/abs/2604.17308>.
- Xueyang Zhou, Yangming Xu, Guiyao Tie, Yongchao Chen, Guowen Zhang, Duanfeng Chu, Pan Zhou, and Lichao Sun. Libero-pro: Towards robust and fair evaluation of vision-language-action models beyond memorization. *arXiv preprint arXiv:2510.03827*, 2025.
- Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning. *arXiv preprint arXiv:2009.12293*, 2020.

A. Skill Library Details

This appendix expands the representative entries shown in Figure 3 into the skill-library taxonomy. Each entry will list: (i) the *problem* extracted from the triggering failure trace; (ii) the *when-to-apply* guard encoding the situational retrieval condition; (iii) the validated *repair snippet*; and (iv) the origin task(s) that produced the entry.

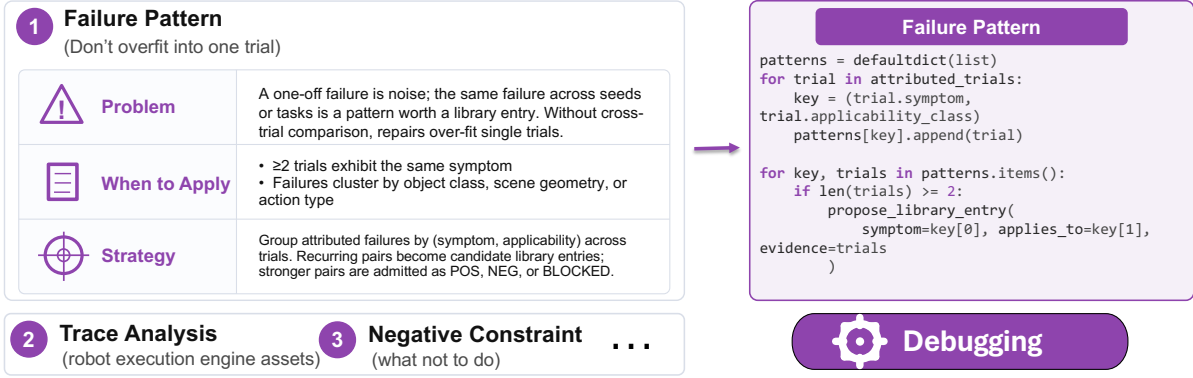


Figure 7: **Debugging skills.** Representative skill-library entries that encode reusable debugging strategies, including failure signatures, when-to-apply guards, and validated repair sketches.

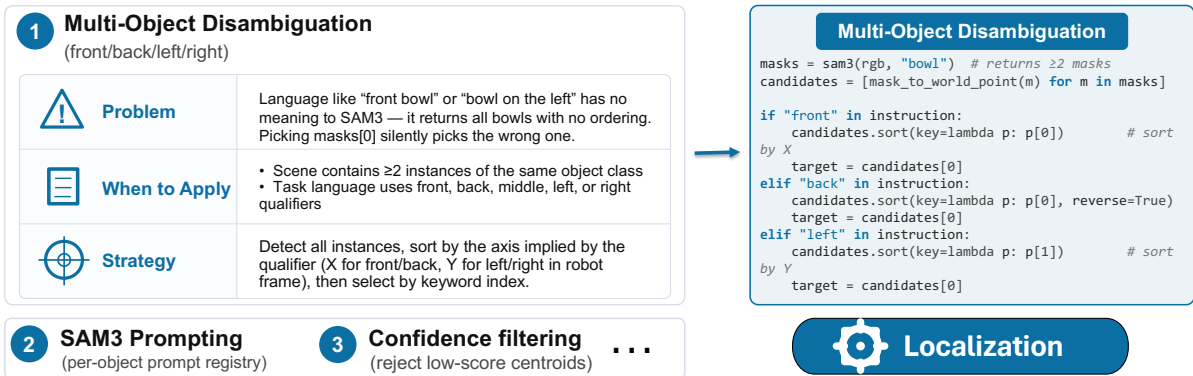


Figure 8: **Localization skills.** Representative entries for grounding ambiguous language and object references into robust perception and localization routines.

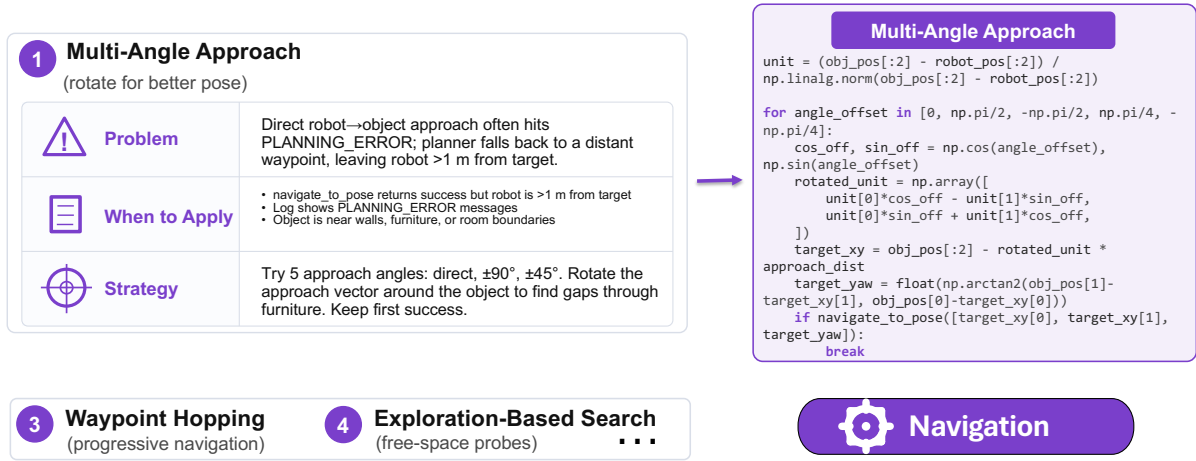
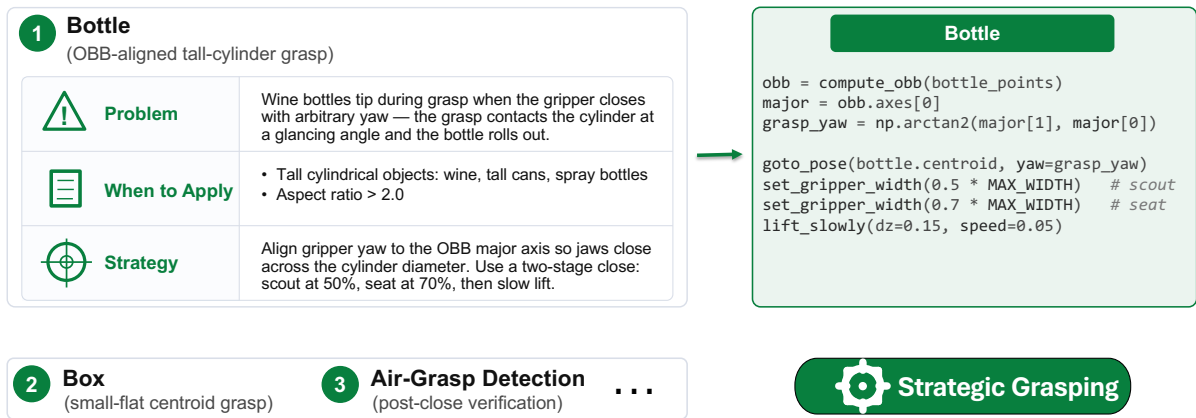


Figure 9: **Navigation skills.** Representative entries for recovering from motion-planning failures and selecting collision-aware approach poses.



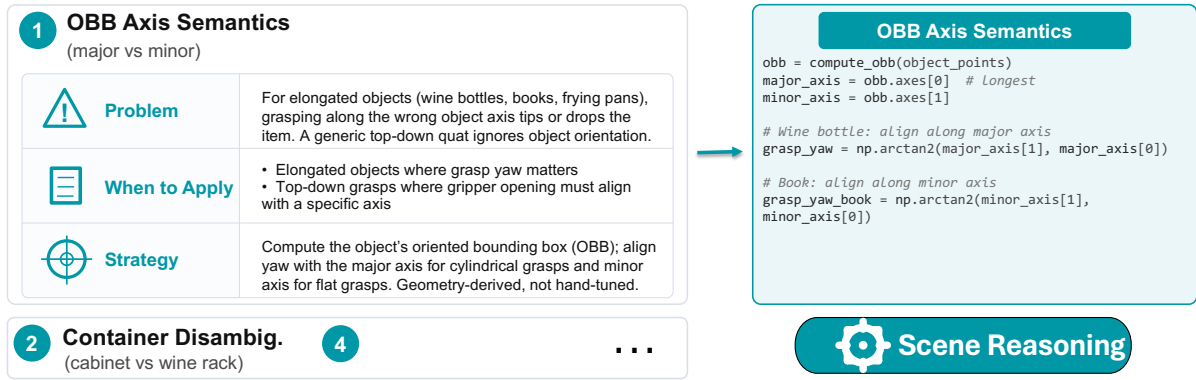


Figure 12: **Scene-reasoning skills.** Representative entries for reasoning over spatial relations, support surfaces, occlusions, and scene-level task constraints.

B. Main Benchmark Result Tables

B.1. Macro-Averaged Main Results

This section reports the exact numerical values visualized in Figure 4 of the main paper.

Table 2: LIBERO-Pro [Zhou et al., 2025] performance of OpenVLA [Kim et al., 2024], π_0 [Black et al., 2025], $\pi_{0.5}$ [Physical Intelligence et al., 2025], CaP-Agent0 [Fu et al., 2026], and ASPIRE on the **libero-object**, **libero-goal**, and **libero-spatial** benchmarks under initial position perturbations (Pos) and instruction perturbations (Task), macro-averaged across 10 tasks per suite. **Overall** columns are macro-averaged across the three suites. All ASPIRE numbers are on the 50-seed held-out evaluation (seeds 1–50). See the per-task breakdown in Appendix D.1.

Method	libero-object		libero-goal		libero-spatial		Overall		
	Pos	Task	Pos	Task	Pos	Task	Pos	Task	All
OpenVLA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
π_0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$\pi_{0.5}$	0.17	0.01	0.38	0.00	0.20	0.01	0.25	0.01	0.13
CaP-Agent0	0.22	0.18	0.26	0.17	0.12	0.14	0.20	0.16	0.18
Aspire	0.98	0.95	0.81	0.45	0.51	0.60	0.77	0.67	0.72

Table 3: Robosuite [Zhu et al., 2020] performance of CaP-Agent0 [Fu et al., 2026] and ASPIRE on 7 manipulation tasks (100-trial held-out evaluation). Values are task success rates in $[0, 1]$. **Average** is the unweighted mean across all 7 tasks.

Task	CaP-Agent0	Aspire
cube_lift	0.97	0.97
cube_stack	0.98	0.99
cube_restack	0.89	1.00
spill_wipe	1.00	0.99
two_arm_handover	0.20	0.92
two_arm_lift	0.74	0.71
nut_assembly	0.00	0.09
<i>Mean</i>	<i>0.68</i>	<i>0.81</i>

Table 4: BEHAVIOR-1K [Li et al., 2024] performance on two representative household tasks: Soda Can pick-up and Radio pick-up. Navigation and Task Success rates in $[0, 1]$ are reported separately; all numbers are on the 25-seed held-out evaluation (seeds 1–25); ASPIRE runs interactive block-by-block generation with the skill library accumulated on seeds 26–35.

Task	Human		CaP-Agent0		Aspire	
	Nav	Task	Nav	Task	Nav	Task
Soda Can pick-up	0.80	0.72	0.84	0.72	0.92	0.88
Radio pick-up	0.88	0.36	0.80	0.56	1.00	0.88

C. LIBERO-Pro Long Zero-Shot Transfer

C.1. Macro-Averaged Main Results

Table 5: Macro-averaged zero-shot transfer of the ASPIRE skill library (accumulated on LIBERO-90) to LIBERO-Pro Long [Zhou et al., 2025]. N denotes the number of LIBERO-90 tasks whose repair skills seed the library; no additional debugging is performed at test time. Columns report macro-averaged success rate on Pos (position perturbation) and Task (instruction perturbation) axes, 10 tasks per axis.

Method	LIBERO-Pro Long Zero-Shot Transfer		
	Pos	Task	Overall
OpenVLA	0.00	0.00	0.00
π_0	0.00	0.00	0.00
$\pi_{0.5}$	0.08	0.01	0.05
CaP-Agent0	0.052	0.024	0.038
ASPIRE ($N=0$, zero-shot)	0.00	0.094	0.047
ASPIRE ($N=25$, zero-shot)	0.056	0.218	0.137
ASPIRE ($N=50$, zero-shot)	0.138	0.292	0.215
ASPIRE ($N=90$, zero-shot)	0.226	0.383	0.305

C.2. Per-Task Breakdown

Table 6: Zero-shot transfer of the ASPIRE skill library (accumulated on LIBERO-90) to LIBERO-Pro Long, per task and snapshot size N (seeds 1–50, $n=50$). **Pos**: positional-perturbation variant. **Task**: semantic-perturbation variant.

Task	Pos				Task			
	$N=0$	$N=25$	$N=50$	$N=90$	$N=0$	$N=25$	$N=50$	$N=90$
Stove + moka pot	0.00	0.30	0.84	1.00	0.00	0.02	0.68	0.26
Bowl in drawer	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Mug in microwave	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Both mokas on stove	0.00	0.00	0.00	0.00	0.00	0.00	0.84	1.00
Soup + cream cheese	0.00	0.12	0.02	0.00	0.00	0.68	0.60	0.70
Soup + tomato sauce	0.00	0.14	0.20	0.00	0.78	0.68	0.02	0.70
Cream cheese + butter	0.00	0.00	0.06	0.94	0.00	0.00	0.34	0.86
Mug on two plates	0.00	0.00	0.16	0.32	0.16	0.28	0.08	0.00
Mug + pudding	0.00	0.00	0.06	0.00	0.00	0.00	0.08	0.04
Book in caddy	0.00	0.00	0.04	0.00	0.00	0.52	0.28	0.26
<i>Mean</i>	<i>0.00</i>	<i>0.06</i>	<i>0.14</i>	<i>0.23</i>	<i>0.09</i>	<i>0.22</i>	<i>0.29</i>	<i>0.38</i>

D. Ablation Details

D.1. LIBERO-Pro Position Perturbation (Swap) Per-Task Breakdown

Table 7: LIBERO-Pro per-task ablation, **Position perturbation** (seeds 1–50, $n=50$).

Aspire w/o Robot execution engine and Evolutionary search: zero-shot Claude Opus 4.6 with 15 example programs. **Aspire w/o Evolutionary search:** Robot Execution Engine repaired program (Execution engine + skill library). **Evo. search:** raw evolutionary-search result (best validated candidate), or “–” if Evolutionary Search was not run for that task. **Aspire:** per-task winner selected on a held-out validation set (seeds 66–80) between the repaired program and Evolutionary Search.

Task	w/o Engine & Evo	w/o Evo	Evo. search	Aspire
<i>libero-object</i>				
Alphabet soup	0.00	1.00	–	1.00
BBQ sauce	0.06	1.00	–	1.00
Butter	0.00	1.00	–	1.00
Chocolate pudding	0.00	1.00	–	1.00
Cream cheese	0.92	1.00	–	1.00
Ketchup	0.02	0.92	–	0.92
Milk	0.06	0.92	–	0.92
Orange juice	0.46	1.00	–	1.00
Salad dressing	0.18	0.96	–	0.96
Tomato sauce	0.66	0.98	–	0.98
<i>Mean</i>	<i>0.24</i>	<i>0.98</i>	–	<i>0.98</i>
<i>libero-goal</i>				
Open middle drawer	0.28	0.02	0.72	0.72
Open top drawer+bowl	0.00	0.86	–	0.86
Push plate to stove	0.00	0.00	0.00	0.00
Bowl on plate	0.04	0.36	0.92	0.92
Bowl on stove	0.16	0.56	0.96	0.96
Bowl on cabinet top	0.72	0.74	1.00	1.00
Cream cheese in bowl	0.00	0.74	0.92	0.74
Wine on rack	0.06	0.46	0.98	0.98
Wine on cabinet top	1.00	0.24	1.00	1.00
Turn on stove	0.40	0.68	0.90	0.90
<i>Mean</i>	<i>0.27</i>	<i>0.47</i>	<i>0.82⁽ⁿ⁼⁹⁾</i>	<i>0.81</i>
<i>libero-spatial</i>				
Betw. plate+ramekin	0.02	0.10	0.94	0.94
From table center	0.00	0.32	0.82	0.32
In top drawer	0.02	0.22	0.10	0.10
Next to cookie box	0.02	0.16	0.12	0.16
Next to plate	0.08	1.00	1.00	1.00
Next to ramekin	0.00	0.20	0.40	0.20
On cookie box	0.00	0.32	0.24	0.24
On ramekin	0.58	0.42	0.94	0.94
On stove	0.04	0.86	–	0.86
On wooden cabinet	0.10	0.56	0.34	0.34
<i>Mean</i>	<i>0.09</i>	<i>0.42</i>	<i>0.54⁽ⁿ⁼⁹⁾</i>	<i>0.51</i>
<i>Overall mean</i>	<i>0.20</i>	<i>0.62</i>	<i>0.68⁽ⁿ⁼¹⁸⁾</i>	<i>0.77</i>

D.2. LIBERO-Pro Task Perturbation (Goal) Per-Task Breakdown

Table 8: LIBERO-Pro per-task ablation, **Task perturbation** (seeds 1–50, $n=50$). **Aspire w/o Robot execution engine and Evolutionary search**: zero-shot Claude Opus 4.6 with 15 example programs. **Aspire w/o Evolutionary search**: Robot Execution Engine repaired program (Execution engine + skill library). **Evo. search**: raw evolutionary-search result (best validated candidate), or “–” if Evolutionary Search was not run for that task. **Aspire**: per-task winner selected on a held-out validation set (seeds 66–80) between the repaired program and Evolutionary Search.

Task	w/o Engine & Evo	w/o Evo	Evo. search	Aspire
<i>libero-object</i>				
Alphabet soup	0.00	1.00	–	1.00
BBQ sauce	0.88	1.00	–	1.00
Butter	0.78	0.98	–	0.98
Chocolate pudding	0.00	1.00	–	1.00
Cream cheese	0.00	0.72	0.74	0.72
Ketchup	0.00	0.96	–	0.96
Milk	0.00	1.00	–	1.00
Orange juice	0.00	1.00	–	1.00
Salad dressing	0.00	0.92	–	0.92
Tomato sauce	0.04	0.96	–	0.96
<i>Mean</i>	<i>0.17</i>	<i>0.95</i>	–	<i>0.95</i>
<i>libero-goal</i>				
Open middle drawer	0.00	0.00	0.00	0.00
Open top drawer+bowl	0.00	0.50	0.46	0.46
Push plate to stove	0.26	0.58	0.76	0.76
Bowl on plate	0.00	0.88	0.82	0.88
Bowl on stove	0.00	0.00	0.00	0.00
Bowl on cabinet top	0.00	0.00	0.04	0.04
Cream cheese in bowl	0.00	0.82	0.74	0.74
Wine on rack	0.00	0.74	0.86	0.86
Wine on cabinet top	0.00	0.14	0.72	0.14
Turn on stove	0.24	0.52	0.66	0.66
<i>Mean</i>	<i>0.05</i>	<i>0.42</i>	<i>0.51⁽ⁿ⁼¹⁰⁾</i>	<i>0.45</i>
<i>libero-spatial</i>				
Betw. plate+ramekin	0.14	0.08	0.84	0.08
From table center	0.26	0.46	0.64	0.64
In top drawer	0.04	0.96	–	0.96
Next to cookie box	0.00	0.94	–	0.94
Next to plate	0.08	0.70	0.92	0.70
Next to ramekin	0.00	0.60	0.42	0.60
On cookie box	0.00	0.06	0.70	0.70
On ramekin	0.20	0.10	0.46	0.46
On stove	0.00	0.32	0.42	0.42
On wooden cabinet	0.13	0.46	0.98	0.46
<i>Mean</i>	<i>0.04</i>	<i>0.47</i>	<i>0.67⁽ⁿ⁼⁸⁾</i>	<i>0.60</i>
<i>Overall mean</i>	<i>0.09</i>	<i>0.61</i>	<i>0.59⁽ⁿ⁼¹⁹⁾</i>	<i>0.67</i>

D.3. Evolutionary Search Progress Per-Task Breakdown

Table 9: Evolutionary search progress on selected LIBERO-Pro tasks. Each column shows the held-out success rate (seeds 1–50) of the best candidate at that iteration. Blank cells indicate search terminated early.

Suite	Task	Iter 0	Iter 1	Iter 2	Iter 3	Iter 4
goal_swap	Bowl → plate	0.62	0.60	0.60	0.18	0.86
goal_swap	Wine bottle → rack	0.40	0.76	0.74		
goal_swap	Bowl → stove	0.62	0.82			
goal_task	Push plate → stove	0.00	0.80			
spatial_swap	Bowl on cabinet → plate	0.16	0.38	0.38	0.46	0.30
spatial_swap	Bowl next to cookie → plate	0.16	0.20	0.04	0.04	0.40
spatial_task	Bowl on cookie → plate	0.26	0.04	0.70		
spatial_task	Bowl on ramekin → plate	0.02	0.00	0.22	0.36	0.18

E. Agent Pipeline Skills

E.1. Skill Library Prompts

Actor agents report candidate repairs using a structured findings schema: the observed failure mode, the validated repair, transferable patterns, task-specific quirks, and the validation success rate on debug configurations. The coordinator prompt then audits each candidate for reusability, checks API-policy compliance, and admits only validated repairs that are likely to transfer beyond the originating task. Parallel actor agents are instructed to write task-level repairs and findings, while the coordinator serializes skill admission to avoid conflicting library writes. See subagent skills in E.3 and E.4 for exact formats. See code release for all agent skill files.

E.2. Agent System Prompt and Persistent Memory

CLAUDE.md

```
# CLAUDE.md
This file is the constitution for Claude Code working in this repository. Read it fully before doing
anything.

## STARTUP: Read Project Memory First

At the start of every session, immediately read ./claude/memory/MEMORY.md before doing anything else.
It contains project vision, key conventions, and the skills index.

## What Is ASPIRE

ASPIRE is a framework where LLMs write Python code to control robots through a curated tool API. Code is
executed in a sandboxed simulator, the simulator verifies success, and the agent improves through
failure diagnosis and skill accumulation.

## FORBIDDEN APIs

These APIs access simulator ground truth and are STRICTLY FORBIDDEN in all fix code, debug scripts, and
skill implementations. Using them invalidates benchmark results, as they don't transfer to real
robots.

'''
[FORBIDDEN] env.handle.env.sim - no MuJoCo sim object access
[FORBIDDEN] sim.data.body_xpos - no ground-truth object positions
... (rest of APIs omitted)
'''

Rule of thumb: If a real robot with a camera could do it, it's allowed. If it reads the physics engine
's internal state, it's forbidden.

Also forbidden: Reading simulator asset files (.bddl, .xml, .urdf, MuJoCo model files) to infer
object geometry, success predicates, or scene structure. Treat these as inaccessible - diagnose
purely from observations and traces.

## ALLOWED APIs

Full source: integrations/franka/libero_reduced_skill_library.py (and libero_reduced.py for base class
). Read the source for exact signatures, return types, and edge cases.

'''
[ALLOWED] get_observation() - RGB, depth, intrinsics, extrinsics, robot state
[ALLOWED] segment_sam3_text_prompt(rgb, text) - SAM3 text-prompted segmentation
... (rest of APIs omitted)
'''

You are encouraged to design new helper functions (e.g. localize_object(), make_topdown_quat())
when you find yourself repeating patterns. Add them to the relevant skill in ./claude/skills/ so
future sessions can reuse them.

## Skills
```

All skills live in `./claude/skills/<name>/SKILL.md` and are auto-discovered. Read the relevant skill before starting work.

```
| Skill | What it covers |
|---|---|
... (skills list omitted)
| **add more as needed** | **document new patterns here as you discover them** |
```

Log Everything

Append to `./docs/logs/YYYY-MM-DD.md` after any significant work: experiments run, fixes tried, skills updated, results observed. Be detailed - include trial numbers, success rates, error messages, file paths.

MEMORY.md

ASPIRE Project Memory

What We're Building

A Self-Improving Coding Agent for robot manipulation. The agent debugs its own failures, accumulates reusable skills, and improves its perception tools.

- Foundation: Rich debugging playground (traces, REPL)
- Layer 1: Auto-growing skill library
- Layer 2: Program search (Pass@K parallel search)

Key Conventions

- ****NEVER** git commit unless user explicitly asks**, do NOT auto-commit after skill updates or log writes
- ****NEVER** read sim asset files** (`.bddl`, `.xml`, `.urdf`, MuJoCo models), diagnose from observations and traces only
- ****Always** write reusable analysis as scripts** (not inline `python3 -c "..."`) and reference them in the relevant skill
 - ... (rest of conventions omitted)

Skills Available

... (skills list omitted)

api-reference.md

Observation and Perception

```
| Function | Returns |
|---|---|
| 'get_observation()' | RGB image, depth image, camera intrinsics, camera pose, and robot state. |
| 'segment_text_prompt(rgb, text_prompt)' | A list of masks with bounding boxes, scores, and labels. |
| 'segment_point_prompt(rgb, point_coords)' | A list of masks from point-conditioned segmentation. |
| 'point_prompt_vlm(image, text_prompt)' | A pixel coordinate for the prompted object or region. |
```

Grasp Planning and IK

```
| Function | Returns |
|---|---|
| 'plan_grasp(depth, intrinsics, segmentation)' | Candidate grasp poses and scores in the camera frame. |
| 'select_top_down_grasp(poses, scores, ...)' | The best top-down grasp transform and score. |
| 'solve_ik(position, quaternion_wxyz)' | A 7-DoF joint configuration, or failure if unreachable. |
```

Camera-to-world conversion is required after grasp planning.

```
'''python
camera_to_world = obs["agentview"]["pose_mat"]
grasp_world = camera_to_world @ grasp_camera
'''
```

Motion Execution

Function	Returns
'move_to_joints(joints)'	Blocking joint-space motion.
'open_gripper()' / 'close_gripper()'	Gripper command.
'goto_home_joint_position()'	Move to a predefined safe home configuration.

Geometry Utilities

Function	Description
'decompose_transform(T)'	Convert a 4x4 transform into position and quaternion.
'mask_to_world_points(mask, depth, intrinsics, extrinsics)'	Project a 2D mask into a world-frame point cloud.
... (rest omitted)	

Task Language

```

'''python
env.task_language
'''

```

The runtime task language is the reliable source of the actual goal, especially when benchmark filenames do not match remapped task instructions.

Success Conditions

Placement Type	Tolerance
Regular object placement	Tight XY tolerance plus contact.
Site-object placement	Larger site-local tolerance.

Scene Coordinate Frames

Use observation-derived depth projection for object positions and avoid hard-coded scene coordinates. Reference frame constants are environment-specific and should not be used as object localization shortcuts.

'... (benchmark-specific table of scene constants omitted) ...'

Output Directory Structure

```

'''text
<OUTPUT_DIR>/
  <suite>/<task>/<run_id>/
    run/
      <done_flag>
      trial_<seed>_<attempt>_reward_<score>_taskcompleted_<flag>/
      code.py
      trace.json
      keyframes/
      video_combined.mp4
      visual_feedback.png
      summary.txt
      all_responses.json
'''

```

The trial folder name records the seed, attempt index, reward, and task-completion flag. The attempt index distinguishes clean exits from crashes, and 'reward' is the scalar task score.

Trace Format

```

'''json
{
  "step": 5,
  "function": "segment_text_prompt",

```

```

"args": {"rgb_shape": [512, 800, 3], "text_prompt": "bowl"},
"duration_ms": 1463.8,
"result": {
  "num_masks": 200,
  "mask_0_score": 0.917,
  "mask_0_bbox": [369.2, 257.1, 438.4, 310.5],
  "mask_0_area_pct": 0.68
},
"keyframe_saved": true
}
'''

```

Key Source Files

File	Role
'<batch_runner>'	Runs suites, tasks, and model configurations.
'<trial_loop>'	Resets the environment, executes generated code, and saves artifacts.
'<single_trial_runner>'	Replays saved code or runs a fresh trial.
'<single_trial_analyzer>'	Performs structured trace analysis for one trial.
'<multi_candidate_trace_analyzer>'	Performs per-candidate trace analysis for evolutionary search runs.
'<progress_scanner>'	Aggregates success rates across an output directory.
'<robot_api_source>'	Defines the robot control and geometry API.
'<trace_logger_source>'	Records API calls, return values, timings, and keyframes.
'<evaluation_config>'	Defines benchmark and replay configuration.

E.3. ASPIRE Fix Loop Skills

coordinator.md

```

> Purpose: Coordinate repair of failed robot trials from a baseline policy and validate finalized
  fixes on a held-out seed set.
> Baseline results: '<BASELINE_OUTPUT_DIR>'
> Progress tracker: '<PROGRESS_FILE>'
> Subagent prompt: '<SUBAGENT_PROMPT_TEMPLATE>'

```

Initialization: Verify Perception Services

Before dispatching subagents, confirm that all required perception services are running. A service-specific health check should return the expected "up" status.

If any service is down, start it before proceeding. Subagents may fail silently without these dependencies.

Main Loop

```

'''text
read progress → assign free devices → dispatch subagents → go idle
on notification: update progress/skills + redispatch device --
'''

```

One task is handled by one subagent on one compute device. Repeat until no eligible tasks remain.

Coordinator Rules

- Dispatch subagents rather than debugging directly.** The coordinator assigns work, waits for completion notifications, updates shared knowledge, and dispatches the next task.
- Go idle after dispatching.** Completion notifications indicate when a device is free; avoid polling or intervening while subagents are running.
- Keep available devices occupied.** Any free device should receive the next pending task.
- Do not inspect task-specific debugging artifacts.** Files such as traces, generated programs, summaries, or keyframes are the subagent's responsibility. The coordinator may read the subagent's final findings when updating shared skills.

5. ****Never re-dispatch completed tasks.**** Completed validation tasks must not be run again, because duplicate trial directories can corrupt aggregate results.

Workflow

1. Read the Progress Tracker

```
'''bash
cat <PROGRESS_FILE>
'''
```

Identify tasks marked 'pending' or 'stage1-complete'.

2. Check Free Compute Devices

Assign tasks only to devices that are free. Never assign two tasks to the same device.

3. Dispatch One Subagent per Free Device

```
'''python
Agent(
    description="Fix loop: <suite>/<task>",
    subagent_type="general-purpose",
    prompt=<filled_subagent_prompt>,
    run_in_background=True,
)
'''
```

Send all dispatches together, then stop. The coordinator resumes when completion notifications arrive.

4. On Completion: Redispatch and Update Skills

When a subagent completes, its result should include the device identifier. The coordinator then:

1. Reads the progress tracker.
2. Dispatches the next eligible task on the freed device.
3. Reads the completed task's 'findings.md'.
4. Promotes generalizable patterns into the shared skill library.
5. Goes idle again.

If the result does not include a device identifier, query the device monitor to infer which device is free.

5. Update Shared Skills

After each completion, read:

```
'''text
<BASELINE_OUTPUT_DIR>/<SUITE>/<TASK>/findings.md
'''
```

Promote only generalizable patterns to the skill library.

'... (skill-library category table omitted) ...'

Validation Runs

For pure validation runs where final repair code already exists, direct scripts are more reliable than autonomous subagents. The validation script should run the held-out seed set exactly once and write to the official output directory.

'... (task-specific validation script, log parsing, and environment variables omitted) ...'

Important rule: pass only the validation base directory if the replay script appends '<suite>/<task>' internally.

Suite Name Collision Warning

Some benchmark suites may contain identical task names under different perturbation conditions. Always include the full suite name when briefing a subagent.

Perturbation Notes

- Position-perturbation suites randomize object poses by seed; perception should handle this naturally.
- Language-perturbation suites remap the task goal; the executable task instruction should be read from the runtime environment rather than inferred from a filename.

subagent.md

The following prompt is filled with one suite, task, and device assignment before being passed to a background subagent.

```
'''text
```

Task Assignment

```
SUITE: <SUITE_NAME>
```

```
TASK: <TASK_NAME>
```

```
DEVICE: <DEVICE_ID>
```

```
Working directory: <PROJECT_ROOT>
```

```
'''
```

Evaluation-Set Lockout

The held-out validation seeds are locked during debugging.

- During Stage 1, replay only the debugging seeds.
- Do not replay validation seeds for testing, tuning, temporary experiments, or alternate code versions.
- Stage 2 uses the validation seeds exactly once, after 'fix_code.py' is finalized, and writes to the official validation output directory.
- Stage 2 must run to completion before the subagent returns.

Violating this rule invalidates the benchmark result.

Role

You are a task-level repair subagent for a robotics benchmark. Your job is to diagnose failed robot trials for one task, write a generalizable repair program, validate it according to the protocol, and report concise results.

Context

The system asks language models to write Python control programs for a robot arm through a curated perception and manipulation API. Programs run in simulation, and failures are diagnosed through traces, summaries, and visual keyframes. The baseline policy has already run a set of debugging seeds ; your job is to repair the task program without using simulator ground truth.

Forbidden APIs

The repair program must not access simulator internals, ground-truth object states, hidden task predicates , asset files, or low-level physics state. Examples include:

```
'''text
simulator body positions
site positions
'... (additional forbidden APIs omitted) ...'
'''
```

These APIs would not transfer to a real robot and invalidate the result.

Allowed APIs

The repair code may use only observation-derived perception, manipulation primitives, and standard numerical computation:

```

'''text
get_observation()
segment_text_prompt(rgb, text)
'... (additional allowed APIs omitted) ...'
'''

```

Always use the project virtual environment or pinned runtime, not the system Python.

Before any replay, verify that the required perception services are running.

Stage 1: Debug on Non-Held-Out Seeds

Step 0: Check Whether Repair Code Already Exists

If it exists, skip Stage 1 and check whether Stage 2 has already completed.

Step 1: Fast Path from Successful Baseline Trials

First, check whether any baseline debugging seed already succeeded:

If successful trials exist:

1. Read the successful generated program.
2. Extract the final code block if multiple attempts are present.
3. Test it on a small number of failed debugging seeds.
4. If it succeeds, save it as the task-level 'fix_code.py' and proceed to Stage 2.
5. If it fails, continue to the full debug loop.

If no debugging seeds succeeded, continue directly to the full debug loop.

```

'... (exact replay command and temporary output paths omitted) ...'

```

Step 2: Diagnose Failed Debugging Seeds

For each failed seed, inspect the last attempt for that seed:

- 'trace.json': function calls, return values, perception failures, IK failures, and gripper signals.
- 'code.py': the baseline strategy.
- 'keyframes/': visual observations at perception steps.
- 'summary.txt': stdout, stderr, and reward.

Interpretation examples:

```

'''text
large gripper width after closing → likely object grasped
medium gripper width → marginal grasp
small gripper width → likely air grasp
zero masks → poor perception prompt or occlusion
missing IK solution → target outside reachable workspace
early crash → inspect stdout/stderr summary
'''

```

Step 3: Write and Test Repair Code

Before writing code, read the relevant skill-library entries for localization, grasping, transport, and manipulation patterns.

Write 'fix.py' from the diagnosis and replay it on debugging seeds only.

If the repair fails, read the new trace and iterate. A live inspection REPL may be used for observation-derived checks such as segmentation counts.

```

'... (interactive REPL command and shell-specific buffering notes omitted) ...'

```

Hard limit: at most three replay attempts per debugging seed. Any run against a seed counts as an attempt, regardless of output directory or code version. After the cap, write a 'BLOCKED.md' report and move

```

on.

‘BLOCKED.md’ format:

‘... (blocked-seed report template omitted) ...’

#### Step 4: Synthesize the Task-Level Repair Program

After debugging all seeds, synthesize one generalizable ‘fix_code.py’. The code must not contain seed-specific branches.

Save it to ‘<BASELINE_OUTPUT_DIR>/<SUITE>/<TASK>/fix_code.py’. Also write <BASELINE_OUTPUT_DIR>/<SUITE>/<TASK>/findings.md’.

Use this findings format:

‘‘‘markdown
## Task: <SUITE> / <TASK>

### Root Cause(s)
- <concise description of each failure mode>

### What Fixed It
- <change that resolved each root cause>

### Perception Prompts That Worked
| Object | Prompts (priority order) | Notes |
|---|---|---|
| <object> | "<prompt1>", "<prompt2>" | <caveats> |

### Generalizable Patterns
- <patterns likely to apply beyond this task>

### Task-Specific Quirks
- <details specific to this task>

### Stage 2 Success Rate
<N>/<TOTAL_VALIDATION_SEEDS>
‘‘‘

If all debugging seeds are blocked, skip Stage 2 for this task.

#### Stage 2: Validate on Held-Out Seeds

Stage 2 is a one-shot validation operation. Do not debug Stage 2 runs.

Before running, check whether validation has already completed:

If enough unique validation seeds are already present, do not re-run Stage 2. Report the existing results and return. Otherwise, run validation sequentially:

‘... (validation loop, log parsing, success counting, and environment variables omitted) ...’

Verify that outputs landed in the expected directory before reporting results.

### Final Action

Run the shared progress-generation script as the last action. It should update the progress tracker atomically and be safe under concurrent subagent execution.

### Return Format

Return only a brief structured summary:

‘‘‘text
SUITE: <SUITE>
TASK: <TASK>
DEVICE: <DEVICE>

```

```

Stage 1: fix_code.py written: yes/no
  Seeds fixed: <count or list>
  Seeds blocked: <count and one-line cause>

Stage 2: <N>/<TOTAL_VALIDATION_SEEDS>
  Output dir: <VALIDATION_OUTPUT_BASE>/<SUITE>/<TASK>/

Key findings:
- <generalizable root cause and fix>
- <useful perception prompt, if notable>
- <pattern that should inform future tasks>
'''

```

E.4. ASPIRE Evolutionary Search Skills

Evolutionary search (Section 2.3) maintains a persistent task-analysis document A_i that is carried across rounds. A_i consists of (i) a per-task scene description populated once from an initial snapshot (object shapes, goal geometry, obstacles, blocked approach directions), (ii) running hypotheses and the candidate metadata that tests them, and (iii) a ledger of *eliminated* directions (through testing) and *blocked*, *untested* directions (failed due to a workspace constraint). UPDATEANALYSIS rewrites A_i each round from the new traces, keyframes, and library retrievals. This document is what lets a later round avoid re-testing an eliminated branch while still retrying blocked branches once a new technique (e.g. a wrist rotation) becomes available.

```

coordinator.md

> What: Run Evolutionary search iterative debugging with multiple candidates per iteration on a list
  of low-performing tasks.
> Why: Intensive debugging for difficult tasks; multi-candidate search can find strategies the single-
  pass actor misses.
> Subagent template: '<EVOSEARCH_SUBAGENT_PROMPT_TEMPLATE>'

### Initialization: Verify Perception Services

Before dispatching subagents, verify that all required perception services are running.
All services must be up before dispatching.

'... (service health-check command and machine-specific ports omitted) ...'

### Directory Layout

'''text
<EVOSEARCH_OUTPUT_DIR>/
  <suite>/<task>/<run_id>/
    task_analysis.md
    iter_00/ iter_01/ ...
      candidate_A/code.py
      iter_summary.json
  <suite>/<task>/
    evosearch_best_code.py
    findings.md

<VALIDATION_OUTPUT_DIR>/
  <suite>/<task>/... (rest omitted)
'''

### Task List

Run '<progress_generation_script>' to verify exact task names before dispatch.

```

Tasks below the chosen baseline success threshold are candidates for repair.

‘... (experiment-specific task ordering and threshold table omitted) ...‘

The Loop

```

“‘text
read progress → assign free devices → dispatch subagents → go idle
on notification: redispatch freed device to next pending task -|
“‘

```

One task = one subagent = one compute device.

The subagent runs iterations on the debug set and then runs Stage 2 on the held-out set before returning. When the coordinator gets a completion notification, the device is already free; dispatch the next task.

Coordinator Rules

1. **Dispatch subagents -- never run iterations yourself.**
2. **Go idle after dispatching.** You will be notified when a subagent finishes.
3. **Keep available devices occupied.**
4. **On each notification: check which device freed up, dispatch next pending task to it.**
5. **Never re-dispatch a done task.** ‘done‘ means Stage 2 complete with enough unique held-out seeds on disk.

Workflow

1. Check Progress

Check which tasks have a saved ‘evosearch_best_code.py‘ and how many held-out seeds have already been evaluated.

‘... (progress-check shell loop omitted) ...‘

2. Check Free Compute Devices

Assign tasks only to devices that show no active jobs.

‘... (device-monitoring command omitted) ...‘

3. Dispatch Subagents

Use the stronger reasoning model selected for the experiment.

```

“‘python
Agent(
    description="Evolutionary search actor: <suite_short>/<task_short> device<N>",
    subagent_type="general-purpose",
    model="<HIGH_CAPACITY_MODEL>",
    prompt=<filled_evosearch_subagent_prompt>,
    run_in_background=True,
)
“‘

```

Send all dispatches in one message, one per free device, then stop.

4. On Each Completion: Redispatch

When a subagent notification arrives, it has already completed both Stage 1 and Stage 2.

Read ‘<EVOSEARCH_OUTPUT_DIR>/<SUITE>/<TASK>/findings.md‘, note Stage 1 and Stage 2 rates, check free devices, dispatch the next pending task, and go idle.

Stopping Criteria Reference

Subagents stop Stage 1 when the best candidate reaches the success threshold on debug seeds, the maximum number of iterations is reached, or the task is blocked.

If Stage 1 is blocked, the subagent skips Stage 2 and returns immediately.

```
subagent.md
```

The following prompt is filled with one suite, task, device assignment, baseline rate, and output location before dispatch.

```

'''text
## Task Assignment

SUITE: <SUITE>
TASK: <TASK>
DEVICE: <DEVICE_ID>
TASKSHORT: <SHORT_UNIQUE_NAME_FOR_LOGS>
BASELINE_RATE: <BASELINE_RATE>
EVOSEARCH_DIR: <EVOSEARCH_OUTPUT_DIR>
EVALDIR: <VALIDATION_OUTPUT_DIR>

Working directory: <PROJECT_ROOT>
'''

### Evaluation-Set Lockout During Iterations

The held-out evaluation seeds must not be run during iterations.
All iteration evaluations use only the debug seed set.
Stage 2 uses the held-out seeds after iterations converge and the final code is chosen.
Violation invalidates the benchmark.

### Role

You are a debugging subagent.
Your job is to run evolutionary search iterative debugging on debug seeds until convergence or plateau,
save the best code, run Stage 2 on held-out seeds, and return a structured findings report with both
Stage 1 and Stage 2 results.
Use the project virtual environment or pinned runtime, never the system Python.

### Context

The actor pipeline achieved '$BASELINE_RATE' on this task.
Your target is to exceed that significantly, ideally reaching the Stage 1 success threshold on debug seeds.

Read the skill library before writing candidates.
Verify that the required perception services are running before replay.

'... (experiment-specific config path and service-port checks omitted) ...'

### Forbidden APIs

'''text
simulator body positions
site positions
'... (rest of forbidden APIs omitted)
'''

### Stage 1: Evolutionary Search Iterations on Debug Seeds

#### Step 0: Check Whether Best Code Already Exists

If 'evosearch_best_code.py' exists, skip to the return step.

#### Step 0b: Task-Language Remapping Check

The task name might not match the actual goal.
Always check and print the runtime task language.
If it differs from '$TASK', the runtime task language is ground truth; base all strategy on it and record
the actual language in 'task_analysis.md'.

#### Step 1: Read Skills and Existing Baseline

```

Read skill-library entries for grasping, localization, transport, manipulation, and any task-specific companion skills.
 Also read existing repair code if it exists.
 If baseline code is found, record its path in 'task_analysis.md' and seed it as 'candidate_A' verbatim.

'... (skill-file reads and baseline fallback command omitted) ...'

Step 2: Create Run Directory and Scene Snapshot

Create a timestamped run directory under '<EVOSEARCH_OUTPUT_DIR>/<SUITE>/<TASK>/'.
 Before writing candidates, run a scene snapshot on one debug seed and inspect the wide and wrist-view images.
 Populate 'task_analysis.md' with object shape, grasp strategy, goal geometry, placement strategy, hypotheses for the first iteration, and an iteration log.

'... (snapshot command and task_analysis.md template omitted) ...'

Step 3: Write K Candidates for the First Iteration

Always seed 'candidate_A' from the existing baseline repair code if it exists.
 Each candidate must test a distinct hypothesis.
 No two candidates should fail at the same stage for the same reason.
 Each candidate should include a docstring describing the hypothesis, how it differs from prior candidates, and the expected failure mode if wrong.

Step 4: Evaluate the Iteration

Evaluate all candidates on the fixed debug seed set.
 Use the same debug seeds across all iterations so cross-iteration comparison is fair.
 Read the leaderboard after completion and analyze traces.

'... (multi-candidate evaluation command, leaderboard parser, and trace analyzer command omitted) ...'

Step 5: Iterate

Stop Stage 1 when any stopping criterion is met.
 When progress stalls, use the remaining iteration budget to explore structurally new approaches rather than stopping early.
 A plateau on the current approach family means the current family may be wrong, not that the task is unsolvable.

The debug seeds are a small, potentially unrepresentative sample.
 The code will be evaluated on unseen held-out seeds; write for those, not for the debug seeds.
 Prefer strategies that work for mechanistic reasons over strategies that exploit patterns specific to debug-seed failures.
 Avoid hard-coded thresholds, image-region masks, or offsets derived by fitting to observed debug-seed failures.

Per iteration:

1. Update 'task_analysis.md' with new geometry from keyframes.
2. Append an iteration log with the leaderboard, eliminated hypotheses, and open questions.
3. Write K new candidates seeded from the top survivors of the current iteration.
4. Evaluate all candidates on the same debug seed set.
5. Read trace-analysis output and watch for arm blocking, gripper-width signals, perception failures, and IK failures.

Step 6: Save Best Code, Run Stage 2, Write Findings

Save the best final candidate as '<EVOSEARCH_OUTPUT_DIR>/<SUITE>/<TASK>/evosearch_best_code.py'.
 If the best candidate does not beat the baseline candidate on debug seeds, fall back to the stronger baseline so Stage 2 uses the best available code.
 If all approaches fail, mark the task blocked and skip Stage 2.

'... (best-candidate sanity check and copy commands omitted) ...'

Stage 2: Evaluate Best Code on Held-Out Seeds

```

Run the selected best code on each held-out seed exactly once.
Do not use Stage 2 results to revise the code.
Poll until the validation loop completes, then read final success counts.

'... (detached validation script, polling commands, and final count commands omitted) ...'

### Findings Report

Write 'EVOSEARCH_OUTPUT_DIR/<SUITE>/<TASK>/findings.md':

'''markdown
## Task: <SUITE> / <TASK>
## Baseline rate: <BASELINE_RATE>
## Best evosearch rate on debug seeds: <N>/<DEBUG_TOTAL> (<pct>%)
## Stage 2 on held-out seeds: <N>/<HELD_OUT_TOTAL> (<pct>%)

### Evolutionary Search Run
- Run dir: <RUN_DIR>
- Iterations completed: <N>
- Stopping reason: <solved|max_iterations|blocked>

### What Fixed It vs Baseline
- <key strategy change>

### Perception Prompts That Worked
| Object | Prompts | Notes |
|---|---|---|

### Failure Modes Eliminated
- <approach that failed and why>

### Generalizable Patterns
- <anything worth adding to the skill library>

### Skill Library Updates Made
- <if any skills were updated during this run>
'''

### Return Format

'''text
SUITE: <suite>
TASK: <task>
DEVICE: <N>
Baseline rate: <from actor pipeline>

Stage 1:
  Best candidate: <candidate_X> at iter_NN
  Best pass rate: <N>/<DEBUG_TOTAL> (<pct>%)
  Stopping reason: <solved|max_iterations|blocked>
  Iterations run: <N>
  Run dir: <RUN_DIR>

Stage 2: <N>/<HELD_OUT_TOTAL> (<pct>%)

Key findings:
- <strategy that worked vs baseline failure mode>
- <perception prompts or grasp parameters>
- <generalizable pattern for the skill library>
'''

```

E.5. Initial Skill Library Templates

```
grasp initial SKILL.md
```

```

---
name: grasp
description: Structural template for pick-and-place programs - topdown-quat construction, pre-grasp
  approach, lower-close sequence, lift-transport-place pattern. No task-specific parameters; those grow
  through experiment.
---

# Grasp - Structural Template for Pick-and-Place

> **Purpose:** Code skeletons that every pick-and-place task reuses. No task-specific z-offsets or yaws -
  those go in the per-task 'task_code.py'.
> **Ownership:** Coordinator adds generalizable patterns. Subagents do not edit this file.

---

## Top-Down Quaternion

```python
import numpy as np
from scipy.spatial.transform import Rotation

def make_topdown_quat(yaw_deg=0):
 """Build a top-down end-effector quaternion (xyzw→wxyz convention used by solve_ik)."""
 R = Rotation.from_euler('z', yaw_deg, degrees=True).as_matrix() @ \
 np.array([[1, 0, 0], [0, -1, 0], [0, 0, -1]])
 q = Rotation.from_matrix(R).as_quat() # scipy returns xyzw
 return np.array([q[3], q[0], q[1], q[2]]) # reorder to wxyz
```

**Yaw selection:** default 0. If the grasp fails because the gripper fingers collide with a neighboring
  object, try 90 or 45 degrees.

---

## Standard Pick-and-Place Skeleton

```python
Assume: obj_center, obj_mask, tgt_center, tgt_pts already localized (see localize skill)
Assume: depth, K (intrinsic), E (camera extrinsic pose_mat) from get_observation()

1. Plan grasp via GraspNet on object mask
2. Pre-grasp approach: gripper above object, open
3. Lower to grasp pose, close gripper
4. Lift (avoid dragging object sideways into neighbors)
5. Move above target (same z as lift)
6. Lower to release height (surface_z + margin)
7. Release
8. Let physics settle (important for drop-success reading)
... (skeleton code omitted)
```

---

## Why 'select_top_down_grasp'?

'plan_grasp' returns grasps in camera frame, often angled or side-approach. We want top-down for table-top
  pick-and-place. The selector transforms to world frame and filters for top-down direction (z-axis
  roughly anti-aligned with gravity). When it returns None (no valid top-down), fall back to 'E @
  grasp_poses[grasp_scores.argmax()]'.

---

## When to NOT Use This Template

```

```

- **Shelves / drawers / microwaves** - use 'manipulation' skill, not this.
- **Stacking** - need stacking-specific surface_z logic (stack on top of existing object, not table).
- **Narrow containers (baskets with walls)** - need yaw that avoids wall collision; release height above
  basket rim.

---

## Per-Object Grasp Registry (grows per chunk)

Discovered per-object grasp parameters. Coordinator adds entries from 'findings.md'.

Object	z_offset	yaw	Notes	First Seen In

<!-- PLACEHOLDER: Registry starts empty. Populated by coordinator after each chunk. -->
<!-- Format: object name | z_offset from OBB top | yaw_deg if non-zero | why | task that validated it -->

---

## Anti-Patterns

- 'set_joint_qpos' to "teleport" the gripper into grasp pose - forbidden.
- Hardcoded grasp positions from task BDDL - forbidden (reading .bddl is banned).
- Lifting to fixed absolute z (e.g., z=0.9) - use 'grasp_pos[2] + 0.15' to be scene-relative.
- Skipping the pre-grasp 'z_approach=0.15' - causes gripper collisions with neighbors.
- Single-try 'select_top_down_grasp' without 'argmax' fallback - some scenes have no valid top-down grasp.

---

## Debugging Grasp Failures

Symptom	Likely Cause	Check
Gripper closes on air	z too high (didn't descend to surface)	Print 'grasp_pos[2]' vs. 'surface_z';
use OBB 'extent[2]/2' below top		
Grasp succeeds but drops during lift	Gripper half-closed on slippery/elongated object	Try
perpendicular yaw (90 degrees) or fingers-along-long-axis approach		
'plan_grasp' returns empty	Mask too small or below grasp threshold	Dilate mask; log 'mask.sum()' -
should be >200 px		
'solve_ik' returns None	Pose out of workspace	Lower z by 5cm, re-check x/y reachable from robot base

```

localize initial SKILL.md

```

---
name: localize
description: Object localization via SAM3 - prompting strategies, multi-prompt fallback, 3D centroid
  extraction, disambiguation, per-object prompt registry. Grows through experiment.
---

# Localize - SAM3 Prompting & Object Localization

> **Purpose:** This skill tracks working SAM3 prompts per object + the standard localization helpers.
> **Ownership:** Coordinator adds entries to the prompt registry after reviewing subagent findings.
  Subagents do NOT edit this file.
---

## Standard Localization Helper

```python
import numpy as np

def localize_object(rgb, depth, K, E, prompts):
 """Try prompts in order, return (center, pts, mask) for first hit with >=10 points."""
 depth_img = depth[:, :, 0] if len(depth.shape) == 3 else depth
 if isinstance(prompts, str):

```

```

 prompts = [prompts]
 for prompt in prompts:
 masks = segment_sam3_text_prompt(rgb, prompt)
 if not masks:
 continue
 best = max(masks, key=lambda d: d["score"])
 mask = best["mask"].astype(np.uint8)
 pts = mask_to_world_points(mask, depth_img, K, E)
 if pts is None or len(pts) < 10:
 continue
 center = get_oriented_bounding_box_from_3d_points(pts)["center"]
 return center, pts, mask
 return None, None, None
'''

Usage:
'''python
obs = get_observation()
cam = obs["agentview"]
center, pts, mask = localize_object(
 cam["images"]["rgb"], cam["images"]["depth"],
 cam["intrinsic"], cam["pose_mat"],
 ["<specific prompt>", "<fallback prompt>"]
)
if center is None: raise RuntimeError("Object not found")
'''

Why OBB center over mean: 'get_oriented_bounding_box_from_3d_points' gives a more robust center for
elongated/occluded objects than 'pts.mean(axis=0)'.

Fallback: Molmo Pixel Grounding

When SAM3 returns no masks even for sensible prompts, try Molmo to get a pixel, then convert with SAM3
point prompt:

'''python
def localize_via_molmo(rgb, depth, K, E, phrase):
 px = point_prompt_molmo(rgb, phrase) # (x, y) or None
 if px is None: return None, None, None
 masks = segment_sam3_point_prompt(rgb, [px])
 if not masks: return None, None, None
 mask = masks[0]["mask"].astype(np.uint8)
 depth_img = depth[:, :, 0] if len(depth.shape) == 3 else depth
 pts = mask_to_world_points(mask, depth_img, K, E)
 if pts is None or len(pts) < 10: return None, None, None
 return get_oriented_bounding_box_from_3d_points(pts)["center"], pts, mask
'''

Disambiguation: Two Similar Objects in Scene

When a scene contains two visually similar objects, 'max(score)' often picks the wrong one. Use bbox
pixel area or bbox y-position (upper = farther):

'''python
def select_by_smallest_bbox(masks, max_cy=None, min_area=50):
 """Pick mask with smallest bbox pixel area (most compact matching shape).
 max_cy: if set, only accept masks with bbox center-y below this row."""
 ... (implementation omitted)
'''

Key: 'area_pct' in SAM3 result dicts clamps to 99.00 for all masks - compute area from 'box' directly.

```

```

Disambiguation: By 3D Position (when two objects share prompts)

python
Example: "black bowl" returns both bowls, want the one at the front
... (implementation omitted)
python

Prompt Registry (grows per chunk)

Discovered working prompts, indexed by object. Coordinator appends entries after each chunk review.

Format per row: object | priority-ordered prompts | which task validated it | notes.

| Object | Working Prompts | First Seen In | Notes |
|---|---|---|---|

<!-- PLACEHOLDER: Registry starts empty. Populated by coordinator after each chunk. -->
<!-- Format: object name | prompts in priority order (quoted) | first task that validated | disambiguation
 notes if needed -->

How to Add a New Prompt (Coordinator Only)

When a subagent's 'findings.md' "Generalizable Patterns" section mentions a new working prompt:

1. **Verify generalizability** - would this prompt plausibly work for >=2 task types? If it's ultra-specific to one task, consider keeping it inline in 'task_code.py' instead of here.
2. **Append to registry** - one table row, priority-ordered list of prompts.
3. **Add a disambiguation note** only if the task exposed a specific confusion (e.g., two bowls in frame).
4. **Keep examples short** - a full code block per prompt is overkill. The helper pattern stays at the top of this file; the registry is just prompt strings.

Anti-Patterns (Do NOT Do These)

- 'body_xpos' for object localization - forbidden. If you're tempted, the perception is wrong; fix the prompt.
- Hardcoded pixel coordinates - always derive from SAM3 or Molmo.
- Caching object positions across seeds - re-localize every episode; objects are randomized.
- Using 'segment_sam3_text_prompt' mask dict without '.astype(np.uint8)' conversion - silent failures downstream.

```

#### manipulation initial SKILL.md

```

name: manipulation
description: Non-pick-and-place manipulation patterns - drawer opening/closing, knob/switch turning,
 pushing, articulated-object interactions. Grows with validated parameters per object type.

Manipulation - Articulated Objects & Non-Prehensile Actions

> **Purpose:** Tracks validated parameters for articulated manipulation. Approaches are discovered through
 experiment and recorded in the registry below.
> **Ownership:** Coordinator adds validated parameters per object-type. Subagents do not edit.

Examples of Task Types Requiring This Skill

- **Drawer open/close** - pull/push a drawer by its handle
- **Knob/switch turn** - rotate a stove knob or flip a switch to trigger a state change
- **Hinged door** - open a microwave, cabinet door, or similar hinged panel

```

```

- **Non-prehensile push** - slide an object to a target without grasping it

These require non-top-down approach angles, arc motions, or contact-without-grasp patterns.

Parameter Registry (grows per chunk)

Validated parameters per object / action. Coordinator adds after verifying from 'findings.md'.

| Object / Action | Parameters | First Seen In | Notes |
|---|---|---|---|

<!-- PLACEHOLDER: Registry starts empty. Populated by coordinator after each chunk. -->
<!-- Format: object/action | key parameters (distances, angles, offsets, approach direction) | first task
| notes on what failed before -->

Anti-Patterns

- 'set_joint_qpos' to force a drawer/door open - forbidden. Must use physical manipulation.
- Reading BDDL to get pull distances or knob angles - forbidden. Discover via OBB or trial-and-error.
- Forgetting to retreat after manipulation - arm stays inside drawer/microwave, blocks next perception.

```

#### transport initial SKILL.md

```

name: transport
description: Motion patterns for moving the end-effector between poses - multi-step waypoints, safe
transit, interpolated Cartesian moves, collision avoidance. Grows with discovered patterns.

Transport - End-Effector Motion Between Poses

> **Purpose:** Transit patterns after a grasp is achieved and before a place/release. Keeps the object
above clutter, avoids scene collisions, handles narrow workspaces.
> **Ownership:** Coordinator adds validated waypoint patterns. Subagents do not edit.

Basic Lift-Transit-Descend

``python
Precondition: object grasped (close_gripper done), grasp_pos known, quat known
Target: tgt_center (3D world coordinate), surface_z (target surface height)

Waypoint 1: lift straight up (avoid dragging)
lift_z = grasp_pos[2] + 0.15
joints = solve_ik([grasp_pos[0], grasp_pos[1], lift_z], quat.tolist())
if joints is not None: move_to_joints(joints)

Waypoint 2: move laterally at lift height (avoid hitting other objects on surface)
joints = solve_ik([tgt_center[0], tgt_center[1], lift_z], quat.tolist())
if joints is not None: move_to_joints(joints)

Waypoint 3: descend to release height
release_z = surface_z + 0.05
joints = solve_ik([tgt_center[0], tgt_center[1], release_z], quat.tolist())
if joints is not None: move_to_joints(joints)
``

Why 3 waypoints, not direct goto_pose: a direct move from grasp to release tends to take a diagonal
path that clips tall objects, basket walls, or the drawer opening. Three cardinal waypoints (up →
over → down) keeps the object in free air during transit.

```

**## Home Reset Between Subtasks**

For long-horizon tasks (LIBERO-Long-Pro), reset the arm between subtasks to a known-safe config:

```

'''python
def safe_reset_between_subtasks():
 open_gripper()
 goto_home_joint_position()
 for _ in range(3): get_observation() # let physics settle
'''

```

**\*\*When to call:\*\*** after releasing subtask-1's object, before starting subtask-2's localization. Prevents subtask-2 from trying to perceive through the arm in frame.

---

**## Far-Workspace Safety: 'solve\_ik' over 'goto\_pose'**

'goto\_pose' is unreliable in the far workspace (e.g., back of counter, far side of table). Use 'solve\_ik' + 'move\_to\_joints' for these transits:

```

'''python
UNRELIABLE (far workspace):
goto_pose(far_target_pos, quat)

RELIABLE:
joints = solve_ik(far_target_pos.tolist(), quat.tolist())
if joints is None:
 # Try slightly closer z or x
 fallback = far_target_pos.copy(); fallback[2] -= 0.02
 joints = solve_ik(fallback.tolist(), quat.tolist())
if joints is not None:
 move_to_joints(joints)
else:
 raise RuntimeError(f"IK infeasible at {far_target_pos}")
'''

```

---

**## Waypoint Registry (grows per chunk)**

Validated multi-step transit patterns for scenes where direct motion fails.

Scenario	Pattern	First Seen In	Notes
---	---	---	---

<!-- PLACEHOLDER: Registry starts empty. Populated by coordinator after each chunk. -->

<!-- Format: scenario description | waypoint pattern used | first task | why default 3-waypoint failed -->

---

**## Anti-Patterns**

- Single 'goto\_pose' across the whole scene - clips clutter. Use waypoints.
- Lift waypoint at same z as grasp\_pos - drags the object sideways.
- Descending straight into a container without a rim-clearance check - grasp collides with walls.
- Forgetting 'for \_ in range(3): get\_observation()' after 'open\_gripper()' - object doesn't settle; next observation is stale.
- Using 'goto\_pose' in far workspace where 'solve\_ik'+ 'move\_to\_joints' is required.