ROBORSI: STABLE, EFFICIENT, AND REUSABLE ROBOT SELF-EVOLUTION IN COMPLEX REAL-WORLD ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A generalist robot in complex real-world environments should not only perform diverse tasks but also improve through experience, turning what it learns during execution into capabilities that later tasks can reuse. Robot agents that act through code can already repair programs from execution feedback, yet it remains a central challenge to organize this experience around the task structure that gives it meaning, so that each repair is attributed to the responsible capability, supported by execution evidence, and validated before it is reused. We introduce RoboRSI, a robot self-improvement system built on Top-Down Skill Refinement (TSR). TSR decomposes tasks into compound, atomic, and base skills with scoped responsibilities and explicit input–output contracts, attributes each execution outcome to the responsible branch, and confines revision to that branch. Building upon this structure, a Manager, Planner, Engineer, and Reviewer coordinate planning, execution, diagnosis, and the validated release of new skills, while people steer the process through objectives and corrections; stable skill sequences are further consolidated into reusable compound skills. On a mobile manipulator, RoboRSI develops multi-object household cleanup over 104 rounds. In simulation, it achieves the highest success rate on LIBERO, LIBERO-PRO, LIBERO-Plus, and RoboTwin, exceeding the strongest baseline by 2.7 to 11.0 percentage points.

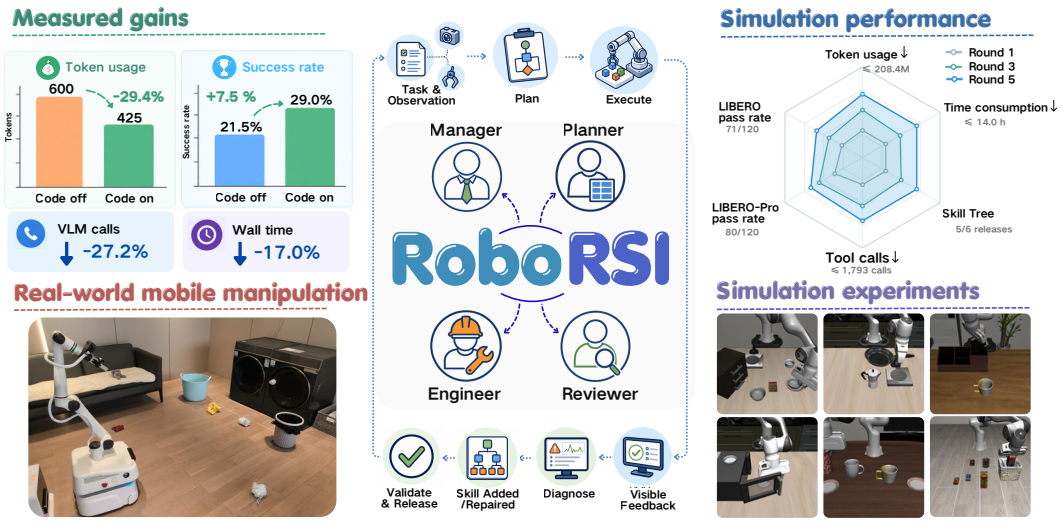


Figure 1: **RoboRSI at a glance.** Center: the multi-agent loop connects task execution, feedback, and skill refinement. Left: physical mobile manipulation in household cleanup (lower) and, from the simulation ablation, the effect of compound skills (Code on versus off) on task success and execution cost (upper). Right: capability growth during one day of self-iteration (upper, Appendix E.2) and simulation task environments (lower).

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

1 INTRODUCTION

The vision of generalist robotics is to develop robots that perform a broad range of tasks reliably in complex real-world environments, where objects, layouts, and instructions keep changing, while improving through experience. This shifts the goal of robot agents from episode-level task completion toward sustained capability accumulation: failed grasps, stale object locations, and recovery situations are valuable only if the robot can identify the capability involved, refine it using execution evidence, and make the resulting behavior available to later tasks.

A promising route lets foundation models act through code. Code as Policies (Liang et al., 2023) composes perception and control primitives into executable robot programs, and CaP-X (Fu et al., 2026) studies how abstraction, grounding, and interaction shape such programs. Subsequent work closes the loop with execution feedback: ETA/OpenETA (Chen et al., 2026b) interleaves planning and tool execution with fresh observations, ASPIRE (Lu et al., 2026) adds repaired programs to a growing skill library, and ENPIRE (Xiao et al., 2026) carries this loop into physical experimentation. Execution feedback thus shifts from a signal for correcting a single attempt to a resource for building capabilities that persist across tasks.

Execution experience, however, becomes a reusable capability only when it is organized around the task structure that gives it meaning. A repair must remain connected to the objective that motivated it, belong to the capability responsible for the observed behavior, and be validated before it is reused. Without such structure, repeated attempts fill the context with complete trajectories, local fixes drift from the overall objective, and people are drawn back into resetting environments, reading logs, and deciding where each fix belongs.

Our observation is that this structure must serve both the people directing the robot and the agents implementing its behavior, which calls for two complementary abstractions. *Human-friendly steering* lets people express objectives, domain knowledge, constraints, and safety judgments at the level of the service they need. *Agent-friendly structure* gives agents task-relevant context, explicit interfaces, and bounded responsibilities for execution and revision.

Based on this observation, we introduce **RoboRSI**, a **multi-agent system** for robot self-improvement organized by **Top-Down Skill Refinement (TSR)**. TSR decomposes each task into compound, atomic, and base skills with scoped responsibilities and explicit input–output contracts, attributes each execution outcome to the responsible branch, and confines the revision to that branch: an inadequate decomposition revises the task structure, an unreliable composition the corresponding compound skill, and a failed operation the responsible atomic or base skill. A Manager, Planner, Engineer, and Reviewer carry out planning, execution, diagnosis, and validated release, and stable skill sequences are consolidated into compound skills that later executions select and monitor instead of reconstructing every tool decision.

We evaluate RoboRSI on a mobile manipulator that performs multi-object household cleanup across a scene change while its skills are developed over 104 rounds, and in simulation against robot-agent baselines on LIBERO, LIBERO-PRO, and RoboTwin, with frozen capability transfer to LIBERO-Plus. RoboRSI achieves the highest success rate on all four benchmarks, exceeding the strongest baseline by 5.3 points on LIBERO, 11.0 on LIBERO-PRO, 5.7 on LIBERO-Plus, and 2.7 on RoboTwin.

Our contributions are threefold:

- **Complementary abstractions for robot self-improvement:** we identify that execution experience becomes reusable only when it is tied to the task structure that gives it meaning, and that this structure must serve both human-friendly steering and agent-friendly structure.
- **RoboRSI Framework:** a system that realizes these abstractions through Top-Down Skill Refinement, which verifies each atomic task's postcondition before completion and revises each failure at its earliest failing node, and a multi-agent execution–refinement loop that validates and releases skills across tasks.
- **Real-world and simulated evaluation:** skill development on a mobile manipulator over 104 rounds, the highest success on four simulation benchmarks, and ablations of role separation, skill refinement, and compound skills.

2 Related Work

Code as Policy. Code as Policies (Liang et al., 2023) showed that language models can compose perception and control APIs into robot programs; CaP-X (Fu et al., 2026) and GRACE (Sun et al., 2025) study the primitives such programs are built from, and ASPIRE (Lu et al., 2026) keeps repaired programs in a growing skill library. Once programs are reused, a repair to one skill affects every task that calls it; TSR gives each capability an interface and a declared responsibility so that a repair is made at the responsible branch.

Recursive Self-Improvement (RSI). Language agents improve through verbal feedback (Shinn et al., 2023; Madaan et al., 2023) or accumulated code skills (Wang et al., 2024), and multimodal and embodied agents acquire skills from external resources and execution feedback (Fan et al., 2026; Chen et al., 2026a; Xu et al., 2026; Ju et al., 2026; Xiao et al., 2026). Robot failure detectors check preconditions, constraints, or outcomes during execution (Singh et al., 2023; Liu et al., 2023b; Guo et al., 2024; Zhou et al., 2025; Duan et al., 2025). In physical execution, a failure may originate several steps before it is observed, and a change to a shared skill affects other tasks; RoboRSI therefore attributes each failure to its earliest failing node and validates revisions against the recent history of the affected tasks.

Large Language Models for Robot. Language models entered robotics as planners grounded in admissible actions and environment feedback (Huang et al., 2022a;b) and later composed value maps and keypoint constraints through code (Huang et al., 2023; 2024). Vision–language–action and world-action models learn control end to end (Brohan et al., 2023; Kim et al., 2024; Black et al., 2025; Physical Intelligence et al., 2025; Zhou et al., 2026; Gemini Robotics Team et al., 2025), while recent agents use general-purpose models directly for robot decisions (Shi et al., 2025; Chen et al., 2026b; Cheng et al., 2026; Huang et al., 2026; Li et al., 2026; Ye et al., 2025; 2026). RoboRSI complements these directions at the system level by maintaining reusable skills whose execution trajectories can also train such policies. Appendix A gives an extended discussion.

3 Method

Consider a coding agent connected directly to robot tools and asked to improve a household cleanup task in a complex real-world home over many runs. The context fills with records of earlier runs; a failure observed late in a sequence is patched where it appears, for example by adding retries to the release step, although it originated in perception or navigation; and because the objective is implicit in the latest prompt, successive edits optimize for the latest scene. People are then drawn back into reading logs and deciding where each fix belongs. RoboRSI removes this burden by making both the intent of the people and the work of the agents explicit.

3.1 Overall Framework

RoboRSI turns repeated robot execution into a growing library of reusable skills, organized around the two abstractions of Section 1. **Human-friendly steering** is the channel through which people enter objectives, constraints, domain knowledge, and corrections at the level of the task; each correction is kept as maintained guidance on the responsible skill. **Agent-friendly structure** is the task–skill hierarchy in which the agents work: every task and skill has an interface, a declared responsibility, and a record of recent executions, so each agent receives only the part of the hierarchy that its current step concerns.

Skill hierarchy. The hierarchy contains four kinds of nodes (Figures 2 and 3). A *task family*, such as household cleanup, groups tasks that share goals and constraints. Each task is decomposed into *atomic tasks*, scoped goals with an observable outcome such as placing an object in a container, each achieved by an *atomic skill* with a postcondition. Atomic skills call *base skills*, the perception and control primitives exposed as tools. A stable sequence of skills is consolidated into a *compound skill*, a parameterized code policy invoked as a single unit. Every skill declares its inputs, outputs, and responsibility, and a skill may be shared by several branches.

Figure 2 follows one development round: the agents decompose a stated goal and plan over released skills, execute them down to base skills, locate the earliest failing node from visible evidence and the tool trace, and validate and release the resulting updates for the next round. People steer at the first stage and whenever a correction is needed.

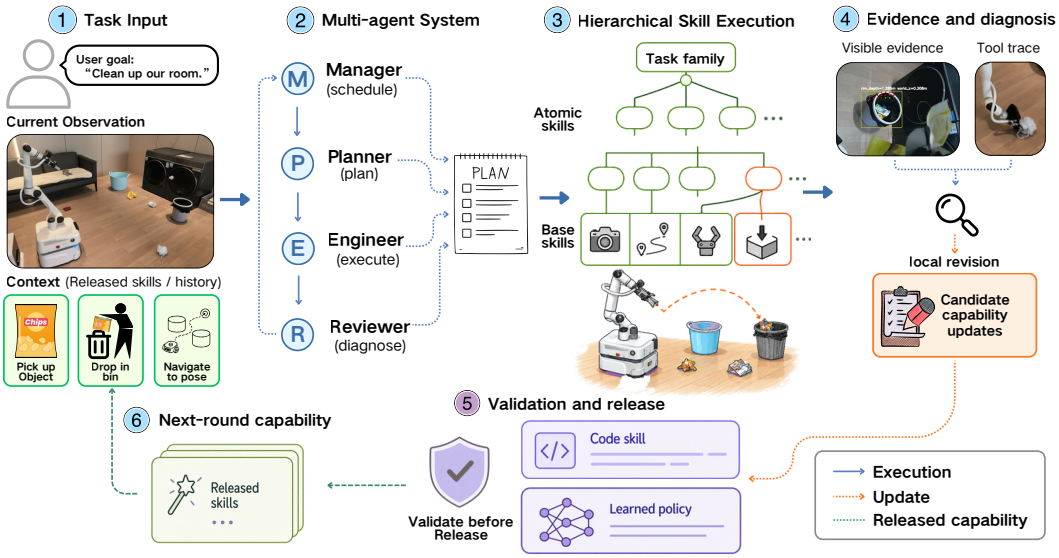


Figure 2: **RoboRSI framework.** Given a task, the Manager organizes its skill hierarchy, the Planner composes an execution plan, and the Engineer runs the required skills. The Reviewer diagnoses failures from observations and tool traces, returning them for TSR-guided revision. Tested updates enter the next iteration, and stable branches become parameterized skills for subsequent tasks.

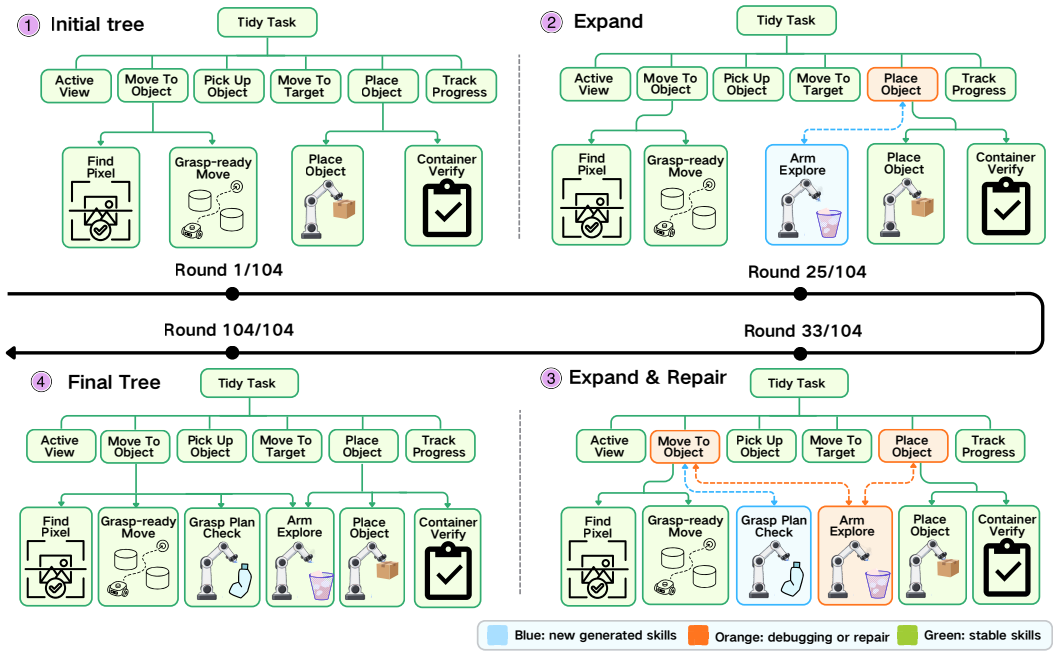
3.2 Top-Down Skill Refinement

Coding agents can solve local programming problems yet lose the task objective across repeated edits, misinterpret physical conditions, or tailor a fix too closely to one failed scene. TSR therefore organizes implementation and revision around the skill hierarchy of Section 3.1: the task family and its atomic tasks preserve the global objective, while atomic, compound, and base skills provide bounded places for implementation, experience, and revision (Figure 3a); Figure 3b follows two revisions on the physical robot.

Design principles. Three principles connect local revision to reliable execution and reuse. *Responsibility and interface trust:* each skill reports its outcome through an explicit interface, so callers rely on its postconditions instead of duplicating its checks, and diagnosis follows the call path to the responsible component instead of spreading fixes across callers. *Parameterization and history-guided revision:* object classes, conditions, target relations, and execution order become parameters, scene geometry comes from current observations, and each change is checked against recent successful uses of the skill. *Persistent human knowledge:* a human correction becomes a runtime check, a test, or maintained guidance on the responsible skill, so people can concentrate on goals and exploration direction.

Refinement step. We write the released hierarchy at round t as a graph $G_t = (V_t, E_t)$ whose nodes are tasks and skills and whose edges are possible calls; each node v carries an implementation or guidance c_v and a declared postcondition ϕ_v. Executing a task produces a record τ_t of observations and tool calls. We list the nodes executed on the task's branch as $\pi_t = (v_1, \ldots, v_m)$ in the order in which they return, so that each callee precedes its caller; a retried node appears once per attempt. The Reviewer evaluates the postconditions along this path and returns the earliest failing node

$$v_t^\star = v_{k^\star}, \qquad k^\star = \min\{k : \phi_{v_k}(\tau_t) = 0\}. \tag{1}$$



(a) Task–skill structures during physical task development



(b) Two TSR revisions on the physical robot

Figure 3: **TSR during physical cleanup.** (a) Four representative task–skill structures show the progression from an initial task proposal to a developed skill library; colored blocks mark the skill operations at each snapshot. (b) Round 25 (top): the robot holds a wrapper at the correct bin without releasing it (i), although the arm has reached its observation pose (ii); the motion call reported a tolerance miss and the fallback exhausted its budget. TSR traces the failure to this observation motion and revises only the Place Object branch, which rechecks the reached pose and executes the complete code-produced release sequence; the next placement succeeds (iii). Round 33 (bottom): the destination bucket appears only at the image edge in the chassis view (iv) but completely in the in-hand view of the same sweep (v); the consumer rejected the producer's destination, and the run ended with the second item held (vi). The revision leaves identity to the producer and prefers complete views. Both revisions passed offline replay of the four most recent runs and the existing regression tests (Appendix F.5).

The revision scope $S_t \subseteq V_t$ is the sub-hierarchy owned by $v_t^\star$, and ∂S_t denotes its interfaces to the rest of G_t. A revision round is then

$$\Delta_t = \mathcal{R}\big(G_t|_{S_t}, \partial S_t, \tau_t\big), \qquad G_{t+1} = G_t \oplus \mathcal{V}(\Delta_t; \mathcal{H}_t). \tag{2}$$

Here $\mathcal{R}$ is the Reviewer's patch proposal, which may change only nodes in S_t; if the interface itself must change, the scope is first expanded to include its callers. $\mathcal{H}_t$ collects recent failed and successful executions of every task whose call graph reaches S_t. The validation operator $\mathcal{V}$ returns Δ_t if it passes review, functional tests, and replay on $\mathcal{H}_t$ without regression, and the empty patch otherwise; $\oplus$ applies the accepted patch. Eq. (1) places each repair at the origin of the failure rather than where it is observed, the restriction to S_t bounds its effect to the tasks that reach S_t, and validation on $\mathcal{H}_t$ preserves behavior that already works (Appendix B).

From stable branches to reusable capabilities. Stable branches become compound skills, shifting online reasoning from individual tool choices to skill selection and monitoring; execution trajectories can also train neural policies exposed as skills in the same hierarchy (Appendix C.5).

3.3 Multi-Agent System

Four agents carry out each round within this structure. The **Manager** decomposes the goal into atomic tasks, identifies the base skills each one needs, maintains skill versions, reviews each proposed revision, may request or author changes to it, and decides whether it is admitted, which branch it may change, and whether it is released. The **Planner** receives one atomic task, the current observation, and the interfaces of the skills below it, and returns an executable plan. The **Engineer** executes the plan through the tool interface and implements missing skills. The **Reviewer** judges from the observations and tool trace whether an atomic task reached its expected outcome and, for a failure, identifies where execution first diverged, and writes a revision patch for the responsible branch that it submits to the Manager. Each agent works from the part of the hierarchy that its step concerns, so its context stays short and every change can be traced to a diagnosis.

4 Experiment

We report physical deployment and simulation results, then compare individual episodes and summarize the ablations.

4.1 Physical Robot Case Study

Setting. We study household floor cleanup with a WheelSingleArm M1 mobile manipulator. The robot searches for objects, approaches and grasps them, and transports them to designated receptacles. The case study covers 104 physical development runs with 24 hours of cumulative operation. Before each run, the scene is restored to a common base layout, while the objects and receptacles are partially randomized; the navigation map remains fixed. Platform details and the reset procedure are given in Appendix D.

Iterative cleanup. Figure 4 follows ten selected runs from the 104-run development period. In the earliest selected run, the robot placed no objects. Later runs progressed to placing several objects, and some completed their assigned cleanup tasks and returned to the dock. The change was not steady: a later run recorded only one placement after complete runs had already appeared. We counted an object only when on-site personnel observed it enter its designated receptacle. The figure displays at most four placements per run, although the task manifests varied in size.

The skill-tree snapshots in Figure 3a provide a complementary view of this development process. The hierarchy introduces *Arm Explore* under *Place Object* to obtain a view of the target receptacle. Later revisions add *Grasp Plan Check* under *Move To Object* and make *Arm Explore* available to both the approach and placement branches. These changes target two recurring requirements of the cleanup procedure: checking a candidate grasp before motion and grounding placement in an observation of the destination. The snapshots document how the corresponding capabilities were organized and reused; the selected executions do not isolate the effect of an individual revision.

Physical execution. The two-scene demonstration provides an extended execution example. The robot placed all five designated objects in the first scene and all four in the second, with each placement confirmed on site. Between scenes, an operator rearranged the objects and receptacles; the skill code, prompts, and policy configuration were not modified. The cleanup executions required

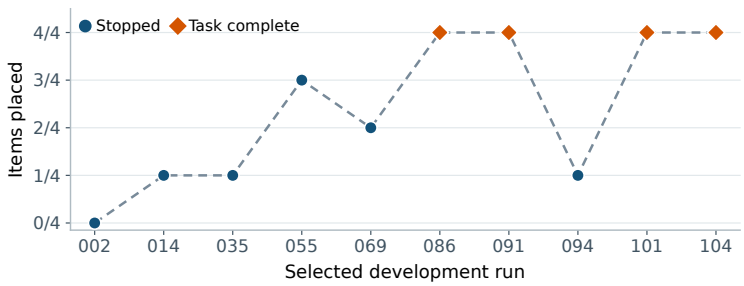


Figure 4: **Placement progress in selected physical cleanup runs.** Each point shows the number of objects placed among the first four items considered in that run; placements were confirmed on site. Marker colors indicate the audited task outcome. Points are equally spaced in selected-run order, while tick labels give the development run numbers. Dashed segments connect selected observations only. The four-item denominator is a common display reference: task manifests and object configurations vary across runs.

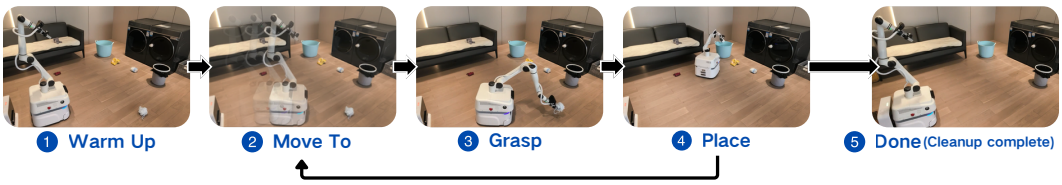


Figure 5: **Two-scene physical mobile cleanup.** Selected frames show preparation, approach, grasping, placement, and the recorded final scene. The return arrow indicates repeated object handling. Objects and receptacles were rearranged between scenes, while the agent configuration remained unchanged.

no manual correction or restart. Following the scene change, the robot repeated the search, approach, grasp, transport, and placement sequence. Figure 5 illustrates stages of this recorded execution.

4.2 Main Simulation Results

Evaluation settings. Self-improvement is part of the evaluated capability: on LIBERO, LIBERO-PRO, and RoboTwin, RoboRSI starts from its initial skill library and improves online on the evaluation tasks, summarizing failures and revising skills as evaluation proceeds, whereas the baselines have no self-improvement mechanism (Appendix H). LIBERO-Plus instead tests transfer: the library obtained on LIBERO is held fixed and receives no updates on the perturbed instances. All methods use GPT-5.6-SOL. Maestro, the most costly baseline, is evaluated on LIBERO and LIBERO-PRO only.

Task execution and coverage. RoboRSI achieves the highest success and coverage on every benchmark (Table 1). The largest advantage is on LIBERO-PRO, where it exceeds OpenETA by 11.0 points in success and 18.3 points in coverage, so the gain extends to a larger set of solved tasks. On LIBERO, it exceeds OpenETA by 14.0 points on Spatial and 8.0 points on Object (Appendix E.1).

Where the LIBERO-PRO gains come from. RoboRSI is best in every suite and under every perturbation (Table 2). The margins are largest on the Object suite ($+16.5$) and under object perturbations ($+15.3$), which require re-identifying the target in the current scene.

Transfer under perturbations. On LIBERO-Plus, the frozen source library exceeds OpenETA by 5.7 points and solves at least one perturbed instance of 28 of the 30 tasks. The gains are largest under background ($+13$), language ($+12$), robot initial state ($+9$), noise ($+8$), and lighting ($+7$) perturbations (Figure 6b reports all seven types).

Table 1: **Simulation results.** Success is successful/valid episodes (rate); coverage is solved/total tasks (rate), counting each task once after its first success. LIBERO uses 30 tasks×5 episodes, LIBERO-PRO uses 120 tasks×5 episodes, LIBERO-Plus uses 840 perturbation instances of 30 tasks, and RoboTwin uses 50 tasks×3 episodes for the baselines; for RoboRSI we report the first 154 episodes of its online run on the same 50 tasks. RoboRSI improves online on LIBERO, LIBERO-PRO, and RoboTwin and uses its fixed LIBERO library on LIBERO-Plus. Maestro is evaluated on LIBERO and LIBERO-PRO because of its computational cost.

Benchmark	Method	Success	Coverage
LIBERO	CaP-X	36/150 (24.0%)	16/30 (53.3%)
	Maestro	62/150 (41.3%)	23/30 (76.7%)
	OpenETA	76/150 (50.7%)	23/30 (76.7%)
	RoboRSI	**84/150 (56.0%)**	**24/30 (80.0%)**
LIBERO-PRO	CaP-X	111/600 (18.5%)	46/120 (38.3%)
	Maestro	198/600 (33.0%)	77/120 (64.2%)
	OpenETA	231/600 (38.5%)	80/120 (66.7%)
	RoboRSI	**297/600 (49.5%)**	**102/120 (85.0%)**
LIBERO-Plus	CaP-X	130/840 (15.5%)	21/30 (70.0%)
	OpenETA	306/840 (36.4%)	27/30 (90.0%)
	RoboRSI	**354/840 (42.1%)**	**28/30 (93.3%)**
RoboTwin	CaP-X	3/150 (2.0%)	3/50 (6.0%)
	OpenETA	32/150 (21.3%)	17/50 (34.0%)
	RoboRSI	**37/154 (24.0%)**	**26/50 (52.0%)**

Table 2: **LIBERO-PRO by task suite and perturbation type.** Success rate (%); each suite contains 200 episodes and each perturbation type 150 episodes per method. Bold: best in the column; gray: RoboRSI's gain over the strongest baseline in percentage points.

	Task suite			Perturbation type				
Method	Spatial	Object	Goal	Language	Object	Position	Task	All
CaP-X	24.5	13.5	17.5	21.3	16.7	17.3	18.7	18.5
Maestro	39.5	38.0	21.5	34.7	30.0	32.0	35.3	33.0
OpenETA	50.0	36.5	29.0	42.7	33.3	36.0	42.0	38.5
RoboRSI	**57.0** (+7.0)	**54.5** (+16.5)	**37.0** (+8.0)	**52.0** (+9.3)	**48.7** (+15.3)	**45.3** (+9.3)	**52.0** (+10.0)	**49.5** (+11.0)

Why the baselines fail. About two thirds of the failures of OpenETA and Maestro on LIBERO-PRO (66% and 64%) and of OpenETA on LIBERO-Plus (66%) are premature completions: the agent declares the task done after a tool reports a release, while the simulator judges it unfinished (Figure 6a). For RoboRSI, premature completion accounts for 10% and 14% of failures. The methods differ in how they decide that a task is complete: the baselines accept a successful tool return, whereas every RoboRSI atomic task has a postcondition checked against fresh observations before the agent proceeds (Eq. (1)). The failure composition reflects this completion rule, and RoboRSI's remaining failures mostly end with the budget exhausted during recovery rather than a false claim of success. On episodes with an execution failure, it still completes 67.0% on LIBERO-PRO and 63.2% on LIBERO-Plus, against 55.0% and 51.6% for OpenETA (Figure 6c).

4.3 Case Study

Figure 7 compares the methods on three episodes with the same initial state (details in Appendix F.1). The baselines stop on a tool's success return, whereas RoboRSI's skills report the failed outcome and its agents re-observe and retry until the postcondition holds. Revisions also transfer: `grasp_object`, revised on one development task, is invoked in 42 successful episodes on other tasks.

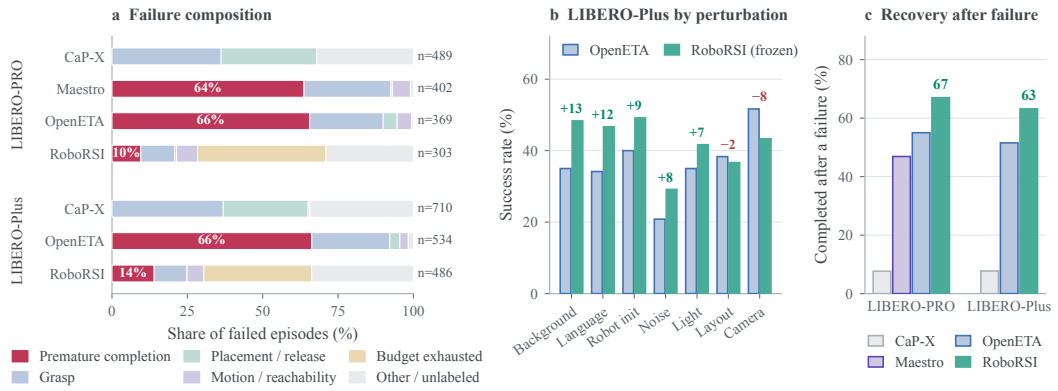


Figure 6: **Failure analysis.** (a) Composition of failed episodes on LIBERO-PRO and LIBERO-Plus; n is the number of failed episodes, and the premature-completion share is labeled. (b) Success on LIBERO-Plus by perturbation type, 120 instances each; numbers give RoboRSI's difference from OpenETA in percentage points. (c) Share of episodes that still succeed after an execution failure, over the 209 LIBERO-PRO and 258 LIBERO-Plus (task, seed) pairs in which every method has a complete trace and at least one execution failure (Appendix E.3).

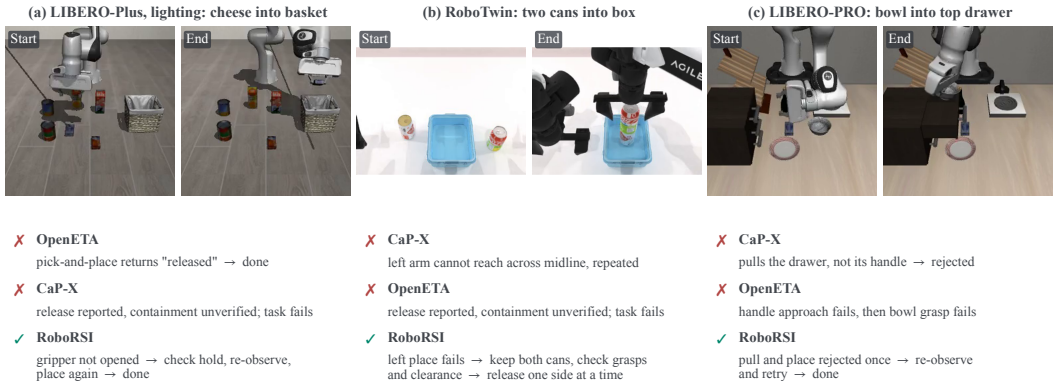


Figure 7: **Simulation cases with the same initial state across methods.** Images show the first and last recorded frames of the RoboRSI episode; text summarizes the decisive tool calls of each method.

4.4 Ablation Study

We ablate the components that drive self-improvement (Appendix C). *Role separation:* with separate Planner, Engineer, and Reviewer roles, development on RoboTwin solves 36 of 50 tasks, against 9 for a single agent. *TSR:* on the same failed task, evidence, and release gate, TSR localizes two responsible nodes and its revision is released, whereas no flat revision passes. *Compound skills:* on a separate 120-task panel, they raise success from 21.5% to 29.0% over 600 matched episode pairs and cut median token use, model calls, and wall time by 29.4%, 27.2%, and 17.0%. Backbones and ACT policies (Zhao et al., 2023) are compared in the appendix.

5 Conclusions and Future Work

RoboRSI combines Top-Down Skill Refinement and multi-agent coordination to organize execution, diagnosis, local revision, and reusable capability, toward stable, efficient, and reusable robot self-evolution in complex real-world environments. Future work will transfer simulation-acquired experience to physical robots and share skills across robots and platforms.

Reproducibility Statement

Section 3 and Appendix B define the skill hierarchy, the attribution rule, and the refinement and validation operators. Section 4 and Appendix E specify every evaluation panel, including task sets, episode counts, the definitions of success and coverage, and the failure classification rules; Tables 1, 2, and 5 report success counts together with their denominators. Appendix D describes the physical platform and protocol, and Appendices F and F.5 give the skill code, revisions, and validation records used in the case studies. The code, de-identified experiment records, and videos of the real-robot and simulation executions are submitted as supplementary material; Appendix G describes their contents and the reproduction steps.

AI Use Statement

RoboRSI uses large language models as components of the system under study: the Manager, Planner, Engineer, and Reviewer are implemented with GPT-family models, which generate plans, code-based skills, diagnoses, and revisions as described in Section 3. Separately from the system itself, we used generative AI tools in preparing this work. Coding agents assisted in running the experiments, aggregating episode logs, and implementing the associated scripts, including the failure classifier and the plotting code, and parts of the RoboRSI codebase were supplemented with AI assistance. Generative AI tools also assisted in designing analyses, specifying follow-up experiments, and summarizing and interpreting experimental results, all of which the authors reviewed. We also used generative AI tools to polish the text and to search the literature. We have reviewed all AI-assisted work: the authors checked and tested the AI-assisted code, every reported number was checked against the logged simulator verdicts and physical execution records, and cited works were verified against their sources. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

References

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a Visual Language Model for Few-Shot Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Kevin Black, Noah Brown, Danny Driess, et al. 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, et al. RT-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning (CoRL)*, 2023.

Bofan Chen, Boxuan Zhang, Fei Tang, Zhengxi Lu, Yong Du, Tongbo Chen, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Reflect, Revise, Reuse: Training-Free Skill Evolution for GUI Agents. *arXiv preprint arXiv:2609.17653*, 2026a.

Yitong Chen, Zezheng Huai, Sixian Li, Yubang Wang, Haozhe Zhang, Yifei Zhang, Hechang Chen, Jingjing Gong, Yu-Gang Jiang, and Xipeng Qiu. ETA: A new agentic paradigm for embodied tasks. *arXiv preprint arXiv:2608.03924*, 2026b.

Dongzhou Cheng, Taoran Yi, Ye Fang, Xingwu Zhang, Fan Feng, Yixuan Li, Gengxiong Zhuang, Rongze Wang, Shuai Yang, Wei Song, Weizhi Xue, Minyan Wu, Jie Gui, Jiaqi Wang, and Tong Wu. In-Context Robot Learning with VLM Agents. *arXiv preprint arXiv:2609.19138*, 2026.

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. PaLM-E: An Embodied Multimodal Language Model. In *International Conference on Machine Learning (ICML)*, 2023.

Jiafei Duan, Wilbert Pumacay, Nishanth Kumar, Yi Ru Wang, Shulin Tian, Wentao Yuan, Ranjay Krishna, Dieter Fox, Ajay Mandlekar, and Yijie Guo. AHA: A vision-language-model for detecting and reasoning over failures in robotic manipulation. In *International Conference on Learning Representations (ICLR)*, 2025.

Yijia Fan, Zonglin Di, Zimo Wen, Yifan Yang, Mingxi Cheng, Qi Dai, Bei Liu, Kai Qiu, Yue Dong, Ji Li, and Chong Luo. RESOURCE2SKILL: Distilling Executable Agent Skills from Human-Created Multimodal Resources. *arXiv preprint arXiv:2606.29538*, 2026.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Jim Fan. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026.

Gemini Robotics Team et al. Gemini Robotics 1.5: Pushing the Frontier of Generalist Robots with Advanced Embodied Reasoning, Thinking, and Motion Transfer. *arXiv preprint arXiv:2510.03342*, 2025.

Gemini Team et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Yanjiang Guo, Yen-Jen Wang, Lihan Zha, and Jianyu Chen. DoReMi: Grounding language model by detecting and recovering from plan-execution misalignment. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024.

Jinbang Huang, Yuanzhao Hu, Zhiyuan Li, Ran Qi, Yixin Xiao, Zhanguang Zhang, Mark Coates, Tongtong Cao, and Yingxue Zhang. RoboHarness: Memory-Driven Orchestration of Heterogeneous Robot Policies for Long-Horizon Planning. *arXiv preprint arXiv:2607.18060*, 2026.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *International Conference on Machine Learning (ICML)*, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, et al. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, 2022b.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Fei-Fei Li. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. In *Conference on Robot Learning (CoRL)*, 2023.

Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Fei-Fei Li. ReKep: Spatio-Temporal Reasoning of Relational Keypoint Constraints for Robotic Manipulation. In *Conference on Robot Learning (CoRL)*, 2024.

Ruofei Ju, Xinrui Wang, Xin Ding, Yifan Yang, Hao Wu, Shiqi Jiang, Qianxi Zhang, Hao Wen, Xiangyu Li, Weijun Wang, Kun Li, Yunxin Liu, Haipeng Dai, Wei Wang, and Ting Cao. EmbodiSkill: Skill-Aware Reflection for Self-Evolving Embodied Agents. *arXiv preprint arXiv:2605.10332*, 2026.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, et al. OpenVLA: An open-source vision-language-action model. In *Conference on Robot Learning (CoRL)*, 2024.

Zhuoran Li, Zhiyang Li, Kaijun Zhou, and Jinyu Gu. RoboHarness: A Memory-Augmented Policy Harness for Vision-Language-Action Model Robustness via In-Context Adaptation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 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.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual Instruction Tuning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023a.

Zeyi Liu, Arpit Bahety, and Shuran Song. REFLECT: Summarizing robot experiences for failure explanation and correction. In *Conference on Robot Learning (CoRL)*, 2023b.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi Jim Fan, and Guanzhi Wang. ASPIRE: Agentic /skills discovery for robotics. *arXiv preprint arXiv:2607.00272*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a Vision-Language-Action Model with Open-World Generalization. In *Conference on Robot Learning (CoRL)*, 2025.

Junyao Shi, Rujia Yang, Kaitian Chao, Selina Bingqing Wan, Yifei Shao, Jiahui Lei, Jianing Qian, Long Le, Pratik Chaudhari, Kostas Daniilidis, Chuan Wen, and Dinesh Jayaraman. Maestro: Orchestrating robotics modules with vision-language models for zero-shot generalist robots. *arXiv preprint arXiv:2511.00917*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating situated robot task plans using large language models. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

Mingyang Sun, Jiude Wei, Qichen He, Donglin Wang, Cewu Lu, and Jianhua Sun. Executable Analytic Concepts as the Missing Link Between VLM Insight and Precise Manipulation. *arXiv preprint arXiv:2510.07975*, 2025.

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. *Transactions on Machine Learning Research*, 2024.

Wenli Xiao, Jia Xie, Tonghe Zhang, Haotian Lin, Letian Fu, Haoru Xue, Jalen Lu, Yi Yang, Cunxi Dai, Zi Wang, Jimmy Wu, Guanzhi Wang, S. Shankar Sastry, Ken Goldberg, Linxi Fan, Yuke Zhu, and Guanya Shi. ENPIRE: Agentic Robot Policy Self-Improvement in the Real World. In *Conference on Robot Learning (CoRL)*, 2026.

Xu Xu, Jinxiu Liu, Zhangbo Qiao, Jiaxing Lu, Xiangyu Zhang, Yubin Gu, Fangwei Ning, and Yan Shi. OmniHarness: Harnessing Generalizable Visual Generation via Symbolic Policy Learning. *arXiv preprint arXiv:2609.16057*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Haoming Ye, Yunxiao Xiao, Cewu Lu, and Panpan Cai. UniDomain: Pretraining a Unified PDDL Domain from Real-World Demonstrations for Generalizable Robot Task Planning. *arXiv preprint arXiv:2507.21545*, 2025.

Haoming Ye, Yunxiao Xiao, Cewu Lu, and Panpan Cai. UniPlan: Vision-Language Task Planning for Mobile Manipulation with Unified PDDL Formulation. *arXiv preprint arXiv:2602.08537*, 2026.

Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. In *Robotics: Science and Systems (RSS)*, 2023.

Enshen Zhou, Qi Su, Cheng Chi, Zhizheng Zhang, Zhongyuan Wang, Tiejun Huang, Lu Sheng, and He Wang. Code-as-Monitor: Constraint-aware visual programming for reactive and proactive robotic failure detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.

Jiaming Zhou, Qihang Zhang, Gangwei Xu, Cunxin Fan, Yujie Zhao, Ruilin Wang, Yiming Luo, Shuai Yang, Xing Zhu, Yujun Shen, Junwei Liang, and Yinghao Xu. Zero-WAM: In-Context World-Action Modeling from Human Videos for Open-Ended Task Generalization. *arXiv preprint arXiv:2608.26103*, 2026.

A Extended Related Work

Code as Policy. Code as Policies (Liang et al., 2023) showed that language models can write robot programs that compose perception and control APIs, turning high-level instructions into executable behavior. Follow-up work examines what these programs should be built from: CaP-X (Fu et al., 2026) studies the abstraction level of primitives, perceptual grounding, and interaction, and GRACE (Sun et al., 2025) encodes affordances and geometry as executable concepts. ASPIRE (Lu et al., 2026) goes further by repairing failed programs and keeping the repaired skills in a growing library. Once programs are kept and reused, however, a repair to one skill can affect every task that calls it, and deciding where a repair belongs becomes as important as writing it. RoboRSI addresses this through TSR, which gives each capability an interface and a declared responsibility, so that a repair is made at the responsible branch and composes with the skills around it.

Recursive Self-Improvement (RSI). RSI studies how an agent's experience improves its later behavior. Language agents first did so through verbal feedback: Reflexion stores reflections on failed attempts, and Self-Refine iteratively critiques and revises its own outputs (Shinn et al., 2023; Madaan et al., 2023). Voyager moved from text to executable skills, accumulating a code library through open-ended interaction (Wang et al., 2024). Multimodal agents extend this idea with skills acquired from external resources, execution feedback, and reusable procedures, as in RESOURCE2SKILL, EvoSkill-GUI, and OmniHarness (Fan et al., 2026; Chen et al., 2026a; Xu et al., 2026). Robot-specific work detects and explains failures during execution: ProgPrompt adds assertions to generated programs (Singh et al., 2023), REFLECT summarizes robot experience to explain and correct failures (Liu et al., 2023b), DoReMi and Code-as-Monitor monitor language- or code-specified constraints to trigger recovery (Guo et al., 2024; Zhou et al., 2025), and AHA trains a vision-language model to detect and reason about manipulation failures (Duan et al., 2025). These methods detect a failure within an episode; RoboRSI additionally decides which skill should change and validates the change before later tasks reuse it. In embodied settings, EmbodiSkill diagnoses errors at the level of individual skills (Ju et al., 2026), and ENPIRE couples physical evaluation with coding-agent refinement of policies (Xiao et al., 2026). Physical execution adds a difficulty that software environments largely avoid: a failure may originate several steps before it is observed, and a change to a shared skill affects tasks that were not part of the failed run. RoboRSI therefore attributes each failure to its earliest failing node, validates revisions against the recent history of the affected tasks, and reuses capabilities through explicit skill interfaces.

Large Language Models for Robot. Large language models first entered robotics as planners. Early work grounded generated plans in admissible actions (Huang et al., 2022a), and Inner Monologue fed environment feedback back into planning (Huang et al., 2022b). Advances in multimodal understanding (Alayrac et al., 2022; Liu et al., 2023a; Gemini Team et al., 2024), embodied grounding (Driess et al., 2023), and interleaved reasoning and acting (Yao et al., 2023) then allowed models to reason directly over observations, and VoxPoser (Huang et al., 2023) and ReKep (Huang et al., 2024) used generated code to compose 3D value maps and relational keypoint constraints for motion planning. A parallel line trains robot foundation models end to end. Vision–language–action models such as RT-2, OpenVLA, π_0, and $\pi_{0.5}$ learn visuomotor control from large robot datasets (Brohan et al., 2023; Kim et al., 2024; Black et al., 2025; Physical Intelligence et al., 2025); world-action models such as Zero-WAM add predictive visual dynamics (Zhou et al., 2026); and Gemini Robotics and Gemini Robotics-ER 1.5 combine control with embodied reasoning (Gemini Robotics Team et al., 2025). As general-purpose models have become more capable, recent systems use them directly for robot decisions and actions. Maestro orchestrates perception and control modules (Shi et al., 2025), ETA/OpenETA closes the loop between planning, execution, and feedback (Chen et al., 2026b), GPT-Policy performs in-context control (Cheng et al., 2026), two RoboHarness systems adapt through memory (Huang et al., 2026; Li et al., 2026), and UniDomain and UniPlan connect symbolic planning with visual grounding (Ye et al., 2025; 2026). RoboRSI complements these directions at the system level: TSR and multi-agent execution maintain reusable skills, and the trajectories collected during execution can train policies that return as tools in the same hierarchy.

B TSR FORMULATION AND ANALYSIS

Task–skill graph and scoped patches. At iteration t, the released hierarchy is represented by $G_t = (V_t, E_t, \{c_v\}_{v \in V_t})$, where V_t contains task and skill nodes, E_t contains possible call edges, and c_v is a node's implementation or guidance. The execution record τ_t includes the goal, observations, and tool trace. Diagnosis identifies the affected scope $S_t \subseteq V_t$; $G_t|_{S_t}$ contains its nodes and internal calls, while ∂S_t contains the interfaces connecting it to the unchanged hierarchy.

The revision procedure $\mathcal{R}$ proposes a graph and code patch Δ_t with $\mathrm{supp}(\Delta_t) \subseteq S_t$. The support identifies existing nodes whose implementation, guidance, or outgoing calls change. New helper nodes may be attached within that scope. The operator $\oplus$ applies both code edits and branch expansion. If diagnosis implicates a caller or interface outside S_t, the scope expands before the next proposal.

History and validation. The case collection in Eq. 2 is $\mathcal{H}_t = \mathcal{D}_t^- \cup \mathcal{D}_t^+(S_t)$, comprising recent failures that motivate the edit and selected successful cases from affected callers. Review and functional tests determine whether $\mathcal{V}$ returns the proposed patch or an empty patch. Applying an empty patch leaves the released graph unchanged; accepted revisions become available to subsequent execution. Shared skills require checking the relevant caller paths even when only one implementation changes. The next iteration uses feedback from the currently released version.

Dependency-local effects. Equation 2 separates the scope of an edit from the scope of its possible effects. Let r_q be the entry node for task q and let $\mathrm{Reach}_{G_t}(r_q)$ include that node and all nodes reachable through possible calls. The affected task set is $\mathcal{A}_t = \{q : \mathrm{Reach}_{G_t}(r_q) \cap S_t \neq \emptyset\}$. This uses possible dependencies, not only calls observed in one rollout: a shared base skill may affect several tasks under different observations.

Consider a fixed task distribution $p(q)$, and let $J_q(G)$ be the expected failure rate for task q under graph G. Assume the graph captures all behavioral dependencies, the patch leaves interfaces and implementations outside its scope unchanged, and evaluation uses the same reset and exogenous-randomness distributions. In particular, edits must not alter untracked shared state, global prompts, or tool configurations. Under these conditions, tasks outside $\mathcal{A}_t$ cannot encounter a changed implementation. Their execution distributions are unchanged, so their contributions cancel in the aggregate difference:

$$\begin{aligned} J(G_{t+1}) - J(G_t) &= \sum_{q \in \mathcal{A}_t} p(q)\big[J_q(G_{t+1}) - J_q(G_t)\big], \\ |J(G_{t+1}) - J(G_t)| &\le \sum_{q \in \mathcal{A}_t} p(q), \qquad J(G) = \sum_q p(q) J_q(G). \end{aligned} \tag{3}$$

The bound follows because each failure rate lies in $[0, 1]$. Its magnitude scales with the probability mass of affected tasks and covers both improvements and regressions. A task-specific branch can have a small affected set, whereas a widely shared base skill can have a large one. This motivates selecting regression cases by dependency coverage rather than by the number of edited functions. If a revision changes shared state or an interface, its dependency scope must be expanded before this analysis applies. Development cases guide revision, while held-out tasks measure generalization.

Token accounting. Construction cost includes skill generation, diagnosis, review, and validation. Execution cost includes all model calls within the executing agent, including model-backed perception, monitoring, and fallback; separately measured planning and review are excluded from this execution subtotal. Unmetered calls are reported separately. Comparisons require matched task distributions and must include failed executions, since a shorter failed attempt can consume fewer tokens without improving efficiency. Task success, runtime, and the frequency of fallback to online reasoning provide context for interpreting any reduction in execution tokens.

Execution and outcome interfaces. Agents communicate through plans, structured skill inputs and outputs, and execution records. Physical deployment uses human guidance and safety supervision; simulation can automate resets and outcome checks. Planner, Engineer, and Reviewer receive observations and tool feedback, while hidden simulator state and success-condition internals remain outside their context. The simulator scores the task separately after execution.

Table 3: **Single-agent and multi-agent development on RoboTwin.** Tasks solved at least once over the full online run, out of 50; the multi-agent condition is the RoboTwin run of Table 1, whose first 154 episodes are reported there.

Condition	Roles	Tasks solved
Single agent	Engineer	9/50 (18.0%)
Multi-agent	Planner, Engineer, Reviewer	**36/50 (72.0%)**

Operational details. Each skill has a descriptor that declares its arguments, typed return fields, applicability conditions, and a validation harness (simulator task, seeds, arguments, and any setup skill). The return fields report whether the intended effect was verified; for example, a placement skill returns whether the target was reached, the gripper opened, and the release was verified, and it does not open the gripper unless its reach and hold checks pass. An atomic task counts as complete only when its postcondition holds on fresh observations taken after the last action. A candidate revision is staged in an isolated copy of the library and reviewed by the Manager. If it changes a base skill, the harness of that skill runs in the simulator; the complete atomic task is then re-run on its development seeds. Release requires simulator success on at least two declared seeds, restoration of the unmodified source after the trial, and Manager approval with no outstanding changes; a candidate that fails cannot be resubmitted unchanged. In simulation, $\mathcal{H}_t$ consists of these harness and development runs; on the robot, it consists of the four most recent eligible runs, replayed offline (Appendix F.5). A compound skill is proposed once a task has at least three successful executions whose tool sequences share at least 60% of their longest common subsequence and a common skeleton of at least three steps. The generated skill replays the skeleton and recomputes every argument from perception at run time, never storing coordinates from an episode, and it passes the same review and validation as any other revision.

C Ablation Studies

We study role separation, hierarchical refinement, code consolidation, and backbone choice separately, and additionally examine policy learning from execution data. The backbone study uses LIBERO-PRO and RoboTwin task panels with three evaluation episodes per task, 360 and 150 episodes per backbone.

C.1 Role Separation

We compare development on the 50 RoboTwin tasks with a single agent and with separate roles (Table 3). In the single-agent condition, one Engineer plans, executes, and judges its own attempts. In the multi-agent condition, a Planner composes the plan, the Engineer executes it, and a Reviewer diagnoses each outcome from observations and tool traces. The single agent solves 9 of the 50 tasks, whereas the multi-agent system solves 36. Separating diagnosis from execution gives every failure an independent review before the next attempt, instead of leaving the executing agent to judge its own result.

C.2 Top-Down Skill Refinement

TSR determines where the system improves. We therefore compare TSR with flat revision on how well each turns an observed failure into a validated, released improvement, rather than on task success. Both conditions start from the same code and memory and receive the same public traces of a failed `libero_pick_place` execution on `libero_goal/8`. They may modify Atomic and Base skills, and use the same Planner, Engineer, Reviewer, and Manager review. A candidate is released only if the Base functional checks pass and the complete Atomic task succeeds on both development seeds. Under TSR, the Reviewer first builds a responsibility tree from the Atomic failure and selects the layers to revise; under flat revision, it proposes a revision directly from the failure.

TSR attributed the failure to two nodes: the Atomic plan treated a do-not-regrasp flag set at pickup as permanent after a verified release, and the Base hold check used the fixed pickup patch as an



Figure 8: **Effect of code consolidation.** (a) Task success over 600 matched episodes per condition. (b) Median token, model-call, and time cost on the 118-task efficiency sample, normalized to Code off.

invariant during transport. Its candidate changes only these nodes, succeeds on both seeds, and is released (Table 4). The flat candidate also rewrites the placement policy and succeeds on one seed; on the other, the agent declares success with the object off the target. A further flat candidate, restricted to the Atomic plan after Manager review, again succeeds on one seed, so no flat candidate is released.

Table 4: **TSR and flat revision on the same failed task.** Files and lines count changes relative to the common starting code; Atomic success is over the two development seeds.

Condition	Files	Changed lines	Atomic success	Released
TSR	3	50	**2/2**	yes
Flat, candidate 1	4	104	1/2	no
Flat, candidate 2	1	63	1/2	no

C.3 Code Consolidation

We compare execution with compound skills available to the agent (Code on) and withheld (Code off) on a separate panel of 120 tasks selected from LIBERO-Plus, distinct from the task sets in Table 1, so its absolute success rates are not comparable with the main results. The base tools, the GPT-5.6-SOL backbone, the 120 tasks, and the five seeds are the same in both conditions, so every Code-on episode has a matched Code-off counterpart. Figure 8 reports success over these 600 matched pairs and normalized median costs on a separate 118-task sample.

Code consolidation raises success from 21.5% (129/600) to 29.0% (174/600), a paired gain of 7.5 percentage points (95% CI $[3.7, 11.5]$; McNemar $p < 10^{-4}$), and increases the number of tasks solved at least once from 58 to 71 of 120. The gain is spread across many episodes: 86 matched pairs succeed only with Code on, against 41 that succeed only with Code off. Median token consumption, model calls, and wall time decrease by 29.4%, 27.2%, and 17.0%, respectively.

The success gain follows from how errors arise in long tool sequences. Without code, the agent reconstructs the sequence in every episode, and each step is another opportunity to choose the wrong tool, pass an inconsistent argument, or skip a check between steps; over sequences of many steps, these errors accumulate. A compound skill fixes the validated order of calls and the checks between them, so the agent decides only which skill to invoke and with which parameters. The physical Round 25 failure in Section 3.2 shows the same effect: a model-written release sequence dropped steps that the code-produced sequence contained. The cost reductions have the same source, since one skill call replaces several model decisions. Wall time falls less than tokens and model calls because the motion executed in the simulator is unchanged; only the reasoning between motions is removed. The agent still responds to execution feedback and returns to step-by-step reasoning when a compound skill reports a failure.

Table 5: **Backbone comparison.** The study uses a separate panel of 120 LIBERO-PRO tasks×3 seeds run independently of Table 1, so only the relative differences between backbones are compared. LIBERO-PRO success over 360 episodes per backbone, overall and by task suite; RoboTwin successes over 150 episodes. Model calls are medians per LIBERO-PRO episode; budget exhausted counts LIBERO-PRO episodes that ended at the interaction limit. Bold: best in the column.

	LIBERO-PRO success (%)				RoboTwin	LIBERO-PRO episodes	
Backbone	All	Spatial	Object	Goal	success	Model calls	Budget exhausted
GPT-5.6-SOL	38.3	38.3	48.3	28.3	**24/150**	115	79
GPT-5.5	**48.6**	53.3	**55.0**	**37.5**	19/150	122	80
GPT-5.4	40.3	**55.8**	39.2	25.8	21/150	124	101

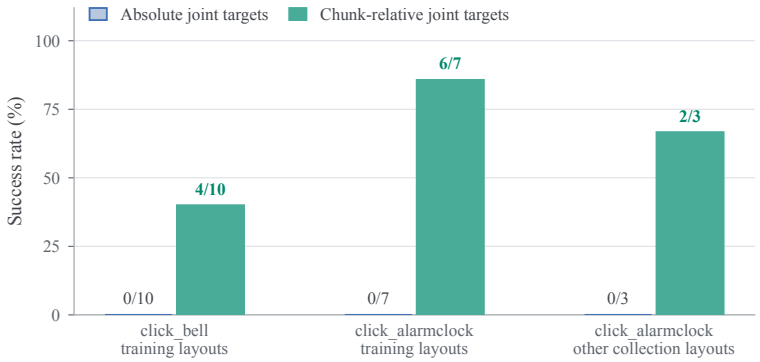


Figure 9: **ACT policies trained on RoboRSI executions.** Success with absolute and chunk-relative joint targets under the same demonstrations, architecture, and training budget.

C.4 Backbone Model

We compare GPT-5.6-SOL, GPT-5.5, and GPT-5.4 across the reasoning roles with the same initial skills, task distribution, and interaction budget (Table 5). GPT-5.5 reaches the highest LIBERO-PRO success, 48.6%, and leads on the Object and Goal suites, while GPT-5.6-SOL is best on RoboTwin. The number of model calls hardly differs, with medians of 115 to 124 per LIBERO-PRO episode, so the stronger backbone does not win by reasoning longer. The difference lies in how often an episode runs out of budget: GPT-5.4 exhausts it in 101 episodes and GPT-5.5 in 80, so the stronger backbone reaches a verified outcome within the budget more often. Because the ranking changes between benchmarks while the task structure and skill library stay the same, the choice of backbone interacts with the benchmark rather than favoring one model throughout.

C.5 Learning Policies from Execution Data

Successful RoboRSI executions provide demonstrations for training compact visuomotor policies, which can then be exposed as skills in the hierarchy. We train ACT policies (Zhao et al., 2023) on 10 successful `click_bell` and 7 successful `click_alarmclock` executions and compare two action representations with the same data, architecture, and 10,000 training steps. With absolute joint targets, neither task is solved in any evaluated layout. Expressing the joint targets relative to the arm configuration at the start of each action chunk raises success to 4/10 on the bell training layouts, and on the alarm clock to 6/7 on training layouts and 2/3 on collection layouts that did not supply training data (Figure 9). With so few demonstrations, an absolute target requires the policy to regress the full arm configuration at every step, and small regression errors move the end effector away from the small contact region that these pressing tasks require. A chunk-relative target encodes only how the arm moves from its current configuration, which the demonstrations constrain far more tightly: on the training observations, the joint-command error falls by 57%, from 0.0086 to 0.0037 rad. Appendix E.5 gives the training details.

D PHYSICAL EVALUATION SETTING & PROTOCOL

Platform and observation interface. The physical experiments use two robot platforms. The mobile cleanup case study uses a WheelSingleArm M1 comprising an Athena Pro Max mobile base, a RealMan arm, and a Zhiyuan gripper. An in-hand camera and a chassis-mounted camera provide RGB-D observations. The agent can access arm-joint and end-effector states, gripper state, chassis pose, and navigation and docking status. Camera-to-arm calibration was performed before the experiments. All mobile development runs used the same prebuilt navigation map.

The dual-arm setup comprises two AgileX PiPER robotic arms with standard 100-mm PiPER grippers. An Intel RealSense D435i facing the workspace and an Intel RealSense D405 mounted on each wrist provide RGB-D observations. All three camera streams are available to the agent, together with joint states, end-effector poses, and gripper states for both arms. Camera-to-arm calibration was performed before the experiments.

Mobile task distribution and development protocol. The mobile cleanup development record spans 104 runs. Each run starts from a common docking pose. Before execution, on-site personnel restore the scene to a common base layout while varying the number, identities, positions, and orientations of the objects, as well as the positions of their destination receptacles. The task manifest specifies the objects and destinations for that run and remains fixed during its execution. The robot is expected to place the specified objects in their designated receptacles and return to the docking pose.

RoboRSI may revise its skills between runs, while the skill version remains fixed within each run. Revisions are made when needed rather than after every run. Candidate revisions are checked against retained, reviewed execution records before use in subsequent runs; earlier tasks are not physically repeated for this check. Thus, the 104 runs trace development of an evolving system across varying task manifests, rather than repeated trials of a frozen policy on an identical task.

Outcome annotation and figure construction. A placement is counted when on-site personnel observe the object enter its designated receptacle. For every run selected for Figure 4, each counted placement was confirmed in this way. The figure shows ten runs selected retrospectively to represent early partial execution, multi-object placement, setbacks during development, and later completed tasks. Selection was made after inspection of the development record, not according to a prespecified sampling rule.

The plotted value is the number of confirmed placements among the first four items handled in a run. Four items provide a common display scale; a run with fewer than four items in its manifest is not treated as a four-item trial. Marker appearance indicates the audited task outcome. Because object sets and reset configurations vary across runs, the plotted values describe placement progress during development rather than success rates on a fixed task.

Mobile execution records. The recorded demonstration comprises two cleanup scenes with five and four target objects, respectively (Figure 5). On-site personnel confirmed that every object was placed in its designated receptacle. Between scenes, an operator rearranged the objects and receptacles; the skill code, prompts, and policy configuration remained unchanged. Neither cleanup execution required manual takeover or correction.

A separate physical evaluation attempt illustrates a placement failure: the robot grasped a bath pouf but released it outside the target bucket. The execution trace exposed a mismatch between the release check and the required final state, directing a subsequent revision of the placement geometry. The revision was checked using retained observations, but was not evaluated in a matched post-repair physical trial. This failure case is reported separately from the two-scene demonstration and the 104-run development trace.

E SUPPLEMENTARY EXPERIMENTS

This appendix reports the LIBERO suite breakdown, a one-day self-iteration run, the failure classification, exploratory development, and small-scale policy learning.

E.1 LIBERO Results by Suite

Table 6 splits the LIBERO results of Table 1 by suite; each suite has 10 tasks and 50 episodes per method.

Table 6: **LIBERO by suite.** Successful/valid episodes per suite.

Method	Spatial	Object	Goal	All
CaP-X	12/50	15/50	9/50	36/150
Maestro	22/50	24/50	16/50	62/150
OpenETA	23/50	29/50	**24/50**	76/150
RoboRSI	**30/50**	**33/50**	21/50	**84/150**

E.2 One-Day Self-Iteration

To examine self-iteration without human steering, we let RoboRSI iterate on its own for one day on a 120-task LIBERO catalog and on the 120 LIBERO-PRO tasks. In each round, the current library runs on the tasks not yet solved, and revisions that pass validation are released for the next round. Figure 10 reports the cumulative number of tasks solved at least once. Within five rounds, coverage grows from 32 to 71 of 120 LIBERO tasks and from 43 to 80 of 120 LIBERO-PRO tasks; the five LIBERO rounds take 7.0 hours of active execution. These counts accumulate across released library versions and are separate from the evaluation in Table 1.

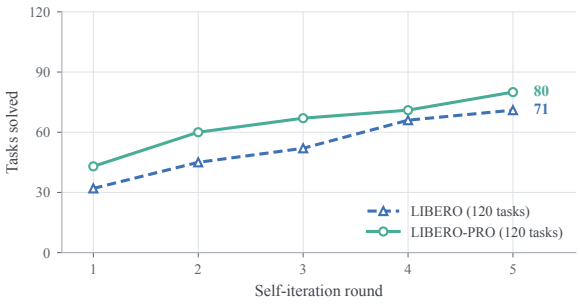


Figure 10: **One-day self-iteration.** Cumulative number of tasks solved at least once after each self-iteration round; each task is counted at its first recorded success.

E.3 Failure Classification

Each failed episode receives one category from deterministic rules over its public tool results and the simulator's terminal outcome. Explicit fields such as an empty grasp, a dropped object, a rejected release, or an unreachable target determine grasp, placement/release, and motion/reachability failures; for complete traces, the last explicit failure step is used. An episode in which the agent declares success while the simulator reports failure is a premature completion, and an episode that ends at the interaction limit is budget exhausted. Episodes whose traces support none of these rules are unlabeled. Categories are counted separately for each evaluation panel; the analysis covers 5,008 failed episodes. Completion after a failure (Figure 6c) is computed on the (task, seed) pairs for which every method has a complete trace and encounters at least one execution failure, 209 on LIBERO-PRO and 258 on LIBERO-Plus; it is the fraction of these episodes that still end in simulator success.

E.4 Exploratory Development

Early LIBERO-Plus runs also allowed capability updates on the target benchmark. Their configurations allowed repeated attempts and target-side updates, whereas the main LIBERO-Plus transfer panel keeps source capabilities fixed.

E.5 Learned Policies from Execution Data

Demonstrations are successful RoboRSI executions on fixed collection layouts: ten for `click_bell`, yielding 2,161 training samples, and seven for `click_alarmclock`. Both action representations use the same ACT architecture, with a ResNet-18 visual backbone trained from scratch, a transformer width of 256, action chunks of 100 steps of which 25 are executed per inference, a KL weight of 1, and 10,000 training steps. The absolute version predicts joint position commands directly. The chunk-relative version predicts joint positions relative to the arm configuration at the start of each chunk, with gripper and velocity outputs unchanged. Final checkpoints run through a standalone inference server without online agent assistance, with a budget of 2,000 control ticks per episode. Evaluation covers the ten training layouts of `click_bell` and the ten collection layouts of `click_alarmclock`, seven of which supplied training demonstrations.

F Case Studies: Skill Code and Failure-Driven Revision

We first compare RoboRSI with the baselines on individual simulation episodes, then examine an execution failure and a local skill revision on the same development task and initial states. Separate cases then document skill publication and review-driven code revision.

F.1 Simulation Episodes Across Methods

We examine the three episodes of Figure 7, in which RoboRSI succeeds while the baselines fail from the same initial state, and then follow a revised skill as it is reused on tasks it was not developed on (Figure 11).

Terminal checks under a lighting perturbation. On LIBERO-Plus, the task is to place a cream-cheese package in a basket. OpenETA's combined pick-and-place tool returned that the object had been released at the destination, and the agent declared success at once. CaP-X's placement also reported a release while its containment check remained unverified. Both episodes failed. RoboRSI, with its frozen source library, received a placement result stating that the gripper had not opened. Rather than ending, it checked whether it still held the package, re-observed the basket from the head and top views, placed the package again, and succeeded.

Dual-arm state after a failed placement. On RoboTwin, both cans must be picked up with separate arms and placed into a plastic box. CaP-X repeatedly tried to reach the box with the left arm across the midline, which was infeasible. OpenETA stopped after the gripper reported a release, with containment unverified, and the task failed. When RoboRSI's first left-arm placement failed, it kept both cans in hand, verified both grasps and the clearance between the loaded arms, and then released the cans one side at a time.

Recovery in a multi-step task. On LIBERO-PRO, the robot must open the top drawer and put a bowl inside. CaP-X invoked the pulling skill on the drawer rather than its handle and was rejected; OpenETA repeatedly failed to approach the handle and then failed to grasp the bowl. RoboRSI also had a pull and a placement rejected, but continued to observe and act until the task succeeded.

In all three episodes, the baselines fail by trusting a tool's return value, whereas RoboRSI's skills expose failed outcomes and its agents verify postconditions before proceeding, the mechanism behind the failure composition in Figure 6a.

Reusing a revised skill across tasks. Revisions made during development are released as shared skills and reach tasks they were never developed on. The base skill `grasp_object` was revised on a single development task: the Manager's patch makes every retry verify the target's identity from a fresh observation and forbids replaying a grasp point that has already failed, and the revision passed its gate before release. After release, 42 successful episodes on other tasks invoke the revised skill. Figure 11 shows three of them, from the Goal, Object, and Spatial suites, each of which had failed in the earlier frozen evaluation. Because the skill is shared through the hierarchy, a repair made in one task becomes available to every branch that calls it.

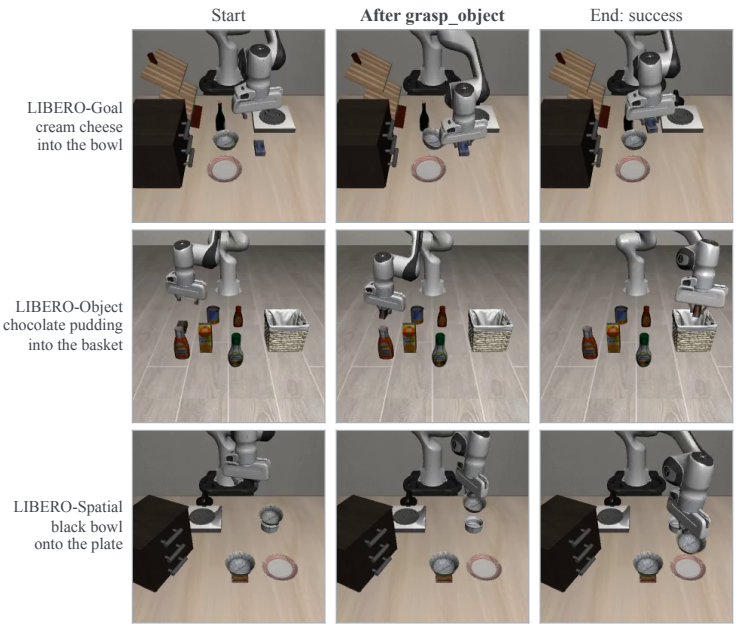


Figure 11: **Reuse of a revised skill on new tasks.** The revised `grasp_object`, developed on a single task, is invoked on three tasks outside its development set, each of which had failed in the earlier frozen evaluation. Columns show the first recorded frame, the state after the `grasp_object` call, and the final successful state.

F.2 Execution Repair: Removing a Premature Target Rejection

This case follows an earlier revision of `guarded_compact_feature_tap` on `click_bell`. Both candidates were reviewed by GPT-5.6-SOL Manager calls and evaluated on the same three declared initial states, invocation arguments, and independent simulator-success criterion. Each version ran once per initial state. The original candidate succeeded in one of the three trials (1/3); the revised candidate succeeded in all three (3/3), with no crashes. It met the predeclared requirement of at least two successes and was then published. These six runs form the development comparison below.

Failure and responsible node. In the first failed trial, the original skill observed a bell bounding box $[249, 185, 306, 240]$ in a 320×240 image. Its lower edge exceeded the hard-coded threshold of 237, so the target-association check rejected the object before issuing a tap. The task consequently failed. The responsible code lay in the skill's association checks following `get_object_bbox`; the bell-pressing goal and tool interface did not need to change. The original size and boundary checks were:

```
if width < 8 or height < 8 or width > 100 or height > 100:
    return fail("object association is not within fixed compactness bounds")
if x0 < 2 or y0 < 2 or x1 > 317 or y1 > 237:
    return fail("object association touches the fixed head-camera border guard")
```

Local revision and observed execution. After receiving the failed runs' public tool traces, the Manager generated a revision that allows bounded lower-edge truncation for a compact object. The revised checks below retain size limits and reject extensive truncation; feature-in-box, depth, and arm-state checks remain elsewhere in the skill, and the revision also adds motion-endpoint checks.

```
if width < 8 or height < 8 or width > 100 or height > 100:
    return fail("object association is outside fixed compactness bounds")
if x0 < 2 or y0 < 2 or x1 > 318 or y1 > 240:
    return fail("object association violates fixed image guards")
if y1 == 240 and height > 70:
    return fail("lower-edge truncation is too extensive")
```

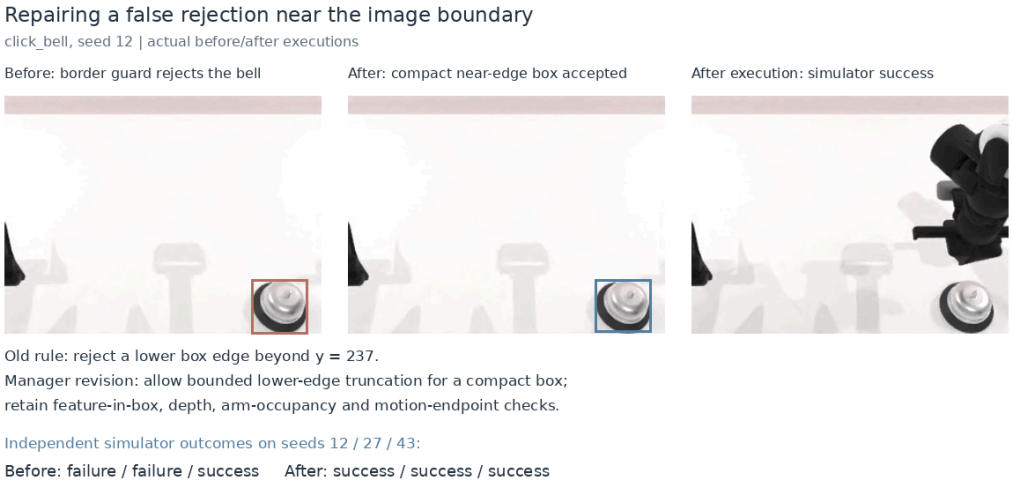


Figure 12: A local skill revision enables execution in the bell-pressing case. Left: the original target-association check rejects the near-boundary box before action. Middle: the revised check accepts the compact target. Right: the post-execution observation from the successful revised run. Boxes mark the detected targets; the simulator supplies the task outcomes.

In the revised run from the same initial state, the observed box was $[249, 185, 306, 239]$ and the localized feature was $(257, 195)$. The skill accepted the target, called `move_to_pixel` with `action=tap`, and received the hover–close–press–retreat execution report. The independent simulator verdict was success. Figure 12 shows the corresponding observations. Replaying the old code on this same recorded post-revision observation still rejects the box because $239 > 237$; the revised code replays all 22 recorded public calls, including the tap. The replay uses stored tool outputs to compare the two implementations on the same observation.

Evaluation setting. A supervisor supplied the complete tool arguments, outputs, and failed trace to the Manager and organized the two validation runs. The Manager revised the code, followed by a separate review. The patch updated the boundary checks and recovery logic; in the second trial, both versions attempted a tap. The native tools, dispatch, and harness code were unchanged. Runs used different GPUs and slightly different initial images, including a one-pixel difference in the box boundary of the first trial. Process limits were 300 and 900 seconds, with actual durations of approximately 231 and 297 seconds.

F.3 Generated Compound Skill: Validation, Publication, and Invocation

A later, distinct `exact_waypoint_feature_tap` skill provides a concrete example of executable capability accumulation. A GPT-5.6-SOL Manager call generated the revised code, and a separate Manager call reviewed it. The descriptor exposes four inputs: arm, object description, functional feature description, and camera. The implementation obtains a fresh image, localizes the object and its functional point, recovers depth, checks motion endpoints, executes a tap, and checks retreat and the terminal state. Its structured outputs describe execution, recovery, and failure reasons. The simulator evaluates task completion.

The following source excerpt shows the fixed execution sequence after perception. Here `xyz` is the currently observed feature position, `u, v` its image coordinates, and `screen_tcp` checks the candidate endpoint. Input validation, perception, and subsequent terminal checks are omitted from this excerpt. The code preserves the procedure, while object association and action targets are recomputed on each call.

```
top_down = [0.5, -0.5, 0.5, 0.5]
approach_tcp = [xyz[0], xyz[1], xyz[2] + 0.10] + top_down
press_tcp = [xyz[0], xyz[1], xyz[2] - 0.005] + top_down
```

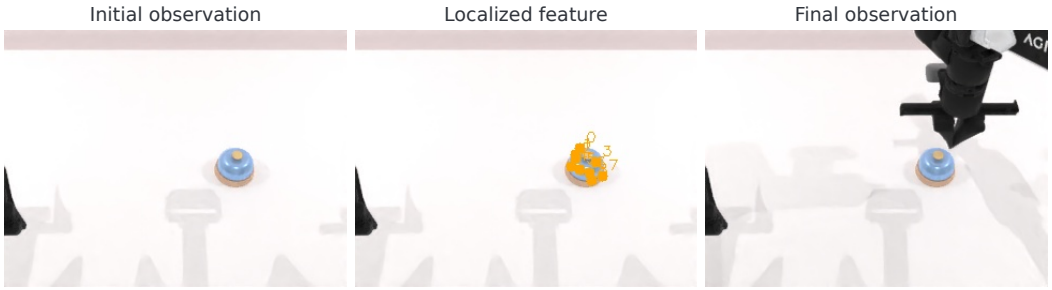


Figure 13: Observations from the published tap skill in a successful `click_bell` development trial: the initial view, feature localization, and post-execution view. The outcome is scored by the simulator; images are 320×240 pixels.

```
if screen_tcp(approach_tcp) is None:
    return fail("exact tap approach/retreat endpoint is unsafe or unknown")
if screen_tcp(press_tcp) is None:
    return fail("exact tap press endpoint is unsafe or unknown")

tap_result, _ = _dispatch_tool(
    state,
    "move_to_pixel",
    {
        "arm": arm,
        "u": u,
        "v": v,
        "action": "tap",
        "height_above_m": 0.0,
        "camera": "head_camera",
    },
)
tapped = tap_result.get("ok") is True
```

Validation and publication. The predeclared development check required independent simulator success on at least two of three `click_bell` trials. Two of the three trials succeeded, and no run crashed. The 2/3 result passed that check, after which the reviewed implementation was published as a versioned skill. This later skill uses its own three-trial development panel.

Figure 13 shows the observations from a successful development trial.

F.4 Before/After Code: Review-Driven Terminal Checks

The earlier candidate was rejected during Manager review. Its recovery expression short-circuited: when a movement or pose check failed, the later occupancy and clearance queries could be skipped. Elsewhere, clearance could be queried with empty-arm assumptions after occupancy was not confirmed. The old recovery condition was:

```
recovered = (
    recovery_move.get("ok") is True
    and recovery_measure.get("ok") is True
    and recovery_measure.get("pose_matches") is True
    and both_empty()
    and current_clear()
)
```

The model-generated revision evaluates terminal occupancy after each recovery attempt, then checks clearance only if both arms are confirmed empty. It uses these recorded checks when deciding whether recovery was established. The helper and revised condition are shown below; intervening motion calls are omitted.

```
def terminal_empty_clear():
    empty = both_empty()
    if not empty:
        return False, False
    return True, empty_current_clear()
```

```
recovery_empty, recovery_clear = terminal_empty_clear()
recovered = (
    recovery_move.get("ok") is True
    and recovery_measure.get("ok") is True
    and recovery_measure.get("pose_matches") is True
    and recovery_empty
    and recovery_clear
)
```

A separate Manager review accepted this version, which passed the 2/3 simulator development check before publication. The previous candidate stopped at code review and had no simulator runs. The full patch also changes diagnostics and removes an optional visual-difference check. An operator separately maintained the camera, low-level interfaces, and supervisor. The code excerpts show the terminal-check revision.

F.5 Physical Skill Revisions in Rounds 25 and 33

Both records come from the household cleanup task on the mobile manipulator (Section 4.1) and are shown in Figure 3b. Each revision was admitted with its trigger run, the four most recent eligible runs as a replay buffer, and the changed files; live iteration was paused until validation finished.

Round 25: placement observation and release. *Execution.* The robot held a food wrapper in front of the correct bag-lined trash bin, which remained visually distinct from a nearby light-blue bucket. The compiled observation move reported a 0.052 m and 17.1° tolerance miss, although the current joint and end-effector state and the third-person video showed the arm at the requested pose. The unrestricted fallback repeated opening estimation from the same view, invoked Arm Explore with an invalid self-inspection target, and exhausted 28 merged tool calls. A workspace probe had already stored a complete three-step release sequence in runtime memory, but the model passed an incomplete copy to `place_object`. Both calls failed closed before motion; the gripper never opened, and the robot returned to its docked home pose with the item still held. *Diagnosis.* The earliest failing node was completion of the placement observation motion. Its false negative expanded into duplicated geometry estimation and an incomplete release plan. *Revision.* (i) A failed observation move receives one read-only pose recheck, and a physically reached pose continues without replaying the placement chain. (ii) An unreached pose returns a non-releasable checkpoint to the Planner, with Engineer fallback disabled. (iii) Probing and release above a container require opening geometry from the current observation rather than semantic rim identity alone. (iv) `place_object` replaces an incomplete model-written sequence with the complete sequence held in runtime destination memory. *Validation.* Five new regression tests, 172 placement tests, 863 manager tests, and 97 real-robot interface tests passed. The next run required a fresh human reset of the scene.

Round 33: destination identity and navigation handoff. *Execution.* A single sweep over five headings recorded all eight expected scene entities. The first item was released and its detachment confirmed, but the run ended after about 25 minutes with the second item still held. Two owners failed. The inventory producer chose an isolated but edge-cropped view of the destination bucket although the same sweep contained a complete view, and the consumer then treated the blue color ratio as a mandatory check and rejected the producer-certified destination. Earlier, a navigation to the item had been attempted but not completed and produced no post-navigation image; this state was routed to Engineer fallback, which repeated status, power, and camera queries and exhausted the merged budget before a single compiled retry succeeded. *Revision.* Inventory selection prefers complete, non-edge observations; query color becomes advisory during materialization because the producer owns semantic identity, while current-frame navigation and manipulation checks remain in force; and incomplete navigation without a post-navigation image returns a typed checkpoint to the Planner instead of entering fallback. The revision adds no new object or destination branch, no episode coordinates, no historical pose gate, and no change to global color thresholds. *Validation.* An offline replay of the trigger run selected the complete bucket view, materialized the destination, and routed the incomplete navigation to the Planner. The replay buffer contained four consecutive blocked runs with different terminal failures: a true destination rejected by a score veto, a defensive post-release check, exhausted placement regions, and the color veto above. Twenty-eight focused



(a) Round 23: item released outside the container

(b) Round 31: grasp approach under a speed limit

(c) Round 37: wrapper lands beside the bin

(d) Round 99: item still held after termination

Figure 14: **Physical corrections that required human knowledge.** (a) Resetting the scene after a confirmed misplacement; (b) a safe grasp speed specified by the operator; (c) a wrapper released outside the bin; (d) holding protection after termination.

tests, 866 manager tests, and 97 real-robot interface tests passed. Resets before and after the run were planned; no unplanned human prompt occurred during execution.

F.6 Corrections That Required Human Knowledge

Four of the 53 classified revision triggers in the physical development (Section 4.1) required knowledge that the system could not obtain from its own records (Figure 14). In Round 23, a released item landed outside the intended container. The operator specified that a confirmed misplacement contaminates the scene and must be reset by a person, rather than being treated as a new source object for the robot to recover. In Round 31, the grasp approach had to respect a default linear speed of 0.02 m/s that the validated motion sequence did not yet consume; the operator supplied and confirmed this safety limit. In Round 37, a wrapper fell beside the bin after release, and the operator judged the placement a failure that required a scene reset, not a success or a step for automatic recovery. In Round 99, the robot still held an item when the task stopped, and the item was later lost while the robot waited at its dock. The operator specified that holding protection must inherit the reviewed grip width and force and, when the hold is lost, freeze motion and request confirmation instead of moving or releasing. Each correction is kept as a reusable contract, checkpoint, or test on the responsible skill, so the same prompt is not needed again.

G Supplementary Material and Reproduction

The supplementary material contains the code, de-identified experiment records, and videos.

Code. The RoboRSI code includes the Manager, Planner, Engineer, and Reviewer roles, the task–skill hierarchy with its base, atomic, and compound skills, the LIBERO, LIBERO-PRO, LIBERO-Plus, and RoboTwin adapters, and the evaluation and audit tools. Self-improvement runs use the `evolve` mode; frozen transfer uses the `eval` mode, in which no skill, memory, or proposal is written back. Success is taken only from the simulator's post-episode verdict, and an audit command recomputes success from the append-only episode journal. The accompanying README gives the installation steps and the commands for each benchmark, including a script that installs the LIBERO-PRO assets, starts the required services, runs the campaign, and audits its journal.

Experiment records. The records include per-task results on LIBERO-PRO and per-instance results on LIBERO-Plus (Table 1); the failure category of every failed episode and the per-episode recovery records (Figure 6); per-episode backbone results (Table 5); the matched Code-on/off counts; the code patches, layer selection, validation results, and Manager reviews of the TSR controlled repair (Table 4); the ACT evaluation episodes; and the tool traces and frames of the simulation case studies (Appendix F.1). Machine paths, host names, and user names are replaced by placeholders; run identifiers and code revisions are kept.

Videos. The videos show the two-scene real-robot cleanup (Section 4.1), the evolution of the physical task–skill tree (Figure 3a), and successful LIBERO-Plus episodes under camera, robot-initialization, layout, and lighting perturbations. The robot base is blurred for anonymity, and the interval in which the scene is rearranged is removed.

H EVALUATION DESIGN

Choice of baselines. RoboRSI accumulates skills serially: each episode runs with the current library, and revisions that pass validation are released to the episodes that follow. To our knowledge, no existing robot-agent system accumulates skills in this serial form. ASPIRE (Lu et al., 2026), the closest self-improving code agent, learns every task in parallel with a dedicated agent that runs a population-based search over candidate programs, admits the resulting skills after the search, and evaluates one generated program per task; ENPIRE (Xiao et al., 2026) refines learned policies and their training code through physical experimentation. Both rely on development loops that differ from a single serial stream, so they cannot be run under the same protocol. We therefore compare with the strongest agents that act through code or tools, CaP-X, Maestro, and OpenETA, with the same backbone.

Self-improvement as the evaluated capability. All methods are evaluated under the same conditions: the same backbone, tool interface, per-episode interaction budget, task panels, and simulator verdict. The baselines execute with a fixed set of tools and skills and do not revise or add skills across episodes; RoboRSI differs only in that it can use the outcome of an episode to revise its skills for later episodes. Improving on the tasks at hand is the capability that RoboRSI contributes, so the comparison on LIBERO, LIBERO-PRO, and RoboTwin measures what self-improvement adds over strong fixed agents. LIBERO-Plus complements this by evaluating the library with self-improvement disabled on perturbed instances.

Verification before completion. Checking the postcondition of every atomic task against fresh observations before the agent proceeds is part of RoboRSI's design (Section 3.2 and Appendix B), not an external addition. The lower share of premature completions in Figure 6a is one measured effect of this design.

Evidence for the components. Appendix C isolates the components of the system: role separation (36 versus 9 of 50 RoboTwin tasks), TSR against flat revision on the same failure, compound skills over 600 matched episode pairs ($+7.5$ points, McNemar $p < 10^{-4}$), the backbone model, and policies learned from execution data.

Scale of the evaluation. The main comparison comprises 5,974 evaluation episodes: 150 per method on LIBERO and 600 on LIBERO-PRO, each with five initial states per task, 840 perturbation instances per method on LIBERO-Plus, and 150 episodes per baseline and 154 for RoboRSI on RoboTwin. The ablations add 2,770 episodes, and the physical study covers 104 development rounds. In total, RoboRSI is evaluated on 230 tasks across four benchmarks with 1,744 episodes. For reference, the main LIBERO evaluation of OpenETA uses 40 tasks with ten seeds each (400 episodes), and its self-evolution studies use at most ten tasks (Chen et al., 2026b); ASPIRE evaluates 69 tasks on LIBERO-PRO, Robosuite, and BEHAVIOR-1K (Lu et al., 2026); the memory-augmented harness of Li et al. (2026) is evaluated on 25 tasks, and Huang et al. (2026) on LIBERO, LIBERO-Plus, and five long-horizon tasks.

Implementation details. The postcondition of each skill, the release gate, the construction of the replay history, and the consolidation of compound skills are specified in the released code and in the descriptor of each skill (Appendix G).