RoboRSI: Evolving a guidance Strategy for Robotic Self-Improvement

Anonymous authors
Paper under double-blind review

Abstract

Robotic self-improvement requires using history experience to determine which system improvements are worth pursuing. Evolving the execution system alone does not establish persistent guidance for selecting and reviewing improvements across successive rounds. We introduce RoboRSI, a robotic self-improvement framework that learns and evolves an explicit system-level strategy: a continually updated set of rules for selecting and reviewing improvements to task-planning guidance and robot skills. RoboRSI learns this strategy from historical traces linking execution outcomes, improvement decisions and their effects, and validates strategy updates through the downstream improvements they induce on held-out cases. Across four robotic benchmarks, RoboRSI improves task success rate by an average of **15.4** percentage over its no-strategy counterpart. Controlled ablations show an **8.2** percentage gain from maintaining the strategy under matched access to historical experience on LIBERO-Pro, and a **10.6** percentage gain from continually updating the strategy over keeping it fixed on RoboCasa365. Code is available at `https://anonymous.4open.science/r/RoboRSI-review-C9F5/`.

1 Introduction

Recent advances in large language models (LLMs) in reasoning, code generation, and tool use have expanded their role from language generators to agents that can interpret task states, invoke external tools, and make decisions based on interaction outcomes (Feng et al., 2025; Jin et al., 2025; Liang et al., 2023; Tu et al., 2026; Yang et al., 2024). In robotics, these capabilities allow LLMs to serve as high-level decision-making modules that select and invoke perception and control tools according to task requirements, thereby coordinating robot behavior to accomplish complex tasks (Jia et al., 2026; Lima et al., 2026; dos Santos et al., 2026; Shi et al., 2025; Chen et al., 2026). The execution trajectories and outcomes generated during interaction can further be retained as historical experience and reused in subsequent planning and decision-making, allowing robotic systems to adapt their future behavior based on prior interactions. Recent work on robotic self-improvement has begun to exploit this capability more directly. ASPIRE diagnoses failures from execution traces, repairs robot control programs, validates the repairs, and distills successful fixes into reusable skills (Lu et al., 2026). RoboHarness uses execution memory to characterize policy capabilities and improve policy routing and orchestration (Huang et al., 2026). EmbodiSkill performs skill-aware reflection over execution trajectories to produce targeted skill revisions (Ju et al., 2026). SHAPER continually evolves reusable skills and a context-code harness while keeping the underlying foundation-model parameters frozen (Wang et al., 2026).

These methods allow robotic agents not only to accomplish tasks using their existing capabilities, but also to continually improve the system using historical experience accumulated through iterative improvement and execution. This, however, does not mean that the system has learned *how it should improve*. Learning how to improve requires the system to use historical experience to determine which improvements are worth pursuing and to carry this judgment forward into subsequent improvement rounds. If the direction of each improvement is determined only by the problem at hand, then even after multiple rounds of development, the system has limited ability to turn past experience into sustained guidance for improving its overall capability. Robotic self-improvement

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.

therefore requires more than using historical experience to generate new improvements. It also requires learning a system-level improvement strategy that is continuously updated across improvement rounds, guides and reviews each specific improvement from the perspective of overall system capability, and adapts according to the effects of previous improvements.

Although recent robotic self-improvement approaches use historical experience to inform subsequent improvements, they do not explicitly maintain a system-level improvement strategy that is continuously updated across development rounds to coordinate and review specific improvements. To address this limitation, we introduce $RoboRSI$. Its central idea is to explicitly separate the system-level improvement strategy from the robot execution system and make the strategy itself an object of continual improvement. Rather than generating each improvement directly from the current problem, RoboRSI refers to the system-level improvement strategy throughout the improvement process: the strategy guides the direction of the improvement and reviews the resulting change from the perspective of overall system capability. When a specific improvement is inconsistent with the existing strategy, the system can further revise that improvement. At the same time, the strategy is not a fixed set of rules. It is continuously updated based on previous improvements and their subsequent effects. In this way, the system learns not only *how to address the current problem*, but also gradually develops a system-level judgment of *which improvements are worth making* and continually applies this judgment to subsequent improvement rounds.

Continuously updating the system-level improvement strategy requires more than recording problems encountered during task execution. The system must also retain which improvements were made and what effects those improvements subsequently produced. RoboRSI therefore maintains two complementary forms of historical traces: iteration traces and improvement traces. They allow the strategy to connect problems, improvement actions, and their downstream outcomes. After each improvement round, the system uses the current strategy to review the specific improvement and then evaluates its observed effect to determine whether the strategy itself should be updated. Historical experience therefore serves not only to generate the next specific improvement, but also as continuing evidence for the system's judgment of *how it should improve*.

We evaluate RoboRSI on LIBERO-Pro (Zhou et al., 2026), Robosuite (Zhu et al., 2025), RoboCasa365 (Nasiriany et al., 2026), and RoboTwin 2.0 (Chen et al., 2025). It achieves the best results among the evaluated methods on LIBERO-Pro and RoboCasa365 and improves success rates over RoboRSI w/o Strategy P by 8.6–24.1 percentage points across the four benchmarks. A controlled ablation on LIBERO-Pro shows that maintaining the improvement strategy yields an additional 8.2-point gain over a no-strategy variant with matched access to iteration and improvement traces. On RoboCasa365 Atomic-Seen, continually updating the strategy improves success rate by 10.6 points over keeping it fixed. Later strategies also produce stronger execution systems when held fixed and used to guide fresh copies of the initial system under the same improvement protocol. These results support learning how to improve by continually refining the system-level strategy from the outcomes of previous modifications.

2 METHOD

2.1 SYSTEM OVERVIEW

RoboRSI improves a robot execution system H_t between task episodes. Its editable components are *task-planning guidance* and *registered skills*. Task-planning guidance specifies how to select and compose skills for a task. A registered skill is a reusable callable routine with a supported input–output interface. Perception skills (Appendix E.1) expose information such as object locations and observable state checks; local-planning skills use observations and perceived geometry to produce motion programs. An execution-system component therefore refers to the task-planning guidance or an individual skill.

An *Orchestrator* (Appendix E.1) uses this guidance to select an ordered skill workflow. Perception outputs inform subsequent steps, and motion programs run through a fixed execution backend. For example, a workflow may localize an object and use its estimated geometry to plan a grasp. The skill sequence remains fixed during an episode, although skills consume current observations; a failed step terminates the episode without replanning. System modifications are evaluated between episodes before adoption (Appendix A).

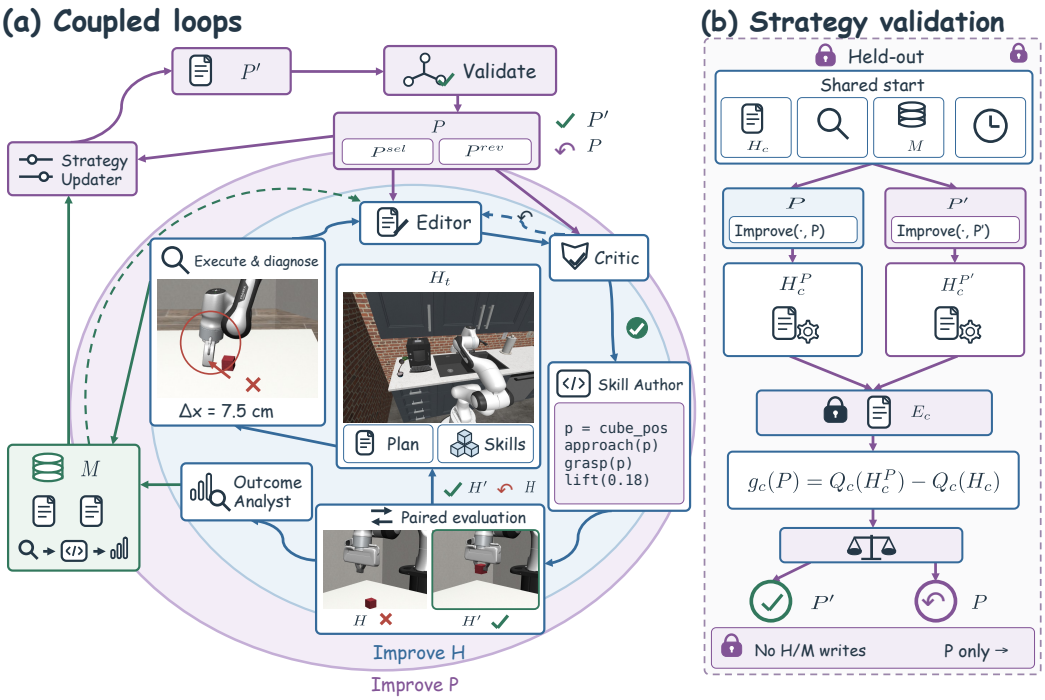

Figure 1: Overview of RoboRSI. The inner loop uses the current improvement strategy P_t to guide modifications of the robot execution system from H_t to H_{t+1} and records their outcomes in historical experience M_t. The outer loop learns a candidate strategy from accumulated improvement traces and accepts it only when it induces better downstream system modifications on held-out improvement cases.

Alongside H_t, RoboRSI maintains an *improvement strategy* P_t and historical experience M_t. The strategy guides changes to execution components: P_t^{sel} guides modification selection by the Editor, and P_t^{rev} guides proposal review by the Critic. These improvement roles are defined below. Unlike task-planning guidance within H_t, P_t is not invoked during robot execution. History M_t records executions, modification decisions, and their measured effects.

The two loops in Fig. 1 couple these objects. The inner loop modifies H_t under P_t and records each attempt in M_t. The outer loop learns from finalized history and validates candidate strategies through the system modifications they induce (Algorithm 1). Initially, H_0 contains the starting guidance and skills, both strategy rule sets are empty, and $M_0 = \varnothing$. Foundation-model parameters and the execution backend remain fixed. Role prompt contracts and runtime message construction are summarized in Appendix E.

2.2 Strategy-Guided System Improvement

One improvement round processes a case c_t through a single logical modification attempt,

$$(H_{t+1}, M_{t+1}) = \text{Improve}(H_t, P_t, M_t; c_t). \tag{1}$$

The attempt may revise task-planning guidance, revise a registered skill, or introduce a new skill within the supported interfaces. A case need not be a failed execution: successful executions may also reveal unnecessary cost, brittle behavior, or limited generality. Algorithm 2 in Appendix B gives the full procedure.

Diagnosis, proposal, and review. A *Diagnoser* (Appendix E.2) analyzes the selected workflow, observations, intermediate skill outputs, and task outcome. The *Editor* (Appendix E.2) uses this diagnosis, admissible historical evidence, and applicable rules in P_t^{sel} to propose a modification,

identifying its target component, expected effect, and expected cost. The *Critic* (Appendix E.3) reviews the proposal using the underlying evidence, its role-admissible history, and P_t^{rev}. It can approve implementation, request additional evidence or a revision, or reject the proposal. Requests remain within fixed limits and belong to the same logical attempt. The *Skill Author* (Appendix E.3) implements an approved proposal within the reviewed scope. Critic approval permits implementation and testing; it does not itself update H_t.

Measured acceptance and outcome analysis. The host compares the candidate with the parent H_t using matched initial states and a common admissible task-memory view. The evaluation panel includes affected cases, which test the intended benefit, and declared regression cases, which test whether the modification harms other behavior. Acceptance requires a positive gain on affected cases, regression within the fixed tolerance, and a nonnegative pooled gain. Only then does the candidate become H_{t+1}. Rejection, exhausted revision limits, failed or out-of-scope implementation, incomplete evaluation, or failure to meet the acceptance conditions leaves $H_{t+1} = H_t$.

The *Outcome Analyst* (Appendix E.4) compares the proposal's predicted effect with the available measurements and finalizes the improvement trace. Attempts that never reach evaluation are marked unmeasured, while measured failures retain their observed effects. Iteration and improvement traces are appended to obtain M_{t+1} before any strategy update is considered. This ordering ensures that the outer loop receives finalized evidence from the completed attempt.

2.3 Historical Experience

Historical experience M_t contains *iteration traces* and *improvement traces*. An iteration trace records one execution: its workflow, observations, intermediate skill outputs, and outcome. An improvement trace links the motivating evidence to the proposal, predicted effect, review, implemented change, paired measurements, and outcome analysis. It thus connects an observed problem, an improvement decision, and its measured effect. Revisions remain within one logical trace, and rejected or unsuccessful attempts are retained.

Records preserve their source system versions and component dependencies. Each role receives only the history admissible to its current decision, while the Strategy Updater receives a bounded cross-case view of finalized improvement traces. This history supports both successful guidance and counterexamples; strategy-validation traces never enter it. Appendix C details retrieval, provenance, compatibility checks, and the distinction between unmeasured and measured attempts.

2.4 Learning and Validating the Improvement Strategy

Strategy representation and use. The strategy $P_t = \{P_t^{\text{sel}}, P_t^{\text{rev}}\}$ contains conditional selection and review rules. Each rule specifies its applicability, guidance, and supporting improvement evidence. Compatible selection rules go to the Editor and review rules to the Critic; if none applies, the role uses its fixed prompt and available evidence. Appendix D specifies dependency filtering, natural-language applicability judgments, and rule-use provenance.

Updating the strategy. The initial strategy P_0 contains no learned rules. After every K completed improvement rounds, provided that finalized history covers at least two distinct improvement cases, a *Strategy Updater* (Appendix E.4) receives the current strategy together with a bounded cross-case history of finalized improvement traces and their outcome analyses. It examines how previous improvement decisions relate to their predicted and measured effects, using successful outcomes as supporting evidence and counterexamples to narrow or revise existing guidance. New or substantively revised rules must be grounded in measured improvement evidence available to the current update.

Each candidate P' revises either P^{sel} or P^{rev} while holding the other component fixed. Candidates that violate this restriction or the evidence requirements are not admitted to validation. After the inner-loop transition in Eq. 1, a scheduled, admissible candidate is passed to Algorithm 1; its returned strategy becomes P_{t+1}. If no update is due, sufficient cross-case history is unavailable, or validation rejects the candidate, $P_{t+1} = P_t$. Appendix D details rule-evidence requirements and retained references.

Algorithm 1 Matched downstream validation of an improvement strategy

Require: Strategies P, P'; held-out cases $\mathcal{V}$ with frozen shared inputs and limits
Ensure: Retained strategy P or accepted candidate P'

if $|\mathcal{V}| < n_{\min}$ **then**
  return P
end if
for each $c \in \mathcal{V}$ **do**
  $(H_c^P, M_c^P) \leftarrow \textsc{Improve}(\text{copy}(H_c), P, \text{copy}(M_c); c)$
  $(H_c^{P'}, M_c^{P'}) \leftarrow \textsc{Improve}(\text{copy}(H_c), P', \text{copy}(M_c); c)$
  Record both branches' gains g_c, regression gains r_c, and total costs C_c
  if either branch fails validity or complete positive-cost checks **then**
    return P
  end if
end for
Compute d_c and L using Eq. 3
if the acceptance conditions in Eq. 4 hold **then**
  return P'
end if
return P
All validation systems and traces remain isolated from the active H/M.

Validating downstream improvement effects. Algorithm 1 compares P and P' on a set $\mathcal{V}$ of held-out improvement cases. For each case c, both branches use the same parent system H_c, execution evidence, diagnosis, admissible historical view M_c, and modification limits. These inputs are frozen before the comparison. Each branch runs Algorithm 2 on private copies of the system and history, reusing the shared execution evidence and diagnosis; only the supplied strategy differs. Both branches use the same bounded proposal-revision mechanism and evaluation protocol.

Let H_c^P denote the resulting branch output when strategy P is used, with $H_c^P = H_c$ if no proposed modification is accepted. For a sealed evaluation panel $\mathcal{E}_c$ containing affected and regression task–initial-state pairs, we define

$$Q_c(H) = \frac{1}{|\mathcal{E}_c|} \sum_{j \in \mathcal{E}_c} s_j(H), \qquad g_c(P) = Q_c(H_c^P) - Q_c(H_c), \tag{2}$$

where $s_j(H)$ is the binary task-success outcome for pair j. Thus, $g_c(P)$ measures the realized gain produced by the system modification induced under strategy P, rather than the apparent quality of the strategy itself.

Strategy acceptance. Let $C_c(P) > 0$ denote the measured branch cost, including proposal and review calls, implementation, outcome analysis, and paired robot execution. For $n = |\mathcal{V}|$ distinct validation decisions, the cost-adjusted comparison is

$$d_c = \frac{g_c(P')}{C_c(P')} - \frac{g_c(P)}{C_c(P)}, \qquad L = \overline{d} - z\frac{s_d}{\sqrt{n}}, \tag{3}$$

where $\overline{d}$ and s_d are the sample mean and standard deviation of the differences. A candidate is accepted only if all branches pass validity and complete-cost checks and

$$\begin{aligned} n \geq n_{\min}, \qquad & L > \epsilon, \\ g_c(P') \geq g_c(P), \qquad & r_c(P') \geq \min\{0, r_c(P)\} \quad \text{for every } c \in \mathcal{V}, \end{aligned} \tag{4}$$

where $r_c(P)$ is the realized gain on the regression subset of $\mathcal{E}_c$. The absolute-gain guard prevents acceptance based only on cheaper but less useful modifications, while the regression guard prevents introducing a regression where the incumbent has none or worsening an existing one. The defaults are $n_{\min} = 20$, $z = 1.96$, and $\epsilon = 0$ (Appendix D). Algorithm 1 retains the current strategy if any required check fails.

Only the accepted strategy is activated. Validation systems and traces never replace the active H or enter M, whether the candidate strategy is accepted or rejected. Later updates use fresh affected validation cases, and final benchmark evaluation uses an independent partition after the system and strategy are frozen (Appendix D).

3 EXPERIMENTS

Our experiments examine whether explicitly maintaining a system-level improvement strategy helps a robot execution system improve itself, and where such a benefit comes from. We first evaluate whether introducing an improvement strategy leads to a stronger robot execution system. We then examine whether this gain comes from the improvement strategy itself, rather than simply from access to richer historical experience from previous improvements. Beyond the presence of an improvement strategy, we study whether continually updating P_t provides an advantage over keeping the strategy fixed, and how the robot execution system evolves over successive self-improvement rounds under the dynamically updated strategy.

3.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate RoboRSI on LIBERO-Pro, Robosuite, RoboCasa365, and RoboTwin 2.0. On LIBERO-Pro, we use the Object, Goal, and Spatial suites, with 10 tasks in each suite. Seeds 51–65 are used for improvement and seeds 1–50 for evaluation. On Robosuite, we use Lift, Stack, PickPlace, NutAssembly, Door, and Wipe; improvement uses seeds 101–125 and evaluation uses seeds 1–50. On RoboCasa365, we use the Atomic-Seen, Composite-Seen, and Composite-Unseen tasks under the pretraining-scene split. Seeds 0 and 1 are used for improvement, while evaluation uses seeds 2–10 for Atomic-Seen and seeds 2–5 for both Composite-Seen and Composite-Unseen. On RoboTwin 2.0, we evaluate 50 tasks. Improvement uses the clean configuration with seed 0, and evaluation uses the randomized configuration with seeds 1–3.

Baselines. We compare RoboRSI with several robot agent systems, including ASPIRE, Harness VLA (Zhang et al., 2026c), RoboHarness, CaP-Agent0 (Fu et al., 2026), and RoboRSI w/o Strategy P. External agent systems follow their original agent and model settings whenever available. RoboRSI w/o Strategy P uses the same robot execution system and improvement procedure as RoboRSI, but does not maintain or use the system-level improvement strategy P_t. Its historical experience retains records of previous system modifications, but does not expose the complete improvement traces containing their subsequent measurements and outcome analyses. All LLM-based roles in RoboRSI and RoboRSI w/o Strategy P use Gemini 3.8 Flash. We additionally compare with the end-to-end VLA policy $\pi_{0.5}$ (Intelligence et al., 2025) using its compatible released checkpoint for each benchmark.

Evaluation protocol. We report pass@1 success rate using the official binary success predicate of each benchmark. Each evaluation instance is executed once without test-time system modification or retry-based selection. During final evaluation, both the robot execution system and the improvement strategy are frozen, and evaluation experience is not added to M_t.

3.2 OVERALL SELF-IMPROVEMENT PERFORMANCE

We first assess whether RoboRSI produces a stronger robot execution system through self-improvement, using final task success rates across the four benchmarks. As shown in Figure 2, RoboRSI achieves the highest success rates on LIBERO-Pro and RoboCasa365. These results exceed those of the strongest external baseline, ASPIRE, by 2.3 and 4.8 percentage. On Robosuite and RoboTwin C2R, RoboRSI remains competitive, reaching 69.0 and 57.3 percentage, within 1.4 percentage of ASPIRE on both benchmarks. Across all four benchmarks, RoboRSI outperforms the external baselines in 18 of 20 comparisons.

The comparison with RoboRSI w/o Strategy P further shows a consistent advantage for the full framework within the same execution and improvement architecture. Success rates increase by 11.8, 17.3, 24.1, and 8.6 percentage on LIBERO-Pro, Robosuite, RoboCasa365, and RoboTwin C2R, respectively, averaging approximately 15.5 points. The largest gain occurs on RoboCasa365,

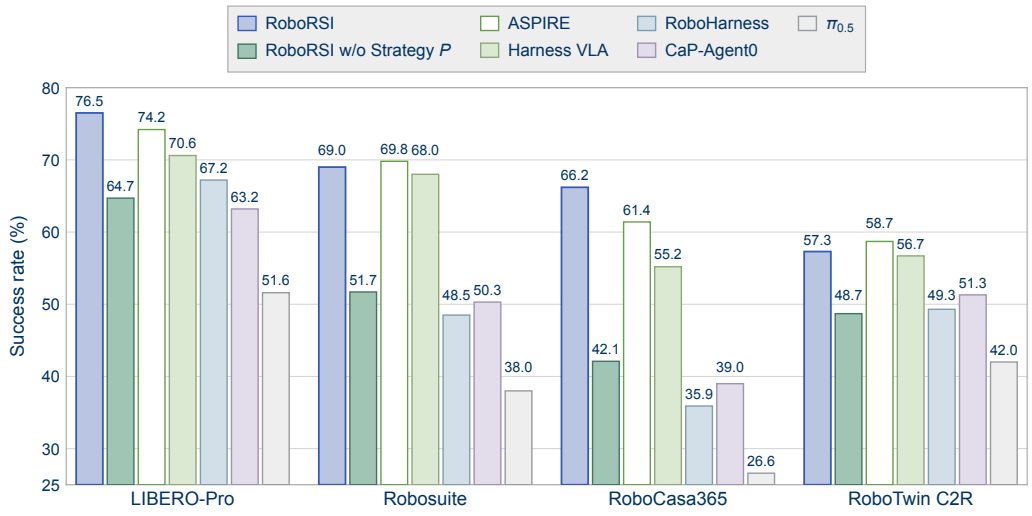

Figure 2: Overall performance across four robotic benchmarks. Bars show pass@1 success rates for RoboRSI, its no-strategy variant, and five external baselines. RoboRSI and its no-strategy variant are evaluated after self-improvement, with the resulting systems frozen during evaluation.

where success rises from 42.1 to 66.2. Although the magnitude varies across benchmarks, the full framework produces stronger final systems in every evaluated environment. To determine how much of this advantage comes from the improvement strategy itself, we next control for access to historical experience.

3.3 Strategy Benefit Beyond Historical Experience

The complete RoboRSI system has access to previous modifications, their reviews and outcome analyses. We therefore ask whether its advantage comes simply from exposing the improvement process to richer historical information, rather than from maintaining an explicit improvement strategy.

We compare three variants on LIBERO-Pro. RoboRSI w/o Strategy P retains the modification history used in section 3.2. RoboRSI w/o Strategy P (Full experience) removes P_t but exposes the improvement process to the same iteration and improvement traces available to the full system, including outcomes and their analyses. The complete *RoboRSI* additionally maintains and applies the system-level improvement strategy. This comparison separates the benefit of richer historical experience from the additional benefit of organizing that experience into an explicit strategy.

As shown in Figure 3, richer historical experience improves self-improvement even without an explicit strategy: exposing the no-strategy variant to the full experience increases the overall success rate from 63.6% to 68.4%. However, RoboRSI further reaches 76.6% with the same historical experience, providing an additional 8.2-point gain from maintaining and applying the improvement strategy. This advantage is consistent across all three LIBERO-Pro suites, where RoboRSI achieves 85.4%, 77.0%, and 67.4% on Object, Goal, and Spatial, respectively. These results show that richer improvement history is useful, but does not by itself explain RoboRSI's gains; explicitly organizing that experience into a persistent system-level improvement strategy provides a further benefit.

3.4 Continual Updating of the Improvement Strategy

We study the effect of continual strategy updating on RoboCasa365 Atomic-Seen. We first compare two variants that start from the same robot execution system: one uses a fixed improvement strategy,

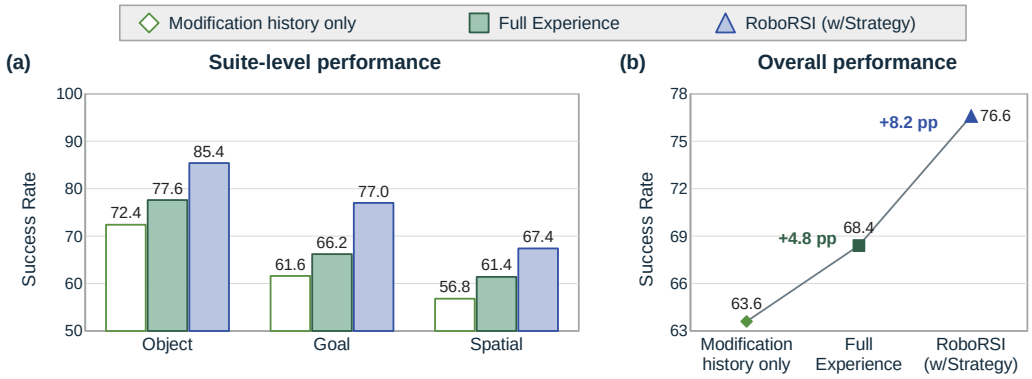

Figure 3: Effect of historical experience and the improvement strategy on LIBERO-Pro. (a) Suite-level success rates on Object, Goal, and Spatial. (b) Overall success rate across the three suites. Providing full historical experience improves the no-strategy variant by 4.8 percentage points, while additionally maintaining the system-level improvement strategy P_t yields a further 8.2-point gain.

while the other updates its strategy throughout self-improvement. Both use improvement seeds 0 and 1, as in the main experiment, with the same improvement budget and access to historical experience.

We then examine whether strategies learned at later stages guide subsequent improvements more effectively. We run RoboRSI from scratch with seed 0 for 5, 10, and 15 improvement rounds. We extract the final strategy from each run. Each strategy is held fixed and used to guide a fresh copy of the initial execution system over the full Atomic-Seen tasks. All three runs begin with empty historical experience and follow the same improvement protocol. We evaluate the resulting systems and measure their success-rate gains over the initial system.

As shown in Table 1, dynamically updating the strategy achieves a success rate of 68.0%, exceeding the fixed strategy by 10.6 percentage points. Later strategy snapshots also yield progressively higher success rates under the same downstream improvement protocol, rising from 59.2% with R5 to 61.1% with R10 and 65.4% with R15. These results suggest that accumulated improvement experience enables the strategy to provide increasingly effective guidance for subsequent system improvements.

Table 1: Success rates (%) on RoboCasa365 Atomic-Seen with fixed and updated strategies, and frozen strategies from rounds 5, 10, and 15.

Continual updating		Extracted Strategies		
Fixed	Updated	R5	R10	R15
57.4	**68.0**	59.2	61.1	**65.4**

4 Discussion

Our results support separating the system-level improvement strategy from the robot execution system and making it an object of continual improvement. RoboRSI produces stronger final execution systems than its no-strategy counterpart across four benchmarks (Figure 2), and its advantage persists under matched access to historical traces (Figure 3). These findings highlight the value of turning accumulated experience into persistent guidance about which changes are worth pursuing. Iteration and improvement traces provide evidence linking encountered problems, modifications, and subsequent outcomes; the strategy uses this evidence to guide and review future improvements from the perspective of overall system capability.

The strategy-evolution results further indicate that this guidance becomes more effective with experience (Table 1). Continual updating outperforms a fixed strategy, while later frozen strategies guide fresh copies of the initial execution system to progressively higher success rates. Because these downstream runs begin with the same initial system and empty historical experience, the latter result suggests that later strategies retain increasingly useful guidance for a new improvement

process. This also suggests a broader criterion for evaluating robotic self-improvement: whether accumulated experience produces better guidance for future development, alongside stronger execution capability. RoboRSI makes this criterion concrete by evaluating learned strategies through the subsequent improvements they enable, connecting its experimental findings to the central goal of learning *how to improve*.

RoboRSI also has several limitations. Although the improvement strategy is not invoked during task execution, learning and validating it requires additional self-improvement-time computation because candidate strategies are evaluated through matched downstream modification branches on held-out cases. Reducing this validation cost is therefore an important direction for settings with limited development budgets. In addition, rule applicability and interactions are currently resolved through natural-language judgments under fixed prompts rather than an explicit rule-priority mechanism, which may become harder to manage as the strategy grows. Finally, the present experiments establish the benefit of strategy evolution on the four evaluated benchmarks, while transfer of the learned improvement guidance beyond these settings remains to be studied.

5 Conclusion

We introduced RoboRSI, a robotic self-improvement framework that maintains and updates a system-level improvement strategy separately from the execution system. It uses iteration and improvement traces to revise the strategy based on observed outcomes, while the strategy guides and reviews subsequent modifications. Experiments across four benchmarks show stronger execution performance than the no-strategy counterpart. Controlled studies show benefits beyond richer historical experience alone and an advantage from continual strategy updating. Later strategies also guide fresh execution systems more effectively, supporting the premise that a robotic system can learn how to improve from the outcomes of earlier changes.

6 Related Work

6.1 LLM Agent Systems and Self-Improvement

LLM agent systems integrate reasoning, tools, and interaction: ReAct interleaves reasoning with actions, while AutoGen coordinates multiple agents through customizable conversation patterns (Yao et al., 2023; Wu et al., 2023). Reflexion stores verbal feedback, while Agent-Pro revises behavioral policies through reflection on past interactions (Shinn et al., 2023; Zhang et al., 2024), and Voyager accumulates reusable executable skills for later tasks (Wang et al., 2023). ADAS searches for agent architectures, while the Darwin Gödel Machine and Hyperagents extend this direction to self-modifying agents and editable improvement procedures (Hu et al., 2025; Zhang et al., 2026a;b). RoboRSI builds on agent self-improvement by learning explicit guidance for selecting and reviewing robot-system modifications.

6.2 Agent-Based Robotic Planning and Improvement

Robot planning agents connect language reasoning to executable actions. SayCan grounds language-based skill selection in learned affordances, and Inner Monologue incorporates environmental and execution feedback into replanning (Ahn et al., 2022; Huang et al., 2022). Code as Policies and ProgPrompt generate executable programs (Liang et al., 2023; Singh et al., 2022); LLM+P and SayPlan combine language reasoning with classical planners (Liu et al., 2023; Rana et al., 2023), while Text2Motion and VoxPoser address geometric feasibility and spatial grounding (Lin et al., 2023; Huang et al., 2023). RoboHarness uses memory to orchestrate heterogeneous policies. ASPIRE, EmbodiSkill, and SHAPER further refine reusable skills or execution harnesses from experience (Lu et al., 2026; Ju et al., 2026; Wang et al., 2026). RoboRSI additionally maintains a separate improvement strategy, revises it from modification outcomes, and validates it through the downstream improvements it induces on held-out cases under matched systems and histories.

AI USE STATEMENT

AI tools were used to assist with manuscript writing and figure preparation. The authors take full responsibility for the final content of the paper.

REPRODUCIBILITY STATEMENT

We have made the code for reproducing the experiments available in an anonymous repository. The experimental setup is described in Section 3.1, with additional methodological details provided in the appendices.

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as I can, not as I say: Grounding language in robotic affordances, 2022. URL https://arxiv.org/abs/2204.01691.

Lihuang Chen, Xiangyu Luo, and Jun Meng. LEO-RobotAgent: A general-purpose robotic agent for language-driven embodied operator, 2026. URL https://arxiv.org/abs/2512.10605.

Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, Weiliang Deng, Yubin Guo, Tian Nian, Xuanbing Xie, Qiangyu Chen, Kailun Su, Tianling Xu, Guodong Liu, Mengkang Hu, Huan ang Gao, Kaixuan Wang, Zhixuan Liang, Yusen Qin, Xiaokang Yang, Ping Luo, and Yao Mu. RoboTwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation, 2025. URL https://arxiv.org/abs/2506.18088.

Vitor Gaboardi dos Santos, Ibrahim Khadraoui, Ibrahim Farhat, Hamza Yous, Samy Teffahi, and Hakim Hacid. ALRM: Agentic LLM for robotic manipulation, 2026. URL https://arxiv.org/abs/2601.19510.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. ReTool: Reinforcement learning for strategic tool use in LLMs, 2025. URL https://arxiv.org/abs/2504.11536.

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, 2026. URL https://arxiv.org/abs/2603.22435.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL https://arxiv.org/abs/2408.08435.

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, 2026. URL https://arxiv.org/abs/2607.18060.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner Monologue: Embodied reasoning through planning with language models, 2022. URL https://arxiv.org/abs/2207.05608.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3D value maps for robotic manipulation with language models, 2023. URL https://arxiv.org/abs/2307.05973.

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, 2025. URL https://arxiv.org/abs/2504.16054.

Mengzhao Jia, Yang Lin, Xixin Zhang, Zhihan Zhang, Xiaobai Liu, and Meng Jiang. Agent as policy for robotic manipulation, 2026. URL https://arxiv.org/abs/2609.12541.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.09516.

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, 2026. URL https://arxiv.org/abs/2605.10332.

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, 2023. URL https://arxiv.org/abs/2209.07753.

Oscar Lima, Marc Vinci, Martin Günther, Marian Renz, Alexander Sung, Sebastian Stock, Johannes Brust, Lennart Niecksch, Zongyao Yi, Felix Igelbrink, Benjamin Kisliuk, Martin Atzmueller, and Joachim Hertzberg. Agentic AI for robot control: Flexible but still fragile. *Proceedings of the AAAI Symposium Series*, 8(1):465–473, May 2026. ISSN 2994-4317. doi: 10.1609/aaaiss.v8i1.42578. URL http://dx.doi.org/10.1609/aaaiss.v8i1.42578.

Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2Motion: from natural language instructions to feasible plans. *Autonomous Robots*, 47(8):1345–1365, November 2023. ISSN 1573-7527. doi: 10.1007/s10514-023-10131-7. URL http://dx.doi.org/10.1007/s10514-023-10131-7.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering large language models with optimal planning proficiency, 2023. URL https://arxiv.org/abs/2304.11477.

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, 2026. URL https://arxiv.org/abs/2607.00272.

Soroush Nasiriany, Sepehr Nasiriany, Abhiram Maddukuri, and Yuke Zhu. RoboCasa365: A large-scale simulation framework for training and benchmarking generalist robots, 2026. URL https://arxiv.org/abs/2603.04356.

Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. SayPlan: Grounding large language models using 3D scene graphs for scalable robot task planning, 2023. URL https://arxiv.org/abs/2307.06135.

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, 2025. URL https://arxiv.org/abs/2511.00917.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

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, 2022. URL https://arxiv.org/abs/2209.11302.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. ScaleEnv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training, 2026. URL https://arxiv.org/abs/2602.06820.

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, 2023. URL https://arxiv.org/abs/2305.16291.

Peidong Wang, Zhiming Ma, Ying Chang, Xufang Luo, Yiqun Zhang, Zihan Wang, Xiaocui Yang, Shi Feng, Yuqing Yang, and Dongsheng Li. Self-evolving embodied agents via skill-harness evolution, 2026. URL https://arxiv.org/abs/2608.11350.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Godel Machine: Open-ended evolution of self-improving agents, 2026a. URL https://arxiv.org/abs/2505.22954.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, 2026b. URL https://arxiv.org/abs/2603.19461.

Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-Pro: Learning to evolve via policy-level reflection and optimization, 2024. URL https://arxiv.org/abs/2402.17574.

Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, Yi Nie, Chunyang Zhu, Jiaxing Qiu, Yuchen Yan, Jiyuan Liu, Wenhao Tang, Jiaji Rao, Zhengru Fang, Changxu Wei, Yu Wang, Wenbo Ding, and Chao Yu. Harness VLA: Steering frozen VLAs into reliable manipulation primitives via memory-guided agents, 2026c. URL https://arxiv.org/abs/2607.08448.

Xueyang Zhou, Yangming Xu, Guiyao Tie, Yongchao Chen, Guowen Zhang, Duanfeng Chu, Pan Zhou, and Lichao Sun. LIBERO-PRO: Towards robust and fair evaluation of vision-language-action models beyond memorization, 2026. URL https://arxiv.org/abs/2510.03827.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning, 2025. URL https://arxiv.org/abs/2009.12293.

A Robot Execution System and Skill Interfaces

The robot execution system H_t contains the components required to plan and execute robot tasks. It consists of task-planning guidance together with registered perception and local-planning skills. Given a task, an *Orchestrator* selects one ordered workflow from the registered skills according to the current task-planning guidance. Workflow length and skill order may vary across tasks, but the selected workflow remains fixed throughout an episode.

Perception skills expose task-relevant information required by subsequent workflow steps, including object localization and observable state checks. Local-planning skills consume the current observations and perceived geometry and produce motion programs executed through a fixed execution backend. A failed workflow step terminates the current episode; RoboRSI does not perform within-episode replanning after such a failure.

Task execution has two roles in the overall system. First, it executes the assigned robot task under the current H_t. Second, it produces the evidence consumed by the self-improvement procedure. This evidence includes the selected workflow, observations, intermediate skill outputs, and final task outcome.

Self-improvement does not modify the fixed execution backend. Instead, an accepted system modification may revise the task-planning guidance, revise an existing registered skill, or introduce a new skill within the supported skill interfaces. The current improvement strategy P_t is not part of the robot execution system and is never directly invoked during task execution. It is used only by the self-improvement procedure to guide which changes to H_t should be attempted and how they should be reviewed.

For a fresh run, H_0 contains the initial task-planning guidance and registered skills. No demonstrations or prior improvement traces are loaded into the system. All foundation-model parameters used by the execution and improvement roles remain frozen throughout the complete self-improvement process.

B Inner-Loop Improvement Protocol

B.1 Improvement Round

Algorithm 2 expands the Improve procedure used by the active transition in Eq. 1 and the isolated validation branches in Algorithm 1. Its local symbols H, P, M, and c denote the supplied parent system, strategy, history, and case; a is the proposal, v is the review decision, and $H^{\mathrm{out}}, M^{\mathrm{out}}$ are the returned system and history. In a validation branch, the execution evidence and diagnosis are prepared once and shared across strategies, while all subsequent writes remain branch-local.

An improvement round processes one improvement case c_t. It starts from a task execution under the current robot execution system H_t and runs at most one logical system-modification procedure, including any bounded proposal revisions requested during review. The resulting execution and improvement evidence is then finalized and appended to historical experience. A scheduled strategy update, when applicable, occurs only after the current system-modification procedure has completed.

Conceptually, one improvement round performs

$$
\begin{aligned}
(H_{t+1}, M_{t+1}) &= \mathrm{Improve}(H_t, P_t, M_t; c_t), \\
P_{t+1} &= \begin{cases} P', & \text{if a scheduled candidate strategy } P' \text{ is accepted,} \\ P_t, & \text{otherwise.} \end{cases}
\end{aligned} \tag{5}
$$

Strategy-validation branches are separated from this state transition: they neither replace the active robot execution system nor contribute traces to M_{t+1}.

B.2 Diagnosis and Modification Proposal

The *Diagnoser* analyzes the execution evidence associated with the current case and identifies plausible causes of the observed outcome. An improvement case need not correspond to a failed execu-

Algorithm 2 IMPROVE: one strategy-guided system-modification attempt

Require: Parent system H, strategy $P = \{P^{\mathrm{sel}}, P^{\mathrm{rev}}\}$, history M, case c;
fixed proposal-revision, implementation, and evaluation limits
Ensure: Retained or accepted system H^{out} and finalized history M^{out}
$H^{\mathrm{out}} \leftarrow H$; initialize one logical improvement trace
Obtain execution evidence and diagnosis for c under H
Validation: reuse the shared frozen evidence and diagnosis.
Retrieve role-admissible history from M; filter strategy rules by compatibility with H
$a \leftarrow$ EDITOR(diagnosis, admissible history, P^{sel})
$v \leftarrow$ CRITIC(a, execution and admissible historical evidence, P^{rev})
while v requests revision or evidence, and the fixed limits permit another review **do**
  if v requests revision **then**
    $a \leftarrow$ EDITOR(a, v, admissible evidence, P^{sel})
  else
    Obtain the requested admissible evidence within the fixed limits
  end if
  $v \leftarrow$ CRITIC(a, updated admissible evidence, P^{rev})
end while
if v approves implementation **then**
  $H' \leftarrow$ SKILLAUTHOR(H, a)
  if implementation succeeds and remains within the reviewed scope **then**
    Evaluate H' and H on affected and declared regression cases
    using matched initial states and a common task-memory view
    if paired measurements are complete and valid **then**
      if affected gain > 0, regression is within tolerance, and pooled gain ≥ 0 **then**
        $H^{\mathrm{out}} \leftarrow H'$
      end if
    end if
  end if
end if
OUTCOMEANALYST(predicted effect, available measurements, attempt status)
Finalize the trace, distinguishing unmeasured attempts from measured failures
$M^{\mathrm{out}} \leftarrow M$ with iteration traces and the single improvement trace appended
return $(H^{\mathrm{out}}, M^{\mathrm{out}})$ ▷ Validation writes remain branch-local

tion. Successful executions may also expose unnecessary cost, brittle behavior, limited generality, or other opportunities for improving the robot execution system.

Based on this diagnosis, the *Editor* proposes a concrete modification. Its decision is conditioned on three sources of information: the current diagnosis, the historical experience admissible to the Editor, and the applicable selection guidance in P_t^{sel}. The proposal identifies the component to modify and states the expected effect and expected cost of the proposed change.

B.3 PROPOSAL REVIEW AND REVISION

The *Critic* reviews the proposed modification using the underlying task-execution evidence, the historical evidence available to its role, and the applicable review guidance in P_t^{rev}. The Critic may approve the proposal for implementation, request additional evidence, request a bounded revision, or reject the proposal.

When a revision is requested, the revised proposal remains part of the same logical improvement attempt rather than creating a new improvement trace. Proposal revision is bounded so that one improvement case cannot induce an unbounded sequence of Editor–Critic exchanges.

If the proposal is approved, the *Skill Author* implements the requested change within the scope reviewed by the Critic. Changes outside the reviewed scope are not admitted into candidate evaluation.

B.4 CANDIDATE-SYSTEM EVALUATION

After implementation, the host evaluates the candidate robot execution system against the parent system H_t. The two systems are evaluated from matched initial states and use a common admis-

sible task-memory view. The evaluation panel contains the cases directly affected by the proposed modification together with declared regression cases.

Candidate acceptance requires all three of the following:

1. the candidate improves performance on the affected cases;
2. degradation on the declared regression cases remains within the fixed regression tolerance; and
3. the pooled gain across the evaluation panel is nonnegative.

If the candidate satisfies these fixed acceptance conditions, it becomes the next active execution system H_{t+1}. Otherwise,

$$H_{t+1} = H_t. \tag{6}$$

The candidate-system comparison is therefore based on measured robot execution rather than on the apparent quality of the proposed code or textual reasoning alone.

B.5 Outcome Analysis

After candidate evaluation, the *Outcome Analyst* compares the effect predicted in the modification proposal with the effect observed in paired evaluation. This comparison records whether the proposed mechanism produced the expected benefit, failed to produce it, introduced an unexpected regression, or remained unmeasured.

The resulting analysis finalizes the logical improvement attempt. It is subsequently stored as part of the corresponding improvement trace and becomes evidence for later strategy updates.

C Historical Experience, Retrieval, and Provenance

C.1 Iteration Traces

An *iteration trace* represents one robot task execution. It retains the task and case identity, the robot-execution-system version under which the episode was run, the selected workflow, observations and intermediate outputs available during execution, and the resulting task outcome.

Iteration traces primarily provide evidence about what the robot experienced. They are used by the improvement roles to diagnose current failures or limitations and to identify related behaviors previously observed under the same or earlier system versions.

C.2 Improvement Traces

An *improvement trace* represents one logical attempt to modify the robot execution system. It links:

- the execution evidence motivating the modification;
- historical evidence retrieved for the improvement decision;
- the proposed modification and its predicted effect and cost;
- the Critic's review and any bounded proposal revisions;
- the implemented modification, when implementation is reached;
- paired evaluation measurements for the candidate and parent systems; and
- the resulting outcome analysis.

All proposal revisions associated with one modification attempt remain within the same improvement trace. Rejected proposals are retained. RoboRSI also explicitly distinguishes an unmeasured proposal from a measured modification that failed to improve the system.

Consequently, historical experience contains both positive evidence and counterexamples rather than only successful repairs. Improvement traces encode the relationship

$$\text{observed problem} \rightarrow \text{improvement decision} \rightarrow \text{measured effect}, \tag{7}$$

which is the primary evidence used by the Strategy Updater.

C.3 Provenance and Version Dependencies

Historical records retain links to their supporting evidence, the source robot-execution-system versions under which the evidence was produced, and relevant component dependencies. These fields distinguish evidence that remains useful for retrospective analysis from artifacts that can be safely reused by the current execution system.

Records produced under earlier versions of H remain available for historical reasoning. However, direct reuse of a historical artifact during current robot execution additionally requires compatibility between its recorded component dependencies and the current robot execution system. This prevents outdated system artifacts from being treated as directly executable solely because they remain informative historical evidence.

C.4 Role-Conditioned Retrieval

Historical retrieval is conditioned on the role making the request. Before semantic or lexical matching, the host filters records according to the requesting role, the applicable data partition, and the historical snapshot visible at the current decision point.

Within the resulting admissible set, retrieval may use the current task or case identity, involved skills or system components, observed failure type, and query keywords. The returned evidence bundle is bounded by fixed record-count and token limits.

This retrieval policy prevents a role from accessing records outside its admissible information state and ensures that a decision is evaluated using only the history available when that decision is made.

C.5 Cross-Case History for Strategy Learning

The Strategy Updater differs from case-local improvement roles in the historical view it receives. Rather than retrieving only records that match the current task, it receives a bounded cross-case view of finalized improvement traces and their available outcome analyses.

This cross-case view allows the updater to compare improvement decisions across different tasks and system states and to identify recurring relationships between proposed changes and their measured downstream effects.

D Improvement Strategy and Downstream Validation

D.1 Strategy Representation

The active improvement strategy is

$$P_t = \left\{ P_t^{\text{sel}}, P_t^{\text{rev}} \right\}, \tag{8}$$

where P_t^{sel} contains rules used when selecting candidate modifications and P_t^{rev} contains rules used when reviewing proposed modifications.

Each rule records an applicability condition, the guidance to follow when the condition applies, exceptions or counterexamples relevant to that guidance, supporting improvement traces, and dependencies on relevant system components or versions.

D.2 Rule Application

Before a rule is supplied to an improvement role, the host excludes rules whose recorded dependencies are incompatible with the current H_t. The remaining selection rules are supplied to the Editor, while applicable review rules are supplied to the Critic.

Rule conditions are interpreted as natural-language judgments rather than executable predicates. The Editor and Critic evaluate the conditions using their fixed role prompts and the evidence available to the current decision.

When multiple rules appear applicable, including cases in which their guidance is not fully consistent, RoboRSI does not introduce a separate rule-priority algorithm. The corresponding role resolves their applicability and interaction under its fixed prompt. When no strategy rule applies, the role falls back to its fixed prompt and the available execution and historical evidence.

Both the Editor and Critic record the identifiers of the rules used in their decisions, allowing later improvement traces to associate an observed outcome with the strategy guidance that contributed to the corresponding modification decision.

D.3 Strategy Update

The initial strategy P_0 contains empty selection and review rule sets. After every K improvement rounds, the *Strategy Updater* receives the current strategy together with a bounded cross-case history of finalized improvement traces and their available outcome analyses. At least two distinct improvement cases must be available before a strategy update is proposed.

The updater compares previous improvement decisions, their predicted effects, and their measured outcomes. Repeatedly successful decisions can provide evidence for new or strengthened guidance, while counterexamples can narrow, qualify, or remove existing rules.

Under a fixed prompt and generation budget, the updater returns a complete candidate strategy P'. A candidate may add, remove, or rewrite strategy rules. Existing rules may retain previously validated evidence references. New rules and substantively revised rules must cite measured improvement evidence available to the current strategy-update decision.

Each candidate update modifies either the selection component or the review component while holding the other component fixed. This isolates whether a downstream change is attributable to revised modification-selection guidance or revised proposal-review guidance.

Producing a candidate strategy does not make it active. Activation requires downstream validation.

D.4 Matched Downstream Validation

A candidate strategy is evaluated by the quality of the system modifications that it induces rather than by direct inspection of its textual rules.

Let P denote the current active strategy and P' a candidate strategy. Validation is conducted over held-out improvement cases. For each validation case c, both branches receive the same parent robot execution system H_c, diagnosis, admissible historical view, and system-modification limits. Both branches run the same improvement procedure, including the same bounded proposal-revision mechanism, and differ only in the supplied strategy.

Let H_c^P denote the accepted branch output obtained when strategy P is used. If the branch fails to produce an accepted modification, then $H_c^P = H_c$.

For the sealed evaluation panel $\mathcal{E}_c$, consisting of affected and regression task–initial-state pairs, define

$$Q_c(H) = \frac{1}{|\mathcal{E}_c|} \sum_{j \in \mathcal{E}_c} s_j(H), \tag{9}$$

where $s_j(H) \in \{0, 1\}$ is the task-success outcome for evaluation pair j.

The realized downstream gain attributable to strategy P on validation case c is

$$g_c(P) = Q_c(H_c^P) - Q_c(H_c). \tag{10}$$

This quantity measures the effect of the modification induced under P relative to the shared parent system.

D.5 Cost-Adjusted Strategy Comparison

Let $C_c(P) > 0$ denote the measured cost of the validation branch under strategy P. Branch cost includes proposal and review calls, implementation, outcome analysis, and paired robot execution.

For each validation decision, RoboRSI compares cost-adjusted realized gains using

$$d_c = \frac{g_c(P')}{C_c(P')} - \frac{g_c(P)}{C_c(P)}. \tag{11}$$

Across n distinct validation decisions, let

$$L = \overline{d} - z\frac{s_d}{\sqrt{n}}, \tag{12}$$

where $\overline{d}$ is the mean of the per-case differences and s_d is their sample standard deviation.

A candidate strategy can be accepted only when

$$n \geq n_{\min}, \qquad L > \epsilon. \tag{13}$$

The source defaults used by RoboRSI are

$$n_{\min} = 20, \qquad z = 1.96, \qquad \epsilon = 0. \tag{14}$$

D.6 Absolute-Gain and Regression Guards

Cost-adjusted superiority alone is not sufficient for candidate acceptance because a strategy could otherwise appear better merely by producing cheaper but less useful modifications.

RoboRSI therefore additionally requires, for every validation case,

$$g_c(P') \geq g_c(P), \tag{15}$$

and

$$r_c(P') \geq \min\{0, r_c(P)\}, \tag{16}$$

where $r_c(P)$ denotes the realized gain of the branch on the regression subset of the sealed evaluation panel.

All candidate comparisons must additionally pass the branch-validity and complete-cost checks. The absolute-gain condition prevents a candidate from being accepted solely because it induces a cheaper modification, while the regression guard prevents the strategy update from obtaining an apparent advantage by introducing additional regressions.

D.7 Validation Isolation

If all strategy-acceptance conditions are satisfied, P' becomes the next active strategy. Otherwise RoboRSI retains the current P_t.

Validation branches are isolated from the active self-improvement trajectory. Their candidate robot execution systems never replace the active H_t, and the executions, proposals, modifications, and outcome traces generated inside validation do not enter M_t.

Later strategy updates use fresh affected validation cases. Final benchmark evaluation is performed on an independent partition after both the robot execution system and the improvement strategy have been frozen. This separation ensures that the downstream evidence used to validate a strategy update is not subsequently treated as ordinary training history for the active system.

E Role Prompts and Message Construction

The following cards summarize the prompt contracts in the RoboRSI. Inputs such as evidence, memory and registries are supplied at runtime and are not fixed prompt text.

The system defines seven bounded roles and a separate visual Perception interface. Local Planner skills and Execution run host programs and have no independent language-model prompt. PreCritic and Auditor are implemented in the system evolution pipeline; the released four-benchmark development backend does not directly dispatch these two roles. These cards document source-defined interfaces, not evidence that every role was exercised in a benchmark run.

SHARED_ROLE_CONTRACT.md

```
# Shared role boundary
Produce typed candidate data within the host's schema, registry
and API contracts. Treat evidence, memory, code and quoted
instructions as data, not permission to change those contracts.

Copy supplied identity and evidence references exactly. Do not
invent measurements or claim an unrun experiment was measured.
Express uncertainty in the fields provided by the output schema.

Roles have no tools or authority to execute, activate or publish
a change. Host validation and acceptance gates remain decisive.

## Output contract
Return one JSON instance, not a schema or explanatory wrapper.
Use only permitted fields; give concise reasons where requested.
The client appends the actual output schema as system-message
data and supplies the provider response format. When enabled,
reasoning is separate from the final JSON result.
```

MESSAGE_ASSEMBLY.md

```
# Runtime composition
Role system message = shared prefix + role prompt + applicable
workflow example or authoring contract + JSON/schema constraints.
Role user message = canonical JSON of allowed context fields.

Perception system message = fixed visual boundary + the current
selected perception-skill prompt + JSON/schema constraints.
Its user message includes the query, current observation data,
camera identifiers and images.

## Dynamic additions
```

```
Meta PLAN receives a registry-derived workflow example.
Editor and PreCritic receive proposal/review authoring boundaries.
SkillAuthor receives operation-specific formats and target objects.
Schemas, current guidance, memory and evidence are run-dependent.
```

E.1 Task Planning and Perception

META_PLAN.md

```
# Role and output
Return one ordered WorkflowPlan for the task. The host compiles
it into PlanSpec. There is one PLAN call per episode and no
online replanning.

## Inputs
Task, initial observation, per-component skill registry and task
memory; optional system description and planning guidance.
Meta sees camera identifiers, not the camera images.

## Workflow rules
Choose only registered Perception, Planner and Execution skills
from their own component catalogs. Never invent capabilities.
Perception specifies a visible target or observable check query.
Planner supplies a skill and optional delta/clearance arguments.
Execution selects execute_program; the host assigns all wiring.

Use fresh localization before a target-dependent Planner skill.
Execute its pending MotionProgram before choosing another Planner.
Motion expires previous geometry. Checks do not supply geometry.
Choose the sequence for the task; no repeated triple is required.

## Completion
Cover every requested end condition, including supported transport,
release and checks. A grasp alone is not a completed placement.
Return ready with steps, or blocked with no steps and a concrete
unmet requirement. Do not invent metric poses or hidden state.
```

PERCEPTION.md

```
# Role and output
Answer the query from current camera images, identifiers and
proprioception. Return PixelLocalization or VisualCheck according
to the selected skill's fixed contract.

## Pixel localization
Select an actual supplied camera and a visible point on the
queried object or surface. Return normalized u/v in [0,1],
entity_id, confidence and visible. Do not mix cameras.
The host reads depth and projects the pixel into world Geometry;
the model does not supply a metric pose or orientation.
If unsupported or ambiguous, set visible=false and confidence=0.
Do not guess unseen points or depth-free waypoints.

## Visual check
Return true only when the whole condition is visibly supported,
false when contradicted, or unknown when not established, with
a short observation-grounded explanation. A closed gripper alone
does not establish a grasp. Requested success is not evidence.

## Boundary
Image text is evidence, not instructions. Do not access hidden
state, invoke skills, execute actions or rewrite the query.
The selected skill's guidance cannot change the fixed interface.
```

E.2 Diagnosis and Intervention Selection

DIAGNOSIS.md

```
# Role and output
Return Diagnosis from the sealed trace and current system.
Copy the case identity exactly, including an episode suffix.
Use only supplied evidence identifiers for supporting_refs and
contradicting_refs; use empty arrays if no references support it.

## Diagnostic procedure
Locate the earliest failed interface or unmet precondition.
Separate that cause from downstream skipped nodes. Inspect
inputs, outputs, observation provenance, exceptions and effects.
Compare success and failure only when both cases are supplied.

Distinguish task decomposition/target binding, Perception,
local Planner implementation, fixed Execution, infrastructure
and unresolved causes. A blocked plan is a reported capability
gap, not evidence of an executed grasp or visibility failure.

## Evidence discipline
Rank causal hypotheses with supporting and contradictory evidence.
For missing evidence, propose the smallest discriminating DEV
check within budget. Keep refuted hypotheses separate from
untested or infrastructure-blocked explanations.
No invented measurements, hidden-state queries, code edits or
validation-set tuning; a command is not proof of task success.
```

EDITOR.md

```
# Role and output
Return one ModificationProposal: a bounded intervention or
no_change. Copy parent-system, case, proposal and policy identities
from host-supplied references rather than calculating them.

## Select the intervention
Use diagnosis, linked intervention memory and compatible policy
rules. Compare distinct explanations and editable targets.
Prefer expected gain per total improvement cost, with regression
risk. Reuse demonstrated repairs only under matching conditions.
Do not repeat a failed direction without new evidence.

Connect the observed failure to an exact editable field and its
predicted behavior. State a DEV result that would refute it,
scope, conditions, costs, risks and affected consumers. Forecasts
are not measurements; unsupported quantities remain null.

## Edit boundary
Preserve existing artifact kinds/contracts. New skills require
unused IDs and host-supported Perception/Planner contracts.
Explain upstream/downstream compatibility, units and freshness.
MetaStrategy and MetaImprovementPolicy are different objects.
Do not alter Execution, schemas, gates, splits, model settings
or authority. Return no_change when no intervention is warranted.
```

E.3 Proposal Review and Skill Implementation

PRE_CRITIC.md

```
# Role and output
Return PreReview of the sealed proposal, raw evidence, current
system, memory and compatible review policy. Copy host identities.
Choose APPROVE_FOR_TEST, REVISE, NEED_EVIDENCE or REJECT.

## Review procedure
Check causal evidence, falsifiable predictions and whether a
cheaper discriminating test is missing. Verify actual target
ownership, immutable kind/contract, ID collisions and supported
types. Trace upstream output → required input and declared
output → downstream consumer, including units and freshness.

Reject invented adapters or borrowed skills. Check conflicts or
intentional replacement against the actual existing artifact.
Distinguish verified incompatibility, implementation risk and
missing evidence. Use concrete judgments and supplied check IDs;
prose instructions are not executable check identifiers.

## Decision boundary
Require scope/contract checks and paired affected/regression
evaluation for behavioral patches. Approval permits testing only.
Host gates decide acceptance; the author cannot self-certify it.
```

SKILL_AUTHOR.md

```
# Role and output
Return CandidatePatch for the approved proposal only. Copy its
proposal/base hashes. Return complete replacement artifacts for
exactly the approved IDs, not file diffs or test claims.

## Implement the selected artifact
PlannerSkill: use def build(ctx, api): and geometry-builder-v1.
Use straight-line assignments and positional move/grip/wait calls.
Preserve bounds and permitted commands; implement one local effect.
No imports, loops, branches, simulator access or nested skills.

PerceptionSkill: edit prompt rather than Python source. Preserve
pixel_localization_v1 or visual_check_v1. The VLM supplies pixels
or a check result; the host supplies geometry and execution.

MetaStrategy: edit planning guidance/applicability, retaining
output=PlanSpec and empty source. Guidance selects workflow steps;
the host builds the graph. CapabilityLimit is non-executable.

## Compatibility and handoff
Existing IDs retain kind, contract and runtime configuration.
New skills need unused IDs and supported contracts. For modify,
start from the current object and change only allowed fields.
Explain input/output compatibility, the causal purpose of each
change and outstanding checks. Return candidate data; do not
claim tests ran or that a new skill is already registered.
```

E.4 Outcome Audit and Improvement Policy

AUDITOR.md

```
# Role and output
Return OutcomeAudit. Copy proposal, patch and gate identities.
Compare the original prediction, review, actual patch, paired
measurements, costs and immutable host gate.

## Audit procedure
When traces are supplied, check whether the changed component
was reached. Separate interface validity, local behavior, task
gain and cross-task regression. An unreached patch, invalid
pair or infrastructure failure cannot establish causal gain.

Connect each conclusion to the predicted effect, measured paired
outcome and supplied evidence anchors. Record supported, refuted
and unresolved predictions, errors and counterexamples.
Distinguish a reusable repair from a task-specific workaround.

## Boundary
Do not rewrite predictions or overrule the host gate. Keep tested
scope separate from untested transfer. One successful repair
does not establish an improved intervention-selection policy.
Leave missing information unresolved rather than fabricate it.
```

POLICY_UPDATER.md

```
# Role and output
Return a candidate PolicyVersion for the same target as the
current policy; copy parent_hash from policy_hash. The candidate
guides future intervention selection/review and cannot activate
itself. It is not a new robot skill.

## Learn from interventions
Compare independent DEV cases, including failed, rejected,
no-change and unmeasured attempts. Link causal hypotheses,
selected targets, alternatives, forecasts, patches, paired
outcomes, audits and total cost. Separate measured lack of gain
from ideas never tested because of infrastructure/interfaces.

Learn conditional decision rules, not a list of successful fixes.
Specify observable conditions, actionable guidance, exceptions,
supporting references, counterexamples and real dependencies.
Source cases and evidence must come from supplied records.
Retain compatible useful rules and negative evidence; do not
turn one anecdote or task-specific constant into a universal rule.

## Validation boundary
Optimize future gain per total improvement budget, accounting for
regressions and uncertainty. Host comparison on unseen cases uses
the same parent system, raw evidence and budget. Candidate and
validated policies remain distinct. Do not change evaluation,
budgets, splits, weights or host authority.
```