SELF-IMPROVING ROBOT SKILLS FOR LONG-HORIZON TASKS VIA RECURSIVE STATE-AWARE CREDIT ASSIGNMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon robot programs can fail when early operations leave states that compromise later execution. We present Recursive State-Aware Credit (RSAC) for improving robot skills through program revision, with the language model and action primitives fixed. Continuations of unchanged program suffixes from saved stage exits estimate how intermediate states support complete-task success. State-aware credit (SA) separates local passage deficits from poor exit-state quality. RSAC recursively compares gains in terminal-outcome dependence across stage boundaries to select an editing region for the current program. Execution experience guides localized revisions, and paired full-task validation determines adoption for the next iteration. Across eight simulated quadrupedal robot tasks, with ten independent optimization runs per task and configuration, RSAC achieves 44.4% task-averaged success, compared with 30.3% for ASPIRE and 35.4% for one-step localization. Controlled interventions and online revisions connect the diagnostic evidence to concrete program changes and improved task performance.

1 INTRODUCTION

Long-horizon robot tasks require perception, manipulation, and navigation to interact through a continuously evolving physical state. Executable programs provide a flexible interface for composing robot primitives and revising their parameters and control logic in response to execution feedback. Prior work has established language-to-program robot control and grounded language-level decisions in robot capabilities and observations (Liang et al., 2023; Ichter et al., 2023; Huang et al., 2023). This raises a distinct question: how can an executable robot program improve from repeated execution experience while the underlying language model and action primitives remain fixed?

A central difficulty is that long-horizon task outcomes depend not only on whether each stage succeeds, but also on the states passed between stages. A grasp may satisfy its local condition yet leave an object orientation that complicates transport and makes placement difficult several operations later. In multi-object tasks, later actions may also disturb objects whose goals were previously satisfied. Local pass rates therefore capture milestone completion but not whether successful stage exits are useful for the remaining program. Skill-chaining work has studied compatibility between terminal and initial state distributions (Lee et al., 2022). For program self-improvement, this creates a localization problem: a failure may become visible downstream even when the state that makes it likely was produced several stages earlier.

A useful localization must lead to a concrete program revision. Successful and failed trajectories provide examples of behavior to preserve and conditions to repair, while prior edits supply feedback on directions already explored. Experience-driven agents and artifact optimizers demonstrate the value of these records (Wang et al., 2024; Lu et al., 2026; Yuksekgonul et al., 2025; Agrawal et al., 2026; Yang et al., 2026). Our framework addresses this localization problem using state evidence, with version-associated experience providing context for edits in the selected region.

We develop a self-improvement framework organized around three decisions: where to edit, how to revise the program, and whether the revision should be retained. Repeated fixed-suffix continuations from saved stage-exit states estimate how well those states support completion of the remaining task. State-aware credit identifies stages with either local pass deficits or poor exit-state quality. Recursive

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.

State-Aware Credit (RSAC) then recursively compares gains in terminal-outcome dependence across stage boundaries to select an editing region according to the current program's state evidence. The selected region constrains localized parameter, operation, and logic changes. Retrieved trajectories and revision history provide editing context, and full-task validation determines whether a candidate revision is retained.

An accepted revision changes the states encountered in later executions, providing new evidence for the next round of diagnosis and editing.

We evaluate the framework in a MuJoCo-based quadrupedal robot environment with fixed primitives and high-level programs. The task suite spans observation, manipulation, transport, obstacle traversal, and sequential delivery, creating settings in which intermediate states and previously achieved conditions affect later execution. Our contributions are:

- A state-aware credit and RSAC localization mechanism that distinguishes local stage failure from poor exit-state quality and recursively compares boundary gains to adapt the editing region to the current program.
- An iterative robot-program self-improvement framework in which execution, state diagnosis, localized revision, and full-task validation repeatedly generate the evidence used for subsequent updates.
- An evaluation across eight simulated quadrupedal robot tasks, with ten independent optimization runs per task and configuration, component ablations, and controlled and online revisions linking state evidence to task performance.

2 Related Work

Executable robot policies and interactive feedback. Code as Policies generates robot policy code that composes perception and control APIs (Liang et al., 2023). SayCan combines language-model task knowledge with skill affordances (Ichter et al., 2023), while Inner Monologue incorporates environment feedback into language-based planning (Huang et al., 2023). More recently, CaP-X provides a framework for robotic coding agents; its training-free CaP-Agent0 combines multi-turn feedback, visual differencing, synthesized skills, and parallel reasoning (Fu et al., 2026). These works show how executable code can be generated or revised using grounded feedback. Our focus is complementary: we study how repeated executions of a retained program can be converted into state-level evidence for diagnosing and improving that program over multiple iterations.

Experience accumulation and reusable skills. Voyager accumulates an executable skill library through iterative code generation and environmental feedback in Minecraft (Wang et al., 2024). In robotics, ASPIRE combines fine-grained execution traces, program repair and validation, reusable skill accumulation, and evolutionary exploration (Lu et al., 2026). Reflexion uses verbal reflections on task feedback to inform subsequent attempts (Shinn et al., 2023). Our framework retains version-associated trajectories and revision feedback to support editing. Its localization signal comes from current-program state continuations and their relation to terminal task outcomes.

Feedback-based optimization of external artifacts. Self-Refine iterates between feedback and revision using a language model (Madaan et al., 2023). TextGrad propagates textual feedback through compound systems to optimize textual variables (Yuksekgonul et al., 2025), and GEPA uses execution traces and reflection to evolve prompts (Agrawal et al., 2026). SkillOpt optimizes a natural-language skill document for a frozen agent through bounded edits, held-out selection, rejected-edit memory, and slower-timescale updates (Yang et al., 2026). Our optimization object is instead an executable robot program whose revisions alter the physical states encountered during future execution. Our localization problem concerns physical state transitions: the stage at which a failure becomes visible need not be the stage whose code should be revised. We combine cross-stage localization with experience-conditioned editing and full-task validation.

Skill interfaces and continuation-based credit. T-STAR regularizes terminal-state distributions to improve compatibility between separately learned skills in long-horizon manipulation (Lee et al., 2022). Its focus is on learning compatible skill interfaces. Credit Assignment with Resets uses

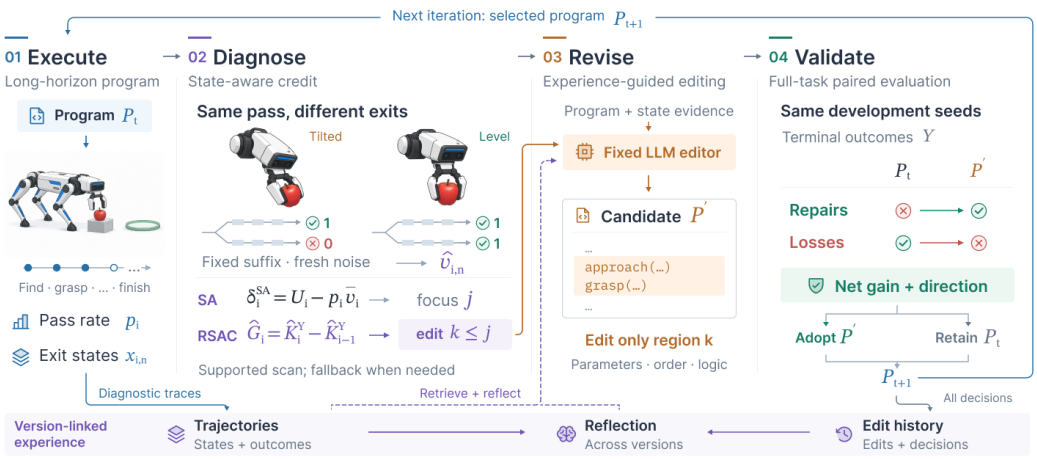


Figure 1: **Robot-program self-improvement.** Execute the current program, estimate downstream value from saved stage exits, use SA and Recursive State-Aware Credit to select an editing region, and validate a localized revision on the complete task. Version-associated trajectories and editing feedback support candidate generation. The accepted program supplies execution evidence for the next iteration.

intermediate reasoning states and resampled continuations to assign credit during language-model policy optimization (Samanta et al., 2026). Our setting differs in both the optimization object and the role of continuation. We restore simulator snapshots at successful stage exits and repeatedly execute an unchanged program suffix to estimate how those states support final task completion. State-aware credit and RSAC use this evidence to localize program revisions, while version-associated memory and full-task validation connect those revisions to an iterative self-improvement process.

3 Method

3.1 Problem and Improvement Loop

Given an initial robot program P_0, a task distribution $\mathcal{D}$, and a fixed library of action primitives, we seek a program with higher complete-task success under a bounded editing and interaction budget. A task instance and its execution randomness are denoted by $\xi \sim \mathcal{D}$, and $Y(P, \xi) \in \{0, 1\}$ records terminal success. The objective is

$$J(P) = \mathbb{E}_{\xi \sim \mathcal{D}}[Y(P, \xi)]. \tag{1}$$

Edits change primitive parameters, operation sequences, or local program logic; the language model and low-level controller remain fixed.

At optimization iteration t, the framework executes the current program P_t, diagnoses its behavior, proposes localized revisions, and validates them on the complete task (Figure 1). SA identifies an initial phase of concern, and RSAC compares boundary gains in terminal-outcome dependence to select the editing region. Execution experience supplies examples and revision feedback to the editor. An adopted revision becomes P_{t+1} and produces the next round of evidence. The index t counts program-improvement iterations, rather than robot control steps. Below we describe diagnosis within one iteration and omit t from the stage statistics.

3.2 State-Aware Diagnosis

Local progress and downstream state value. A task contains H ordered semantic stages. Stage i specifies a condition that may span several primitive calls. We record whether an execution reaches the stage, whether it passes, and the state when its condition first becomes satisfied. Its pass rate is

$$p_i = \frac{N_i^{\text{pass}}}{N_i^{\text{reach}}}, \qquad N_i^{\text{reach}} > 0. \tag{2}$$

Passing a stage does not guarantee that its exit state supports later operations. For example, a successful grasp can leave an orientation that makes placement difficult. Later actions can also invalidate previously achieved conditions, so final task success Y remains the evaluation target.

We save simulator snapshots at successful stage exits. From each retained exit $x_{i,n}$, we repeatedly execute the unchanged remaining program with fresh execution noise. These *fixed-suffix continuations* hold the state and code fixed while measuring the reliability of subsequent execution. If C_i exits are retained and exit n has $M_{i,n}$ valid continuations, their terminal outcomes give

$$\widehat{v}_{i,n} = \frac{1}{M_{i,n}} \sum_{m=1}^{M_{i,n}} Y_{i,n,m}, \qquad \bar{v}_i = \frac{1}{C_i} \sum_{n=1}^{C_i} \widehat{v}_{i,n}. \tag{3}$$

Thus $\widehat{v}_{i,n}$ estimates completion probability from one exit, and $\bar{v}_i$ measures the average downstream quality of the sampled successful exits. An execution that fails within the stage lowers p_i; a continuation that fails later contributes $Y_{i,n,m} = 0$.

A high-value exit reference. We compare average exit quality with a reference U_i formed from the better observed exits. The reference fraction is $\rho = 0.20$, selecting

$$L_i = \max\{1, \mathrm{round}(\rho C_i)\} \tag{4}$$

exits. With the diagnostic cap $C_i = 8$, this selects two exits. The fraction controls how strongly the reference concentrates on high-value states; U_i is an empirical reference for the current program. To reduce selection bias, we split each exit's branches into two groups: one ranks exits and the other evaluates the selected L_i, then the roles are exchanged and the two estimates averaged. The resulting reference is clipped between $\bar{v}_i$ and the direct top-exit mean. The reference therefore uses separate branch outcomes for selection and evaluation.

Two sources of task-performance deficit. The SA score combines local failures with the quality of states passed downstream:

$$\begin{aligned} \delta_i^{\mathrm{SA}} &= U_i - p_i \bar{v}_i \\ &= \underbrace{(1 - p_i) U_i}_{\text{local pass deficit}} + \underbrace{p_i (U_i - \bar{v}_i)}_{\text{exit-quality deficit}} . \end{aligned} \tag{5}$$

The *local pass deficit* weights the fraction of executions that fail within the stage by the downstream reference value. It is large when a potentially useful stage is difficult to pass, such as an unreliable grasp. The *exit-quality deficit* measures the gap between average and high-value successful exits, weighted by how often the stage passes. It can be large even when local passage is reliable, such as when most grasps succeed but leave poor placement states. Together the terms describe a value gap conditional on reaching the stage.

SA selects the eligible stage with the largest δ_i^{SA} as the initial focus. This identifies where the current program exhibits a large deficit. RSAC then examines preceding state transitions to determine where a revision should be made.

3.3 Recursive State-Aware Credit

Distinguishing state formation from transmission. A downstream deficit can reflect a state produced several phases earlier. In a Grasp–Carry–Place chain, Carry may preserve a grasp configuration whose consequences become visible only at Place. RSAC repeats an entry–exit comparison along the predecessor chain to identify where terminal-outcome dependence increases.

For a fixed program, let $\mathcal{P}$ be the common trajectory population used for the boundary comparison, X_i its state at boundary i, and $q_i^Y(x) = \Pr_{\mathcal{P}}(Y = 1 \mid X_i = x; P_t)$. Define

$$K_i^Y = \frac{\mathrm{Var}_{\mathcal{P}}[q_i^Y(X_i)]}{\mathrm{Var}_{\mathcal{P}}(Y)}. \tag{6}$$

All compared boundaries use the same original diagnostic trajectories and terminal task outcome. Trajectories that have already failed before a boundary remain in its statistics as absorbing failure states

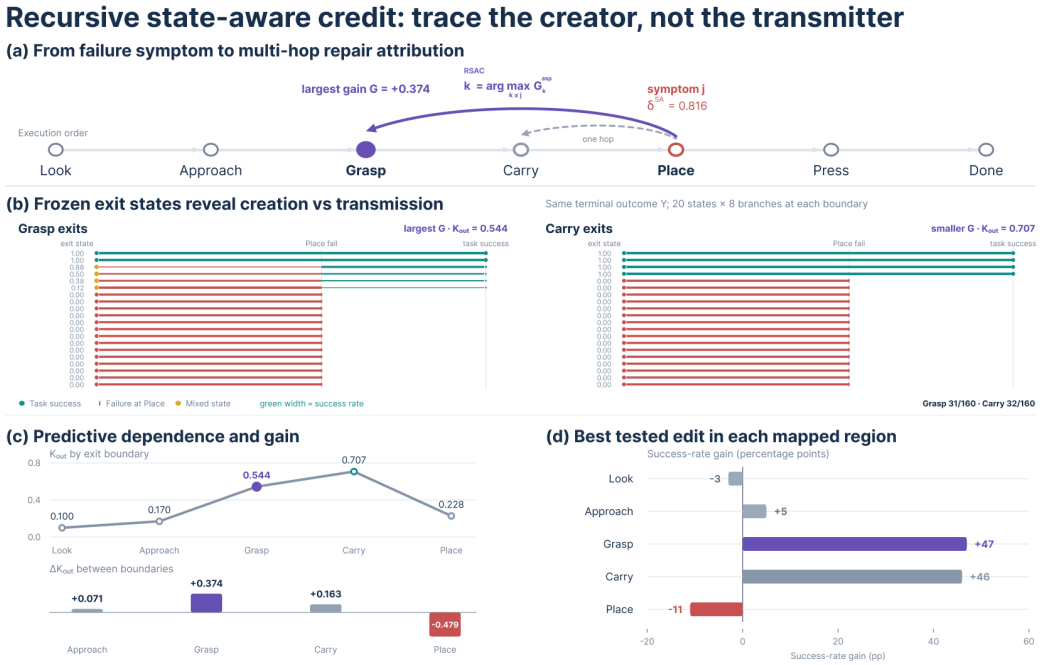


Figure 2: **Recursive state-aware credit and tested upstream revisions.** On a fixed `apple_chain_z10` program, SA focuses on Place, and RSAC selects Grasp by its maximum adjacent gain, while Carry has higher exit dependence. The diagnostic uses 20 successful exits per boundary and eight continuations per exit; dependence estimates use the common trajectory population, including the weight of earlier failures. In a separate edit sweep, the best tested Grasp- and Carry-region edits improve success from 14% to 61% and 60%, respectively.

with zero continuation success probability. Sampling and weighting are specified in Appendix A.2. K_i^Y measures how much of the outcome variation is explained by differences between boundary states. A large K_i^Y means that the state strongly distinguishes later success from failure; it does not mean that the stage itself has a high success rate. For target selection, we use a Jeffreys-smoothed plug-in estimate $\widehat{K}_i^Y$ from continuation outcomes. Smoothing stabilizes extreme empirical success rates; the estimate still includes uncertainty from finite branch sampling.

The preceding stage's exit provides the next stage's entry. RSAC compares terminal dependence across this shared boundary:

$$\widehat{K}_{\mathrm{in},i}^Y := \widehat{K}_{i-1}^Y, \qquad \widehat{G}_i := \widehat{K}_i^Y - \widehat{K}_{i-1}^Y, \qquad i = 2, \ldots, H. \tag{7}$$

A large positive gain highlights a stage across which states become more discriminative of the terminal outcome. In contrast, high dependence at both entry and exit with a small gain suggests transmission of already informative state differences. This distinction supplies boundary evidence for selecting the editing stage. The complete selection rule can retain the initial focus, select its immediate predecessor, or select an earlier upstream stage, according to the diagnostic evidence for the current program. Under the population and replay assumptions in Appendix A.6, the population gain satisfies $\mu G_i = R_{i-1} - R_i$, where $\mu = \Pr_{\mathcal{P}}(Y = 1)$ and R_i is the recovery probability obtained by replaying from boundary i, conditional on the original execution having failed. Thus, μG_i quantifies the marginal recovery from re-executing phase i rather than retaining its original exit (Theorem 1). This connection provides an operational basis for using adjacent gains to prioritize program regions for localized revision.

Selecting the editing region. If the initial focus is the first stage, RSAC retains it as the editing target. Otherwise, RSAC checks its entry boundary. A boundary is supported only when both stages have finite dependence estimates and sufficient exit and branch samples; an unsupported initial boundary invokes the fallback below. At a supported initial boundary, entry dependence at or below

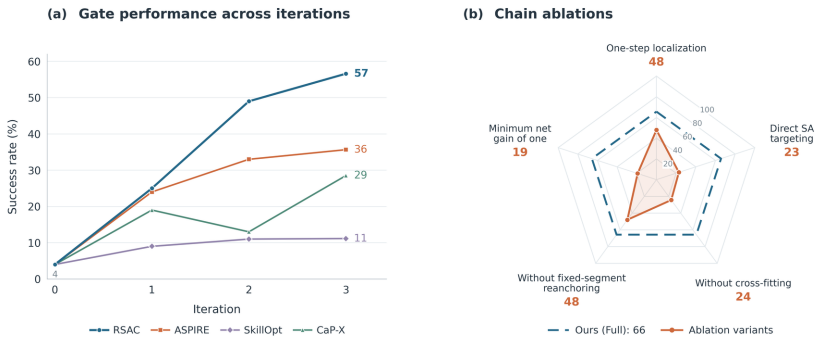


Figure 3: **Program improvement and ablations.** (a) Gate SR averaged at each iteration over the same ten independent optimization runs per method, with matched rollout counts; iteration 0 denotes initialization. (b) Chain ablation means over ten runs per configuration. CaP-X denotes CaP-Agent0. SDs are reported in Tables 1 and 2.

α_{in} retains the focus. Higher entry dependence initiates a backward scan from the focus, including its own boundary, to evaluate adjacent gains. The scan stops at the first unsupported boundary.

Let $\mathcal{C}$ denote the resulting contiguous set of supported boundaries. The maximum-gain candidate is

$$k^\star = \arg\max_{i \in \mathcal{C}} \widehat{G}_i. \tag{8}$$

Ties select the downstream stage. The procedure considers the whole supported segment rather than stopping at the first above-threshold gain. It returns $k = k^\star$ when the largest gain meets the configured acceptance threshold $g_{\min}$. If the initial boundary is unsupported, the candidate set is empty, or the gain is below threshold, it uses a one-step fallback: move to the immediate predecessor when that stage is available and the SA entry-dependence score is finite and exceeds α_{redir}; otherwise retain the initial focus. The fallback score is the entry-dependence signal supplied by SA, distinct from the terminal statistic used for the gain scan.

The final stage k determines the code region made available to the editor. Figure 2 connects this boundary evidence to a finite set of tested revisions. Sampling, smoothing, and the full decision specification are collected in Appendix A.

3.4 Experience-Guided Revision and Validation

The editor receives the current program, the selected region k, stage statistics, and examples of high- and low-value states. Retrieved successful and failed trajectories retain their producing program versions, allowing the editor to compare behaviors and reuse earlier evidence. Previous edit outcomes and available cross-version reflections provide additional guidance. The fixed language model uses these inputs to propose changes to primitive parameters, inserted or reordered operations, and local logic within the selected region.

Candidates are evaluated against the current program using the same development seeds. Paired outcomes count both newly successful executions and losses of previous successes; the adoption rule uses their net gain and direction. If a candidate passes validation, it becomes P_{t+1}; otherwise P_t is retained. The decision and its execution feedback enter the revision history, while subsequent diagnostic executions supply evidence for the next iteration. Diagnostic trajectories support editing, development trials support adoption, and held-out executions evaluate final performance. Retrieval, prompting, and reflection details are provided in Appendix B.

4 Experiments

We evaluate complete-task performance, examine variants of the diagnostic and revision procedure, and connect state evidence to concrete program changes. Controlled state-handoff edits complement logged online revisions by showing how changes within a stage affect later task outcomes.

Table 1: Complete-task SR (%; mean ± sample SD across independent runs). Each final program is evaluated on 200 unseen seeds. Each task–method pair has ten runs. Bold indicates the largest mean. For the macro row, we first average the eight tasks at each run index, then report the mean and sample SD of the ten resulting values.

Task	CaP-Agent0	ASPIRE	SkillOpt	Ours (RSAC)
`apple`	28.6 ±24.5	41.5 ±29.8	19.3 ±23.8	**65.7 ±19.4**
`haul`	17.0 ±8.1	25.1 ±9.7	21.8 ±6.0	**39.1 ±11.6**
`zone`	26.0 ±10.4	37.9 ±17.5	16.6 ±8.8	**41.9 ±12.4**
`wall`	41.0 ±7.4	26.0 ±4.2	21.2 ±10.0	**45.9 ±13.4**
`deliver`	22.4 ±10.8	22.7 ±7.5	19.4 ±6.9	**26.7 ±7.2**
`pick`	13.0 ±12.4	25.0 ±13.9	14.5 ±10.8	**38.8 ±7.0**
`gate`	28.5 ±23.1	35.7 ±28.9	11.2 ±9.5	**56.6 ±27.6**
`survey3`	25.5 ±8.9	28.3 ±21.8	16.6 ±9.4	**40.7 ±7.1**
Macro average	25.2 ±8.6	30.3 ±7.9	17.6 ±7.3	**44.4 ±3.5**

4.1 Experimental Setup

Environment and tasks. Experiments use a MuJoCo-based quadrupedal robot environment (Todorov et al., 2012). Fixed primitives support observation, approach, grasping, transport, placement, and return. The language model updates high-level programs while the motion controller and language model weights remain fixed. We consider eight task families: `apple`, `haul`, `zone`, `wall`, `deliver`, `pick`, `gate`, and `survey3`. Task conditions define semantic stages, enabling diagnosis of the states exchanged between successive program regions. Appendix H describes all eight task families and illustrates their action sequences.

Evaluation and aggregation. The primary metric is complete-task success rate (SR). The main comparison summarizes independent experimental runs, with each final program evaluated on 200 unseen seeds. We report the mean and sample standard deviation across ten runs per task and method. These standard deviations describe variation across independent runs. To obtain the macro summary, we first average the eight tasks at each run index and then compute the mean and sample standard deviation across the ten task-averaged values. Each component-ablation variant is evaluated in a fresh batch of ten independent optimization runs on each of the eight tasks. Full-RSAC references use the corresponding ten-run main-comparison results.

Comparison methods. **CaP-Agent0**, the training-free agent in CaP-X, combines multi-turn code revision, visual feedback, synthesized skills, and parallel reasoning (Fu et al., 2026). This is the fixed-model CaP-X configuration; CaP-RL is a separately trained variant. **ASPIRE** combines trace-guided robot-program diagnosis, repair and validation, reusable skills, and evolutionary exploration (Lu et al., 2026). **SkillOpt** optimizes a natural-language skill document through bounded edits and validation, retaining rejected edits and slower cross-epoch guidance (Yang et al., 2026). Its adaptation here applies the editing procedure to executable robot programs. Model settings, baseline adaptations, and the comparison protocol are summarized in Appendix D.

4.2 Comparison with Existing Methods

Table 1 summarizes the external methods and RSAC across the eight task families.

RSAC has the highest mean SR on all eight tasks, with a macro SR of $44.4 \pm 3.5\%$, exceeding ASPIRE by 14.2 percentage points before rounding. Task means range from 26.7% on `deliver` to 65.7% on `apple`. Run-to-run variation is largest on `gate` and `apple`, with sample SDs of 27.6 and 19.4 points, respectively.

Progress under matched rollout counts. Figure 3(a) averages Gate SR at corresponding iterations across the same ten independent runs per method. From initialization, each run undergoes three program-improvement rounds with equal rollout counts across methods at each round. RSAC improves throughout and achieves the highest final mean SR.

Table 2: Component ablations across eight tasks. Task rows report complete-task SR (%; mean ± sample SD) over ten independent optimization runs per task and configuration. Full RSAC reuses the main-comparison results; each variant uses a fresh ten-run batch. The final row is the unweighted mean of the eight task means. Bold indicates the largest mean in each row.

Task	Full RSAC	One-step localization	Direct SA targeting	Without cross-fitting	Without reanchoring	Net gain of one
Chain	**65.7** ±19.4	48.0 ±16.6	22.8 ±14.1	24.4 ±12.3	48.0 ±15.0	19.2 ±2.4
Haul	**39.1** ±11.6	30.7 ±15.1	32.9 ±14.2	34.8 ±13.2	36.1 ±17.2	27.2 ±13.5
Zone	**41.9** ±12.4	34.7 ±9.8	36.7 ±12.7	32.6 ±11.8	30.0 ±12.4	32.1 ±9.2
Wall	**45.9** ±13.4	29.6 ±13.4	38.0 ±12.5	35.3 ±8.1	40.3 ±18.8	40.9 ±9.1
Deliver	**26.7** ±7.2	21.9 ±5.9	23.5 ±4.0	20.9 ±6.4	17.5 ±6.3	21.7 ±6.1
Pick	**38.8** ±7.0	32.3 ±7.3	33.5 ±4.7	34.1 ±7.9	33.8 ±8.6	34.7 ±6.9
Gate	**56.6** ±27.6	48.4 ±18.8	49.3 ±24.3	49.2 ±20.0	44.8 ±29.1	36.4 ±23.6
Survey	**40.7** ±7.1	37.7 ±14.1	37.0 ±11.4	33.2 ±4.3	28.1 ±11.2	29.9 ±11.9
Task mean	**44.4**	35.4	34.2	33.0	34.8	30.2

4.3 Component Ablations

We compare five variants across all eight tasks to assess localization, reference estimation, code-region tracking, and adoption (Table 2). Chain, Zone, and Survey denote `apple`, `zone`, and `survey3`. **One-step localization** replaces the backward gain scan with a one-step rule based on entry dependence. **Direct SA targeting** edits the stage with the largest SA score without upstream localization. **Without cross-fitting** uses the naive top-exit mean for U_i. **Without fixed-segment reanchoring** disables code-region reanchoring across revisions. **Minimum net gain of one** lowers the required paired net gain from three successful executions to one.

Across all eight tasks, Full RSAC has the highest mean SR among the six configurations (Table 2). Its task-averaged SR is 44.4%, compared with 35.4% for one-step localization and 34.2% for direct SA targeting. Full RSAC improves over one-step localization on every task, with gains from 3.0 to 17.7 percentage points and a task-averaged gain of 9.0 points.

The appropriate editing stage depends on the task structure and the current program. RSAC and one-step localization can select the same phase in a given optimization iteration. When boundary evidence points elsewhere, RSAC can retain the initial focus or select a different upstream phase. The ablation therefore compares the two phase-selection rules within the complete optimization process. These results support the effectiveness of adapting the editing location to boundary evidence. Figure 3(b) shows the corresponding Chain means.

Removing cross-fitting or fixed-segment reanchoring reduces task-averaged SR to 33.0% and 34.8%, respectively, while relaxing the adoption requirement to a net gain of one yields 30.2%. Each variant has a lower mean than Full RSAC on all eight tasks. These comparisons support the contributions of reference-value estimation, code-region tracking, and the adoption criterion within the complete optimization loop.

4.4 Parameter Sensitivity

We evaluate parameter sensitivity on Pick (`pick`) and Zone (`zone`) using Full RSAC and eight configurations that each change one parameter from the defaults in Table 5. Table 3 lists the individual changes and their results.

All configurations use the same initialization protocol and total rollout budget, with a limit of 12 program-improvement iterations, up to five candidate plans per iteration, a 40-seed development panel, and $N_{\text{diag}} = 16$ diagnostic rollouts per iteration. Cross-fitting, Jeffreys smoothing, fixed-segment reanchoring, and the history channel remain enabled, and candidate adoption requires a paired net gain of at least three successful executions. Each task–configuration pair is summarized over ten independent optimization runs. The Full reference uses the corresponding ten-run results from Table 1.

Table 3: Parameter sensitivity on Pick and Zone. Complete-task SR is reported in percent as mean ± sample SD over ten independent optimization runs per task and configuration. Each variant changes only the listed parameter from Full. Bold indicates the largest mean in each task.

Configuration	Parameter change	Pick	Zone
FULL	Default	**38.8** ±7.0	41.9 ±12.4
G_LOW	$g_{\min} : 0.20 \to 0.10$	32.0 ±15.4	39.8 ±9.0
G_HIGH	$g_{\min} : 0.20 \to 0.40$	37.7 ±17.9	**42.6** ±16.2
MORE_BRANCHES	$M : 8 \to 16$	29.2 ±3.7	39.3 ±11.1
MORE_EXITS	$C_{\max} : 8 \to 16$	36.5 ±12.4	39.1 ±10.0
ENTRY_LOW	$\alpha_{\text{in}} : 0.50 \to 0.30$	31.9 ±14.2	39.9 ±10.7
ENTRY_HIGH	$\alpha_{\text{in}} : 0.50 \to 0.70$	30.1 ±15.0	41.2 ±12.4
REF_NARROW	$\rho : 0.20 \to 0.125$	29.4 ±11.2	41.3 ±10.1
REF_WIDE	$\rho : 0.20 \to 0.50$	30.1 ±13.3	39.5 ±8.8

Full achieves the highest mean on Pick (38.8%) and is close to the highest mean on Zone, where increasing $g_{\min}$ to 0.40 gives 42.6%, compared with 41.9% for Full. Sensitivity is task dependent: the tested configurations span mean SRs of 29.2–38.8% on Pick and 39.1–42.6% on Zone. These results support the default setting as a competitive common configuration across the two tasks. Under the fixed total rollout budget, increasing the number of suffix branches allocates more rollouts to diagnosis and leaves fewer for candidate evaluation and validation. The MORE_BRANCHES results therefore reflect the tradeoff between diagnostic sampling and the budget available for subsequent program improvement.

4.5 State Handoffs and Revision Mechanisms

State quality beyond local passage. In a controlled experiment on a fixed apple_chain_z15 program, we insert a pitch reset after a successful grasp and before the transport and placement sequence. Across 100 paired evaluation seeds, complete-task success rises from $7/100$ to $28/100$, with 22 repaired failures and one lost success. Both programs pass Carry in 31 executions. Thus, the same aggregate local pass count can accompany substantially different downstream success, illustrating the information captured by exit-state quality. On a second fixed program, place_apple_zone, a pitch reset increases success from $57/100$ to $73/100$, whereas placing the same type of operation at another tested location yields $46/100$. The effect depends on where the state handoff is changed.

From diagnosis to an adopted edit. Logged online revisions connect the diagnostic region to an actual code change and a subsequent adoption decision. In a Chain iteration, both the initial SA focus and the final editing region are Carry. Adding a short backward walk raises the recorded development SR from 65.0% to 72.5%; the final paired adoption check records nine repaired failures and six lost successes. In a Pick iteration, the initial focus is Place and the one-step fallback selects Fetch. The adopted edit reduces the approach stopping distance from 0.80 to 0.50, with development SR increasing from 37.5% to 87.5% and a final paired check of five repairs and one loss. These are distinct online episodes with their own development evaluations; they are separate from the fixed-program interventions above. Together, they show how state diagnosis guides localized changes that are assessed through complete-task execution. Detailed intervention results and editing records appear in Appendix C.

5 Conclusion

We presented Recursive State-Aware Credit for improving long-horizon robot skills through program revision. SA combines local passage statistics with continuation-based exit values; RSAC recursively compares boundary gains to select editing regions. Execution experience guides localized revisions, and paired full-task validation determines adoption. Comparisons across eight tasks and component ablations demonstrate improved complete-task performance, while controlled interventions and online revisions connect the diagnostic evidence to concrete program changes.

AI USE STATEMENT

Generative AI tools assisted with literature retrieval, manuscript organization, Chinese-to-English translation, drafting and editing, reference formatting, and discussion of experimental methodology. Language models also serve as components of the proposed program-diagnosis and editing framework, as described in Section 3.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/0e9e708b6f48e14fd0ac29e167413f76-Paper-Conference.pdf.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026. URL https://openreview.net/pdf/935eb8c21784f82f7110b7ecea0557c7a24be23c.pdf.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 1769–1782. PMLR, 2023. URL https://proceedings.mlr.press/v205/huang23c.html.

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

Youngwoon Lee, Joseph J. Lim, Anima Anandkumar, and Yuke Zhu. Adversarial skill chaining for long-horizon robot manipulation via terminal state regularization. In *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pp. 406–416. PMLR, 2022. URL https://proceedings.mlr.press/v164/lee22a.html.

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 *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://doi.org/10.1109/ICRA48891.2023.10160591.

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. doi: 10.48550/arXiv.2607.00272. URL https://arxiv.org/abs/2607.00272.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large

language models. In *International Conference on Learning Representations*, volume 2024, pp. 26516–26560, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/70c26937fbf3d4600b69a129031b66ec-Paper-Conference.pdf.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Utkarsh Aashu Mishra, Shangjie Xue, Yongxin Chen, and Danfei Xu. Generative skill chaining: Long-horizon skill planning with diffusion models. In *Proceedings of the 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 2905–2925. PMLR, 2023. URL https://proceedings.mlr.press/v229/mishra23a.html.

Ankur Samanta, Akshayaa Magesh, Ayush Jain, Youliang Yu, Daniel Jiang, Kavosh Asadi, Kaveh Hassani, Paul Sajda, Jalaj Bhandari, and Yonathan Efroni. Credit assignment with resets in language model reasoning. *arXiv preprint arXiv:2605.25507*, 2026. doi: 10.48550/arXiv.2605.25507. URL https://arxiv.org/abs/2605.25507.

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*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Program generation for situated robot task planning using large language models. *Autonomous Robots*, 47(8):999–1012, 2023. doi: 10.1007/s10514-023-10135-3. URL https://link.springer.com/article/10.1007/s10514-023-10135-3.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109. URL https://doi.org/10.1109/IROS.2012.6386109.

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. URL https://openreview.net/forum?id=ehfRiF0R3a.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026. doi: 10.48550/arXiv.2605.23904. URL https://arxiv.org/abs/2605.23904.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4. URL https://www.nature.com/articles/s41586-025-08661-4.

Lihan Zha, Yuchen Cui, Li-Heng Lin, Minae Kwon, Montserrat Gonzalez Arenas, Andy Zeng, Fei Xia, and Dorsa Sadigh. Distilling and retrieving generalizable knowledge for robot manipulation via language corrections. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 15172–15179. IEEE, 2024. doi: 10.1109/ICRA57147.2024.10610455. URL https://doi.org/10.1109/ICRA57147.2024.10610455.

A Diagnostic Estimation and RSAC Target Selection

This appendix specifies the diagnostic statistics and target-selection procedure used by RSAC. The main method connects these diagnostics to experience-conditioned program editing and full-task validation; the details below define how the editing target is obtained from a fixed current program.

A.1 Stage Boundaries and Prediction Targets

We use the stage indices $i = 1, \ldots, H$ from the main text. For the detailed decision procedure, let j denote the eligible stage with the largest SA score δ_i^{SA}. Given this initial focus and the available stage statistics, RSAC returns the editing stage k. These stage indices refer to a single optimization iteration.

Two prediction targets occur in the implementation and must be distinguished. For final task success $Y \in \{0, 1\}$, let $\mathcal{P}$ be the common trajectory population used by the compared boundaries. The population diagnostic is

$$\begin{aligned} q_i^Y(x) &= \Pr_{\mathcal{P}}(Y = 1 \mid X_i = x; P_t), \\ K_i^Y &= \frac{\mathrm{Var}_{\mathcal{P}}[q_i^Y(X_i)]}{\mathrm{Var}_{\mathcal{P}}[q_i^Y(X_i)] + \mathbb{E}_{\mathcal{P}}[q_i^Y(X_i)(1 - q_i^Y(X_i))]}. \end{aligned} \tag{9}$$

Here P_t and its suffix are fixed during diagnosis, and X_i denotes the boundary-i state under $\mathcal{P}$. All expectations and variances in the comparison refer to this same population. For nondegenerate outcomes, the denominator equals $\mathrm{Var}_{\mathcal{P}}(Y)$ by the Bernoulli variance decomposition. The ratio measures heterogeneity in final success probabilities across boundary states, rather than the success rate itself.

A separate statistic, K_i^{next}, replaces Y with the indicator that phase $i + 1$ passes. This next-phase statistic supplies the passability entry gate; when next-phase outcomes are unavailable, the implementation uses the final-outcome statistic as its fallback. Consequently, the passability gate and the RSAC selector need not reach the same conclusion about a boundary. RSAC uses K^Y throughout its entry test and gain calculation:

$$K_{\mathrm{in},i}^Y := K_{i-1}^Y, \qquad G_i := K_i^Y - K_{i-1}^Y, \qquad i \geq 2. \tag{10}$$

No synthetic entry value $K_{\mathrm{in},1}^Y = 0$ is assigned to the first stage.

A.2 Successful Exits, Continuations, and Failure Accounting

Diagnostic rollouts execute the current program with seeds indexed as a base seed plus the rollout index. Each rollout records the full snapshot at the first successful completion of each phase. For SA exit-value estimation, at most $C_{\max}$ exits per phase are retained by uniform sampling within the diagnostic collection window, and retained exits receive equal weight within a phase. Each exit is evaluated with up to M suffix continuations; support uses the number of valid exits and the minimum valid branch count across those exits. The same continuation outcomes also support RSAC's common-population boundary comparison. Within each iteration, all compared boundaries refer to the same N_{diag} original diagnostic trajectories. For a fully resolved boundary with S_i live exits and $N_{\mathrm{diag}} - S_i$ trajectories already terminated in failure, each of the C_i uniformly retained live exits receives weight $w_{i,n} = S_i/(N_{\mathrm{diag}} C_i)$. The absorbing failure state receives weight $w_{i,0} = 1 - S_i/N_{\mathrm{diag}}$ and has terminal completion probability zero. SA continues to average conditionally over successful exits. Support thresholds count actual valid exits and branches, rather than absorbing-state mass.

Prefix replay reconstructs program control flow using recorded primitive return values, without advancing the physical simulation. At the branch boundary, execution resumes from the restored physical and execution state, preserving the task configuration, controller state, program variables, and remaining episode horizon. Subsequent primitive calls execute in the simulator. Each branch receives a distinct random seed. This resamples the downstream perception and physical-noise streams while preserving the state and program at the branch boundary.

Failure contributes through distinct channels:

- An execution that never reaches phase i does not enter that phase's reach or pass counts.
- An execution that reaches but does not pass phase i enters its reach count, reducing p_i, and contributes no successful exit to that phase's SA exit-value estimate.
- A failed suffix continuation contributes $Y = 0$ to the branch outcomes of its starting exit, and therefore to that exit's value estimate and the phase's final-outcome statistic.

Thus, local phase failures affect SA through its pass-rate component, while continuation failures affect the estimated downstream values. These SA accounting rules are distinct from the common-population specification of the RSAC boundary comparison.

Adjacent boundaries on a common population. The exit of a phase provides the adjacent phase's entry state, motivating the boundary definition in Equation (10). The compared statistics K_i^Y and K_{i-1}^Y are evaluated on the same trajectory population and use the same terminal outcome. Accordingly, G_i measures the change in terminal-outcome dependence across adjacent boundaries within that population. Support requirements determine whether a boundary comparison can be used, and full-task candidate validation measures the benefit of a concrete edit. The recovery interpretation in Appendix A.6 additionally assumes conditional consistency and faithful replay with fresh independent randomness. It compares recovery on the same original failure population across boundaries, rather than on failures selected separately at each phase.

A.3 Reference Values, Smoothing, and Minimum Support

Let $Y_{i,n,m}$ be the final outcome of branch m from retained exit n of phase i. Its empirical value is

$$\widehat{v}_{i,n} = \frac{1}{M_{i,n}} \sum_{m=1}^{M_{i,n}} Y_{i,n,m}, \qquad \bar{v}_i = \frac{1}{C_i} \sum_{n=1}^{C_i} \widehat{v}_{i,n}. \tag{11}$$

The same construction can be applied to the recorded next-phase outcomes when forming the passability gate.

Cross-fitted reference exits. The high-value reference uses $L_i = \max(1, \mathrm{round}(\rho C_i))$ exits. Let U_i^{top} be the mean of the largest L_i empirical exit values. Branches at each exit are divided into alternating-index groups A and B. One group ranks exits, and the other evaluates the selected set. Writing $\mathcal{T}_A$ and $\mathcal{T}_B$ for the two selected top-L_i sets gives

$$\begin{aligned} U_i^{\mathrm{cf}} &= \frac{1}{2}\left(\frac{1}{L_i}\sum_{n\in\mathcal{T}_A}\widehat{v}_{i,n}^{B} + \frac{1}{L_i}\sum_{n\in\mathcal{T}_B}\widehat{v}_{i,n}^{A}\right), \\ U_i &= \min\{U_i^{\mathrm{top}}, \max(\bar{v}_i, U_i^{\mathrm{cf}})\}. \end{aligned} \tag{12}$$

When there are too few branches for splitting, the implementation uses the direct top-exit reference. Cross-fitting reduces the optimism introduced by selecting and evaluating exits with the same noise realizations; the subsequent clipping keeps the reference between the overall mean and the direct top-exit mean. With $\rho = 0.20$ and $C_i = 8$, the reference averages two exits.

Jeffreys smoothing for boundary diagnostics. For retained live exit n of phase i, let $s_{i,n}$ and $M_{i,n}$ be its successful and valid continuation counts. We use $\widetilde{q}_{i,n} = (s_{i,n} + \frac{1}{2})/(M_{i,n} + 1)$. The absorbing failure state has $\widetilde{q}_{i,0} = 0$ and is not smoothed. Using the common-population weights

defined above gives

$$\begin{aligned}\bar{q}_i &= \sum_{n=0}^{C_i} w_{i,n}\widetilde{q}_{i,n},\\ \widehat{B}_i &= \sum_{n=0}^{C_i} w_{i,n}(\widetilde{q}_{i,n}-\bar{q}_i)^2,\\ \widehat{W}_i &= \sum_{n=0}^{C_i} w_{i,n}\widetilde{q}_{i,n}(1-\widetilde{q}_{i,n}),\\ \widehat{K}_i^Y &= \frac{\widehat{B}_i}{\widehat{B}_i+\widehat{W}_i}.\end{aligned} \tag{13}$$

The weighted variance uses normalized population weights. Throughout the selector, $\widehat{K}_i^Y$ denotes this smoothed statistic computed from terminal outcomes at stage i. The next-phase entry statistic uses its own branch labels and the SA sampling population. Jeffreys smoothing stabilizes extreme empirical exit values.

Finite-branch variation. The population dependence K_i^Y and its plug-in estimate $\widehat{K}_i^Y$ are distinct. To isolate branch noise, suppose every sampled state has M conditionally independent Bernoulli continuations. Let $q = q_i^Y(X_i)$ be its true completion probability and $\widehat{q}$ its empirical branch mean. Taking variance over states and continuation outcomes gives

$$\mathrm{Var}(\widehat{q}) = \mathrm{Var}(q) + \frac{1}{M}\mathbb{E}[q(1-q)]. \tag{14}$$

Observed exit-value dispersion contains both state heterogeneity and finite-branch sampling variation. RSAC computes $\widehat{K}_i^Y$ as a smoothed plug-in diagnostic from these exit values. Consequently, adjacent gain estimates and their maximum vary across diagnostic samples, and empirical gains can take either sign. Under the assumptions of Theorem 1, the population gain G_i is nonnegative. Cross-fitting separates exit selection and evaluation in U_i^{cf}, before clipping to obtain U_i (Equation (12)); Jeffreys smoothing stabilizes the empirical rates used for boundary-dependence estimation.

A phase is *supported* when its statistical row exists, its consumed K is finite, it has at least $C_{\min}$ valid exits, and every included exit has at least $M_{\min}$ valid branches. A boundary is supported only when both of its phases are supported. Invalid or unsupported boundaries terminate the upstream scan.

Uninformative diagnostic outcomes. Missing exits or a non-finite consumed statistic invoke the selector's support fallback. If smoothing produces a finite estimate from constant branch outcomes, it remains subject to the same entry and gain tests; smoothing alone does not turn such an estimate into a non-finite value. RSAC takes an existing initial target j as input, so choosing this target when all SA scores are unavailable or tied is a separate upstream selection rule.

A.4 Complete Target-Selection Procedure

Let $\mathcal{S}$ be the phases with an SA row, α_{in} the terminal-entry threshold, and $g_{\min}$ the minimum accepted gain. The implementation's one-step fallback is

$$r(j) = \begin{cases} j-1, & j-1 \in \mathcal{S}, \quad K_{\mathrm{entry}}(j) \text{ is finite},\\ & \qquad K_{\mathrm{entry}}(j) > \alpha_{\mathrm{redir}},\\ j, & \text{otherwise.}\end{cases} \tag{15}$$

Here $K_{\mathrm{entry}}(j)$ denotes the entry statistic supplied to the existing fallback interface; it is kept distinct from the explicitly terminal-targeted $\widehat{K}_{j-1}^Y$ used by RSAC. This one-step rule is an internal fallback component.

The following procedure returns an editing target and a reason code. All comparisons use the consumed smoothed statistics.

1. If j is absent from the phase schema or $\mathcal{S}$, return $(j, \texttt{invalid_delta_target})$. If $j = 1$, return $(j, \texttt{delta_is_first_phase})$.

Table 4: RSAC target-selection outcomes in evaluation order. $r(j)$ is the one-step fallback in Equation (15).

Condition	Reason code	Target
Invalid initial target	`invalid_delta_target`	j
Initial target is first phase	`delta_is_first_phase`	j
Initial boundary lacks support	`insufficient_j_boundary_support`	$r(j)$
$\widehat{K}^{Y}_{j-1} \leq \alpha_{\text{in}}$	`low_entry_dependence`	j
No identifiable scanned gain	`no_identifiable_gain`	$r(j)$
Largest gain is below $g_{\min}$	`weak_gain`	$r(j)$
Largest gain is at least $g_{\min}$	`strong_generator`	$k^\star$

Table 5: Default diagnostic sampling settings and selector thresholds. The separate fixed-program illustration in Figure 2 uses 20 exits per boundary.

Setting	Symbol	Value
Branches per retained exit	M	8
Maximum retained exits per phase	$C_{\max}$	8
Reference fraction	ρ	0.20
Minimum valid exits	$C_{\min}$	6
Minimum branches per included exit	$M_{\min}$	6
Terminal-entry threshold	α_{in}	0.50
One-step fallback threshold	α_{redir}	0.50
Minimum accepted boundary gain	$g_{\min}$	0.20

2. Compute the one-step fallback $r(j)$. If phase j or $j-1$ lacks support, return $(r(j), \texttt{insufficient_j_boundary_support})$.
3. If $\widehat{K}^{Y}_{j-1} \leq \alpha_{\text{in}}$, return $(j, \texttt{low_entry_dependence})$.
4. Initialize $k^\star$ as unset and $\widehat{G}^\star = -\infty$. For $i = j, j-1, \ldots, 2$:
   (a) If phase i or $i-1$ is unsupported, terminate the loop immediately. Do not resume at an earlier boundary.
   (b) Compute $\widehat{G}_i = \widehat{K}^{Y}_{i} - \widehat{K}^{Y}_{i-1}$.
   (c) If $\widehat{G}_i > \widehat{G}^\star$, set $k^\star = i$ and $\widehat{G}^\star = \widehat{G}_i$. Strict improvement preserves the more downstream candidate on an exact tie.
5. If $k^\star$ is unset, return $(r(j), \texttt{no_identifiable_gain})$.
6. If $\widehat{G}^\star < g_{\min}$, return $(r(j), \texttt{weak_gain})$. Otherwise, return $(k^\star, \texttt{strong_generator})$.

The final reason string is an implementation identifier for an accepted boundary gain. The diagnostic is used to propose an editing location; candidate execution determines the effect of editing there.

Equivalently, after the initial support and entry tests, let $\mathcal{C}(j)$ be the consecutive segment of supported boundaries encountered from j before the first unsupported boundary. The scan computes

$$k^\star = \arg\max_{i \in \mathcal{C}(j)} \widehat{G}_i, \tag{16}$$

with downstream tie-breaking, and accepts this target only if $\widehat{G}_{k^\star} \geq g_{\min}$. It searches this whole segment rather than stopping at the first above-threshold gain. Phase j is included in the candidate set, whereas stage 1 has no defined gain and is never assigned an artificial predecessor statistic.

A.5 Diagnostic Configuration

The selector's default gain threshold is $g_{\min} = 0.20$.

A.6 Population Interpretation of Boundary Gains

We relate common-population boundary gains to changes in conditional completion value and to the recovery of failed executions. The resulting identities connect state formation to the marginal recovery obtained by extending replay across a phase, providing a diagnostic basis for prioritizing editing regions.

Population and boundary values. Fix the current program P_t and the common trajectory population $\mathcal{P}$. Write $\mu = \Pr_{\mathcal{P}}(Y = 1) \in (0, 1)$. When $\mathcal{P}$ is the full forward-execution law under the task distribution, $\mu = J(P_t)$. For this analysis, every boundary state X_i is defined on the same original trajectory population. A trajectory that terminates in failure before boundary i is represented there by an absorbing failure state, whose continuation value is zero. No phase-specific conditioning on successful passage is applied. All expectations and variances below are under $\mathcal{P}$. For $i = 1, \ldots, H$, define

$$q_i^Y(x) = \Pr_{\mathcal{P}}(Y = 1 \mid X_i = x), \qquad K_i^Y = \frac{\mathrm{Var}(q_i^Y(X_i))}{\mu(1-\mu)}. \tag{17}$$

The population gain is $G_i = K_i^Y - K_{i-1}^Y$ for $i \geq 2$. No artificial entry statistic is assigned to phase 1.

Conditional consistency and faithful replay. We retain the conditional consistency assumption

$$\mathbb{E}\big[q_i^Y(X_i) \mid X_{i-1}\big] = q_{i-1}^Y(X_{i-1}), \qquad i = 2, \ldots, H. \tag{18}$$

In addition, assume that each snapshot contains the task, physical, and control information needed to reproduce the conditional law of the remaining execution. Let Y_i^{rep} be the outcome of restoring the original state X_i and replaying the unchanged suffix with fresh future randomness. Given X_i, assume that Y_i^{rep} is independent of the original outcome Y and that $\Pr(Y_i^{\mathrm{rep}} = 1 \mid X_i) = q_i^Y(X_i)$. Probabilities involving replay also integrate over this fresh randomness.

Define the failure-conditioned recovery probability

$$R_i := \Pr(Y_i^{\mathrm{rep}} = 1 \mid Y = 0)\,. \tag{19}$$

All R_i refer to the same original failure event $Y = 0$. Replay from boundary $i-1$ re-executes phase i and its suffix; replay from boundary i retains the original exit of phase i and re-executes only its suffix. These auxiliary replay outcomes give the boundary-gain statistic an operational interpretation.

Theorem 1 (Boundary gains as marginal recovery from rewinding)**.** *Under the population, consistency, and replay assumptions above, for every $i = 1, \ldots, H$,*

$$R_i = \mu(1 - K_i^Y). \tag{20}$$

Moreover, for every $i = 2, \ldots, H$,

$$\begin{aligned} R_{i-1} - R_i &= \mu G_i \\ &= \frac{\mathbb{E}\Big[\big(q_i^Y(X_i) - q_{i-1}^Y(X_{i-1})\big)^2\Big]}{1-\mu} \geq 0. \end{aligned} \tag{21}$$

Thus, μG_i is exactly the marginal increase in recovery probability obtained by re-executing phase i rather than retaining its original exit, conditional on the original execution having failed.

Proof. Conditional independence and faithful replay give

$$\begin{aligned} R_i &= \frac{\Pr(Y_i^{\mathrm{rep}} = 1, Y = 0)}{1-\mu} \\ &= \frac{\mathbb{E}\big[q_i^Y(X_i)\big(1 - q_i^Y(X_i)\big)\big]}{1-\mu} \\ &= \frac{\mu(1-\mu) - \mathrm{Var}(q_i^Y(X_i))}{1-\mu} = \mu(1 - K_i^Y), \end{aligned}$$

where $\mathbb{E}[q_i^Y(X_i)] = \mu$ by the law of total expectation. Subtracting the identities at boundaries $i-1$ and i gives $R_{i-1} - R_i = \mu G_i$. By (18) and the law of total variance,

$$\begin{aligned} G_i &= \frac{\mathrm{Var}(q_i^Y(X_i)) - \mathrm{Var}(q_{i-1}^Y(X_{i-1}))}{\mathrm{Var}(Y)} \\ &= \frac{\mathbb{E}\big[\mathrm{Var}(q_i^Y(X_i) \mid X_{i-1})\big]}{\mathrm{Var}(Y)} \\ &= \frac{\mathbb{E}\Big[\big(q_i^Y(X_i) - q_{i-1}^Y(X_{i-1})\big)^2\Big]}{\mu(1-\mu)} \geq 0. \end{aligned} \tag{22}$$

Multiplying by μ completes the proof. □

State formation versus transmission. Equation (21) implies

$$G_i = 0 \quad \Longleftrightarrow \quad q_i^Y(X_i) = q_{i-1}^Y(X_{i-1}) \quad \text{almost surely.}$$

A phase can therefore have high exit dependence yet zero boundary gain if it only preserves existing continuation-value differences. Replaying such a phase adds no recovery benefit over replaying from its exit. More generally, for $2 \leq a \leq b \leq H$,

$$R_{a-1} - R_b = \mu \sum_{i=a}^{b} G_i.$$

An intervening chain of value-preserving phases contributes zero gain, rather than repeatedly receiving credit for the same state differences. Within a fixed candidate set, maximizing G_i is equivalent to maximizing the marginal recovery increment $R_{i-1} - R_i$.

Implications for editing localization. Adjacent gain separates the dependence gained within a phase from dependence transmitted through later phases. The theorem gives this distinction an operational interpretation: larger gain corresponds to greater marginal recovery when replay extends across that phase. RSAC uses supported gain estimates to prioritize a program region; the editor proposes a localized revision, and paired full-task validation measures its effect and determines adoption.

B Experience-Guided Editing Details

B.1 Version-Associated Records and Retrieval

Each trajectory record links its producing program version to a primitive trace, stage progress, state summaries, and the complete-task outcome. Compact summaries include relative positions, robot pose, grasp relations, and the states of previously completed objects. These summaries support comparisons and language-model inputs. Separately stored execution snapshots contain the physical state, controller history, task events, and recorded prefix returns required for continuation.

Retrieval selects successful executions, high-progress failures, and low-progress failures from the current task. Success retrieval favors recent records; failure retrieval considers progress and distance from the current program version. Program identity is used to control duplicate examples. Failure–success comparisons summarize termination stages, task progress, final robot states, and object displacements, with preference for nearby versions of the same skill. Each example retains its original execution context. Historical trajectories provide behavioral references for editing; the current program's diagnostic executions provide its state statistics.

Diagnostic trajectories enter the run's memory. Development evaluations used to score candidates do not enter diagnostic memory, and final testing uses instances and randomness held outside program selection. If optimization continues after an adoption decision, new diagnostic executions are associated with the program actually retained. Each controlled optimization run maintains its own memory and version history.

B.2 Two-Step Generation and Patch Application

The editor first constructs a structured diagnosis from the current program, selected editing stage k, stage statistics, features of high- and low-value exits, retrieved trajectories, and recent editing feedback. This diagnosis identifies relevant state factors, behaviors to preserve, and possible code changes. Where multi-object monitoring is available, it also includes the states and violations of previously completed objects.

The second step proposes candidate plans comprising parameter changes, inserted or removed operations, reordered calls, and local logic changes. The allowed number of patches per plan decreases across optimization iterations. For iteration t, let $\mathcal{B}_t$ denote current trajectory and state evidence, $\mathcal{E}_t$ retrieved experience, and $\mathcal{H}_t$ editing feedback and any available reflection. With diagnosis d_t and candidate set $\mathcal{Q}_t$, the two calls are

$$\begin{aligned} d_t &= \mathcal{G}_{\text{diag}}(P_t, \mathcal{B}_t, \mathcal{E}_t, \mathcal{H}_t, k), \\ \mathcal{Q}_t &= \mathcal{G}_{\text{edit}}(P_t, d_t, \boldsymbol{\delta}^{\text{SA}}, k). \end{aligned} \tag{23}$$

Here, $\boldsymbol{\delta}^{\text{SA}}$ contains the stage credits. Both operators are prompted calls to the fixed language model. The index t denotes an optimization iteration; k denotes the editing stage selected for that iteration.

Stage boundaries map primitive-call intervals to source-code regions. When this mapping is available, the default patch procedure enforces the selected region and checks for ambiguous replacements. Within a candidate plan, edits are applied and evaluated sequentially; edits that do not reduce development performance are retained within that plan. Candidate plans are then compared before the final adoption decision.

B.3 Validation Records and Cross-Version Reflection

Current and candidate programs are evaluated on the same development seeds. Paired outcomes record both newly successful executions and losses of existing successes; adoption considers their net gain and direction. The candidate, parent version, editing target, evaluation results, and adoption decision are stored together. Rejected edits supply feedback for subsequent diagnosis, and valid evaluation results may be reused when the program is unchanged. Optimization stops when the budget, development target, or stagnation condition is reached.

Once a skill has accumulated multiple versions with execution evidence, reflection can periodically summarize version-level success, progress, termination stages, and representative failures. The language model produces guidance on effective changes, directions to avoid, and subsequent priorities. This guidance remains within the current run and is supplied to later editing calls. Reflection organizes the revision history, while trajectory retrieval provides individual behavioral examples.

C State Handoffs and Program-Revision Evidence

This appendix examines how stage-level execution evidence relates to useful program revisions. Compatibility between successive skill states also motivates learned-interface and long-horizon planning approaches (Lee et al., 2022; Mishra et al., 2023); here we study how changing an executable program affects that compatibility. We first report controlled edits to fixed programs, then describe revisions produced and adopted by the online framework, and finally inspect the historical experience supplied to an editor. These studies connect state handoffs, editing locations, concrete code changes, and full-task outcomes.

Task success rates below refer to completion of the entire task; stage-pass counts are identified separately. The fixed-program interventions and the logged optimization episodes use distinct starting programs and are reported as separate studies. Controlled comparisons evaluate the original and modified programs on the same initial seeds; online examples use the development evaluations recorded at the corresponding editing iteration.

C.1 Controlled Edits to State Handoffs

On `apple_chain_z15`, we modify a fixed program by inserting `pitch_to(0.0)` immediately after a successful grasp and before looking for the placement target. This single operation changes

Table 6: Controlled modifications of a fixed `apple_chain_z15` program on 100 paired seeds. Repaired and lost counts compare each candidate with the original program. A repaired execution changes from failure to success; a lost execution changes from success to failure.

Modification	Successes	Repaired	Lost
Original program	7	—	—
Pitch reset after grasping	28	22	1
Grasp modification only	3	3	7
Press modification only	7	0	0
Return modification only	7	0	0
Grasp modification and pitch reset	22	21	6
Pitch reset and downstream edits	29	23	1

Table 7: Boundary evidence and tested revisions for `apple_chain_z10`. Diagnostic quantities use terminal outcome Y. Edit gains compare the best reported insertion in the indicated region with the unchanged program's 14% success rate.

Region	$\widehat{K}_i^Y$	$\widehat{G}_i$	Edited SR (%)	Gain (pp)
Grasp	0.544	0.374	61	+47
Carry	0.707	0.163	60	+46

the state handed to the transport and placement sequence. Across 100 paired seeds, task success increases from $7/100$ to $28/100$. The edit repairs 22 previously unsuccessful executions and loses one previous success. Both programs pass Carry in 31 executions, whereas successful placement and subsequent task completion increase from 7 to 28. Thus, the aggregate Carry pass count alone does not distinguish programs with substantially different downstream performance.

Table 6 compares this change with other tested modifications. The tested Press and Return edits leave every seed's terminal outcome unchanged. Combining the Grasp modification with the handoff reset yields $22/100$, below the reset alone, illustrating interactions between edits. The Grasp-only change repairs three executions but loses seven, for a net loss of four. These results motivate evaluating complete candidate programs after localized changes, even when individual operations appear useful.

A second fixed-program study on `place_apple_zone` evaluates 100 paired seeds. The original program succeeds in 57 executions. A tested pitch reset raises this to 73, with 28 repairs and 12 losses. A different placement of the same type of operation, associated with grasping, yields 46 successes, with 17 repairs and 28 losses. Applying resets at both tested locations produces 65 successes. The direction of the effect therefore depends on where the operation is inserted, and combining the two locations does not preserve the better individual result. These manually specified edits test the usefulness of changing state handoffs; the online editing examples below examine how the framework produces actual revisions.

C.2 Boundary Diagnosis and a Finite Edit Sweep

The `apple_chain_z10` study in Figure 2 compares a fixed-program diagnostic with a separate sweep of insertion locations. The unchanged program succeeds on $14/100$ evaluation seeds. Inserting a pitch reset in the Grasp region yields $61/100$, with 48 repairs and one loss. Two tested Carry-region insertions yield $60/100$ and $56/100$, with paired repair/loss counts of $50/4$ and $46/4$, respectively.

The diagnostic initially focuses on Place. Among the supported upstream boundaries, Grasp has the largest measured gain, even though Carry has higher exit dependence (Table 7). The larger change in predictive dependence therefore highlights a region with a useful tested revision. The diagnostic uses 20 sampled exits and eight continuations per exit, with the same terminal outcome at every boundary. The maximum boundary gain identifies Grasp as the upstream editing region. The separate edit sweep establishes correspondence between this boundary evidence and effective edits within the tested insertion set.

Table 8: Two online program revisions. Development SRs are the recorded before–after candidate-evaluation values. Paired counts are from the final adoption check. Both candidates were adopted.

Task	Actual modification	Dev SR (%)	Repaired	Lost
Chain	Add a short backward walk in Carry	$65.0 \rightarrow 72.5$	9	6
Pick	Approach distance $0.80 \rightarrow 0.50$ in Fetch	$37.5 \rightarrow 87.5$	5	1

Table 9: Complete-task success before and after the revision on the same 40 seeds. Repaired and lost counts compare the revised program with its parent.

Program	SR (%)	Repaired	Lost
Parent	47.5	—	—
Revised	60.0	9	4

C.3 Online Localization, Editing, and Adoption

Table 8 records two actual editing iterations. Candidate evaluation and the final paired adoption decision are retained as distinct logged quantities. Each before–after comparison uses the development evaluations recorded at that editing iteration.

In the Chain case, all 16 diagnostic executions pass the stages through Carry, but only nine pass Place. SA nevertheless selects Carry as its initial editing focus, reflecting evidence beyond the location of observed stage failures. This illustrative run uses a case-specific gain threshold of $g_{\min} = 0.50$, while the default setting is 0.20. Its maximum upstream gain is 0.441, below the case-specific threshold, so the selector returns `weak_gain`; Carry is retained as the final editing target. Thus, the initial and final editing locations coincide. The editor adds a small backward walk inside the selected Carry region. Development SR rises from 0.650 to 0.725, and the final paired check records nine repairs and six losses, leading to adoption.

In the Pick case, the initial SA focus is Place, with terminal entry dependence 0.666. The diagnostic records a maximum boundary gain of 0.113 and a `weak_gain` decision. The one-step fallback selects Fetch, where the editor changes the apple-approach stopping distance from 0.80 to 0.50. This is the sole recorded code difference for the adopted revision. Development SR increases from 0.375 to 0.875, and the adoption check records five repairs and one loss. These episodes show how the full selection procedure, including its fallback, supplies an editing region that is respected by the actual modification and evaluated through complete-task execution.

A multi-stage upstream revision. An additional logged Chain revision illustrates localization beyond the immediate predecessor. With $C_{\max} = 16$, $M = 8$, and $g_{\min} = 0.20$, SA initially focuses on Press, while the backward scan selects Carry through the predecessor sequence Press–Place–Carry. The logged maximum gain is 0.272, exceeding the acceptance threshold and redirecting editing two stages upstream.

The selected revision modifies the Carry region by inserting `pitch_to(0.4)` and reducing the target-approach stopping distance from 0.5 to 0.2. On the same 40 seeds, complete-task success increases from 47.5% to 60.0% (Table 9). The paired comparison records nine repaired failures and four lost successes, giving a net gain of five and leading to adoption. This case connects multi-stage upstream localization to a concrete revision within the selected region and subsequent full-task validation.

C.4 Localization Behavior in Supplementary Runs

We summarize 135 online target-selection decisions from 12 existing Full-RSAC runs and four supplementary Full-RSAC runs on Chain and Zone (Table 10). The initial focus is retained in 112 decisions (83.0%), and the target moves one stage upstream in 23 decisions (17.0%). No decision in this collection moves the target at least two stages upstream. The counting unit is an online

Table 10: Editing positions relative to the initial SA focus in 16 Full-RSAC runs on Chain and Zone. Counts refer to online target-selection decisions.

Editing position	Decisions	Share (%)
Retain the initial focus	112	83.0
One stage upstream	23	17.0
At least two stages upstream	0	0.0
Total	135	100.0

target-selection decision within an optimization run. These records document both current-stage retention and upstream redirection in the optimization loop.

C.5 Historical Experience in an Editing Input

A `haul_press` episode exposes the version-associated memory used during candidate generation. The run starts with an empty trajectory pool and accumulates 60 records. At a later iteration, the editor receives three concrete experiences: a successful execution from version 2 with complete progress and target displacement $1.27\,\mathrm{m}$; a version-2 placement failure with progress 0.60 and object-to-bin distance $0.28\,\mathrm{m}$; and a version-0 fetching failure with progress 0.10 and object-to-bin distance $1.44\,\mathrm{m}$. The input preserves the producing version and outcome of each trajectory, making earlier failures available alongside more recent behavior.

The resulting candidate changes a walking operation. It passes the candidate-level check with unchanged development SR, $0.75 \rightarrow 0.75$. The final paired comparison records zero repairs and zero losses, so the current program is retained. This episode shows how version-associated experience enters candidate generation and how paired full-task validation determines whether a proposed revision is retained.

D Model and Comparison Protocol

All methods use Qwen3.7-Max. Optimization modifies high-level robot programs while the parameters of the language model and low-level motion controller remain fixed. At each program-improvement iteration, all methods use the same number of rollouts.

External baselines are adapted to the robot-program execution interface. CaP-Agent0 uses execution-failure feedback to guide program revision; ASPIRE performs multi-candidate program search using execution feedback; and SkillOpt applies reflection- and validation-based editing to executable robot programs. Each method retains its respective candidate organization and update procedure. Final programs are evaluated by complete-task success rate.

Candidate adoption in Full RSAC requires a paired net gain of at least three successful executions; the minimum-net-gain-of-one ablation changes this threshold to one. The main comparison and component ablations each use ten independent optimization runs per task and method or configuration, with each final program evaluated on 200 unseen seeds. The Full-RSAC references in the component ablations reuse the corresponding ten main-comparison runs. Aggregation is specified in Appendix E.

E Evaluation Aggregation

For task a, method m, and independent run q, let $r_{a,q,m}$ denote complete-task SR in percent, and let $n_{a,m}$ be the number of reported runs for that task–method pair. Each main-comparison run uses 200 held-out evaluation seeds, so $r_{a,q,m} = 100c_{a,q,m}/200$, where $c_{a,q,m}$ is the success count. We compute

$$\bar{r}_{a,m} = \frac{1}{n_{a,m}} \sum_{q=1}^{n_{a,m}} r_{a,q,m}, \qquad s_{a,m} = \sqrt{\frac{1}{n_{a,m}-1} \sum_{q=1}^{n_{a,m}} (r_{a,q,m} - \bar{r}_{a,m})^2}. \tag{24}$$

In Table 1, $n_{a,m} = 10$ for every task–method pair. The reported sample standard deviations quantify variation across the ten independent optimization runs. Means and sample standard deviations are computed from the reported run-level SRs before rounding the displayed values to one decimal place.

E.1 Run-Wise Macro Aggregation

At each run index $q \in \{1, \ldots, 10\}$, we average the eight task families with equal weight:

$$R_{q,m} = \frac{1}{8} \sum_{a=1}^{8} r_{a,q,m}. \tag{25}$$

The macro row of Table 1 then reports

$$\bar{R}_m = \frac{1}{10} \sum_{q=1}^{10} R_{q,m}, \qquad s_{R,m} = \sqrt{\frac{1}{9} \sum_{q=1}^{10} (R_{q,m} - \bar{R}_m)^2}. \tag{26}$$

The resulting summaries are $44.4 \pm 3.5\%$ for RSAC, $25.2 \pm 8.6\%$ for CaP-Agent0, $30.3 \pm 7.9\%$ for ASPIRE, and $17.6 \pm 7.3\%$ for SkillOpt. With ten runs for every task, the macro mean also equals the unweighted mean of the eight task means. The macro SD is computed over the ten task-averaged run results, rather than by averaging task-level SDs.

The five component-ablation variants in Table 2 each use a fresh batch of ten independent optimization runs per task. Their summaries use the same arithmetic mean and sample-standard-deviation formula with $n_{a,m} = 10$. Full-RSAC references reuse the corresponding main-comparison results. The final row of Table 2 is the unweighted average of the eight task means, computed before rounding. Figure 3(a) reports the mean at each iteration across the same ten independent optimization runs for each method. Panel (b) displays the ten-run Chain means with numerical labels rounded to whole percentages and without standard-deviation bars; the sample standard deviations are given in Table 2.

F Additional Connections to Prior Work

Dependencies between successive skills. Generative Skill Chaining learns skill-centric diffusion models and composes their distributions to construct long-horizon plans subject to geometric constraints (Mishra et al., 2023). This work shares our concern with dependencies between successive operations. Our analysis examines the states produced by a retained executable program: fixed-suffix continuations estimate how stage-exit states support eventual task completion, and this evidence guides the localization of program revisions.

Execution feedback and reusable experience. ProgPrompt represents task plans as programs with assertions that check preconditions and trigger recovery actions (Singh et al., 2023). DROC distills human language corrections into reusable knowledge and retrieves relevant experience for subsequent robot tasks (Zha et al., 2024). Our framework uses autonomous execution records associated with the program versions that produced them. Successful and failed trajectories inform editing, while complete-task validation determines whether a proposed revision is retained. Appendix C.5 illustrates this process through a logged editing call and its subsequent validation decision.

Feedback-driven code optimization. Eureka iteratively optimizes reward code using feedback from reinforcement-learning policy training (Ma et al., 2024). Our optimization object is the executable robot program itself. Revisions directly change the operations and intermediate states encountered during execution. The controlled interventions and online editing episodes in Appendix C examine how such changes affect complete-task outcomes.

G Repository-Based Program Initialization

The repository preserves programs together with their execution and revision records. Task descriptions, tags, and primitive information can identify reference programs for initializing a new task. Program retrieval supplies operation structure, whereas trajectory retrieval supplies evidence about

behavior under particular conditions. The controlled improvement experiments described in the main text start from fixed initial programs; this additional initialization capability is separate from the within-run memory mechanism.

H Task Descriptions and Illustrative Action Sequences

We describe the eight task families evaluated in Table 1. The short identifiers below match those used in the main comparison. The figures illustrate the task sequences, including navigation, manipulation, visual observation, and task-specific ordering or object-preservation requirements.

H.1 Apple Chain (apple)

The robot locates an apple, grasps and lifts it, carries it to a marked target zone, and places it in the zone. It then navigates to a designated location, re-approaches the apple, presses it, and returns to the designated final location. The task therefore extends beyond delivery to include a later contact interaction with the placed object (Figure 4).

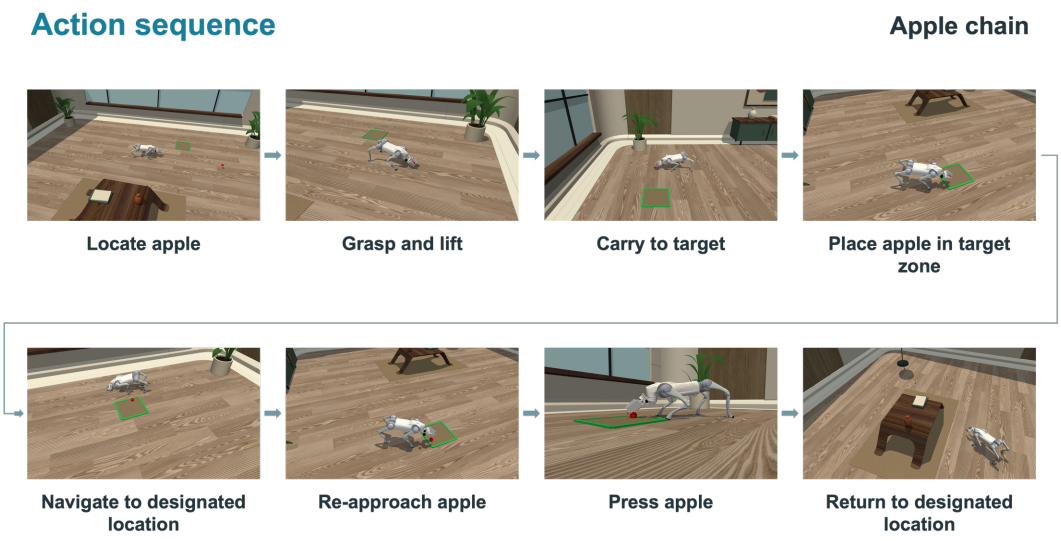


Figure 4: **Apple chain.** Apple grasping, transport, placement, re-approach, pressing, and return.

H.2 Haul with Obstacle Avoidance (haul)

The robot navigates to an apple, grasps it, and carries it toward a distant bin. During transport, it locates the destination and navigates around obstacles while carrying the apple. It then aligns the apple with the bin and releases it. This task combines object transport with navigation through an obstructed scene (Figure 5).

H.3 Dual-Zone Relay (zone)

The robot locates an apple and a cup. It first approaches the apple, transports it to the first target zone, and sets it down and releases it. The robot then turns toward the cup, grasps and lifts it, carries it over the bin, and lowers and releases it into the bin before navigating to a designated final location. The task composes two object-delivery sequences with different destinations (Figure 6).

H.4 Wall Round Trip (wall)

The robot locates an apple on the far side of a sofa that obstructs direct access. It navigates around the sofa, grasps the apple, and returns while carrying it. After navigating toward the placement area, it

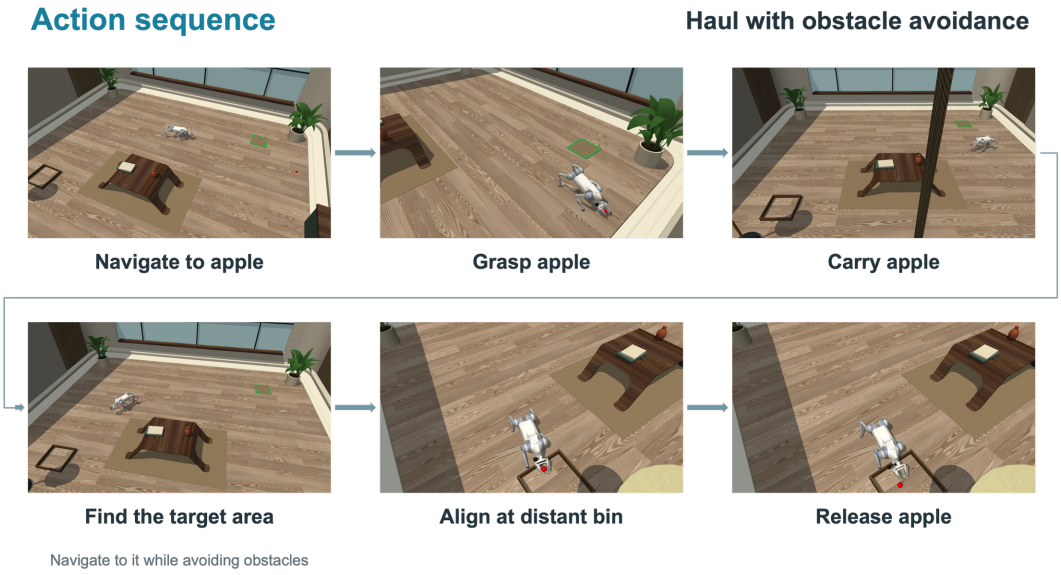


Figure 5: **Haul with obstacle avoidance.** The robot carries an apple through an obstructed scene and releases it into a distant bin.

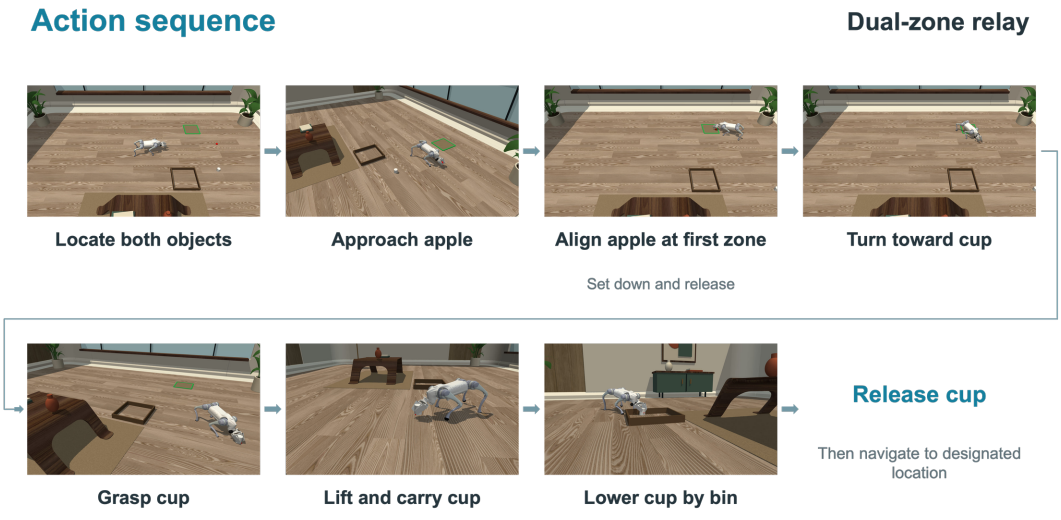


Figure 6: **Dual-zone relay.** Apple placement in the first target zone is followed by cup transport and release into the bin.

places the apple in the target zone and proceeds to a designated final location. The sequence requires outbound navigation and a return trip with the object (Figure 7).

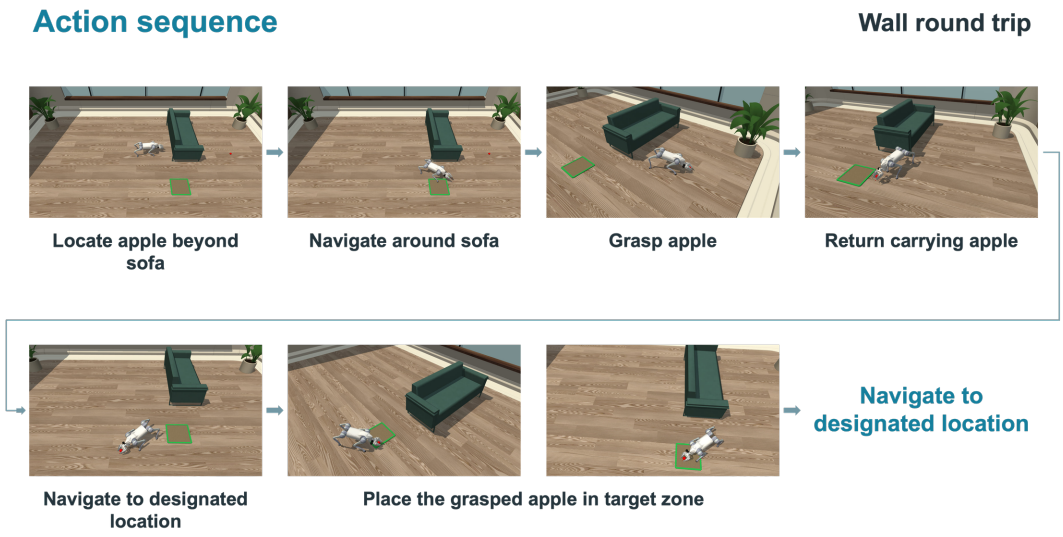


Figure 7: **Wall round trip.** The robot retrieves an apple from beyond the sofa, carries it back, and places it in the target zone.

H.5 Deliver Then Survey (DELIVER)

The robot approaches an apple, grasps and carries it, aligns it over a bin, and releases it into the bin. After delivery, it turns toward a bottle, approaches it, and frames it in the camera view. It then navigates to a designated final location. This task places a visual survey step after an object-delivery sequence (Figure 8).

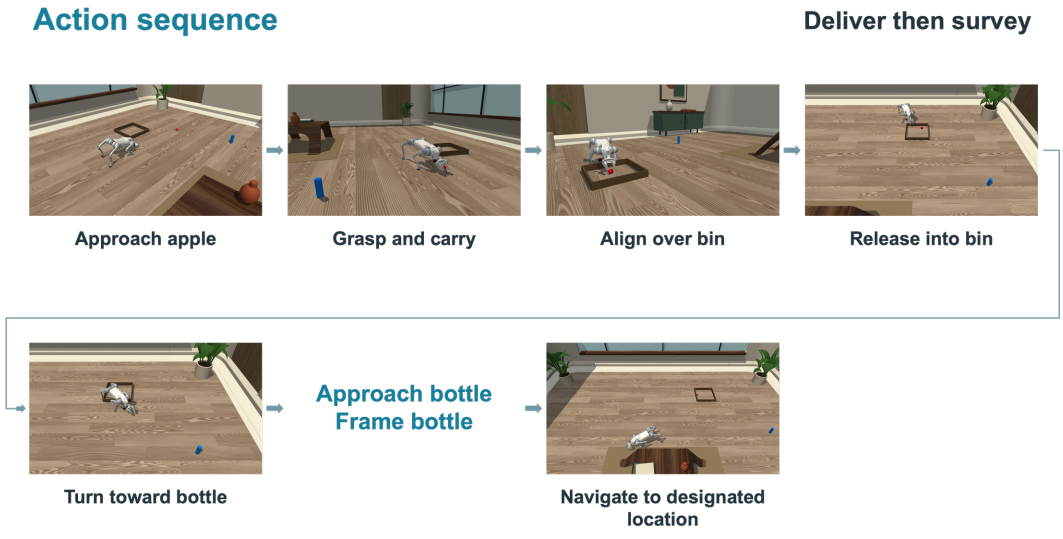


Figure 8: **Deliver then survey.** Apple delivery into the bin precedes approaching and framing the bottle, followed by final navigation.

H.6 Selective Apple Delivery (PICK)

The robot identifies an apple among distractor objects, approaches and grasps it, and carries it to the target zone for placement. It then navigates to a designated final location. The can and book must remain undisturbed throughout execution. Thus, the task combines selective object manipulation with an explicit preservation requirement for the distractors (Figure 9).

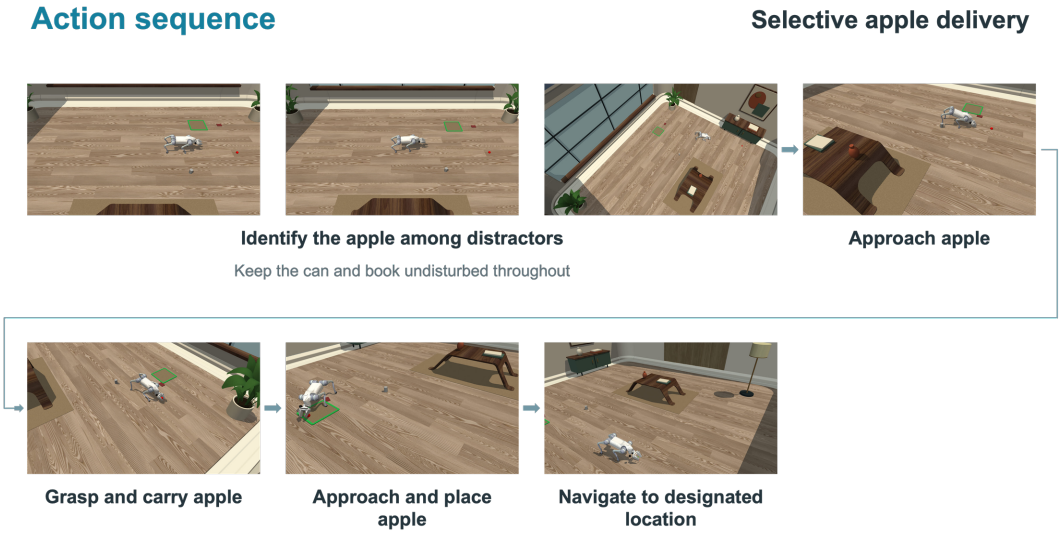


Figure 9: **Selective apple delivery.** The robot delivers the apple while keeping the can and book undisturbed throughout execution.

H.7 Photo-Gated Delivery (GATE)

After locating the apple and its target zone, the robot approaches a can and frames it in the camera view. This visual step must occur before the first apple grasp. The robot then grasps and carries the apple, aligns it with the target zone, and releases it onto the zone. The gate is therefore an observation-before-grasp ordering requirement (Figure 10).

H.8 Three-Stop Visual Survey (SURVEY3)

The robot locates the survey targets and frames the apple in the camera view. It then turns and travels toward the bottle, frames the bottle, and proceeds to the can to frame it. After observing the three targets in the order apple–bottle–can, the robot navigates to a designated final location. The illustrated sequence consists of visual observation and navigation (Figure 11).

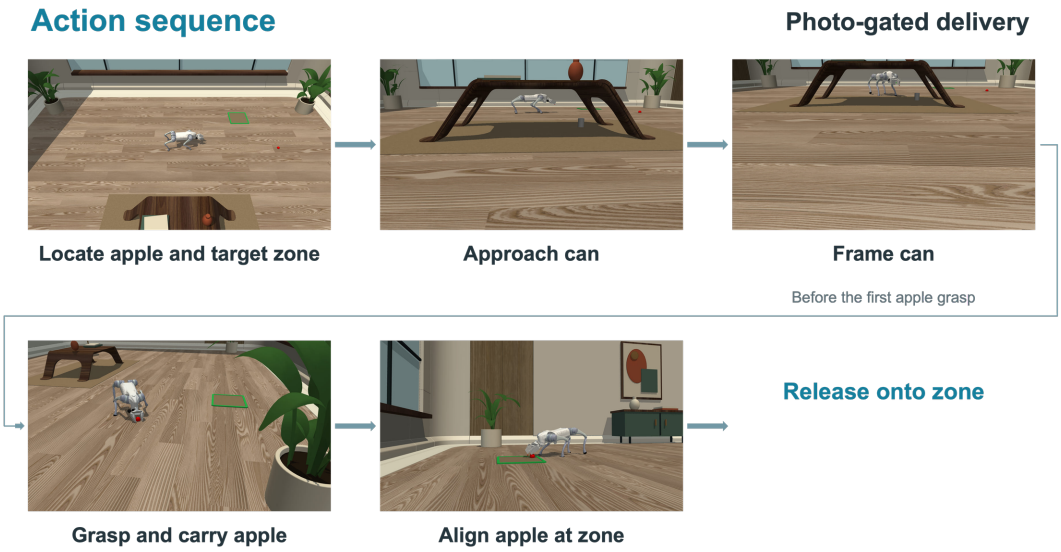


Figure 10: **Photo-gated delivery.** Framing the can is required before the first apple grasp; apple transport and placement follow.

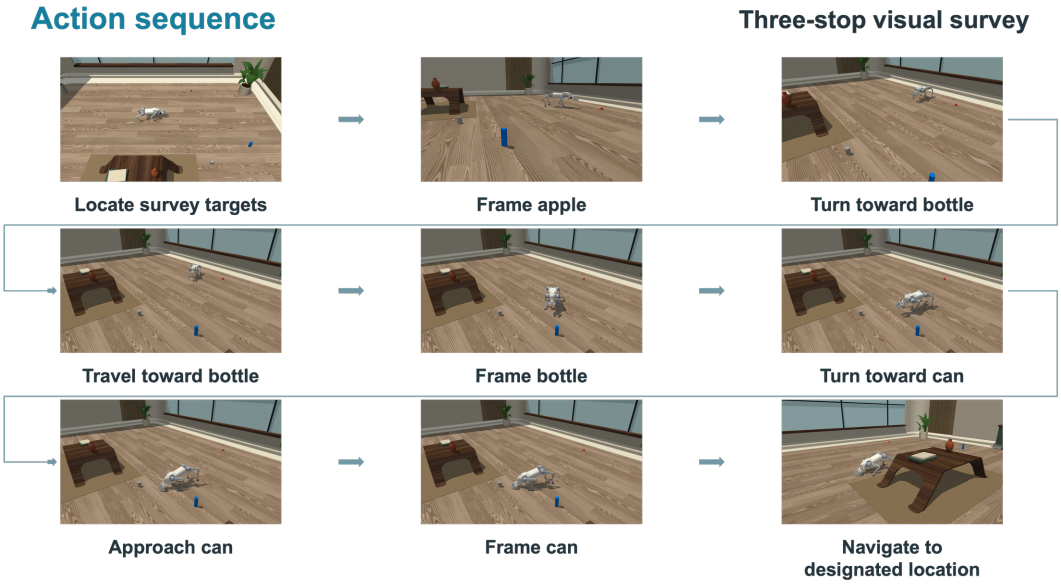


Figure 11: **Three-stop visual survey.** The robot sequentially frames the apple, bottle, and can, then navigates to the final location.