MICA: HARNESS SELF-EVOLUTION THROUGH INTERVENTION-GUIDED CO-ADAPTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Harness self-evolution refines runtime instructions through **persistent context edits**, but individually beneficial edits can fail in combination because one edit can change how another is interpreted or executed. We introduce **MICA** (**M**atched **I**ntervention-guided **C**o-**A**daptation), which evolves the harness by jointly adapting edit structures and templates while keeping language models frozen. **Matched executions** guide a frozen language model to propose composition-mismatch hypotheses and minimal template repairs. With edit structure fixed, MICA compares the original template pair, each unilateral revision, and the **joint revision** to measure the benefit of co-adaptation. Competing structures receive the same **bounded adaptation allowance** before execution-based selection, followed by frozen screening and independent deployment confirmation. Across five environments, MICA achieves macro-average scores of **80.90%** with `GPT-5.5` and **76.80%** with `DeepSeek-V4-Flash`, exceeding the strongest baselines by **4.82%**↑ and **1.72%**↑, respectively. Under common budgets, MICA improves deployed scores by **3.25%**↑ on *Telecom* and **4.23%**↑ on *AppWorld Normal* over the strongest search baseline. Controlled studies demonstrate that matched evidence and paired repair each improve deployment reward, establishing **harness co-adaptation** as an effective approach to harness self-evolution. The code is available at the anonymous code repository.

1 INTRODUCTION

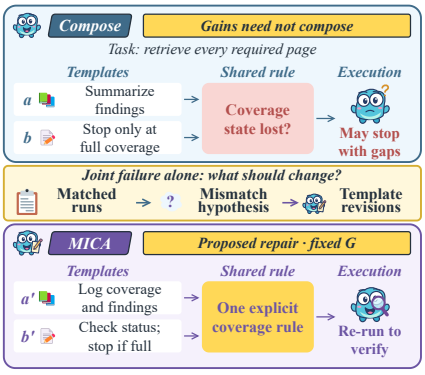


Figure 1: **From composition mismatch to testable repair.** In this illustrative example, a summary may hide the coverage state needed for stopping. Matched runs share task and initial state, supporting testable hypotheses. Fixed-G template revisions require fresh execution to verify the proposed repair.

Harness evolution improves agent behavior by adapting prompts, skills, tools, and execution workflows through experience and feedback while keeping language model parameters fixed (Pan et al., 2026). Within the harness, the context layer supplies reusable instructions and task-relevant information, with persistent context updates retaining lessons from prior executions (Zhang et al., 2026). However, individually beneficial context edits can fail in combination because one edit changes the information available to another or how its instructions are interpreted. Effective context-layer evolution should therefore adapt context edits according to their **joint effects** on agent behavior.

The original context-learning paradigm extracted reusable knowledge from accumulated agent experiences and recalled it to guide subsequent agent decisions without updating model parameters (Zhao et al., 2024). Subsequent work follows two main directions. First, **context learning** accumulates and refines strategies in persistent context, as in Dynamic Cheatsheet (Suzgun et al., 2026) and ACE (Zhang et al., 2026). Second, **candidate optimization** revises and selects prompts, skills, or runtime configurations, as in

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.

GEPA (Agrawal et al., 2026), SkillOpt (Yang et al., 2026), and RHO (Pan et al., 2026). These methods use execution experience to improve reusable contextual guidance or executable candidates.

However, improving context or selecting stronger candidates does not establish which companion edits need revision when their composition fails, as illustrated in Figure 1. A summary can hide coverage information needed by a stopping instruction, while ordering or input bindings can also affect stopping behavior. Aggregate rewards alone cannot distinguish these causes. Context-layer evolution therefore requires composition-level diagnosis and repair to identify how edits interact and verify whether joint revision improves their combined behavior.

To address this limitation, we introduce MICA, **Matched Intervention-guided Co-Adaptation**, for **evolving the context layer of agent harnesses**. MICA jointly adapts context-edit structures and templates while keeping language models frozen. Here, structure specifies edit selection, execution order, input bindings, and dependencies within the context-edit program. Matched executions of the unedited agent, individual edits, and legal compositions guide a frozen language model to propose testable mismatch hypotheses and minimal template repairs. With structure fixed, fresh executions compare the original template pair, each unilateral revision, and their joint revision to assess the benefits of co-adaptation. Competing edit structures receive equal bounded adaptation before reward-based selection. The resulting context-edit programs are frozen for screening and independent confirmation against the unedited agent, so deployment depends on measured gains.

To assess its effectiveness, we evaluate MICA across five environments: *Telecom* and *Airline* from τ^2-*bench* (Barres et al., 2025), *Retail* from τ-*bench* (Yao et al., 2025), *ALFWorld* (Shridhar et al., 2021), and *AppWorld Normal* (Trivedi et al., 2024). Using `GPT-5.5`[1] and `DeepSeek-V4-Flash` (DeepSeek-AI et al., 2026) under Pi[2], MICA reaches macro averages of **80.90%** and **76.80%**, respectively, exceeding the strongest baseline averages by 4.82%↑ and 1.72%↑. Controlled studies examine the contributions of matched evidence and paired repair under `GPT-5.5` and Pi. Under common resource caps, MICA improves mean deployed score over the strongest search baseline by **3.25%↑ on *Telecom*** and **4.23%↑ on *AppWorld Normal***. We make three contributions:

① ***An intervention-based view of composition repair.*** We make the behavior of an edit composition the unit of diagnosis and repair, using matched executions to connect joint outcomes to revisable template constraints.

② ***Structure-conditioned template co-adaptation.*** MICA combines fixed-structure four-condition comparisons with bounded adaptation and measured-reward selection, separating proposal generation from execution-based retention.

③ ***Empirical validation.*** Experiments across five environments show that MICA achieves the highest macro averages in both model configurations. Controlled studies demonstrate gains from matched evidence and paired repair, with higher deployed reward under shared modification permissions and budgets.

2 Related Work

Harness self-evolution uses execution feedback to improve persistent context and runtime configurations while keeping language models frozen. Existing work follows three technical routes: learning in external context, optimizing executable candidates, and deriving feedback from interventions.

(I) Context learning. Reflexion stores verbal reflections in episodic memory (Shinn et al., 2023). ExpeL extracts reusable knowledge from accumulated experiences and recalls these insights and past experiences for subsequent decisions (Zhao et al., 2024). Dynamic Cheatsheet and ACE distill execution experience into reusable strategies and context updates (Suzgun et al., 2026; Zhang et al., 2026). These methods learn in external context, carrying lessons from past executions into later decisions without updating model parameters.

(II) Candidate optimization. DSPy compiles modular LM programs into optimized pipelines (Khattab et al., 2024), and MIPRO jointly searches instructions and demonstrations for multi-stage

[1] https://developers.openai.com/api/docs/models/gpt-5.5
[2] https://github.com/earendil-works/pi

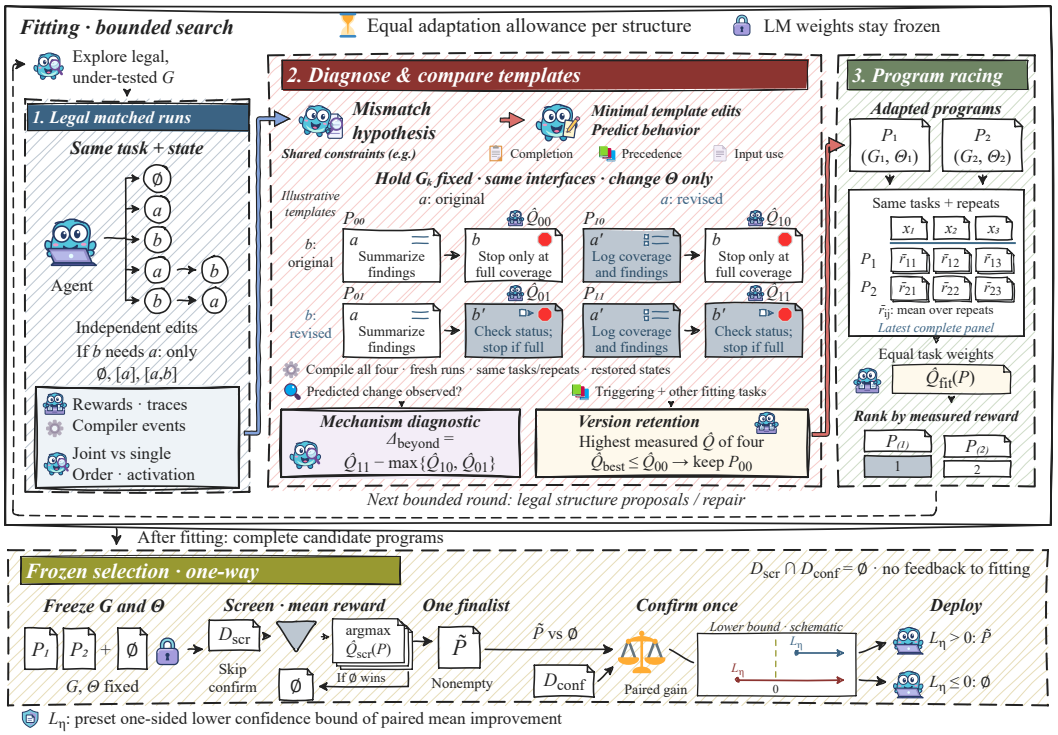


Figure 2: **MICA: matched evidence, co-adaptation, and frozen selection.** Legal matched runs support testable repair hypotheses. Fresh four-condition comparisons under fixed G measure benefit beyond unilateral revision. Retain the highest-reward version only if it improves on P_{00}. Each structure receives equal bounded adaptation allowance before reward-based racing. Freeze G, Θ before disjoint screening and one-time confirmation; $\varnothing$ denotes the unedited program.

programs (Opsahl-Ong et al., 2024). TextGrad propagates textual feedback through computation graphs to improve individual components (Yuksekgonul et al., 2025). GEPA uses reflection to evolve prompts, and SkillOpt uses execution feedback to refine agent skills (Agrawal et al., 2026; Yang et al., 2026). RHO optimizes runtime configurations through retrospective trajectories and self-preference (Pan et al., 2026). At the structure level, AFlow searches code-represented agentic workflows using execution feedback (Zhang et al., 2025). These methods optimize both program content and organization.

(III) Intervention feedback. Studies of intrinsic self-correction find that reasoning revisions without external feedback can fail or reduce accuracy (Huang et al., 2024a). CausalFlow uses step-level counterfactual interventions to attribute failures and construct repairs (Bonagiri et al., 2026), while Causal Agent Replay uses controlled reruns to estimate outcome changes and interactions among execution steps (Shah, 2026). Intervention outcomes provide diagnostic feedback about which execution changes affect task success.

Composition repair asks whether joint template revision improves on either unilateral revision under a fixed structure. MICA uses **matched intervention evidence** for **structure-conditioned template co-adaptation**: aligned executions guide repairs, four-condition comparisons assess joint and unilateral revisions, and measured rewards select template versions.

3 Methodology

3.1 Problem Setup and Overview

MICA enables harness self-evolution through persistent context-editing programs for frozen tool-using agents. Figure 2 summarizes fitting and frozen selection. Each configuration fixes an environment version, base agent, model configuration, and execution framework. Before each action-

generation call, the program edits a copy of the agent's messages for the execution model, preserving environment state, tool implementations, and model parameters.

We represent a program and its optimization objective as

$$P = (G, \Theta), \qquad \max_{\substack{(G,\Theta)\in\mathcal{P}_{\text{legal}} \\ |G|\le m_{\max}}} \mathbb{E}_{x,\omega}[r_{x,\omega}(G,\Theta)]\,, \tag{1}$$

where G specifies edit selection, execution order, input bindings, and dependencies; Θ contains the corresponding context-edit templates; and ω indexes execution randomness. The configuration parameter $m_{\max}$ bounds program length; the composition and repair operators below act on singleton and paired edits. Each template specifies an edit's text, with placeholders instantiated from compiler-validated input bindings. Fitting searches both G and Θ under a finite budget. Every change creates a complete canonical version with its own execution measurements.

Optimization data comprise fitting tasks $\mathcal{D}_{\text{fit}}$ and disjoint screening and confirmation sets, $\mathcal{D}_{\text{scr}}$ and $\mathcal{D}_{\text{conf}}$, grouped by family where defined. Fitting supports generation, diagnosis, adaptation, and racing. Candidate structures and templates are then frozen: screening selects a finalist, and confirmation determines deployment. Development tasks support interface and configuration checks. Frozen evaluation begins after the deployed program, compiler, and model/runtime settings are fixed. Appendix I specifies task pools.

3.2 Parameterized Edits and Executable Dataflow

The frozen generator builds $\mathcal{B}_{\text{init}}$ from fitting-task descriptions, tool specifications, unedited traces, execution status, and rewards. Each primitive declares localization roles and pattern, typed bindings, operation, template, output slot, and dependencies. Bindings read the latest user or tool message, available tool names, non-system trace text, or a predecessor's output. Operations append/prepend system-context text or append explanatory text to a copy of the latest tool result.

For independently executable initialized primitives $\mathcal{B}_{\text{ind}}$, the configuration requires

$$B_{\min} \le |\mathcal{B}_{\text{init}}| \le B_{\max}, \qquad |\mathcal{B}_{\text{ind}}| \ge n_{\text{ind}}. \tag{2}$$

We denote ordered edits by $\langle a, b\rangle$ and the unedited program by $P_\emptyset = \varnothing$. Programs cannot repeat primitives; dependencies, including every referenced slot's producer, must occur in the execution prefix. Each initialized primitive must belong to a legal program within the length limit. Structural proposals vary selection, legal order, typed bindings, or dependencies within the declared interfaces; template proposals preserve placeholder and output contracts. The compiler validates every complete version before execution. Appendix D.1 gives numerical settings.

Before each model call, the compiler clears slots, localizes edits and resolves public inputs from the original snapshot, then edits a copy sequentially. Successful edits publish slots; dependent edits require applied predecessors and available slots. Missing localization, targets, or inputs cause logged skips that propagate to dependents; empty-string outputs remain valid. **Slots store rendered template text**, obtained by direct input substitution without extra agent or language-model calls. The same compiler governs interventions, repair conditions, and deployment (Appendix B).

3.3 Legal Matched Interventions and Diagnostic Evidence

With current templates and bindings fixed, MICA executes both orders for independent edits a, b and the legal prefix for a dependency $a \to b$:

$$\mathcal{S}_{\text{ind}}(a,b) = \{\varnothing, \langle a\rangle, \langle b\rangle, \langle a,b\rangle, \langle b,a\rangle\}, \qquad \mathcal{S}_{\text{dep}}(a,b) = \{\varnothing, \langle a\rangle, \langle a,b\rangle\}. \tag{3}$$

Every branch restores the same verified initial environment state, agent state, tool definitions, and message prefix, including an initial simulator message when present. Later simulator messages follow each branch's interaction. Branch order is randomized within each task/repeat block. A block $\mathcal{S}$ costs $|\mathcal{S}|$ task executions per repeat.

Writing $r_{x,\omega}(P)$ for an observed reward, each complete independent block yields

$$\begin{aligned} I_{x,\omega}(a,b) &= r_{x,\omega}(\langle a,b\rangle) - r_{x,\omega}(\langle a\rangle) - r_{x,\omega}(\langle b\rangle) + r_{x,\omega}(\varnothing), \\ O_{x,\omega}(a,b) &= r_{x,\omega}(\langle a,b\rangle) - r_{x,\omega}(\langle b,a\rangle). \end{aligned} \tag{4}$$

For dependent edits, $\Delta_{x,\omega}(b \mid a) = r_{x,\omega}(\langle a, b\rangle) - r_{x,\omega}(\langle a\rangle)$ gives the increment over the legal prefix. These reward-scale contrasts quantify nonadditivity, order effects, and conditional gains.

The diagnostic record pairs reward contrasts with compiler events, execution trajectories, failure categories, and usage. Early joint termination, action changes under order reversal, and dependent-edit activation ground testable composition-mismatch hypotheses. A semantic-conflict hypothesis requires trace evidence in addition to a negative interaction contrast. Neutral and harmful outcomes remain available for diagnosis and complementary combinations. Appendix B.4 distinguishes infrastructure failures from scored task failures.

3.4 Intervention-guided composition diagnosis and paired template repair

The frozen diagnostic language model returns **(i)** supporting execution evidence with branch and event references, **(ii)** a testable composition-mismatch hypothesis, **(iii)** a minimal template modification, and **(iv)** the observable behavior expected after modification. Execution tests the proposed repair and determines program retention. Appendix C specifies the diagnostic contract.

Repair fixes a compiler-validated two-edit structure G and adapts its templates to shared task constraints, such as completion criteria, instruction precedence, or the use of predecessor text. Let θ_a^0, θ_b^0 be the original templates and θ_a^1, θ_b^1 their proposed revisions. Each repair hypothesis defines the factorial comparison

$$\Theta_{ij} = (\theta_a^i, \theta_b^j), \qquad P_{ij} = (G, \Theta_{ij}), \qquad i, j \in \{0, 1\}. \tag{5}$$

The four conditions compare the original pair, each unilateral revision, and the joint revision. Every condition must preserve the interfaces and pass legality checks. Invalid hybrids undergo revision or rejection within the fixed adaptation allowance. Each complete four-condition set receives fresh executions on common fitting tasks and repeat indices from separately restored states, with a version record for each condition.

Let $\widehat{Q}_{ij}$ be the task-balanced mean reward of P_{ij}. The contrasts $\widehat{Q}_{10} - \widehat{Q}_{00}$ and $\widehat{Q}_{01} - \widehat{Q}_{00}$ isolate unilateral revisions; $\widehat{Q}_{11} - \max\{\widehat{Q}_{10}, \widehat{Q}_{01}\}$ measures the joint revision's advantage over either unilateral revision. MICA retains the highest-reward revised condition when its mean exceeds the original combination on the common panel. Retention depends on reward improvement, while interaction contrasts diagnose complementarity and observable-change checks assess the proposed explanation. Panels include other fitting tasks to measure regressions.

3.5 Coverage exploration, bounded adaptation, and paired execution racing

Coverage and structure proposals. MICA covers every initialized independent edit and declared dependency through legal matched blocks. Dependency coverage requires complete prefix and joint observations on n_{cov} distinct fitting tasks; compiler logs separately measure actual activation. Fitting evidence supports up to K alternative structures per round, including legal order reversals and input-binding changes. The incumbent and alternatives receive the same A-cycle adaptation allowance and generation-call/token caps before comparison. A singleton compares its original and revised template; a two-edit structure uses the four conditions above. Identity has no templates to adapt.

Paired racing. Candidates share a fitting panel $\mathcal{F} \subseteq \mathcal{D}_{\text{fit}}$ and repeat indices:

$$\widehat{Q}_{\text{fit}}(P) = \frac{1}{|\mathcal{F}| R_{\text{fit}}} \sum_{x \in \mathcal{F}} \sum_{\omega=1}^{R_{\text{fit}}} r_{x,\omega}(P). \tag{6}$$

The panel combines the round's triggering tasks with a seeded sample of other fitting tasks. Equal task weights prevent repeated diagnoses from dominating the ranking. Racing uses the latest complete common panel; historical records support diagnosis. Higher-reward programs survive, while a fixed share ϵ of the allocatable post-coverage fitting budget explores under-tested legal combinations. The configuration fixes the round limit L, candidate cap K, adaptation allowance A, repeat count R_{fit}, and exploration share before execution. Generation and reflection calls, all control branches, and repeated executions count toward optimization cost. Budget is reserved for a final common fitting comparison, screening, and confirmation (Appendix D).

Frozen screening and confirmation. When fitting ends, MICA seals the complete structures and templates of at most $C_{\max}$ nonempty candidates plus identity. Screening uses R_{scr} repeats on $\mathcal{D}_{\text{scr}}$ to select a unique finalist $\widetilde{P}$. A nonempty finalist is paired with identity over R_{conf} repeats on the disjoint $\mathcal{D}_{\text{conf}}$. The deployment rule is

$$P^\star = \begin{cases} \widetilde{P}, & \widetilde{P} \neq P_\emptyset \text{ and } L_\eta(\widetilde{P}) > 0, \\ P_\emptyset, & \text{otherwise}, \end{cases} \tag{7}$$

where L_η is the prespecified one-sided lower confidence limit on mean improvement. Confirmation makes a single deployment decision. Screening and confirmation records are isolated from diagnosis and repair, and a failed confirmation retains identity. Appendix E defines tie-breaking and the confidence estimator; Appendix D.1 records the repeat counts and tail probability η.

4 Experiments

We evaluate MICA's benchmark performance, the contributions of matched evidence and paired repair, its efficiency under common optimization budgets, and its robustness across execution frameworks. Our experiments address the following research questions (RQs).

RQ1: How does MICA compare with baselines across benchmarks and language models?
RQ2: How do matched evidence and paired template repair affect MICA's deployed performance?
RQ3: How does MICA compare in reward and optimization cost under shared resource caps?
RQ4: How robust is MICA's performance relative to baselines across execution frameworks?

4.1 Experimental setup

Datasets. We use *Telecom* and *Airline* from τ^2*-bench* 1.0.1 (Barres et al., 2025), *Retail* from the original τ*-bench* (Yao et al., 2025), *ALFWorld* (Shridhar et al., 2021) in native TextWorld (Côté et al., 2018), and *AppWorld Normal* (Trivedi et al., 2024) (data version 0.2.0). We split training tasks approximately 2:1 into fitting and internal-selection pools, keeping families intact. We study mechanisms and efficiency on *Telecom* and *AppWorld Normal*. Appendix I gives task counts, partitions, and grouping rules.

Models and replication. We benchmark `GPT-5.5`[3] and `DeepSeek-V4-Flash` (DeepSeek-AI et al., 2026), using `GPT-5.5` for other studies. We use Pi[4] primarily and DeepSeek Harness[5] for cross-framework comparisons. Each frozen model handles execution, edit generation, diagnosis, and required user simulation. We use a **single optimization seed** per optimizer/configuration for benchmark and cross-framework comparisons. We repeat complete optimization and deployment with **20 independent seeds** per variant/environment for mechanism and efficiency studies. We freeze programs before evaluation. Appendix G details budgets and settings.

Baselines. We compare MICA with Base (the unedited agent) and four optimization baselines across benchmarks and execution frameworks: *(I) Context learning*: ACE (Zhang et al., 2026); and *(II) Candidate optimization*: GEPA (Agrawal et al., 2026), SkillOpt (Yang et al., 2026), and RHO (Pan et al., 2026). We match task splits, tool permissions, and initial framework state across methods, preserving their documented feedback and deployment rules. We assess efficiency using candidate-optimization controls (Failure-Reflection Search and Random-Proposal Racing) and an intervention-feedback control (Sequential-Repair Search); Appendix G.1 defines these controls.

Evaluation and metrics. Following cost-aware agent evaluation (Huang et al., 2024b), we report mean native task rewards (percent) and resource use. Score differences use percentage points; normalized-reward changes retain their original scale. *AppWorld* uses Normal Task Goal Completion (TGC); *Avg.* weights five environments equally. We report **20-seed means** for finalist and deployed scores in mechanism and efficiency studies, computing differences from unrounded counts.

[3] https://developers.openai.com/api/docs/models/gpt-5.5
[4] https://github.com/earendil-works/pi
[5] https://github.com/deepseek-ai/deepseek-harness

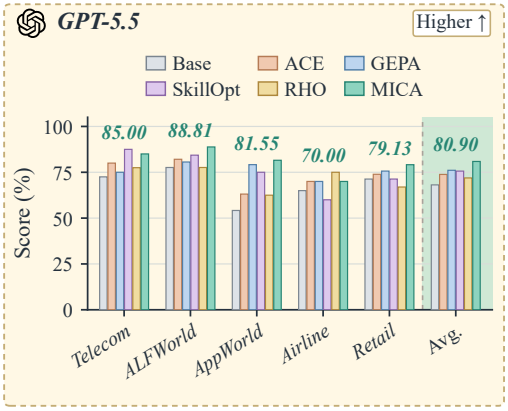


Figure 3: `GPT-5.5` with a single optimization seed. Avg. equally weights the five environments.

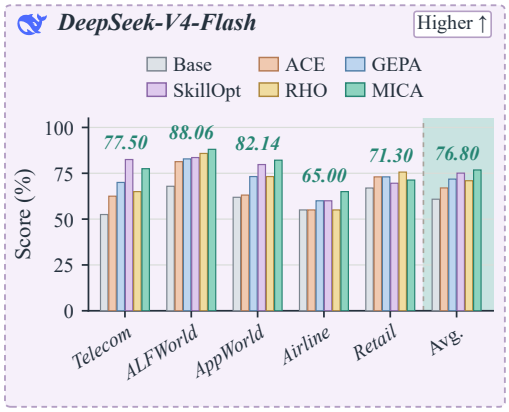


Figure 4: `DeepSeek-V4-Flash` with a single optimization seed. *AppWorld* reports Normal TGC; axes match Figure 3.

Identity retention measures the fraction of runs retaining the unedited program. Behavior checks and regression rates summarize fitting evaluations. Four-condition gains use aggregate fitting means; unilateral maxima follow aggregation. We report efficiency costs as **means per optimization run** and percentages of caps in Figure 7. We account for frozen testing separately. Appendices A and H provide plotted values and score/cost aggregation rules, respectively.

4.2 Main Results (RQ1)

To answer RQ1, we compare MICA with Base and four optimizers across five environments using `GPT-5.5` and `DeepSeek-V4-Flash` under Pi. Figures 3 and 4 report results from one optimization seed per optimizer/configuration; the macro average weights each environment equally.

Obs. 1: MICA leads on the five-environment average with both models. With `GPT-5.5`, it reaches 80.90%, versus 76.08% for GEPA, the strongest baseline overall, a gain of 4.82%↑. With `DeepSeek-V4-Flash`, it reaches 76.80%, exceeding SkillOpt's 75.08% by 1.72%↑. The corresponding gains over Base are 12.78%↑ and 15.95%↑. MICA ranks first in three of five environments with each model, although the leaders vary by environment: SkillOpt leads *Telecom* with both models, while RHO leads *Airline* with `GPT-5.5` and *Retail* with `DeepSeek-V4-Flash`.

4.3 Mechanism Ablation (RQ2)

To answer RQ2, we vary diagnostic evidence and repair strategy using `GPT-5.5` with Pi. Figures 5 and 6 summarize 20 optimization runs per configuration on *Telecom* and *AppWorld Normal*. Appendix F describes the experimental design and evaluation protocol.

Matched Evidence. With paired repair fixed, failure-trace reflection receives failed joint branches and rewards, while matched diagnosis also receives aligned controls. Both variants collect and charge identical legal matched blocks under the same adaptation allowance.

Obs. 2: Matched controls improve repair beyond failure-trace reflection. Matched evidence raises deployed scores from 79.75% to 84.50% on *Telecom* and from 74.52% to 81.07% on *AppWorld*, gains of 4.75%↑ and 6.55%↑. Access to aligned controls thus improves repair under matched collection costs and adaptation allowances.

Paired Repair. With matched evidence fixed, both strategies can revise both templates within the same adaptation allowance, sequentially or jointly.

Obs. 3: Paired repair improves reward and fitting diagnostics over sequential repair. Paired repair raises deployed scores by 2.88%↑ on *Telecom* and 4.46%↑ on *AppWorld*. Behavior-check rates increase from 77.29% to 82.71% and from 75.28% to 82.78%, respectively, while regression rates decrease from 11.88% to 9.38% and from 14.58% to 9.58%. During fitting, these checks assess the proposed explanation separately from the reward-based decision to retain a repair.

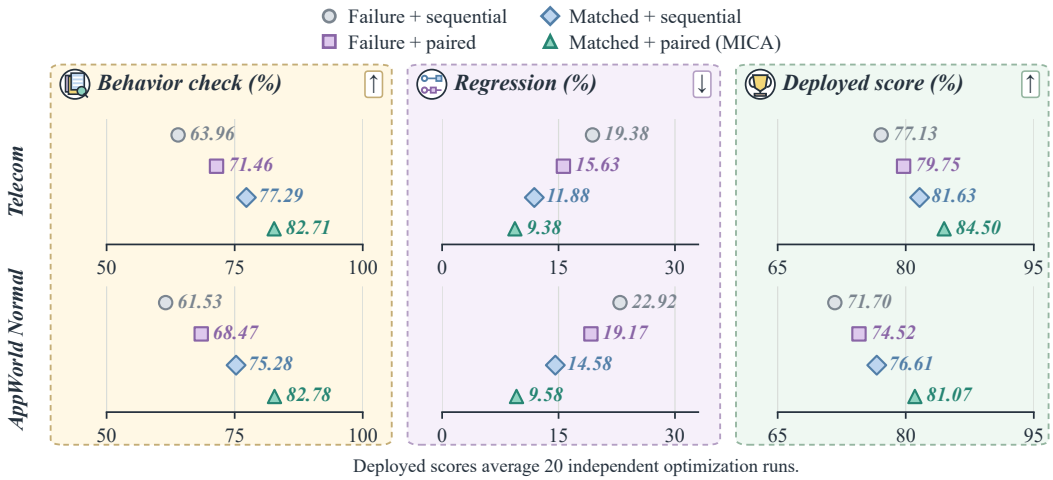


Figure 5: Effects of diagnostic evidence and template repair. Deployed scores are 20-seed means; behavior and regression rates aggregate fitting evaluations. Table 4 gives exact values.

Four-Condition Comparison. We fix G and execute the original pair, each unilateral revision, and the joint revision on common fitting panels. Both edits remain present throughout; only their template versions change. Reported percentage rewards average 20 independent optimization runs per configuration.

Obs. 4: Joint revision yields the highest aggregate fitting reward. In Figure 6, joint revision raises mean reward from 62.71% to 81.67% on *Telecom* and from 50.42% to 79.31% on *AppWorld*. It exceeds the better unilateral condition mean by 10.21%↑ and 14.44%↑, respectively. These aggregate gains support co-adaptation on the evaluated panels. They do not imply that every joint proposal wins; version retention still compares all four measured rewards.

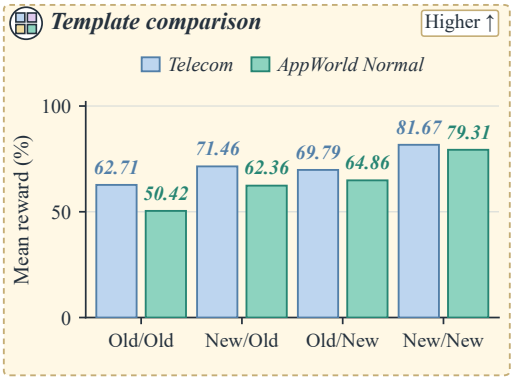


Figure 6: Four-condition fitting rewards. Old/New denote original/revised templates; Table 5 gives repair gains.

4.4 Efficiency Comparison (RQ3)

To answer RQ3, Figure 7 compares complete search procedures over 20 runs per method and environment. Methods share initial interfaces and templates, the compiler, legal modification permissions, bounded adaptation opportunity, selection rules, and per-run resource caps (Appendix G.1).

Deployment Performance. We compare 20-run mean deployed scores, including runs retaining the unedited program, against Sequential-Repair, the strongest search baseline.

Obs. 5: MICA achieves the highest deployed reward under the common resource caps. MICA reaches 84.38% on *Telecom* and 81.22% on *AppWorld*, exceeding Sequential-Repair by 3.25%↑ and 4.23%↑. It retains the unedited program in 2/20 and 1/20 runs, compared with 4/20 and 3/20 for Sequential-Repair. The reward advantage persists through frozen selection and confirmation.

Optimization Cost. Table 6 reports mean per-run costs, including screening and confirmation; frozen testing is accounted for separately.

Obs. 6: MICA achieves higher reward with modest increases in rollout and token use. Relative to Sequential-Repair, MICA uses 0.45% and 0.41% more rollouts on *Telecom* and *AppWorld*, with 0.36% and 0.38% fewer LM calls. Tokens increase from 259.02 M to 263.75 M and from 546.93 M to 556.88 M (1.83% and 1.82%). Although MICA uses the most rollouts and tokens among these methods, its resource use remains close to the strongest baseline.

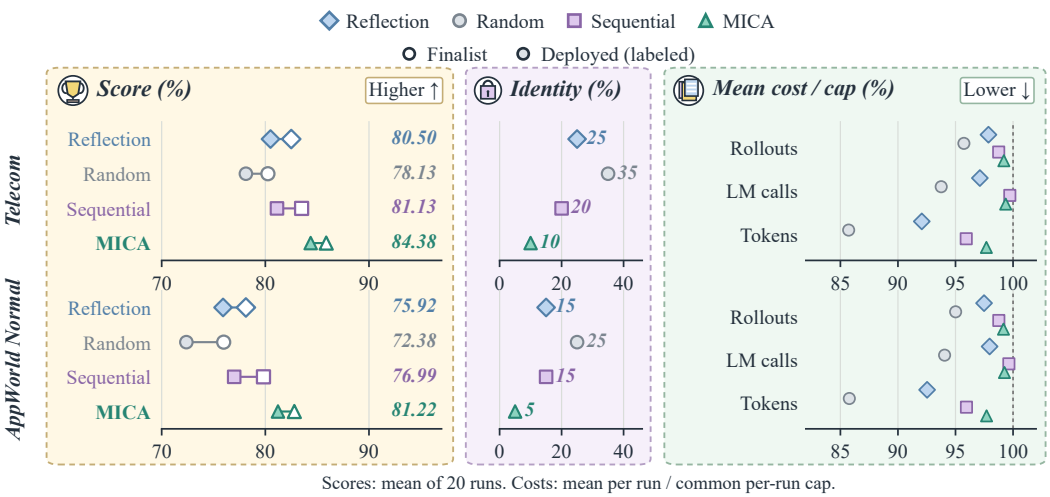


Figure 7: Search performance and optimization cost under shared budgets. Scores are 20-seed means; identity is the fraction of runs retaining the unedited program. Costs show mean per optimization run as a percentage of the common cap (Tables 6 and 13).

4.5 Cross-Framework Robustness (RQ4)

For RQ4, Figure 8 compares `GPT-5.5` under Pi and DeepSeek Harness (DH), with independent optimization and one optimization seed per method–framework–environment configuration.

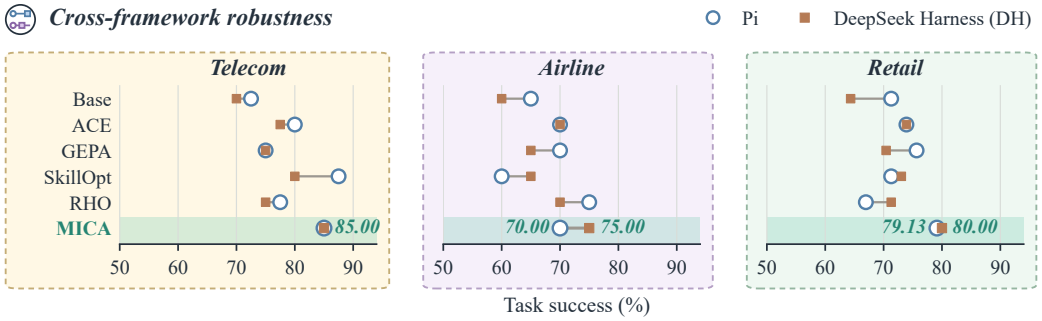


Figure 8: `GPT-5.5` under Pi (open) and DeepSeek Harness (filled), using a single optimization seed per optimizer/framework.

Obs. 7: MICA leads all three DH environments, while rankings vary under Pi. Under DH, MICA reaches 85.00%, 75.00%, and 80.00% on *Telecom*, *Airline*, and *Retail*, exceeding the strongest baselines by 5.00%↑, 5.00%↑, and 6.09%↑, respectively. Under Pi, it leads *Retail* at 79.13%, but SkillOpt leads *Telecom* and RHO leads *Airline*.

Obs. 8: MICA improves over Base in all six evaluated settings. MICA adds 12.50%↑, 5.00%↑, and 7.83%↑ on *Telecom*, *Airline*, and *Retail* under Pi, and 15.00%↑, 15.00%↑, and 15.65%↑ under DH, respectively. This single-seed comparison assesses adaptation within each framework; seed-level stability and frozen-program transfer remain untested.

5 Conclusion

Harness context evolution benefits from **co-adaptation** that accounts for interactions among edits. We introduced MICA to jointly adapt context-edit structures and templates through matched intervention evidence while keeping language models frozen. Experiments show the highest macro-average scores with both models and higher deployed rewards under common budgets, supporting matched evidence and paired repair. Future work will extend co-adaptation to longer edit compositions and test cross-framework transfer of frozen programs.

AI USE STATEMENT

Generative artificial intelligence (AI) tools were used to assist with manuscript writing and language polishing, as well as with the development of research code. AI tools were also used to analyze and organize related literature. They additionally provided inspiration for figure concepts and visual presentation. The figures themselves were drawn manually by the authors. The authors reviewed and revised all AI-assisted text and code and manually checked the reported experimental results.

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*, 2026. URL https://arxiv.org/abs/2507.19457.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-*Bench*: Evaluating conversational agents in a dual-control environment. arXiv preprint arXiv:2506.07982, 2025. URL https://arxiv.org/abs/2506.07982.

Akash Bonagiri, Devang Borkar, Gerard Janno Anderias, Setareh Rafatirad, and Houman Homayoun. CausalFlow: Causal attribution and counterfactual repair for LLM agent failures. arXiv preprint arXiv:2605.25338, 2026. URL https://arxiv.org/abs/2605.25338.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. TextWorld: A learning environment for text-based games. arXiv preprint arXiv:1806.11532, 2018. URL https://arxiv.org/abs/1806.11532.

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. arXiv preprint arXiv:2606.19348, 2026. URL https://arxiv.org/abs/2606.19348.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024a. URL https://arxiv.org/abs/2310.01798.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 20271–20309. PMLR, 2024b. URL https://proceedings.mlr.press/v235/huang24y.html.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/f1cf02ce09757f57c3b93c0db83181e0-Paper-Conference.pdf.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.emnlp-main.525/.

Wenbo Pan, Shujie Liu, Chin-Yew Lin, Jingying Zeng, Xianfeng Tang, Xiangyang Zhou, Yan Lu, and Xiaohua Jia. Evolving agents in the dark: Retrospective harness optimization via self-preference. arXiv preprint arXiv:2606.05922, 2026. URL https://arxiv.org/abs/2606.05922.

Jaineet Shah. Causal agent replay: Counterfactual attribution for LLM-agent failures. arXiv preprint arXiv:2606.08275, 2026. URL https://arxiv.org/abs/2606.08275.

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, 2023. URL https://papers.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. *ALFWorld*: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2010.03768.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.eacl-long.333/.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. *AppWorld*: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.acl-long.850/.

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. URL https://arxiv.org/abs/2605.23904.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ*-bench*: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2406.12045.

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.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5492ecbce4439401798dcd2c90be94cd-Abstract-Conference.html.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2510.04618.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

A EXACT NUMERICAL RESULTS

Tables 1, 2, and 3 give the RQ1/RQ4 values plotted in Figures 3, 4, and 8, at their original reported precision. *AppWorld* benchmark scores report Normal TGC. These RQ1/RQ4 results use a single optimization seed per optimizer/configuration. Tables 4–6 give the 20-seed RQ2/RQ3 results in Figures 5–7; Appendix A.1 retains their underlying counts.

Table 1: Main results with `GPT-5.5` and a single optimization seed per optimizer. Scores are percentages; Avg. is the equal-weight mean of the five environment scores.

Method	*Telecom*	*ALFWorld*	*AppWorld*	*Airline*	*Retail*	Avg.
Base	72.50	77.61	54.17	65.00	71.30	68.12
(I) Context learning						
ACE	80.00	82.09	63.10	70.00	73.91	73.82
(II) Candidate optimization						
GEPA	75.00	80.60	79.17	70.00	75.65	76.08
SkillOpt	**87.50**	84.33	75.00	60.00	71.30	75.63
RHO	77.50	77.61	62.50	**75.00**	66.96	71.91
(III) Intervention feedback						
MICA	85.00	**88.81**	**81.55**	70.00	**79.13**	**80.90**

Table 2: Main results with `DeepSeek-V4-Flash` and a single optimization seed per optimizer. Scores are percentages; Avg. equally weights the five environments.

Method	*Telecom*	*ALFWorld*	*AppWorld*	*Airline*	*Retail*	Avg.
Base	52.50	67.91	61.90	55.00	66.96	60.85
(I) Context learning						
ACE	62.50	81.34	63.10	55.00	73.04	67.00
(II) Candidate optimization						
GEPA	70.00	82.84	73.21	60.00	73.04	71.82
SkillOpt	**82.50**	83.58	79.76	60.00	69.57	75.08
RHO	65.00	85.82	73.21	55.00	**75.65**	70.94
(III) Intervention feedback						
MICA	77.50	**88.06**	**82.14**	**65.00**	71.30	**76.80**

Table 3: `GPT-5.5` under Pi and DeepSeek Harness (DH), using a single optimization seed per optimizer/framework. Entries report environment scores (%).

Method	*Telecom* Pi	*Telecom* DH	*Airline* Pi	*Airline* DH	*Retail* Pi	*Retail* DH
Base	72.50	70.00	65.00	60.00	71.30	64.35
(I) Context learning						
ACE	80.00	77.50	70.00	70.00	73.91	73.91
(II) Candidate optimization						
GEPA	75.00	75.00	70.00	65.00	75.65	70.43
SkillOpt	**87.50**	80.00	60.00	65.00	71.30	73.04
RHO	77.50	75.00	**75.00**	70.00	66.96	71.30
(III) Intervention feedback						
MICA (Full)	85.00	**85.00**	70.00	**75.00**	**79.13**	**80.00**

Table 4: RQ2 mechanism ablation with 20 independent optimization runs (20 optimization seeds) per variant and environment. Deployed score is the 20-seed mean (%); behavior checks and regression rates aggregate fitting evaluations (%). Higher behavior/score and lower regression are better.

Diagnosis evidence	Template repair	Behavior check	Regression rate	Deployed score 20-seed mean
Telecom				
Failure traces	Sequential	63.96	19.38	77.13
Failure traces	Paired	71.46	15.63	79.75
Matched interventions	Sequential	77.29	11.88	81.63
Matched interventions	**Paired (MICA)**	**82.71**	**9.38**	**84.50**
AppWorld Normal				
Failure traces	Sequential	61.53	22.92	71.70
Failure traces	Paired	68.47	19.17	74.52
Matched interventions	Sequential	75.28	14.58	76.61
Matched interventions	**Paired (MICA)**	**82.78**	**9.58**	**81.07**

Table 5: RQ2 four-condition repair diagnostics from 20 independent optimization runs. Structure G is fixed within each comparison. Entries use normalized reward; contrasts are computed from the aggregate condition means before rounding.

Condition or contrast	*Telecom*	*AppWorld Normal*
Original / original: Q_{00}	0.6271	0.5042
New / original: Q_{10}	0.7146	0.6236
Original / new: Q_{01}	0.6979	0.6486
New / new: Q_{11}	**0.8167**	**0.7931**
Overall gain: $Q_{11} - Q_{00}$	0.1896↑	0.2889↑
Gain beyond unilateral repair: $Q_{11} - \max(Q_{10}, Q_{01})$	0.1021↑	0.1444↑

Table 6: RQ3 uses 20 independent optimization runs (20 optimization seeds) per method and environment. Scores are 20-seed means (%); identity retention is the fraction of runs (%). Costs report mean per optimization run under common per-run caps.

Search	Finalist score	Deployed score	Identity retained	Rollouts mean/run	LM calls mean/run	Tokens (M) mean/run
Telecom						
Failure-Reflection	82.50	80.50	25	2,035.55	64,097.15	248.5921
Random-Proposal Racing	80.25	78.13	35	1,991.15	61,880.55	231.4388
Sequential-Repair	83.50	81.13	20	2,054.05	65,814.35	259.0182
MICA	**85.88**	**84.38**	**10**	2,063.35	65,576.95	263.7459
AppWorld Normal						
Failure-Reflection	78.13	75.92	15	2,865.95	68,574.35	527.4382
Random-Proposal Racing	75.98	72.38	25	2,793.35	65,836.65	488.9206
Sequential-Repair	79.82	76.99	15	2,903.65	69,745.85	546.9276
MICA	**82.80**	**81.22**	**5**	2,915.55	69,478.15	556.8842

A.1 UNDERLYING COUNTS FOR THE 20-SEED STUDIES

Tables 7–9 retain the aggregate counts underlying Tables 4–6. Each method/environment has 20 independent optimization runs (20 optimization seeds). Evaluation-score denominators are $20 \times 40 = 800$ on *Telecom* and $20 \times 168 = 3{,}360$ on *AppWorld Normal*. With the same task count per run, the aggregate success fraction equals the mean of the 20 per-seed task-success scores. Diagnostic denominators describe fitting evaluations, not the number of independent optimization runs or unique tasks. Reported percentages and contrasts are rounded only after calculation from these counts.

Table 7: Underlying counts for the RQ2 mechanism ablation. The final column counts frozen-evaluation successes; the first two count behavior checks and regression evaluations.

Diagnosis evidence	Repair	Behavior checks	Regression evaluations	Deployed successes
Telecom				
Failure traces	Sequential	307/480	31/160	617/800
Failure traces	Paired	343/480	25/160	638/800
Matched interventions	Sequential	371/480	19/160	653/800
Matched interventions	Paired (MICA)	397/480	15/160	676/800
AppWorld Normal				
Failure traces	Sequential	443/720	55/240	2,409/3,360
Failure traces	Paired	493/720	46/240	2,504/3,360
Matched interventions	Sequential	542/720	35/240	2,574/3,360
Matched interventions	Paired (MICA)	596/720	23/240	2,724/3,360

Table 8: Underlying counts and differences for fixed-structure four-condition repair. Both unilateral comparisons and the joint comparison use the same denominator within an environment.

Condition or contrast	*Telecom*	*AppWorld Normal*
Q_{00}	301/480	363/720
Q_{10}	343/480	449/720
Q_{01}	335/480	467/720
Q_{11}	392/480	571/720
$Q_{11} - Q_{00}$	91/480↑	208/720↑
$Q_{11} - \max(Q_{10}, Q_{01})$	49/480↑	104/720↑

Table 9: Underlying RQ3 counts across 20 optimization runs. Rollout and LM-call totals divided by 20 give the means in Table 6; token means are reported there directly.

Search	Finalist successes	Deployed successes	Identity runs	Total opt. rollouts	Total LM calls
Telecom					
Failure-Reflection	660/800	644/800	5/20	40,711	1,281,943
Random-Proposal Racing	642/800	625/800	7/20	39,823	1,237,611
Sequential-Repair	668/800	649/800	4/20	41,081	1,316,287
MICA	687/800	675/800	2/20	41,267	1,311,539
AppWorld Normal					
Failure-Reflection	2,625/3,360	2,551/3,360	3/20	57,319	1,371,487
Random-Proposal Racing	2,553/3,360	2,432/3,360	5/20	55,867	1,316,733
Sequential-Repair	2,682/3,360	2,587/3,360	3/20	58,073	1,394,917
MICA	2,782/3,360	2,729/3,360	1/20	58,311	1,389,563

B COMPILER CONTRACT AND BINDING-PRESERVING CANONICALIZATION

B.1 PRIMITIVE INTERFACES AND STATIC VALIDATION

MICA initializes two to four parameterized primitive interfaces and their text templates. A primitive has the record

$$b = (\mathrm{id}, \mathrm{roles}, \mathrm{pattern}, \mathrm{op}, \mathrm{bindings}, \mathrm{template}, \mathrm{out}, \mathrm{deps}).$$

Roles are a subset of system, user, assistant, and tool. The localization pattern is a regular expression evaluated against messages with those roles. The operation is one of `append-system`, `prepend-system`, and `append-tool`. Bindings map template placeholders to typed sources. Every placeholder has exactly one binding; unused bindings are rejected. Each primitive has at most one output slot, which stores its rendered template text.

A program is $P = (G, \Theta)$. Its structure G specifies the selected interfaces, their execution order, input bindings, and dependencies; Θ contains their current templates. The initial interfaces define the allowed operations, source types, and output ports. Fitting can change legal structural choices and template content within this contract. Initialization supplies the interfaces and starting templates for adaptation. Every program contains at most two distinct primitives. The execution model and all language models used for initialization, diagnosis, and repair remain frozen.

The compiler merges canonically identical initialization records and updates their references before checking identifier uniqueness, regular-expression validity, binding types, template placeholders, output ownership, and an acyclic dependency graph. The two-to-four interface count and the requirement for at least two independent primitives apply after this merge. Every slot-consuming binding declares its producer as a predecessor. Every primitive must appear in at least one legal program of length at most two. A dependent primitive therefore has one independent predecessor, and chains requiring three edits are rejected. Structural proposals and each template-repair condition undergo the same checks. Invalid proposals receive structured validation errors within the predeclared model-call allowance.

Table 10: Source descriptors retained in canonical binding records. Selector semantics and placeholder assignments are preserved.

Source type	Canonical descriptor
Latest user	Source tag, original-message selector, and canonical placeholder index.
Latest tool	Source tag, original-message selector, and canonical placeholder index.
Tool names	Source tag and deterministic tool-name ordering.
Non-system trace	Source tag, role filter, chronological message order, and serialization convention.
Predecessor output	Source tag, producer's canonical schema, output-port index, consumer placeholder, and required-edge marker.

B.2 PER-CALL EXECUTION

For each action-generation call, the compiler snapshots the original messages and tool definitions, copies the messages, and clears the slot map. It traverses G in execution order. Localization succeeds when at least one original message with an allowed role matches the pattern. Public bindings resolve from the original snapshot; instructions appended earlier in the call do not become new public inputs.

A primitive is eligible when localization succeeds, its required predecessors have an `applied` event in the current call, all required bindings exist, and its target exists or has the operation's specified creation rule. For system operations, the target is the first system message in the copy; when none exists, a system message is inserted at the beginning. For a tool operation, the target is the latest tool message identified in the original snapshot, mapped to the corresponding message in the copy. A missing tool message skips that edit. Each insertion uses a single newline separator when the target already contains text. Sequential prepends follow their literal semantics: the later prepended text appears before the earlier text.

An eligible primitive renders its template, applies the transformation, emits an `applied` event, and publishes the rendered text in its output slot. Rendering substitutes declared inputs into template

text without an additional model call; slots store that rendered text. Template co-adaptation changes this context text while retaining the interface contract.

Other primitives emit `skip-localization`, `skip-predecessor`, `skip-input`, or `skip-target`; no output is published. A produced empty string is present and available to a consumer. A missing slot is absent and triggers a skip. All writes affect message copies; the environment's actual tool return remains unchanged. Slots are cleared before the next language-model call.

Events record the full program-version identifier, primitive digest, original target index, localization status, resolved source types, required-slot presence, application status, and output length. Instantiated text is retained in access-controlled trace records. These events support application-rate and dependency-activation comparisons alongside task rewards.

B.3 Canonical executable schema and version registry

Canonicalization removes naming variation while preserving executable distinctions. Canonical JSON uses UTF-8 with deterministic escaping and preserves exact code points, newlines, and whitespace inside templates and regular expressions. JSON keys and role sets are sorted; execution order, message chronology, and template token order are preserved. Public source descriptors use the typed forms in Table 10.

Template placeholders are alpha-renamed by first occurrence in the parsed template, retaining their binding descriptors. Primitive identifiers are replaced by recursively canonicalized producer schemas, and output-slot names by canonical output-port indices. A dependency without a consumed slot retains its required-predecessor edge. The acyclic graph makes this construction well defined. The complete program serialization contains G, every template in Θ, and all directed producer-port-to-consumer-placeholder edges.

Changing a user-source binding to a tool-source binding, swapping placeholder sources, reversing execution order, or modifying template text changes the executable record. Consistently renaming identifiers and slots leaves it unchanged. A content digest identifies each complete version. The registry records its parent versions, proposed change, fitting-only evidence references, compilation result, and execution records. Rewards belong to the exact executed version and task/repeat identifier. A new template or structure never inherits its parent's reward. Canonically identical proposals can be deduplicated; observations from distinct executable versions remain distinct.

B.4 Execution status and missing observations

A logical execution is one task rollout under a specified complete program version, initial snapshot, and repeat identifier. The environment's scoring policy determines rewards for task failures, invalid actions, and agent-side time or action-budget exhaustion. These are scored outcomes. A service outage, transport failure, or corrupted restore is an infrastructure failure and receives no invented task reward.

An infrastructure-failed logical execution receives at most two retries from its original snapshot. The first valid completion supplies the reward; successful retries are not additional samples. All attempts, calls, tokens, and time enter the operational-cost ledger. An intervention contrast or four-condition repair comparison requires all constituent rewards on its prescribed task/repeat panel. Incomplete comparisons cannot support a repair-acceptance or racing decision. A required dependency is covered only after its prefix and joint rewards are available on two distinct fitting tasks.

Screening and confirmation require complete prescribed evaluation blocks. An unresolved infrastructure failure does not eliminate only the affected candidate: the optimization run is recorded as incomplete and retains identity. Runs without a valid initialization, required coverage, or complete selection records use the same fallback. Repeated-run studies include every predeclared run and the program it retains. Frozen-test infrastructure failures follow the same retry rule; unresolved outcomes retain their place in the evaluation denominator.

C INTERVENTION-GUIDED DIAGNOSIS AND PAIRED TEMPLATE REPAIR

C.1 MATCHED EVIDENCE AND DIAGNOSTIC OUTPUT

For independent edits, the legal block in Equation (3) executes identity, both singletons, and both orders. For a dependent edit, the same equation defines identity, the legal prefix, and the joint program. Each branch restores the same verified environment state, agent state, tool definitions, and message prefix, including an initial simulator message when present. Branch order is randomized within task/repeat blocks. Subsequent simulated messages follow the branch's interaction.

The diagnostic record aligns branch rewards, compiler events, traces, failure categories, and usage by task, repeat, initial-snapshot digest, and full program version. The frozen diagnostic language model also receives the public task constraints, tool specifications, complete candidate schemas, and current templates. Its output has the four fields in Table 11. The prompt requests evidence-linked hypotheses and executable text changes; it requests neither numerical reward forecasts nor a winning-program decision.

Table 11: Structured output of the fitting-stage diagnostic model. Every proposed repair is tested by execution.

Field	Required content
Execution evidence	Branch and trace/event references supporting the comparison, with the relevant reward difference.
Mismatch hypothesis	A testable account of a shared task constraint violated by the templates, their order, or bindings; an empty hypothesis when the records supply no specific account.
Minimal edit	A legal structural proposal or a template pair for a fixed validated structure, with the exact changed text and preserved interfaces.
Expected observable change	A trace or compiler-event change that would support the hypothesis, together with an outcome that would contradict it.

Reward contrasts identify comparisons to inspect. For example, a lower joint reward accompanied by an early stop motivates a hypothesis about incompatible stopping instructions; an order-dependent action sequence motivates an ordering hypothesis; a skipped consumer motivates a binding or activation hypothesis. Negative reward interaction measures subadditivity on the reward scale, whose sign also reflects reward floors and ceilings. Semantic-conflict hypotheses require supporting trace evidence. Trace differences provide additional observable evidence, while subsequent interventions test the proposed explanation. A singleton with zero or negative gain remains eligible for joint exploration because its contribution can depend on another edit.

C.2 FIXED-STRUCTURE FOUR-CONDITION REPAIR

Let G be a compiler-validated two-edit structure and let its current templates be (θ_a^0, θ_b^0). The model proposes (θ_a^1, θ_b^1) around the shared task constraints. MICA registers and executes

$$\Theta_{ij} = (\theta_a^i, \theta_b^j), \qquad P_{ij} = (G, \Theta_{ij}), \qquad i, j \in \{0, 1\}. \tag{8}$$

All four conditions preserve G, placeholder bindings, output ports, and dependency requirements. Each condition must compile, including the two mixed old/new pairs. A proposal requiring a structural change is evaluated as a separate structure with its own matched adaptation opportunity; it is not inserted into this fixed-G comparison.

The four conditions use fresh executions on the same fitting tasks and repeat identifiers, with separate restores and randomized branch order. The panel includes fitting tasks outside the failures that motivated the repair and follows the common-panel schedule in Appendix D. This measures gains and regressions under the same evaluation coverage. For a common panel $\mathcal{X}_t$, define the task-balanced empirical reward

$$\widehat{Q}_t(P) = \frac{1}{|\mathcal{X}_t|} \sum_{x \in \mathcal{X}_t} \frac{1}{R_{\text{fit}}} \sum_{r=1}^{R_{\text{fit}}} r_{x,r}(P). \tag{9}$$

Each task has equal weight, and repeats are averaged within task. Only the completed common task/repeat panel enters this comparison; additional historical executions of a triggering failure do not increase its weight. The measured repair gain is

$$\widehat{\Delta}_{ij} = \widehat{Q}_t(P_{ij}) - \widehat{Q}_t(P_{00}).$$

MICA retains the best measured version only when its gain over P_{00} is positive; ties retain the original. Comparisons of P_{11} with P_{10} and P_{01} measure the additional observed benefit of adapting both templates beyond either one-sided change. The four-condition interaction is descriptive and is not an acceptance threshold. A repair can improve actual joint reward with a nonpositive interaction, while a positive interaction can accompany a worse joint program.

Execution outcomes and the expected observable changes are recorded separately. A reward gain without the proposed behavioral change weakens the stated explanation even when the version survives fitting. These adaptive comparisons guide search; independent screening and confirmation determine deployment.

D Coverage exploration, bounded adaptation, and paired execution racing

D.1 Numerical configuration of the method

The symbols in Section 3 separate the method definition from its numerical configuration. The primitive-library bounds are $B_{\min} = 2$ and $B_{\max} = 4$, with at least $n_{\text{ind}} = 2$ independently executable primitives after canonical deduplication. The maximum program length is $m_{\max} = 2$: MICA's composition and repair operators use singleton and paired edits. Each dependency requires prefix and joint rewards on $n_{\text{cov}} = 2$ distinct fitting tasks. An independent intervention block contains five branches and a dependent block contains three, so their costs are five and three task executions per repeat, respectively.

The controlled studies initialize four primitives, comprising three independent primitives and one dependent primitive. They use $L = 3$ search rounds, $K = 3$ alternative structures plus the incumbent per round, $A = 1$ adaptation cycle per nonempty structure per round, $R_{\text{fit}} = 2$ fitting repeats, and exploration share $\epsilon = 0.20$. The final common fitting comparison and screening each admit up to $C_{\max} = 3$ nonempty candidates plus identity. Screening uses $R_{\text{scr}} = 2$ repeats; confirmation uses $R_{\text{conf}} = 5$ repeats and one-sided tail probability $\eta = 0.05$. Table 12 gives panel sizes and proposal-resource caps, and Table 15 gives per-stage resource limits. The zero threshold in Equation (7) expresses a positive lower confidence limit on improvement over identity.

D.2 Coverage units and exploration reserve

The coverage universe contains every initialized primitive, valid prefix/joint contrasts on two distinct fitting tasks for each dependency edge, and a reversible independent block. Independent blocks are indexed by unordered pairs because both directions execute together. Additional independent blocks cover primitives absent from the first reversible block. Warm-up uses deterministic greedy set cover, choosing the block with the most newly covered requirements per logical execution and breaking ties by the predeclared seed.

The common repair panel contains 12 *Telecom* tasks: four triggering or accompanying tasks and eight other fitting tasks. *AppWorld* uses 18 tasks from six complete scenarios: six triggering or accompanying tasks from two scenarios and 12 other tasks from four scenarios. The other-task sample follows a seeded order, with complete-scenario grouping in *AppWorld*, cycling after the fitting pool has been visited. Triggering tasks can recur in later rounds with fresh repeat identifiers. Every candidate receives the same panel and repeat coverage. Compiler events distinguish evaluated legal dependencies from producer/consumer pairs that actually apply.

After mandatory coverage, a fixed fraction ϵ of the allocatable post-coverage fitting-execution allowance is reserved for insufficiently tested legal combinations. Screening and confirmation allowances are reserved separately. Exploration chooses the least-exposed affordable combination, with seeded tie-breaking, and executes its complete legal block. Its accounting includes any singleton or identity controls. Neutral singleton rewards do not remove an edit from this pool. Only

complete legal blocks are executed; any allowance too small for the next block remains unused. Exploration therefore preserves opportunities for complementarity outside the current leading structures.

D.3 Equal adaptation opportunities and empirical racing

At each round, the diagnostic model can propose at most K alternative structures from fitting evidence, such as a legal order reversal, edit substitution, or input rebinding. The incumbent and these alternatives each receive the same bounded template-adaptation opportunity before elimination. The default is one adaptation cycle per nonempty structure per round. For two-edit programs, a cycle requests one paired template proposal and evaluates its four conditions. A one-edit candidate uses the same model-call and token cap and compares its original and revised template on the shared panel. Identity has no template and receives no repair budget.

Candidates use the same adaptation-cycle cap, frozen-model call/token cap, and declared execution allowance; the round starts only when the common evaluation allocation is feasible for every admitted candidate. A structure with no evidence-supported repair keeps its current templates. Validation feedback and retries consume the same predeclared allowance; invalid or unproductive proposals do not enlarge it. Baselines that generate candidates receive the same structural and template-edit permissions. A sequential-repair baseline may visit both templates within the same adaptation cycle and total model-call/token cap, with each changed intermediate version separately evaluated. This compares repair strategies under matched editing rights and resource limits.

After adaptation, candidates race on a common fitting-task panel with common repeat identifiers and separate initial-state restores. Equation (9) ranks their measured rewards using only that completed common panel. Ties prefer shorter programs and then canonical serialization. The best version becomes the next incumbent; evaluated alternatives remain in the version archive and can receive subsequent reserved exploration. Each round's comparison panel covers tasks beyond the repair-triggering failures. The pre-run configuration fixes the panel-construction rule, seed, size bound, and overlap policy; actual task identities are recorded before the new versions are evaluated. Reusing a task in a later round requires fresh execution records, and the final fitting comparison evaluates at most three nonempty versions plus identity on 12 *Telecom* or 18 *AppWorld* tasks, with two repeats per task/version.

D.4 Budget feasibility and execution accounting

Table 12 gives the finite-search limits, panel sizes, and per-cycle call allowances. Each cycle permits one diagnostic call and up to three repair or validation-feedback calls. Their combined input/output token allowance is 131,072 for *Telecom* and 196,608 for *AppWorld*. Table 15 allocates the RQ3 rollout, LM-call, and token caps across initialization, fitting, screening, and confirmation. All calls and control branches consume these stage allowances.

The logical execution budget is

$$B_{\text{opt}} = B_{\text{trace}} + B_{\text{fit}} + B_{\text{scr}} + B_{\text{conf}}. \tag{10}$$

Generation-evidence rollouts enter B_{trace}. Coverage, reserved exploration, repair controls, template variants, and candidate racing enter B_{fit}. One independent intervention block uses five executions per task/repeat and one dependent block uses three. A four-condition repair uses $4|\mathcal{X}_t|R_{\text{fit}}$ executions, including a fresh P_{00} control. A new version starts without observed rewards, even when its parent has already been tested. The predeclared final comparison requires every shortlisted version to have observations on the same panel.

The final common fitting comparison costs at most 96 *Telecom* or 144 *AppWorld* rollouts, charged to fitting. For C frozen screening candidates, screening requires $Cn_{\text{scr}}R_{\text{scr}}$ executions. Confirmation of one nonempty finalist requires $2n_{\text{conf}}R_{\text{conf}}$, including identity. With four candidates, screening requires 72 *Telecom* or 96 *AppWorld* rollouts; confirmation requires 150 or 180, respectively. Screening identity as the winner skips confirmation and logs its unused reserve; this budget is not redirected after seeing selection outcomes. No intervention, repair comparison, or common racing block is partly initiated because its remaining branches are unaffordable.

Table 12: Controlled-study search settings. Identity denotes the unedited program. Per-cycle call and token limits cover diagnosis, repair, and validation feedback; environment-execution calls also count toward stage budgets.

Setting	*Telecom*	*AppWorld Normal*
Maximum edits per program	2	2
Initialized primitives	4	4
Initial composition	3 independent, 1 dependent	3 independent, 1 dependent
Search rounds L	3	3
Alternative structures per round K	3 plus incumbent	3 plus incumbent
Adaptation cycles A	1 per nonempty structure per round	1 per nonempty structure per round
Fitting repeats R_{fit}	2 per task/version	2 per task/version
Repair-panel tasks	12	18
Triggering and accompanying tasks	4	6 (2 scenarios)
Other fitting tasks	8	12 (4 scenarios)
Post-coverage exploration ϵ	0.20	0.20
Final fitting-comparison candidates	3 nonempty + identity	3 nonempty + identity
Final fitting-panel tasks	12	18
Screening candidates	3 nonempty + identity	3 nonempty + identity
Screening tasks	9	12
Confirmation tasks	15	18
Screening repeats R_{scr}	2	2
Confirmation repeats R_{conf}	5	5
One-sided tail probability η	0.05	0.05
Bootstrap draws	10,000	10,000
Diagnosis calls per cycle	At most 1	At most 1
Repair/validation-feedback calls per cycle	At most 3	At most 3
Combined calls per cycle	At most 4	At most 4
Combined input/output tokens per cycle	At most 131,072	At most 196,608

The optimization cost also includes every initialization, diagnostic, repair, and validation-feedback language-model call, with tokens and wall-clock time. These calls receive explicit caps alongside rollout allowances; reporting rollout count alone does not establish equal optimization cost. All physical retries, control branches, repeats, and discarded proposals remain in the ledger. Frozen testing and mechanism diagnostics performed after artifact freezing have separate reported budgets and supply no optimizer feedback.

E Independent screening and single-use confirmation

All adaptive changes end when the fitting budget is exhausted. MICA seals the complete structure and templates of each candidate, the version registry, shortlist, and remaining protocol settings. The shortlist contains identity and up to three nonempty versions with the highest task-balanced rewards in the final common fitting race, after canonical deduplication. No language model predicts their numerical scores or decides their final selection.

Screening and confirmation use disjoint subsets of the internal-selection pool: nine and 15 tasks for *Telecom*, and 12 and 18 tasks from four and six complete scenarios for *AppWorld*. *Telecom* uses tasks as its allocation and resampling units; *AppWorld* preserves scenario groups across stages. Development and frozen-evaluation tasks are excluded from selection. Screening and confirmation rewards, trajectories, and compiler events are unavailable to diagnosis and repair.

E.1 Screening one finalist

Every candidate executes on the same screening tasks and repeat identifiers, with separate restores and randomized candidate order within each task/repeat block. Its score is

$$\widehat{Q}_{\text{scr}}(P) = \frac{1}{n_{\text{scr}}} \sum_{x \in \mathcal{D}_{\text{scr}}} \frac{1}{R_{\text{scr}}} \sum_{r=1}^{R_{\text{scr}}} r_{x,r}(P).$$

MICA selects the unique finalist $\widetilde{P}$ with the highest measured score; ties prefer shorter length and then canonical serialization. The default is $R_{\text{scr}} = 2$. Screening changes neither the candidates nor their templates. If identity wins, it is retained immediately.

E.2 One deployment decision

A nonempty finalist is compared with identity on the disjoint confirmation set. For each confirmation task,

$$D_x = \frac{1}{R_{\text{conf}}} \sum_{r=1}^{R_{\text{conf}}} \left[r_{x,r}(\widetilde{P}) - r_{x,r}(\varnothing) \right].$$

The task-balanced mean improvement is $n_{\text{conf}}^{-1} \sum_x D_x$. The default uses $R_{\text{conf}} = 5$, $\eta = 0.05$, and 10,000 paired hierarchical bootstrap draws. For *AppWorld*, each draw resamples scenarios and then tasks within each sampled scenario; *Telecom* resamples tasks directly. Both retain paired finalist/control repeats together. The bootstrap statistic uses the same task-balanced mean as the observed estimate. The lower η-quantile is $L_\eta(\widetilde{P})$, the predeclared one-sided confidence lower limit at the recorded family/task clustering level.

The deployment rule is Equation (7): deploy $\widetilde{P}$ only when $L_\eta(\widetilde{P}) > 0$; otherwise retain identity. The finalist, confirmation tasks, repeat count, estimator, and tail probability are fixed before the single confirmation look. A failed confirmation does not trigger another finalist, another template revision, or a renewed search using the same confirmation records. Incomplete required evaluations follow Appendix B.4. Thus fitting supplies exploration and repair, screening selects one frozen candidate, and confirmation makes one deployment decision.

F Mechanism-isolating experiments

F.1 Shared design and evidence access

The RQ2 suite uses `GPT-5.5` and Pi on *Telecom* and *AppWorld Normal*, with 20 independent optimization runs (20 optimization seeds) per variant and environment. Every variant shares the initial primitive interfaces and templates, compiler, frozen execution and generation models, fitting-task panels, and matched optimization-cost allowances. The two-edit limit applies throughout. Table 4 reports mean deployed scores across the 20 runs and the supplied behavioral and regression rates. Table 5 reports the aggregate four-condition fitting diagnostics.

The controlled evidence-access comparison collects the same legal matched blocks for every variant and charges their full cost to each run. Independent pairs use $\{\varnothing, \langle a\rangle, \langle b\rangle, \langle a, b\rangle, \langle b, a\rangle\}$; dependent pairs use $\{\varnothing, \langle a\rangle, \langle a, b\rangle\}$. Each branch restores the same verified initial state and uses the common task and repeat index. The diagnosis input includes program and template version identifiers, observed rewards, compiler events, execution traces, and failure categories. The failure-trace condition exposes only failed joint branches and their rewards to the repair model. The matched condition exposes the aligned controls as well. Both conditions use the same evidence-selection rule and input-token cap, with any truncation recorded. The hidden control records remain available to the experiment's execution-based evaluator but never enter the failure-trace reflector's prompt. This comparison isolates the value of matched evidence while holding its collection cost constant.

The frozen diagnosis model must return supporting execution evidence, a composition-mismatch hypothesis, a minimal template modification, and a predicted observable change. Evidence is anchored to recorded events, such as termination before a required tool call, an order-dependent change in tool arguments, or a dependent edit skipped because its producer never applied. Reward contrasts identify differences to investigate. A negative interaction contrast alone does not identify a semantic conflict, and neutral singleton reward does not remove an edit from the composition pool. The model neither predicts numerical reward nor chooses the surviving program.

F.2 Paired and sequential adaptation

The repair comparison fixes a compiler-validated structure G before modifying text. Both procedures have permission to modify both templates, preserve the same named inputs, outputs, and

dependencies, and use the same common constraints and diagnostic evidence. Paired repair proposes the two new templates jointly. Sequential repair first proposes a change to one template while keeping its companion fixed, evaluates that version, and then proposes a change to the other. The first template is selected by a preregistered alternating order across optimization seeds. One sequential adaptation cycle therefore permits both templates to be visited. Both procedures receive the same per-cycle frozen-model call and token caps and the same execution cap. The protocol requires their consumption to be recorded separately. A cap that cannot fund both visits and their required evaluations is rejected during preflight, before rewards are observed.

Every proposed text version has a new immutable identifier and receives its own executions; no new version inherits a parent's reward. Intermediate sequential versions count toward the budget. The compiler rejects interface changes before execution. Output slots store only rendered template text, and adaptation does not add calls that compute or publish agent-derived slot values during deployment. The structure, template identifiers, prompt, and model settings for every proposal are recorded.

For each explicit paired-repair hypothesis, the evaluator runs all four versions P_{00}, P_{10}, P_{01}, and P_{11} under the same G. All four pass legality and interface checks, including dependent programs; no standalone dependent edit is introduced. The comparison uses common fitting tasks and repeat indices, with randomized branch order and independent state restoration. Failed compilation consumes proposal resources and is logged separately from a behavioral failure. An invalid four-condition set cannot supply a completed repair contrast.

F.3 Reward and behavioral endpoints

Let $\mathcal{H}$ be a fitting-task panel fixed before evaluating a proposed repair, and let $R_{\text{fit}} = 2$. For $P_{ij} = (G, (\theta_a^i, \theta_b^j))$, its task-balanced panel mean is

$$Q_{ij}^{\mathcal{H}} = \frac{1}{|\mathcal{H}|} \sum_{x \in \mathcal{H}} \frac{1}{R_{\text{fit}}} \sum_{r=1}^{R_{\text{fit}}} r_{x,r}(P_{ij}). \tag{11}$$

Repeated rollouts contribute to each task's mean before tasks are averaged. When several pair hypotheses occur on a task, their metrics are first averaged within that task so that tasks with more diagnoses receive no additional weight. In Table 5, Q_{ij} denotes the reported aggregate condition mean. Its fitting-diagnostic denominators are 480 for *Telecom* and 720 for *AppWorld Normal*. These are diagnostic aggregation counts, not optimization-seed counts or counts of unique tasks. The reported contrasts are computed from the unrounded aggregate condition means:

$$\Delta_{\text{repair}} = Q_{11} - Q_{00}, \tag{12}$$

$$\Delta_{\text{beyond}} = Q_{11} - \max\{Q_{10}, Q_{01}\}, \tag{13}$$

$$C_{\text{repair}} = Q_{11} - Q_{10} - Q_{01} + Q_{00}. \tag{14}$$

Δ_{repair} measures overall reward improvement over the original combination. Δ_{beyond} compares the aggregate joint mean with the larger aggregate unilateral mean; it is not an average of per-seed or per-repair maxima. C_{repair} measures nonadditivity of the template changes on the reward scale and is distinct from the interaction between the original edit primitives. These adaptive fitting diagnostics describe the evaluated repairs; frozen evaluation measures final performance.

Before observing a proposed version's executions, the diagnosis record specifies an observable behavioral predicate and how it will be checked from the trace. Examples include completing all required retrieval pages before termination, using a designated tool argument after changing order, and applying a consumer after its required producer. Behavioral-check rate is the task-balanced frequency with which that predicate holds. Predicates are verified from public execution events or a fixed rubric; the repair model's own assertion is not a verified behavioral outcome. The reported behavior-check denominators in Table 4 are 480 fitting observations for *Telecom* and 720 for *AppWorld Normal*; regression uses 160 and 240 other-task comparisons, respectively. These diagnostic counts are separate from the 20 optimization runs and do not identify unique-task counts. Appendix A.1 records the corresponding numerators.

Every candidate in a comparison uses the same fitting panel. A seeded schedule selects triggering and accompanying tasks and other fitting tasks within the fixed counts in Table 12, before the proposed versions' outcomes are observed. All candidates receive the same number of repeats on every

panel task. Regression rate on other tasks is the fraction whose repeated mean reward under the proposed program is lower than under the original combination. This finite-repeat indicator describes observed regressions. Table 4 reports the aggregate behavioral and regression rates. Acceptance during fitting depends on observed overall reward improvement under the declared comparison rule, including the other-task panel, rather than the sign of C_{repair}. Screening and confirmation are excluded from all repair panels.

G SEARCH AND VERIFICATION EXPERIMENTS

Optimization budgets. MICA initializes two to four primitives with at most two edits per program. Controlled studies initialize four primitives (three independent and one dependent) and use three search rounds, at most three alternative structures plus the incumbent per round, one adaptation cycle per structure, two fitting repeats, and an exploration share of 0.2. Common fitting panels cover triggering failures and other tasks. Screening uses two repeats; one-shot confirmation uses five repeats and $\eta = 0.05$, with both stages isolated from repair. RQ3 caps each run at 2,080 rollouts, 66,000 LM calls, and 270 M tokens on *Telecom*, and 2,940 rollouts, 70,000 calls, and 570 M tokens on *AppWorld Normal*. Generation, reflection, control branches, intermediate versions, repeats, screening, and confirmation all count toward optimization cost. Appendices D–E specify allocation and selection.

G.1 BUDGET-MATCHED SEARCH BASELINES

The RQ3 suite compares complete optimization procedures over 20 independent optimization runs per method and environment. All methods share initialized interfaces and templates, fitting and selection splits, frozen model configurations, compiler, maximum program length, and the per-run resource caps in Table 13. Every candidate-generating baseline can change template content and propose the same legal edit selections, orders, bindings, and dependencies as MICA. Their proposal prompts state these permissions explicitly. The screening and confirmation rules are shared. None receives test rewards during optimization.

Table 13: Common optimization caps per run for every RQ3 method. Each method has 20 independent optimization runs in each environment. Tokens include all optimization language-model calls.

Environment	Rollouts	LM calls	Tokens (M)
Telecom	2,080	66,000	270
AppWorld Normal	2,940	70,000	570

Failure-Reflection Search. A frozen language model receives failed fitting executions, their rewards, and the current complete program, and proposes structural changes or paired template revisions. This candidate-optimization control adapts reflective prompt optimization (Agrawal et al., 2026) to the shared permissions and compiler of this experiment. Its repair prompts omit matched unedited, singleton, reverse-order, and prefix comparisons. Proposed versions are retained by the shared execution race. In the end-to-end comparison, resources saved by its narrower evidence collection are allocated to additional permitted proposals or repeated fitting evaluations according to a schedule fixed before the run. All resulting executions and calls count toward the common cap.

Random-Proposal Racing. Sample legal structural alternatives from a seeded proposal order, and ask the same frozen generator for bounded template revisions using task constraints and current templates without intervention-based diagnosis. Each structure receives the same adaptation allowance before the shared measured-reward race. This candidate-optimization control samples proposals across the full structure and template search space.

Sequential-Repair Search. This intervention-feedback control uses MICA's matched evidence, coverage schedule, structural permissions, and execution race, while adapting one template at a time using the cycle in Appendix F. Both templates can change in one cycle. Intermediate proposals, their evaluations, and all reflection calls count toward the shared cap. The controlled RQ2 comparison isolates repair behavior under shared records; the end-to-end RQ3 comparison measures the resulting deployment reward under the total allowance.

MICA. Use matched intervention evidence to propose a small set of legal structures and a paired template repair for each admitted structure. Equal bounded adaptation precedes comparisons between structures. Actual task-balanced paired execution rewards determine retention. Exploration retains access to insufficiently tested combinations. The language model supplies proposals and observable hypotheses, with no numerical reward-prediction role.

All baselines use common optimization seeds and task schedules where their candidate sets permit pairing. The full initial evidence is charged to methods that receive it. Resource accounting includes generation, reflection, all control branches, all intermediate and four-condition evaluations, repeats, screening, and confirmation. Table 6 reports finalist and deployed scores averaged across 20 runs, mean optimization costs per run, and the percentage of runs retaining identity. The RQ3 protocol requires each run to remain within all three caps in Table 13 and starts a complete comparison only when its remaining resource allowance is feasible.

G.2 Coverage, bounded adaptation, and paired racing

The schedule is fixed before fitting rewards are observed. The design has $L = 3$ search rounds, at most $K = 3$ alternative structures plus the incumbent per round, $A = 1$ adaptation cycle per structure per round, and $R_{\text{fit}} = 2$ fitting repeats. A fixed share $\epsilon = 0.20$ of the allocatable post-coverage fitting budget is reserved for exploration of insufficiently tested legal combinations; coverage warm-up is excluded from this denominator. RQ2 and RQ3 use 20 optimization seeds per method and environment. Table 13 specifies the common per-run caps for RQ3. Tables 12, 14, and 15 specify the panel sizes, task allocations, and resource limits.

Coverage first evaluates the required independently executable edits and dependency edges through complete legal matched blocks. Coverage includes neutral and harmful singleton responses, preserving their eligibility for complementary pairs. After coverage, the frozen model proposes at most three alternatives by changes such as legal order reversal or compatible binding reassignment. Each admitted structure, including the incumbent, receives the same bounded opportunity for template adaptation before it can be eliminated by a structural comparison. Unused proposal opportunities and compiler rejections are logged. If the remaining budget cannot fund equal adaptation and a complete common evaluation block, the new comparison is not started.

Each round evaluates active versions on one shared fitting panel with identical task identities, repeat indices, and repeat counts. Retention uses the task-balanced means from the latest complete common panel, with a preregistered deterministic tie order. Historical logs support diagnosis; unequal exposure across earlier rounds does not enter the ranking. Versions without complete common-panel coverage are evaluated before they can be compared. The best version becomes the next incumbent; other evaluated versions remain archived and eligible for reserved exploration. Identity remains available as the deployment fallback. The fixed exploration share covers least-tested legal combinations by a seeded tie order and is not reassigned solely because a component performed poorly alone. A final common panel compares at most three nonempty versions plus identity before screening. Budget for this panel is reserved in advance. Duplicate complete versions are removed, and each structure-template pair remains individually identified.

G.3 Frozen selection and one-shot confirmation

At fitting-budget exhaustion, serialize and freeze each candidate's full structure, template text, and compiler configuration. Screening evaluates those versions with $R_{\text{scr}} = 2$ repeats on common tasks and repeat indices and selects one finalist by task-balanced mean reward. Ties follow the preregistered order. The confirmation pool shares no tasks with screening, and *AppWorld* scenarios remain disjoint. It evaluates only the frozen finalist and identity, using $R_{\text{conf}} = 5$ paired repeats and restored common initial states. The prespecified one-sided lower confidence limit for mean improvement uses $\eta = 0.05$ and the declared task/family sampling unit. Deployment requires that lower limit to exceed zero. Otherwise the deployed program is identity.

Fitting alone supports adaptive diagnosis and rewriting. Screening and confirmation rewards, compiler events, traces, and failure labels are sealed from the repair module. A failed confirmation ends that optimization run; it does not trigger a further rewrite or another attempt on the same confirmation data. The two selection stages allocate all necessary budget before fitting starts. Insufficient

Table 14: RQ2/RQ3 execution configuration and task allocation. Each *AppWorld* family is a complete three-task scenario; *Telecom* uses task-level allocation.

Item	*Telecom*	*AppWorld Normal*
Execution configuration	`GPT-5.5`, Pi	`GPT-5.5`, Pi
Maximum edits / search rounds	2 / 3	2 / 3
Alternative structures per round	3 plus incumbent	3 plus incumbent
Adaptation cycles per structure/round	1	1
Fitting repeats / exploration share	2 / 0.20	2 / 0.20
Screening / confirmation repeats	2 / 5	2 / 5
One-sided tail probability	0.05	0.05
Fitting tasks (families)	47 (n/a)	60 (20)
Screening tasks (families)	9 (n/a)	12 (4)
Confirmation tasks (families)	15 (n/a)	18 (6)
Development tasks (families)	3 (n/a)	57 (19)
Evaluation tasks (families)	40 (n/a)	168 (56)
Independent optimization runs	20 per method	20 per method
Optimization-seed identifiers	41001–41020	41001–41020
Final shortlist	3 nonempty + identity	3 nonempty + identity
RQ3 total caps (rollouts / calls / tokens)	2,080 / 66,000 / 270 M	2,940 / 70,000 / 570 M
Stage resource caps	Table 15	Table 15
Per-cycle limits and panel sizes	Table 12	Table 12

Table 15: RQ3 resource caps per optimization run and stage. Fitting includes coverage, exploration, repair, and comparison. Calls include all optimization LM calls; token limits combine input and output tokens and are displayed in millions.

Stage	Rollouts	LM calls	Tokens (M)
Telecom			
Initialization traces and generation evidence	64	2,031	8.307692
Fitting	1,794	56,925	232.875000
Screening	72	2,285	9.346154
Confirmation	150	4,759	19.471154
Total	**2,080**	**66,000**	**270.000000**
AppWorld Normal			
Initialization traces and generation evidence	96	2,286	18.612245
Fitting	2,568	61,143	497.877551
Screening	96	2,286	18.612245
Confirmation	180	4,285	34.897959
Total	**2,940**	**70,000**	**570.000000**

remaining resources cannot justify shortening the registered confirmation rule after observing outcomes.

Final evaluation executes both the frozen finalist and the deployed artifact on the designated evaluation tasks without adaptation, separating search quality from the deployment decision. The primary endpoint is deployed reward. Secondary endpoints are finalist reward, identity-retention rate, and paired deployed gain over identity. The reported finalist and deployed scores average the 20 independent optimization runs using the task-balanced aggregation in Appendix H. The benchmark split retains its previously used status, as specified in Appendix I.

G.4 Cross-framework rerun protocol

Section 4.5 retains the `GPT-5.5` results under Pi and DeepSeek Harness, with one optimization seed per optimized method and model/environment/runtime configuration. Each method is optimized separately within its runtime. These RQ4 results and the RQ1 benchmark results are distinct from the 20-seed RQ2/RQ3 comparisons.

H EVALUATION AGGREGATION AND COST ACCOUNTING

H.1 NATIVE BENCHMARK METRICS

The environment scores in Figures 3–8 use task-level rewards from the native benchmark evaluators. A score is 100 times the mean reward; higher is better.

***Telecom* and *Airline*.** The official τ^2-*bench* 1.0.1 evaluator returns a binary task reward based on the task's configured `reward_basis` (Barres et al., 2025). Success requires all configured criteria to pass. Depending on the task, these criteria check the final database state, environment assertions, required information communicated to the user, or specified tool actions. Scores average the resulting task-success rewards.

***Retail*.** The original τ-*bench* evaluator checks that the final database state matches the reference execution and that any required outputs appear in the agent's responses (Yao et al., 2025). The task reward is one when these checks pass and zero otherwise; the reported score is the average task-success reward expressed as a percentage.

***ALFWorld*.** The score is the percentage of tasks for which the native TextWorld `won` flag is true (Shridhar et al., 2021). This flag measures completion of the task goal; partial goal-condition progress is a separate native statistic.

***AppWorld Normal* TGC.** *AppWorld* Task Goal Completion (TGC) is the percentage of tasks that pass all official task tests (Trivedi et al., 2024). We use `test_normal` with data version 0.2.0. The *AppWorld* columns in Tables 1 and 2 report this Normal TGC. Each task variant contributes its own binary success flag. Scenario membership determines split grouping and dependence-aware analysis; Scenario Goal Completion (SGC), which requires all variants in a scenario to succeed, is a separate metric.

These scores measure mean task performance.

H.2 PER-ARTIFACT SCORES AND OPTIMIZATION-SEED AVERAGES

Each RQ2/RQ3 optimization run produces a frozen artifact evaluated on the same environment-specific task set. For method m, optimization seed s, environment e, and evaluation task i, let $r_{m,s,e,i}$ be the native reward and N_e the evaluation size. Its score is

$$S_{m,s,e} = \frac{100}{N_e} \sum_{i=1}^{N_e} r_{m,s,e,i}. \tag{15}$$

Tables 4 and 6 report the mean over $S = 20$ independent optimization seeds:

$$\overline{S}_{m,e} = \frac{1}{20} \sum_{s=1}^{20} S_{m,s,e} = \frac{100}{20N_e} \sum_{s=1}^{20} \sum_{i=1}^{N_e} r_{m,s,e,i}. \tag{16}$$

Here $N_e = 40$ for *Telecom* and $N_e = 168$ for *AppWorld Normal*, giving 800 and 3,360 evaluation outcomes per method, respectively. The pooled success percentages over these denominators equal the means of 20 per-run task-success scores because all runs use the same number of evaluation tasks. The denominators count outcomes across optimized artifacts, while the unique task counts remain 40 and 168. Finalist and deployed scores use the same aggregation with their respective artifacts. All runs contribute, including those retaining identity.

Appendix A.1 gives the supplied aggregate numerators and denominators. Scores and differences in Tables 4–6 are computed from those counts before rounding. The fitting-diagnostic denominators in Table 5 serve a separate purpose and do not replace the 20-run evaluation denominator. Appendix I.2 specifies the fitting, selection, and development roles.

RQ1 and RQ4 retain one optimization seed per optimized method and model/environment/runtime configuration. Writing $S_{m,e}$ for those reported environment scores, the five-environment macro

average and task-pooled average are different estimands:

$$\overline{S}_m = \frac{1}{5}\sum_{e=1}^{5} S_{m,e}, \qquad S_m^{\text{pool}} = \frac{\sum_{e=1}^{5} N_e S_{m,e}}{477}.$$

Only $\overline{S}_m$ is denoted *Avg.* in Figures 3 and 4 and their numerical Tables 1 and 2. The macro difference from Base is the mean of the five environment-level differences. The differences quoted for those retained numerical tables are calculated from their displayed entries: `GPT-5.5` yields 4.82%↑ (80.90 − 76.08), and `DeepSeek-V4-Flash` yields 1.72%↑ (76.80 − 75.08), over their strongest baseline macro averages. Appendix I.3 reports changes from the displayed Base scores using that convention.

An infrastructure retry is another physical attempt at the same logical rollout, not another sample. Unresolved failures retain their place in the declared denominator and follow Appendix B.4.

H.3 Execution and token accounting

An *optimization execution* is one task rollout used for generation evidence, coverage, adaptation, racing, screening, or confirmation. Each control branch, intermediate version, and repeated rollout counts separately. A *test execution* scores a frozen program after the deployment decision and is accounted for separately. Language-model calls and tokens include execution, simulation, generation, and diagnostic reflection; a reflection call consumes optimization resources even when it causes no environment rollout. Optimizer wall time includes compilation and record processing. Table 6 reports total optimization costs averaged per run.

For any cost resource k (rollouts, language-model calls, or tokens), the reported mean and identity-retention percentage are

$$\overline{C}_{m,e}^{(k)} = \frac{1}{20}\sum_{s=1}^{20} C_{m,s,e}^{(k)}, \qquad U_{m,e} = \frac{100}{20}\sum_{s=1}^{20} \mathbf{1}[P_{m,s,e}^{\star} = P_{\emptyset}]. \tag{17}$$

Token means are displayed in millions. Thus a cost entry in Table 6 is a per-run mean, while each cap in Table 13 is an individual-run limit in the RQ3 protocol. Identity retention uses the number of runs retaining the unedited program divided by 20, not a task-level failure denominator. Appendix A.1 records the reported totals.

The RQ3 protocol charges coverage and four-condition controls to the common caps. A branch receives no cost exemption when its reward is reused in an analysis, and a new template version receives no inherited reward. Finalist-only diagnostic testing is separate from deployed-program testing; exact duplicate artifacts can share a recorded frozen evaluation with an explicit ledger reference.

Reported estimates. RQ1/RQ4 report point estimates from one optimization seed per configuration; RQ2/RQ3 report mean scores across 20 independent optimization runs. The confidence limit in Appendix E applies to the within-run deployment decision.

I Dataset splits and evaluation protocol

I.1 Exact task counts and native partitions

Table 16 specifies the five-environment task pools. The completed 20-seed RQ2/RQ3 studies use *Telecom* and *AppWorld Normal*, with 40 and 168 evaluation tasks per run. Training comprises fitting plus internal selection and excludes development. The controlled-study subdivision is given in Table 14: *Telecom* uses 47 fitting, nine screening, and 15 confirmation tasks; *AppWorld* uses 60, 12, and 18 tasks from 20, four, and six scenarios. Pool sizes are distinct from optimization execution counts.

Table 16: Task pools for the five benchmark environments. RQ2/RQ3 evaluate *Telecom* and *AppWorld Normal* over 20 optimization runs. Training equals fitting plus internal selection; development is separate. *ALFWorld* uses official `valid_unseen` as its evaluation holdout.

Environment	Training total	Fitting	Internal selection	Development	Frozen evaluation
Telecom	71	47	24	3	40
ALFWorld	1,972	1,315	657	114	134
AppWorld Normal	90	60	30	57	168
Airline	26	17	9	3	20
Retail	500	333	167	20	115
Total	**2,659**	**1,772**	**887**	**197**	**477**

Telecom. *Telecom* uses τ^2*-bench* 1.0.1. All 74 official training-side tasks are retained: three are reserved for development and 71 form optimization training. The frozen evaluation contains 40 tasks. The official split operates at the task level; seven fault families occur in both optimization training and frozen evaluation. The controlled study partitions *Telecom* at the task level without an additional family grouping.

ALFWorld. *ALFWorld* runs in its native TextWorld text environment. Training comes from official `train`, development from `valid_seen`, and frozen evaluation from `valid_unseen`. We retain the established exclusions for tasks missing executable native games and for training-side tasks whose families overlap the holdout sets. The resulting 1,972 training, 114 development, and 134 evaluation tasks are the complete eligible pools under these eligibility rules. Frozen evaluation covers all 134 executable `valid_unseen` tasks. The evaluation holdout is the official unseen-scene validation partition.

AppWorld Normal. *AppWorld* is pinned to data version 0.2.0 and uses official `train`, `dev`, and `test_normal`. Training contains 90 tasks from 30 scenarios; development contains 57 tasks from 19 scenarios; Normal evaluation contains 168 tasks from 56 scenarios. No `test_challenge` tasks are included. Every scenario has three task variants, and all three remain in the same partition. Fitting contains 20 complete scenarios; internal selection assigns four scenarios to screening and six to confirmation. The benchmark endpoint is Normal TGC, the mean official task-success flag.

Airline. *Airline* uses the revised benchmark in τ^2*-bench* 1.0.1. Excluding one near-duplicate variant and reserving three development tasks from the 30 official training-side tasks leaves 26 optimization-training tasks. The frozen evaluation contains 20 tasks.

Retail. *Retail* uses the original τ*-bench*. The retained pools comprise 500 optimization-training, 20 development, and 115 frozen-evaluation tasks, following the established native-file availability and family rules. Fitting and internal selection contain 333 and 167 tasks.

I.2 Family grouping, information access, and common task order

Let $\mathcal{D}_{\text{train}} = \mathcal{D}_{\text{fit}} \dot{\cup} \mathcal{D}_{\text{sel}}$, with $\mathcal{D}_{\text{sel}} = \mathcal{D}_{\text{scr}} \dot{\cup} \mathcal{D}_{\text{conf}}$. Fitting, screening, confirmation, development, and frozen evaluation use fixed task partitions shared across compared methods. *AppWorld* preserves complete three-task scenarios in every partition, while *Telecom* uses task-level allocation. Table 14 gives their stage counts and optimization-seed identifiers 41001–41020.

Fitting tasks provide edit-generation trajectories, matched intervention evidence, template repair, and paired racing. Internal-selection tasks evaluate candidate programs and determine the final persistent deployment state. Development tasks are limited to checking environment interfaces and general configuration; their trajectories and rewards are excluded from the edit generator and repair module. Before the frozen evaluation, the complete program structure and templates, compiler, and model/runtime settings are fixed. Evaluation rewards do not select programs and do not update persistent learning state.

Screening selects among frozen programs, and confirmation makes one deployment decision. Neither stage returns rewards or traces to diagnosis or repair. Confirmation failure ends the run with identity and cannot trigger another rewrite using the same confirmation data.

Coverage warm-up in Appendix D uses fitting tasks and counts toward optimization cost. No separate smoke-test set is reserved.

This protocol evaluates complete performance on fixed benchmark holdouts that include tasks used in earlier experiments.

I.3 Changes from Base in the current result tables

Table 17 gives percentage-point differences calculated directly from the displayed entries of Tables 1 and 2, for `GPT-5.5` and `DeepSeek-V4-Flash`, respectively. These are arithmetic summaries of the existing experiments.

Table 17: Changes from Base in the reported benchmark results, in percentage points. Each cell subtracts the corresponding displayed Base entry. Δ Avg. subtracts the displayed five-environment macro averages; arrows indicate the direction of each score difference.

			AppWorld			
Method	*Telecom*	*ALFWorld*	Normal TGC	*Airline*	*Retail*	Δ Avg.
`GPT-5.5`						
(I) Context learning						
ACE	7.50%↑	4.48%↑	8.93%↑	5.00%↑	2.61%↑	5.70%↑
(II) Candidate optimization						
GEPA	2.50%↑	2.99%↑	25.00%↑	5.00%↑	4.35%↑	7.96%↑
SkillOpt	15.00%↑	6.72%↑	20.83%↑	5.00%↓	0.00%	7.51%↑
RHO	5.00%↑	0.00%	8.33%↑	10.00%↑	4.34%↓	3.79%↑
(III) Intervention feedback						
MICA	12.50%↑	11.20%↑	27.38%↑	5.00%↑	7.83%↑	12.78%↑
`DeepSeek-V4-Flash`						
(I) Context learning						
ACE	10.00%↑	13.43%↑	1.20%↑	0.00%	6.08%↑	6.15%↑
(II) Candidate optimization						
GEPA	17.50%↑	14.93%↑	11.31%↑	5.00%↑	6.08%↑	10.97%↑
SkillOpt	30.00%↑	15.67%↑	17.86%↑	5.00%↑	2.61%↑	14.23%↑
RHO	12.50%↑	17.91%↑	11.31%↑	0.00%	8.69%↑	10.09%↑
(III) Intervention feedback						
MICA	25.00%↑	20.15%↑	20.24%↑	10.00%↑	4.34%↑	15.95%↑