RSIHub: A Composable Framework for Agent-Evolution Methods

Zimu Wang[1,4,*] **Xiaobo Wang**[2,6,*] **Siyu Ye**[3,7,*] **Jiaqi Li**[6,*]
Silin Chen[4] **Yuling Shi**[4] **Ruobing Wang**[3,7] **Jingjing Zhang**[1]
Chaofan Wang[4] **Kai Wu**[5] **Mengnan Qi**[3,7] **Kai Cai**[3,7]
Xiaodong Gu[4,†] **Zilong Zheng**[6,†]

[1]University of California, Berkeley [2]University of Science and Technology of China
[3]Douyin Group [4]Shanghai Jiao Tong University [5]Tongji University
[6]Beijing Institute for General Artificial Intelligence [7]Simple Agent Lab
[*]Equal contribution [†]Corresponding authors
xiaodong.gu@sjtu.edu.cn zlzheng@bigai.ai

Abstract

Agent-evolution methods use execution feedback to improve prompts, skills, tools, and orchestration, but their implementations often rebuild the same experimental machinery. We present RSIHub, an open-source framework for building, comparing, and extending these methods on shared infrastructure. Recipes compose reusable operators and declare mutable resources; RSIHub supplies execution, evaluation, recovery, and lineage. Researchers can focus on evolution policy while evaluation stays outside the mutable surface and results remain tied to candidates and evaluation conditions. We instantiate AHE, Hyperagents, A-Evolve, and GEPA without modifying the framework core, keeping method-specific logic in recipes and operators. We evaluate them with MiniSWE and Codex on Terminal Bench 2 and τ^3 Banking under fixed model and evaluator configurations. The largest observed overall pass@1 gain is 13.5 percentage points, while method rankings vary across settings and optimization gains do not always transfer to held-out tasks. RSIHub provides a common protocol for comparing evolution policies, inspectable candidate histories, and reusable components for developing new methods. Code, recipes, and documentation are available at https://github.com/simple-agent-lab/RSIHub.

1 Introduction

Language-model agents are developed through an empirical loop: developers inspect executions, revise prompts, skills, tools, or orchestration, and reevaluate behavior [14, 32]. Agent-evolution systems automate parts of this loop through persistent revisions [1, 14, 39, 42]; some also revise the improvement procedure itself [5, 40, 41]. Program-optimization methods apply a similar loop to language-model programs and textual components [11, 37]. Across both settings, an iteration selects a candidate, gathers evidence, proposes and evaluates a revision, and retains the outcome for later search.

Methods differ in search policies and editable resources, yet repeatedly need workspaces, execution, evaluation, artifacts, recovery, and lineage. Bundling these services with method policy can make a new method an infrastructure project and comparison an exercise in reconciling runners. A shared framework would let researchers reuse operators, change one policy component at a time, and retain a common experimental record, so researchers can test new evolution ideas without rebuilding the surrounding infrastructure.

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: Managing Agents that Manage Agents: Workshop on Responsible Use of Meta-Agents.

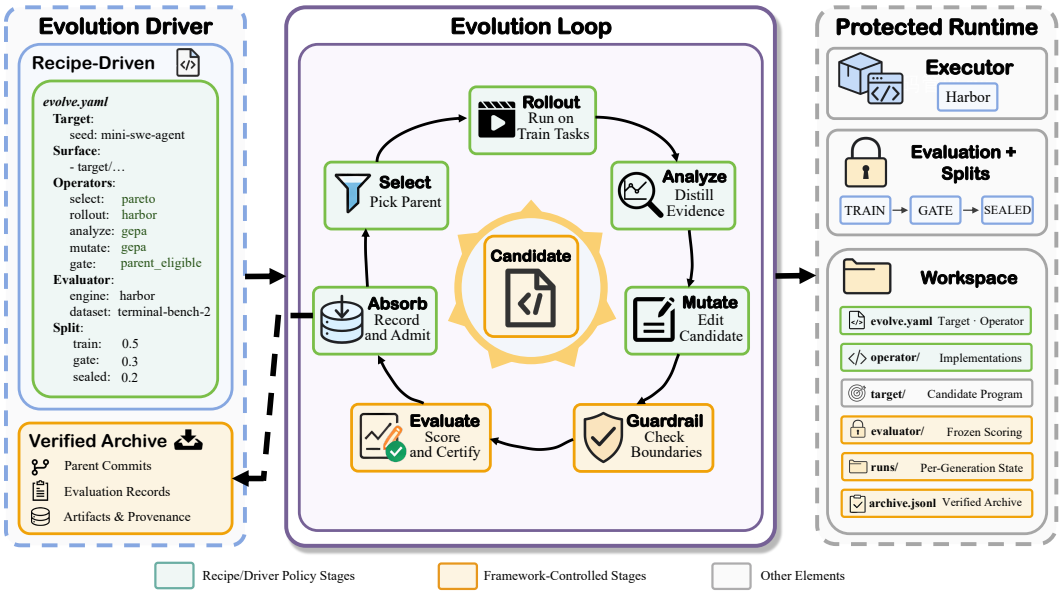


Figure 1: RSIHub separates method policy from framework-managed execution. An evolution recipe binds implementations to lifecycle stages, declares the mutable surface, and specifies the run configuration. The evolution loop invokes these operators, while the framework manages surface enforcement, protected evaluation, and verified archive writes.

RSIHub provides this shared foundation. As Figure 1 illustrates, a *recipe* binds reusable operators, declares the mutable surface, and specifies the target, evaluator, runtime, and data splits. A framework-managed *evolution loop* invokes these operators to select, revise, and retain candidates; a *protected runtime* executes and evaluates them and records their outcomes. Researchers can replace a search or mutation operator while reusing workspace construction, evaluation, artifacts, recovery, and lineage. This organization follows research frameworks in adjacent fields: stable interfaces expose research choices while shared runtime mechanics support their execution [22, 25].

Reuse also requires trustworthy evidence. If method policy can change the evaluator or rewrite outcomes, an apparent gain becomes ambiguous. RSIHub keeps evaluation and its archive under framework control, outside the declared mutable surface. Each result identifies the committed candidate and evaluation conditions that produced it, while a separate Gate operator decides parent eligibility. This separation makes both improvements and failed attempts inspectable as part of the research record.

Concretely, this paper makes three contributions:

- **Composable recipes.** We express method policy through reusable lifecycle operators and explicit mutable surfaces, adapting AHE, Hyperagents, A-Evolve, and GEPA to the same interfaces. Researchers can reuse these recipes or extend individual components.
- **Reusable research infrastructure.** We provide candidate execution, recovery, artifacts, certified evaluation records, and lineage as shared services. This lets method implementations concentrate on evolution policy while inheriting a common path for execution and evidence inspection.
- **Evaluation on common ground.** We compare four recipes across two seed agents and two benchmarks under fixed model and evaluator configurations, and test held-out transfer, execution fidelity, recovery, boundary enforcement, runtime overhead, and integration effort.

Open-source resources. We release RSIHub through Simple Agent Lab under Apache-2.0, with runnable recipes, reusable operators, documentation[1], and an experiment viewer. Researchers can start from runnable recipes, inspect candidate changes and evaluation results, and contribute new recipes and operators.

[1] https://simpleagentlab.com/RSIHub/

2 Related Work

Agent optimization and evolution. Agent improvement spans inference-time feedback, persistent artifact optimization, and self-modification. Reflexion retains linguistic feedback across trials; Self-Refine revises outputs at test time [17, 28]. RAM improves external memory through communications [12]. Related work examines memory mechanisms [9], reflection learning [13], uncertainty-aware preference optimization [30], and reward-model self-improvement [31]. DSPy and TextGrad optimize language-model programs or textual variables [11, 37]. Promptbreeder evolves task and mutation prompts; ACE evolves context playbooks; and CoEvoSkills evolves multi-file skills with a separate surrogate verifier [5, 39, 42]. EffiSkill mines reusable skills for code-efficiency optimization [34]. GPTSwarm optimizes agent graphs, while AgentSquare searches combinations of planning, reasoning, tool-use, and memory modules [24, 44]. EvoAgentX integrates optimizers for evolving multi-agent workflows [33]. In autoresearch, Fork-and-Flush combines workspace inheritance with fresh conversational contexts and branch selection to diversify candidate search [2]. Our recipes adapt selected policies from four approaches. A-Evolve frames deployment-time adaptation as goal-directed optimization over persistent system state [15]. AHE guides revisions to harness files with distilled execution evidence [14]; GEPA uses trajectory reflection and Pareto selection to optimize prompts [1]; and Hyperagents combines a task agent and a self-modifying meta agent in one editable program, allowing the improvement procedure to evolve [41]. Other approaches include STOP, which revises optimizer scaffolding; ADAS, which generates agent programs with a meta agent; DGM, which modifies its coding agent; and AlphaEvolve, which evolves target algorithms [8, 20, 38, 40]. RSIHub provides shared execution, lifecycle, and evidence services for comparing our four policy adaptations.

Tasks, environments, and execution infrastructure. Benchmarks cover software engineering, terminal interaction, and knowledge-grounded dialogue [10, 18, 26]. Others examine multilingual code refactoring [27], repository exploration [43], and repository-level question answering [23]. Harbor standardizes task, dataset, agent, and container-environment interfaces for evaluation and optimization [7]. The OpenHands Software Agent SDK provides composable agent, tool, conversation, and workspace abstractions with local or remote execution [32]. RSIHub delegates container execution, trajectory capture, and per-task verification to Harbor, and adds a versioned evolution lifecycle, recipe-level method composition, and common provenance across methods.

Provenance and measurement validity. The AI4AI survey distinguishes measured gains, retention, and held-out transfer [35]. Software-provenance and reproducibility systems authenticate authorized production steps, bind identities to signed artifacts, or package executions with their dependencies [3, 19, 29]. RSIHub records candidate lineage, distinguishes parent selection from post-evaluation Gate decisions, and binds outcomes to candidates, recipes, and evaluation conditions. Controlled skill-evolution experiments show that persistent improvement can be sparse and depend on feedback, validation, models, and benchmarks [16]. In a synthetic reward-model setting, overoptimizing an imperfect proxy can reduce performance under a fixed gold-standard reward model [6]. After curriculum training in deliberately gameable environments, behavior meeting an operational definition of reward tampering appeared in a small but non-negligible share of cases; the authors note that this definition admits false positives [4]. RSIHub services write certified evaluation fields and filter operator-supplied Record annotations before appending them. This supports attribution under the framework's trust assumptions; benchmark overfitting and broader specification gaming remain evaluator- and study-level concerns.

3 The RSIHub Framework

RSIHub separates method policy from the shared mechanics of candidate execution, evaluation, and record keeping. This section states the principles behind that separation, then describes how the candidate lifecycle and provenance model realize it.

3.1 Design principles

RSIHub is built so that a researcher can assemble an evolution method from interchangeable parts, decide what those parts may change and what they may see, and reuse framework services for recurring mechanics.

Assemble methods from interchangeable parts. RSIHub fixes the slots of the evolution lifecycle and leaves their contents open. A *stage* is a slot such as `select` or `mutate`. An *operator* is one implementation of a stage. A *recipe* names the operator it wants in each slot and configures it. When the required operator implementations are available, a recipe can express the method-specific choices through configuration.

Say what a method may change and see. Two recipe declarations make this an experimental choice. The `surface` block declares the resources a candidate may change, allowing the same operator interfaces to support prompt-only and whole-harness studies. The `mutate` block configures how the editing agent runs and which retained context it receives. Together, these declarations distinguish the persistent resources being evolved from the information used to propose each revision.

Provide the rest of the pipeline once. Workspace construction, candidate snapshots, evaluation execution, artifact retention, terminal-state classification, recovery from interrupted runs, and lineage bookkeeping commonly recur across evolution methods. RSIHub implements these services once and gives every recipe the same execution path. When a study changes one operator without changing the other recipe fields, it isolates one declared policy component while reusing the same framework mechanics.

Keep scoring out of the method's reach. Method policy may revise candidates and the operators that propose them. Measurement remains a framework responsibility: framework services own evaluation and the evidence record outside the mutable surface a recipe declares, while parent eligibility remains a separate downstream policy choice.

Together, these principles separate method-specific choices from framework-managed execution and measurement. The candidate lifecycle below makes this separation operational.

3.2 Composable candidate lifecycle

The lifecycle follows the seven phases shown in Figure 1: Select, Rollout, Analyze, Mutate, Guardrail, Evaluate, and Absorb. Across these phases, recipes bind five required operator interfaces—Select, Rollout, Mutate, Gate, and Record—and may bind four optional ones, marked with * below: Analyze, Validate, Novelty, and Reflect.

RSIHub implements this lifecycle on Harbor [7], which provides container execution, trajectory capture, and per-task verification. Above this substrate, RSIHub manages candidate lineage, train/gate/sealed splits, mutable-surface enforcement, and certified evaluation records.

Before a run begins, the framework resolves each recipe binding and validates its settings against the corresponding operator schema. It records the validated configuration for provenance. Each operator runs in a separate process, and the framework validates its structured output before using it.

The archive separates framework-certified fields, the evaluated candidate, its parent, the evaluation conditions, and the outcome, from method-specific annotations. Operators propose annotations; the framework filters and writes them.

- **Select.** Chooses an eligible parent from the archive according to the recipe's search policy. The framework verifies its identity and forks its exact recorded commit.
- **Rollout.** Runs the parent on training tasks and returns execution evidence as a summary and retained artifact references. The supplied Harbor rollout uses only the training split; a replay-based rollout reuses the parent's recorded evaluation instead of running again.
- **Analyze*.** Distills rollout cases into bounded mutation evidence, such as failure patterns or compact traces, in the representation Mutate consumes.
- **Mutate.** Proposes edits within the declared mutable surface. The operator is typically an agent rather than a fixed transformation: it works over multiple turns, may run small-scale trials on training tasks before proposing a change, and, as in Hyperagents, may edit the evolution operators themselves when the surface allows. It reports changed paths, notes, and usage.

- **Guardrail.** The framework checks the actual tree diff against the mutable surface. Two optional operators may then reject the proposal: Validate* applies a method-specific pre-evaluation check, and Novelty* drops near-duplicates of accepted diffs before they consume evaluation budget.
- **Evaluate.** This framework-managed phase commits the reviewed tree, scores a clean checkout on the gate tasks, and certifies the result. Gate tasks may coincide with training tasks; sealed tasks are reserved for held-out reporting.
- **Absorb.** Gate decides from the certified score whether the child becomes a parent, Record proposes method-specific annotations, and Reflect* distills insights from archive history for later generations. The framework records permitted outputs; none of these operators can change certified evaluation fields.

Algorithm 1 expands these phases into their operator calls and framework actions over the validated recipe R and archive history H. The proposal c remains uncommitted through Guardrail; $\hat{c}$ is the clean commit used for protected evaluation. In the pseudocode, $\textsc{Stops}(x)$ denotes rejection, failure, malformed output, or timeout. An evaluation is *benchmark-complete* when its trial outcomes satisfy the configured coverage policy; only then does Gate run.

A skipped Analyze passes through X, and skipped Validate and Novelty accept. Configured Reflect appends validated operations to a separate playbook without modifying H. Empty proposals and surface violations likewise close as typed, archived attempts. Failures of Rollout, Analyze, or Mutate take the same CLOSEANDRECORD path and are elided for compactness.

Algorithm 1 One RSIHub candidate transaction.

procedure EVOLVEONE(R, H)
  $p \leftarrow \textsc{Select}(H)$
  $c \leftarrow \textsc{ForkCertifiedParent}(p, H)$
  $X \leftarrow \textsc{Rollout}(c)$
  $Z \leftarrow \textsc{Analyze}(c, X)^*$
  $\textsc{Mutate}(c, Z)$
  $s \leftarrow \textsc{CheckSurface}(p, c, R_{\text{surface}})$
  if $\textsc{Stops}(s)$ **then**
    return $\textsc{CloseAndRecord}(H, p, c, s)$
  end if
  $v \leftarrow \textsc{Validate}(c)^*$
  if $\textsc{Stops}(v)$ **then**
    return $\textsc{CloseAndRecord}(H, p, c, v)$
  end if
  $n \leftarrow \textsc{Novelty}(p, c, H)^*$
  if $\textsc{Stops}(n)$ **then**
    return $\textsc{CloseAndRecord}(H, p, c, n)$
  end if
  $\hat{c} \leftarrow \textsc{CommitReviewedTree}(c)$
  $q \leftarrow \textsc{ResolveContract}(R, \hat{c})$
  $y \leftarrow \textsc{Evaluate}(\hat{c}, q)$
  $H \leftarrow \textsc{Certify}(H, p, \hat{c}, q, y)$
  if $y.\text{outcome} \neq \text{benchmark_complete}$ **then**
    return H
  end if
  $a \leftarrow \textsc{Gate}(p, \hat{c}, y, H)$
  if $a.\text{failed}$ **then**
    $H \leftarrow \textsc{CloseAndRecord}(H, p, \hat{c}, a)$
  else
    $H \leftarrow \textsc{AppendGateDecision}(H, a)$
    $e \leftarrow \textsc{Record}(p, \hat{c}, y, a, H)$
    $H \leftarrow \textsc{AppendAllowedRecord}(H, e)$
  end if
  $\textsc{Reflect}(H)^*$
  return H
end procedure

When a recipe places operators inside the mutable surface, the framework must decide which version of an operator runs. For each stage invocation, it loads the implementation from the checkout supplied to that stage. Changes to already-run stages take effect in a later generation, while changes to Validate or Novelty may affect the current proposal. Gate and Record subsequently load from the committed child. The certified record provides a recovery checkpoint after evaluation: it holds the committed child and its outcome before downstream finalization. Gate, Record, and Reflect may add policy outputs but cannot alter certified evaluation fields.

3.3 Provenance

A certified evaluation event ties the committed candidate, its parent, and the recipe configuration to evaluator and task-set fingerprints, trials, score, costs, artifact references, and lineage. A hash-referenced contract can also record evaluator-configuration and dataset-content digests, split and repetition settings, model identity, retry policy, evaluator-runtime digest, and framework version. This identity lets a study distinguish a change in method policy from a change in recorded evaluation conditions.

From experiment records to method design. Researchers can start from a bundled recipe and inspect candidate snapshots, lineage, task outcomes, costs, and trajectories to investigate stalled

progress and revision patterns, combining artifacts with scores as in Fork-and-Flush [2]. For example, researchers can replace the parent-selection operator in a separate run while holding the seed agent, mutable surface, model, evaluator, and task splits fixed. RSIHub supplies execution, recovery, and provenance services, with command-line summaries and a read-only viewer for inspecting the resulting records (Section C). This workflow lets researchers test new policy components while reusing the surrounding experimental infrastructure. At the evaluated commit, the four adaptations used 47–61 recipe lines and reused two operator bindings each, with no framework-source changes (Table 6). Method-specific operator code ranged from 216 to 1,087 lines.

4 Evaluation

In this section, we organize the evaluation around three research questions:

- **RQ1:** Can RSIHub support effective agent evolution?
- **RQ2:** Does RSIHub preserve the capabilities of the agents it executes?
- **RQ3:** Does RSIHub preserve the integrity of agent-evolution experiments?

To answer these questions, we compare evolution recipes across agents, benchmarks, and held-out splits for RQ1, use generation-zero controls against direct execution for RQ2, and conduct fault-injection and boundary-enforcement trials for RQ3.

4.1 Recipe instantiations

RSIHub includes four recipes that adapt selected policies from A-Evolve [15], AHE [14], GEPA [1], and Hyperagents [41]; a hill-climbing recipe provides a minimal baseline. Their operator configurations and mutation targets illustrate how the shared execution path supports different method policies.

Recipe	Search behavior	Mutation focus
`hill_climb`	continue from the best parent	target
`aevolve`	goal-directed persistent adaptation	prompt, target skills
`ahe`	harness revision from execution evidence	target
`gepa`	Pareto selection; minibatch validation	prompt, task skill
`hyperagents`	co-evolve target and evolution policy	target, selected operators

Each recipe is an interface adaptation at the level of operator choices and configuration. Exact correspondence with a source implementation also depends on its prompts, models, budgets, dependencies, and runtime adapters. A user can run an adaptation unchanged, swap one operator, revise its mutable resources, or write a new recipe over the same services.

4.2 Setup

Agents and evolution methods. We evolve MiniSWE [36] and Codex [21] as seed agents with AHE [14], Hyperagents [41], A-Evolve [15], and GEPA [1]. Each adaptation is instantiated as an RSIHub recipe over the same frozen Harbor evaluator [7]. The recipes retain their intended mutable surfaces as in Section 4.1.

Benchmarks and metrics. We evaluate the seed agent and evolved agent on Terminal Bench 2 [18] and τ^3 Banking, which is introduced as the knowledge-grounded τ-Banking domain by τ-Knowledge [26], for one trial per task, and report pass@1. Terminal Bench 2 contains 89 tasks split into 50 train, 19 gate, and 20 sealed tasks; τ^3 Banking contains 97 tasks split into 50 train, 20 gate, and 27 sealed tasks. Train trajectories provide evolution feedback, gate tasks support candidate admission, and sealed tasks are excluded from mutation feedback and used only for held-out reporting. Because the splits have different sizes, each split column in Table 1 is its contribution to the full-benchmark score, e.g., $N_{\text{pass},s}/N_{\text{all}}$.

Models. All experiments use GPT-5.4 with high reasoning effort as the target model for both MiniSWE and Codex. Every recipe uses a Codex mutation agent backed by GPT-5.4 with xhigh

Table 1: Pass@1 before and after evolution on Terminal Bench 2 and τ^3 Banking. All entries are percentages; split columns report their contribution to full-benchmark pass@1. Parenthesized values are absolute changes from the corresponding seed baseline, measured in percentage points. Among evolved variants, the best results per agent are in **bold** and the second best are underlined.

Benchmark	Agent	Method	Train split	Gate split	Sealed split	Overall
Terminal Bench 2	MiniSWE	*Seed*	32.6	12.4	11.2	56.2
		AHE	37.1 (+4.5%)	12.4 (+0.0%)	15.7 (+4.5%)	65.2 (+9.0%)
		Hyperagents	36.0 (+3.4%)	**16.9** (+4.5%)	**16.9** (+5.6%)	**69.7** (+13.5%)
		A-Evolve	**40.5** (+7.9%)	12.4 (+0.0%)	**16.9** (+5.6%)	**69.7** (+13.5%)
		GEPA	36.0 (+3.4%)	12.4 (+0.0%)	15.7 (+4.5%)	64.0 (+7.8%)
	Codex	*Seed*	39.3	15.7	14.6	69.7
		AHE	**42.7** (+3.4%)	12.4 (-3.4%)	**16.9** (+2.2%)	**71.9** (+2.2%)
		Hyperagents	41.6 (+2.3%)	13.5 (-2.2%)	15.7 (+1.1%)	70.8 (+1.1%)
		A-Evolve	40.5 (+1.1%)	**15.7** (+0.0%)	14.6 (+0.0%)	70.8 (+1.1%)
		GEPA	39.3 (+0.0%)	**15.7** (+0.0%)	14.6 (+0.0%)	69.7 (+0.0%)
τ^3 Banking	MiniSWE	*Seed*	12.4	6.2	9.3	27.8
		AHE	16.5 (+4.1%)	7.2 (+1.0%)	5.2 (-4.1%)	28.9 (+1.1%)
		Hyperagents	**17.5** (+5.1%)	8.2 (+2.0%)	7.2 (-2.1%)	**33.0** (+5.2%)
		A-Evolve	12.4 (+0.0%)	6.2 (+0.0%)	**9.3** (+0.0%)	27.8 (+0.0%)
		GEPA	11.3 (-1.1%)	**11.3** (+5.1%)	7.2 (-2.1%)	29.9 (+2.1%)
	Codex	*Seed*	16.5	5.2	3.1	24.7
		AHE	**17.5** (+1.0%)	8.2 (+3.0%)	**8.2** (+5.1%)	**34.0** (+9.3%)
		Hyperagents	15.5 (-1.0%)	6.2 (+1.0%)	7.2 (+4.1%)	28.9 (+4.2%)
		A-Evolve	15.5 (-1.0%)	7.2 (+2.0%)	6.2 (+3.1%)	28.9 (+4.2%)
		GEPA	16.5 (+0.0%)	**9.3** (+4.1%)	7.2 (+4.1%)	33.0 (+8.3%)

reasoning effort. We hold the target and mutation model configurations fixed across benchmarks, seed agents, and evolution methods so that performance differences are attributable to the evolution recipes under shared model configurations.

Optimization and runtime. All recipes propose one child per generation, and evolution runs continue for at most 10 generations. The evolution results use static task sampling and one evaluation per task, while the capability-preservation controls use three independent repetitions. All task execution uses Docker-isolated Harbor environments. The setup script downloads and verifies content-pinned benchmark assets and builds the recipe-specific mutation container, pinning MiniSWE 2.4.5 or Codex 0.143.0.

4.3 Evolution effectiveness and transfer

Protocol. We test the effectiveness conjecture by comparing each evolved recipe with its seed under shared conditions. Overall score changes probe whether the framework can support effective improvement; train, gate, and sealed results reveal whether those improvements survive admission and extend beyond the trajectories used to propose revisions. Crossing agents, benchmarks, and methods makes the conjecture answerable across heterogeneous settings rather than a single favorable configuration.

Overall performance. The results show substantial gains, with their size depending on the benchmark, seed agent, and evolution method. On Terminal Bench 2, every method improves the MiniSWE baseline by 7.8–13.5 percentage points; Hyperagents and A-Evolve reach the highest overall score of 69.7. Gains are smaller from the stronger Codex baseline: AHE reaches 71.9 (+2.2 points), Hyperagents and A-Evolve each add 1.1 points, and GEPA preserves the baseline. The pattern changes on τ^3 Banking. Hyperagents gives the largest MiniSWE improvement, from 27.8 to 33.0, whereas AHE performs best with Codex, raising the overall score from 24.7 to 34.0. GEPA is close behind at 33.0. These shifts indicate that method rankings are conditional on both the task environment and the starting agent.

Task split-level behavior. The split scores expose tradeoffs hidden by the overall metric. With MiniSWE on Terminal Bench 2, all methods improve the train and sealed splits, while only Hyperagents improves the gate split. A-Evolve obtains the largest train gain (+7.9 points) and ties Hyperagents on sealed performance, yet its gate score is unchanged. With Codex, AHE and Hyperagents improve train and sealed performance while decreasing gate performance by 3.4 and 2.2 points, respectively. On τ^3 Banking, Hyperagents produces the largest overall gain from MiniSWE despite a 2.1-point sealed regression; AHE and GEPA also lose 4.1 and 2.1 points on sealed tasks. The Codex results are more consistent across held-out splits: AHE improves train, gate, and sealed scores by 1.0, 3.0, and 5.1 points, while GEPA preserves train performance and improves both gate and sealed scores by 4.1 points.

Implications. No method dominates across benchmarks, seeds, and splits. These setting-dependent outcomes motivate RSIHub's support for changing evolution policies while retaining a common execution and evaluation pipeline. Gate regressions on Terminal Bench 2 and sealed regressions on τ^3 Banking show why inspectable evidence matters: a higher overall score can conceal held-out losses. Split-level results tied to candidate records help researchers diagnose unsuccessful adaptations and choose which policies to investigate further. Substantial gains occur in several configurations, including on sealed tasks, but neither improvement nor transfer is uniform. These descriptive point estimates do not establish the stability of close differences without repeated runs.

4.4 Execution fidelity

Protocol. To test the fidelity conjecture without confounding it with evolution, we compare framework-mediated execution of an unchanged seed agent with Direct execution. Direct execution denotes invoking the frozen evaluator outside RSIHub without creating or recording framework state. Direct-A and Direct-B are two independent direct executions placed before and after the RSIHub execution, respectively. For each condition, we conduct three repetitions, each running the same unchanged seed agent in the fixed order Direct-A, RSIHub generation zero, and Direct-B under identical model, task, evaluator, and runtime settings. We report Avg@3 as the mean pass@1 across the three repetitions. Within each repetition, the Direct score is the mean of Direct-A and Direct-B. D–D agreement measures agreement between Direct-A and Direct-B. D–R agreement is the mean of the Direct-A–RSIHub and Direct-B–RSIHub task-level agreements, and excess disagreement is the mean cross-mode disagreement rate minus the D–D disagreement rate.

Table 2: Capability preservation across three repetitions. Avg@3 is mean pass@1; D–D is Direct-A–Direct-B agreement, and D–R averages Direct-A–RSIHub and Direct-B–RSIHub agreement.

Benchmark	Agent	Avg@3			Agreement		Excess disagreement
		Direct	RSIHub	Δ	D–D	D–R	
Terminal Bench 2	MiniSWE	59.6	56.9	-2.6	83.5	84.6	-1.1
Terminal Bench 2	Codex	64.0	61.4	-2.6	88.0	82.0	$+6.0$
τ^3 Banking	MiniSWE	24.6	22.0	-2.6	81.1	80.9	$+0.2$
τ^3 Banking	Codex	29.6	33.0	$+3.4$	82.1	80.1	$+2.1$

Results. On Terminal Bench 2, both agents change by -2.6 points. MiniSWE's D–R agreement is 84.6%, compared with 83.5% D–D agreement. Codex has 82.0% D–R agreement versus 88.0% D–D agreement, yielding $+6.0$ points of excess disagreement; its RSIHub Avg@3 of 61.4 lies within the 60.7–65.2 range across the six direct executions. On τ^3 Banking, MiniSWE and Codex change by -2.6 and $+3.4$ points, with excess disagreement of $+0.2$ and $+2.1$ points. Direct executions themselves disagree on 12.0–18.9% of tasks. These controls characterize execution variability; they do not establish execution-mode equivalence or the statistical significance of the single-run evolution gains. See Section B.1 for details.

4.5 Recovery and integrity

Protocol. We test the integrity conjecture by terminating runs during mutation or evaluation and after writing an evaluation receipt, then resuming and comparing the recovered state with an uninterrupted

control. We also attempt evaluator mutation, writes outside the declared mutable surface, access to gate, sealed, and anchor evidence, and forged archive outcomes. We run 30 independent trials per scenario in each benchmark campaign. A trial passes only if the workspace reaches a valid and attributable state, unauthorized changes cannot affect an accepted result, and the champion remains reproducible from framework evidence.

Table 3: Fault-injection and boundary-enforcement results in two benchmark campaigns. Each correct-outcome entry is out of 30 trials; no trial admitted corrupted state. Latencies are reported separately rather than averaged across machines.

			Terminal Bench 2		τ^3 Banking	
Class	Scenario	Required invariant	Correct	Median (s)	Correct	Median (s)
Recovery	Terminate during mutation	Resume from a certified parent; admit no partial child	30	4.72	30	5.15
Recovery	Terminate during evaluation	Preserve the old attempt and certify one terminal retry	30	3.61	30	4.94
Recovery	Terminate after stamp/receipt	Reconcile receipts without a duplicate event	30	3.27	30	4.81
Boundary	Modify evaluator files	Reject evaluator mutation before certified evaluation	30	4.72	30	2.35
Boundary	Write outside the mutable surface	Exclude all out-of-surface edits from the candidate	30	3.27	30	2.37
Boundary	Access protected evidence during mutation	Keep gate, sealed, and anchor canaries out of the mutation bundle	30	0.23	30	1.63
Integrity	Forge an archive outcome	Fail closed on unstamped events or a forged champion cache	30	0.98	30	0.69

Results. All 420 trials reached the required outcome without admitting corrupted state. Latencies are reported separately because the campaigns ran on different hosts. These results support integrity only within the tested lifecycle boundaries and attack cases. Method authors can reuse these tested recovery and evidence checks across recipes. Section B reports runtime overhead and method-integration effort.

5 Conclusion

RSIHub provides open, reusable infrastructure for agent evolution. All four evaluated adaptations reuse framework services without framework source changes. Gains and held-out transfer vary across settings, motivating shared protocols and inspectable evidence for policy comparison. We invite researchers to build on the released recipes, compare new policy components, and contribute operators that others can reuse.

6 Limitations

RSIHub's evidence guarantees rely on trusted framework services, host and filesystem protections, Git and digest operations, configured external services, and valid evaluators. Our results apply to the workloads, hosts, and lifecycle boundaries tested in this study, and the capability-preservation measurements assess task-level agreement rather than full behavioral equivalence. Runtime fingerprints do not replace host or service attestation, and RSIHub does not contain hostile code, so stronger deployments require independent attestation, isolation, and containment.

7 Responsible Use Statement

RSIHub can enable unsafe capability growth, reward gaming, or exposure of sensitive traces. Higher pass@1 does not establish safe deployment or exclude harmful intermediate actions; banking-benchmark success does not establish suitability for financial decisions. RSIHub supports auditing and rollback, not containment: editable operators and candidate adapters may execute on the host, beyond task-container isolation and feedback filtering.

Deployers should restrict filesystem and network access, exclude production credentials, and require held-out safety evaluation and human review before promotion. Resource budgets and halt and rollback controls are needed, although rollback cannot undo external actions or disclosures. Trajectories and archives need access controls, retention limits, and redaction before sharing.

References

[1] 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, Alex 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://proceedings.iclr.cc/paper_files/paper/2026/hash/0e9e708b6f48e14fd0ac29e167413f76-Abstract-Conference.html.

[2] Ziyang Cai, Christos Ziakas, Vasilis Kontonis, Tim Pearce, Siddhartha Sen, Akshay Krishnamurthy, Shivam Garg, and Dimitris Papailiopoulos. Fork-and-Flush: Escaping idea basins in autoresearch agents. *arXiv preprint arXiv:2610.07447*, 2026. doi: 10.48550/arXiv.2610.07447. URL https://arxiv.org/abs/2610.07447.

[3] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. ReproZip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2085–2088, 2016. doi: 10.1145/2882903.2899401. URL https://doi.org/10.1145/2882903.2899401.

[4] Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, Buck Shlegeris, Samuel R. Bowman, Ethan Perez, and Evan Hubinger. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*, 2024. doi: 10.48550/arXiv.2406.10162. URL https://arxiv.org/abs/2406.10162.

[5] Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 13481–13544, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

[6] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10835–10866, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

[7] Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. Software, version 0.18.0, 2026. URL https://doi.org/10.5281/zenodo.21249820.

[8] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/36b7acf6f6010652b3f2a433774a66fe-Abstract-Conference.html.

[9] Zixia Jia, Jiaqi Li, Yipeng Kang, Yuxuan Wang, Tong Wu, Quansen Wang, Xiaobo Wang, Shuyi Zhang, Junzhe Shen, Qing Li, Siyuan Qi, Yitao Liang, Di He, Zilong Zheng, and Song-Chun Zhu. The AI hippocampus: How far are we from human memory? *Trans. Mach. Learn. Res.*, 2025, 2025. URL https://openreview.net/forum?id=Sk7pwmLuAY.

[10] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

[11] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/f1cf02ce09757f57c3b93c0db83181e0-Abstract-Conference.html.

[12] Jiaqi Li, Xiaobo Wang, Wentao Ding, Zihao Wang, Yipeng Kang, Zixia Jia, and Zilong Zheng. RAM: towards an ever-improving memory system by learning from communications. *CoRR*, abs/2404.12045, 2024. doi: 10.48550/ARXIV.2404.12045. URL https://doi.org/10.48550/arXiv.2404.12045.

[13] Jiaqi Li, Xinyi Dong, Yang Liu, Zhizhuo Yang, Quansen Wang, Xiaobo Wang, Song-Chun Zhu, Zixia Jia, and Zilong Zheng. Reflectevo: Improving meta introspection of small llms by learning self-reflection. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pages 16948–16966. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-ACL.871. URL https://doi.org/10.18653/v1/2025.findings-acl.871.

[14] Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026. doi: 10.48550/arXiv.2604.25850. URL https://arxiv.org/abs/2604.25850.

[15] Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, Xiang Zhang, Suhang Wang, Benoit Dumoulin, and Jian Pei. Position: Agentic evolution is the path to evolving LLMs. *arXiv preprint arXiv:2602.00359*, 2026. doi: 10.48550/arXiv.2602.00359. URL https://arxiv.org/abs/2602.00359.

[16] Yuxuan Liu, Zhaochen Su, Yuhao Zhang, Jiahe Guo, Zhongwei Xie, Huihao Jing, Lingyun Xie, Qing Zong, Yauwai Yim, Zhixiong Zhang, Haoran Li, and Yangqiu Song. Rethinking self-evolving agent skills: Feedback dynamics over multiple rounds. *arXiv preprint arXiv:2608.02636*, 2026. doi: 10.48550/arXiv.2608.02636. URL https://arxiv.org/abs/2608.02636.

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

[18] Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

[19] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2353–2367, 2022. doi: 10.1145/3548606.3560596.

[20] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian

Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. doi: 10.48550/arXiv.2506.13131. URL https://arxiv.org/abs/2506.13131.

[21] OpenAI. Openai codex cli. Software, version 0.143.0, 2026. URL https://github.com/openai/codex/releases/tag/rust-v0.143.0. Released 2026-07-08; accessed 2026-08-24.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[23] Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions? *arXiv preprint arXiv:2509.14635*, 2025. doi: 10.48550/arXiv.2509.14635. URL https://arxiv.org/abs/2509.14635.

[24] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/0ae94013da7cd459402fd77874e09ee3-Abstract-Conference.html.

[25] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

[26] Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik Narasimhan, and Victor Barres. τ-Knowledge: Evaluating conversational agents over unstructured knowledge. *arXiv preprint arXiv:2603.04370*, 2026. doi: 10.48550/arXiv.2603.04370. URL https://arxiv.org/abs/2603.04370.

[27] Yuling Shi, Jinghan Xu, Kelin Fu, Wenhao Zeng, Shilin He, Lei Zhang, Yue Liu, Zelin Zhao, Terry Yue Zhuo, Jialun Cao, Siyu Ye, Tianyu Liu, Kai Cai, Shing-Chi Cheung, and Xiaodong Gu. SWE-Bench ProMax: Benchmarking agents on large-scale multilingual code refactoring. *arXiv preprint arXiv:2608.09802*, 2026. doi: 10.48550/arXiv.2608.09802. URL https://arxiv.org/abs/2608.09802.

[28] 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, pages 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

[29] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium*, pages 1393–1410, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[30] Xiaobo Wang, Zixia Jia, Jiaqi Li, Qi Liu, and Zilong Zheng. Adaptive preference optimization with uncertainty-aware utility anchor. In *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 19204–19225. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-EMNLP.1046. URL https://doi.org/10.18653/v1/2025.findings-emnlp.1046.

[31] Xiaobo Wang, Tong Wu, Min Tang, Jiaqi Li, Qi Liu, and Zilong Zheng. The flip side of RLHF: on-policy feedback for reward model self-supervised improvement. *CoRR*, abs/2605.30888, 2026. doi: 10.48550/ARXIV.2605.30888. URL https://doi.org/10.48550/arXiv.2605.30888.

[32] Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The OpenHands software agent SDK: A composable and extensible foundation for production agents. In *Proceedings of Machine Learning and Systems*, volume 8, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/8ae9cf363ea625161f885b798c1f1f78-Abstract-Conference.html.

[33] Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. EvoAgentX: An automated framework for evolving agentic workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 643–655, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-demos.47. URL https://aclanthology.org/2025.emnlp-demos.47/.

[34] Zimu Wang, Yuling Shi, Mengfan Li, Zijun Liu, Jie M. Zhang, Chengcheng Wan, and Xiaodong Gu. EffiSkill: Agent skill based automated code efficiency optimization. *arXiv preprint arXiv:2603.27850*, 2026. doi: 10.48550/arXiv.2603.27850. URL https://arxiv.org/abs/2603.27850.

[35] Kai Wu, Hao Lyu, Zhen Luo, Chaofan Wang, Siyu Ye, Jinghao Lin, Xiaozhong Ji, Boyuan Jiang, Shengzhi Wang, Zihan Wang, Yiwen Ye, Hao Wang, Zimu Wang, Wenzhe Liu, Ruobing Wang, Kai Cai, Mingliang Xiong, Wen Fang, Mingqing Liu, Yifan Zhang, Lei Yang, Xiaobin Hu, and Qingwen Liu. AI4AI survey: From long-horizon agents to recursive self-improvement—definitions, reliable horizons, and open problems. Preprints.org, 2026. URL https://www.preprints.org/manuscript/202608.2108/v1. Version 1. Preprint.

[36] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652, 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

[37] 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://doi.org/10.1038/s41586-025-08661-4.

[38] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=46Zgqo4QIU.

[39] Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue (Steve) Liu, Xiaoxiao Li, and Philip S. Yu. CoEvoSkills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026. doi: 10.48550/arXiv.2604.01687. URL https://arxiv.org/abs/2604.01687. Accepted at the Conference on Language Modeling (COLM 2026).

[40] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10007327.

[41] Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026. doi: 10.48550/arXiv.2603.19461. URL https://arxiv.org/abs/2603.19461.

[42] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Y. Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/8a94ff6f922d995d7d3f4ebf4143e442-Abstract-Conference.html.

[43] Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Explore: Benchmarking how coding agents explore repositories. *arXiv preprint arXiv:2606.07297*, 2026. doi: 10.48550/arXiv.2606.07297. URL https://arxiv.org/abs/2606.07297.

[44] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62743–62767, 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A Representative Recipe Configuration

The listing below presents a representative GEPA-inspired recipe. Low-level execution settings are omitted for brevity.

```yaml
experiment:
  max_generations: 10
  budget_usd: 150
  children_per_gen: 1
  seed: 0
target:
  seed: builtin-codex
surface:
  include: [target/**]
  exclude: []
operators:
  select:
    operator: pareto
    timeout_s: 600
    config:
      seed: 0
  rollout:
    operator: harbor
    timeout_s: 3600
    config:
      budget_tasks: 10
      task_sampling: generation_shuffle
  analyze:
    operator: gepa
    timeout_s: 600
    config:
      components:
        system_prompt: target/prompt.md
        task_execution_skill: target/skills/task-execution
      max_cases: 10
  mutate:
    operator: gepa
    timeout_s: 3600
    config:
      expose_gate_data: false
      runner: harbor
      agent: codex
      model: gpt-5.4
      agent_kwargs:
        reasoning_effort: xhigh
      editable_roots: [target]
      components:
        system_prompt: target/prompt.md
        task_execution_skill: target/skills/task-execution
      component_strategy: round_robin
      max_examples: 10
  validate:
    operator: minibatch_improvement
    timeout_s: 3600
    config:
      criterion: strict
      n_concurrent: 4
  gate:
    operator: parent_eligible
    timeout_s: 600
  record:
    operator: gepa
    timeout_s: 600
evaluator:
  engine: harbor
  dataset: terminal-bench@2.0
  agent: target.agent:HarborAgent
  split:
    train: 0.562
    gate: 0.213
    sealed: 0.225
    seed: 0
  sampling: static
  tasks_per_round: 10
  n_concurrent: 4
  repetitions: 1
  anchor:
    final: true
```

B Extended Evaluation

B.1 Pairwise Execution Agreement

Table 4: Pairwise cross-mode task agreement over three repetitions. The mean D–R agreement is the arithmetic mean of the two pairwise agreements and is the value reported in Table 2. All entries are percentages.

Benchmark	Agent	Direct-A–RSIHub	Direct-B–RSIHub	Mean D–R
Terminal Bench 2	MiniSWE	84.3	85.0	84.6
Terminal Bench 2	Codex	82.4	81.6	82.0
τ^3 Banking	MiniSWE	82.1	79.7	80.9
τ^3 Banking	Codex	79.4	80.8	80.1

Results. On Terminal Bench 2, Direct-B–RSIHub agreement is 0.7 points higher for MiniSWE, whereas Direct-A–RSIHub agreement is 0.8 points higher for Codex. On τ^3 Banking, Direct-A is 2.4 points higher for MiniSWE, while Direct-B is 1.4 points higher for Codex. Neither direct replicate is uniformly closer to framework-mediated execution, which motivates averaging the two comparisons.

B.2 Runtime Overhead

Protocol. We measured the cost of the services shared across recipes on both benchmark workloads by instrumenting workspace initialization, candidate snapshotting, mutable-surface validation, evaluation recording, integrity verification, and recovery separately from model inference, benchmark execution, and mutation. For Terminal Bench 2, initialization uses 50 microbenchmark repetitions, recovery uses 30 replay trials, and the four regular-operation rows and end-to-end shares use 20 complete Gen-1 runs, five per method. Each regular-operation sample sums the relevant exclusive spans within one generation, and we compute the median and P95 over the 20 per-generation totals. The reported generation share uses the union of the framework's exclusive spans rather than summing nested durations. Because the campaigns ran on different hosts, we report their latency distributions separately.

Table 5: Framework overhead by benchmark campaign. Operation latencies are in milliseconds. On Terminal Bench 2, the framework's exclusive spans account for a median of 0.02% and a P95 of 0.06% of end-to-end generation time; on τ^3 Banking, they account for 0.019%.

	Terminal Bench 2		τ^3 Banking	
Framework operation	Median (ms)	P95 (ms)	Median (ms)	P95 (ms)
Workspace initialization and configuration freeze	155.5	368.8	1229.3	1320.4
Candidate worktree and snapshot creation	756.2	2952.3	19.2	21.6
Mutable-surface validation	27.1	43.2	13.5	16.0
Evaluation stamp and archive update	32.4	69.8	10.8	14.9
State verification and champion recomputation	652.5	1482.2	108.4	113.8
Recovery replay after interruption	1006.8	2451.9	92.3	99.4

Results. Workspace initialization was the largest measured operation in the τ^3 Banking campaign, with a median of 1.23 seconds, whereas candidate snapshot creation and state verification were the largest regular operations on Terminal Bench 2, with medians of 0.76 and 0.65 seconds. These absolute latencies should not be compared directly across the two hosts. More importantly, framework work remained small relative to complete generations: 0.019% on τ^3 Banking and a median of 0.02% on Terminal Bench 2. Model, mutation, and benchmark execution therefore dominate runtime in both measured workloads.

B.3 Method-Integration Effort

Protocol. We evaluated whether the common interfaces reduce method-specific engineering by counting normalized recipe lines, new operator lines, reused operators, changes to framework source

files, and integration tests for each adapted method. Reporting these quantities separately distinguishes declarative composition and reused services from genuinely new method logic.

Table 6: Method-integration effort under the shared recipe and operator interfaces.

Method	Recipe LOC	New operator LOC	Reused operators	Framework files changed	Integration tests
AHE	47	1087	2	0	47
Hyperagents	48	216	2	0	15
A-Evolve	57	684	2	0	14
GEPA	61	745	2	0	16

Results. At the locked experiment commit, the four adaptations required 47–61 lines of recipe configuration and reused two operator bindings each. None required a change to a framework source file, while method-specific operator implementations ranged from 216 lines for Hyperagents to 1,087 lines for AHE. Each adaptation passed its 14–47 method-specific integration tests. The separation between zero framework changes and nonzero method-specific code supports the intended extension model: research logic remains in recipes and operators while candidate execution, evaluation, recovery, and evidence services are reused.

C Inspecting an Execution

RSIHub writes a file-backed experiment record that remains readable during and after a run. Command-line summaries expose population state and integrity findings. The framework identifies the champion as the best accepted candidate. The read-only local viewer in Figure 2 presents experiment status, generation lineage, and a summary of the champion's cumulative target-tree changes. In the illustrated execution, the accepted Generation 8 candidate is displayed as the champion.

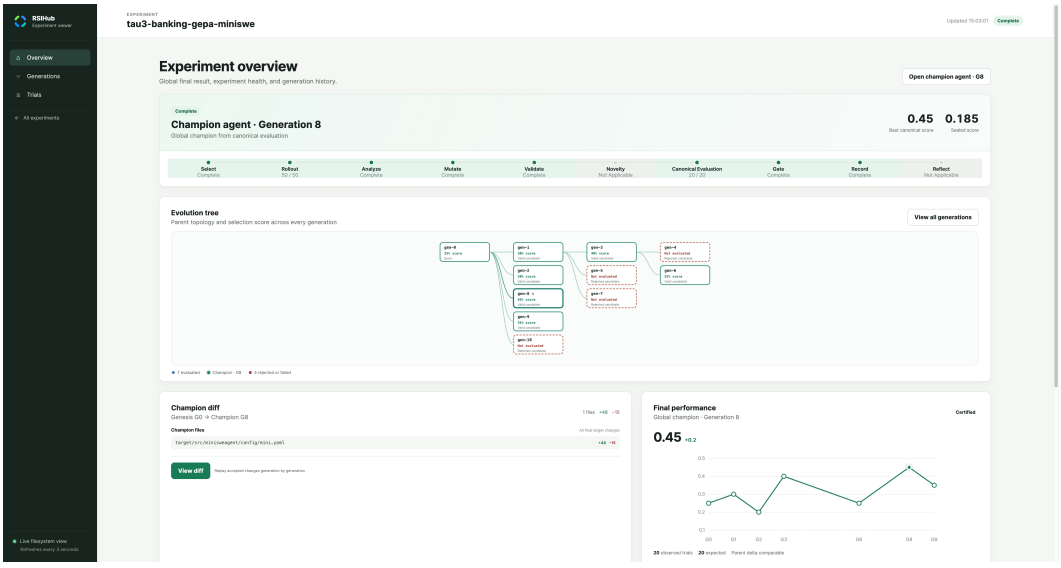


Figure 2: Inspecting one RSIHub execution. The read-only viewer identifies the accepted Generation 8 candidate as champion, shows its lifecycle status and run-specific primary-split and sealed scores, places it in the generation lineage, and summarizes its cumulative target-tree changes. The displayed values describe this execution; aggregate performance is reported in Section 4.

D Executable Contracts and Recovery Semantics

RSIHub exposes each method-controlled stage through a typed operator interface. Operator outputs are validated before the framework uses them; the framework separately enforces the mutable surface, commits the candidate, runs protected evaluation, and protects framework record fields. Table 7 summarizes the nine interfaces in execution order.

D.1 Lifecycle operators

Table 7: Lifecycle operator contracts. "Req." denotes a registry-required operator and "Opt." an operator enabled per recipe.

Stage	Mode	Primary input	Validated result and framework handling
Select	Req.	Archive view and execution context	Non-empty parent-generation list; the framework verifies eligibility and forks the recorded commit.
Rollout	Req.	Candidate checkout and context	Structured summary and artifact paths retained as mutation evidence.
Analyze	Opt.	Candidate checkout and context	Analysis summary and artifact paths bundled before mutation.
Mutate	Req.	Checkout, observation text, and context	Changed paths, notes, and usage metadata; the framework checks the actual tree diff against the declared surface.
Validate	Opt.	Mutated checkout and context	Acceptance flag, reason, and artifacts; rejection stops before protected evaluation.
Novelty	Opt.	Uncommitted checkout and context	Finite novelty score and acceptance flag; rejection stops before commit.
Gate	Req.	Evaluated child row, parent row, and context	`accept`/`reject` decision and reason; the framework keeps the score fields.
Record	Req.	Terminal child row and context	Auxiliary annotation fields filtered against protected record fields.
Reflect	Opt.	Updated archive and context	Validated append-only playbook operations; failure does not unwind the recorded generation.

D.2 Evaluation identity and parent selection

In strict-contract mode, an evaluation's contract accompanies its scalar score. The contract binds the exact candidate to the evaluator, data, sampling, runtime, and model conditions in Table 8; its `contract_id` is the canonical digest of the complete payload. Parent selection combines the explicitly listed fixed-identity projection with integrity and lifecycle checks; the full contract remains attached to each attempt for provenance.

Table 8: Evaluation identity and the parent-selection predicate.

Layer	Recorded identity	Role
Candidate	Candidate commit and tree, generation, and optional dependency digest	Binds evidence to the evaluated source and relevant candidate dependencies.
Evaluator	Frozen evaluator tree, semantic configuration digest, and framework version	Makes evaluator changes visible in per-attempt provenance.
Data and trials	Dataset-content and task-set digests, ordered task members, split, repetitions, seed namespace, and trial identities	Makes task membership and multiplicity explicit.
Runtime and model	Runtime digest, agent and model names, endpoint digest, concurrency, and retry policy	Records execution conditions that can affect evidence collection.
Fixed comparison identity	Evaluator-tree fingerprint, task-set hash, and runtime fingerprint	Defines the cross-generation projection that a selectable parent must match.
Integrity receipt	Framework marker and hash receipt for the serialized evaluation event	Flags an event when its framework's receipt is absent or no longer matches.
Parent eligibility	Complete candidate or genesis evaluation; selection-eligible; no pending Gate decision; valid-parent and `keep`; certified receipt; matching fixed identity; finite score; and generation tag matching the recorded commit	Excludes incomplete, rejected, uncertified, incomparable, non-finite, and retagged candidates.

Sealed evaluation runs as a separate, non-selectable anchor. The implementation stores it as auxiliary evidence, excludes it from the default mutation-feedback bundle, and keeps its `anchor` purpose outside the candidate-or-genesis parent predicate.

D.3 Recovery checkpoints

The archive is a recovery boundary: a complete certified candidate evaluation is appended before Gate and remains pending until Gate records its decision. A pending child is therefore unavailable to Select. Record annotations run after Gate and are filtered separately; Record completion is outside the parent-eligibility predicate. Table 9 summarizes the implemented restart behavior.

Table 9: Checkpoint and recovery behavior.

Observed state	Recovery action	Preserved property
Active child worktree at run start	Stop and require the operator to preserve edits or invoke repair.	The in-progress candidate remains untouched.
Stale managed worktree	Repair removes only clean worktrees; dirty and externally managed worktrees are preserved and reported.	Cleanup is conservative with uncommitted work.
Generation tag without a certified evaluation	Evaluate the existing tagged commit instead of repeating selection and mutation.	Resumption retains the created candidate's identity.
Complete evaluation pending Gate	Resume Gate from the tagged candidate, then attempt terminal Record; keep the row ineligible until the Gate decision.	Parent selection waits for Gate finalization.
Interrupted or terminally failed evaluation	Append available attempt evidence; require an explicit retry to create a new certified attempt.	Failed attempts remain auditable and retries are operator-controlled.
Terminal Record already attempted	Skip a duplicate Record invocation on recovery.	Each terminal Record is invoked at most once.