BENEATH THE SURFACE: WHAT WE OVERLOOK WHEN WE TALK ABOUT AGENT EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent harness evolution enables recursive self-improvement (RSI) without updating model weights. We examine whether higher benchmark scores yield reliable, efficient, and transferable improvement. We evaluate four methods across harnesses, general and coding tasks. Repeated evaluations, controlled agent expansion, component interventions, and admission replays reveal recurring patterns. All sixteen evolved agents gain previously uncovered tasks while losing previously covered ones. In one general-task case, reliability rises by 5.2 points despite losing ten previously covered tasks. Most reliability gains also increase deployment cost. Specific changes usually outperform generic reminders, but their benefits can change when combined. The highest-scoring agent does not yield the largest mean improvement in any of our four pools, and stricter admission can discard useful alternatives. Transfer selectively preserves gains: coverage usually falls on the general-task target across both harness types and paradigms, but rises on the coding target with the outer-layer-only harness. These findings motivate preservation-aware evaluation, tested decision rules, and explicit applicability conditions. As a constructive follow-up, EvoAudit turns these diagnostics into revisions of *select*, *analyze*, *mutate*, and *gate*, achieving the best performance. Our code is available at `https://anonymous.4open.science/r/evolution`.

1 INTRODUCTION

Recursive self-improvement (RSI) studies agents that iteratively revise their own designs based on experience (Zelikman et al., 2024; Hu et al., 2025; Zhang et al., 2026b). Agent harness evolution offers a concrete form of this idea for language-model agents: it keeps model weights fixed while persistently changing the surrounding system that shapes observations and turns model outputs into actions (Lin et al., 2026a; Zhang et al., 2026c). Depending on the harness, these changes can target prompts, skills, tool interfaces, orchestration, or source code, going beyond static prompting and memory-based adaptation (Shinn et al., 2023; Wang et al., 2024). Recent frameworks close the loop between execution, diagnosis, mutation, and evaluation, reporting gains on standard benchmarks (Hu et al., 2025; Shang et al., 2025; Zhang et al., 2025a). These benchmark gains suggest that harness evolution is a promising avenue for pursuing RSI.

However, a higher overall benchmark score do not mean a more useful agent. It averages over tasks and trials, so newly solved tasks can conceal lost ones, while deployment cost and performance beyond the tasks that guided evolution remain unseen (Gao et al., 2026; Wang et al., 2026; Kapoor et al., 2025). Relying on that score alone can favor brittle or expensive agent versions and mistake adaptation to the tasks used during evolution for general improvement. Meaningful progress instead requires gains that are reliable across tasks, affordable to deploy, and effective on tasks that did not guide evolution. We aim to explain which persistent harness changes produce useful gains and how evolution generates and preserves them.

We organize our empirical investigation around three research questions:

- **RQ1 (Phenomenology):** Does agent evolution yield reliable and efficient improvements, or does it suffer from task-coverage trade-offs and hidden costs?
- **RQ2 (Generalization):** Do the gains achieved during evolving survive when the agent is deployed in novel environments or tasks?

- **RQ3 (Mechanisms):** How do specific evolutionary operators contribute to the generation and preservation of beneficial modifications?

We study AHE (Lin et al., 2026a), HyperAgents (Zhang et al., 2026c), A-Evolve (Lin et al., 2026b), and GEPA (Agrawal et al., 2026) with MiniSWE (Yang et al., 2024) and Codex (OpenAI, 2026) on τ^3-Banking (Shi et al., 2026) for general tasks and Terminal-Bench 2 (Merrill et al., 2026) for coding tasks. Repeated evaluations compare evolved and initial agents in reliability, empirical coverage, regressions, and deployment cost. Split and cross-benchmark comparisons assess generalization, while controlled expansion of existing agents, change interventions, and admission replays trace how gains arise and persist.

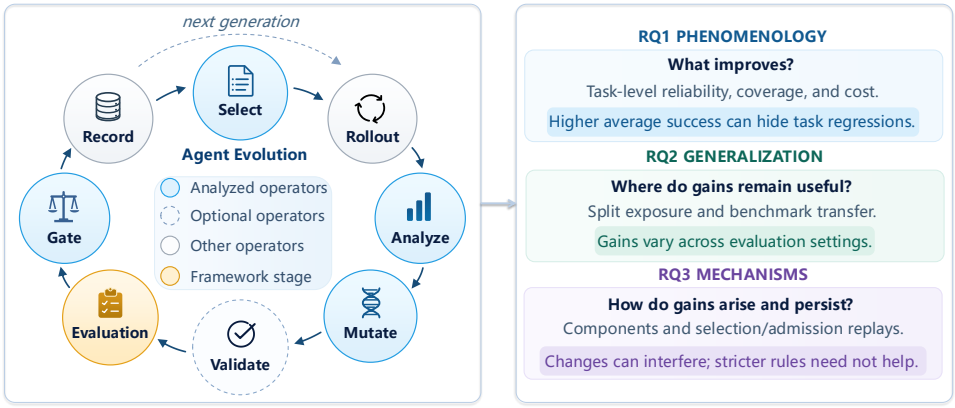


Figure 1: The agent harness evolution procedure (left) and our analyses of Phenomenology, Generalization, and Mechanisms (right).

Harness evolution redistributes task success, with new coverage accompanied by regressions across harnesses, task domains, and evolution paradigms. The practical value of these gains depends on deployment cost and whether they persist beyond the tasks that guided evolution. Specific changes usually outperform generic reminders, yet combining them can alter their value. The strongest existing agent need not offer the greatest further improvement, and stricter admission can discard useful alternatives. Useful improvement therefore requires productive starting points, compatible changes, and search decisions that preserve gains.

We connect these findings to operator redesign as a constructive follow-up. Analyses of Phenomenology, Generalization, and Mechanisms jointly guide revision hypotheses for *select*, *analyze*, *mutate*, and *gate*. Each hypothesis specifies its supporting evidence, expected effects, and applicability conditions. We combine these revisions into *Audit-Guided Evolution* (AGE) to test their practical value.

Our contributions can be summarized as:

- **Revisiting Practical Improvement.** We reveal a gap between benchmark gains and practical utility, showing that useful improvement in agent harness evolution requires preserving useful behavior, managing deployment cost, and sustaining gains beyond the tasks that guide evolution.
- **Understanding the Fragility of Gains.** We identify productive agent selection, change compatibility, and search retention as central challenges. Current performance can misrepresent further improvement potential, useful changes can interfere, and admission can discard stronger versions.
- **Improving Evolution Through Analysis.** We translate these findings into coordinated revisions of diagnosis, mutation, selection, and admission. The resulting AGE algorithm improves reliability and coverage over its reference method across both benchmarks and harnesses.

2 Preliminaries

2.1 Definitions

An agent $A = (M, H)$ consists of a frozen model M and a harness H. Following Lin et al. (2026a); Agrawal et al. (2026), an evolution algorithm E starts from a *Seed* $A_0 = (M, H_0)$ and runs for G

generations. At generation g, a *Parent* is an eligible existing agent selected for modification, and $\mathcal{C}_g$ contains the resulting proposed harnesses. Each $H \in \mathcal{C}_g$ defines a *Candidate* agent (M, H). The final *Champion* is $A_\star = (M, H_\star)$:

$$\mathcal{C}_g = E(H_{<g}, \mathcal{Z}_{<g}, \mathcal{D}_{\text{evo}}; B_{\text{evo}}), \quad H_\star = Q\left(\{H_0\} \cup \bigcup_{g=1}^{G} \mathcal{C}_g\right), \tag{1}$$

where $H_{<g}$ is the history of harness versions before generation g, $\mathcal{Z}_{<g}$ is their evaluation evidence, $\mathcal{D}_{\text{evo}}$ is the task set available during evolution, and B_{evo} is the search budget. The rule Q selects the final Champion harness from the archived versions.

2.2 Experiment Setup

2.2.1 Evolution Process and Methods

Methods and Harnesses. We evaluate **AHE** (Lin et al., 2026a), **HyperAgents** (Zhang et al., 2026c), **A-Evolve** (Lin et al., 2026b), and **GEPA** (Agrawal et al., 2026) using **GPT-5.4**. AHE and HyperAgents are *implementation-centric* recipes that emphasize agent engineering, whereas A-Evolve and GEPA are *reflection-centric* recipes that distill execution feedback into reusable changes. We run each recipe with two harnesses. **MiniSWE 2.4.5** (Yang et al., 2024) is *source-editable* and permits inner-source changes. **Codex 0.143.0** (OpenAI, 2026) is *outer-layer-only* and permits changes to prompts, skills, configuration, and external orchestration. These combinations let us examine different evolution designs under a common process. Appendix B details their implementations.

In this common process, each recipe specifies how seven operators are implemented: *select* chooses a Parent, *rollout* collects execution evidence, *analyze* diagnoses it, and *mutate* proposes a Candidate. The *validate* operator optionally checks the proposal, *gate* decides admission, and *record* archives outcomes. We focus the mechanism analysis on *select*, *analyze*, *mutate*, and *gate* because their decisions determine which Parents are used, how evidence guides changes, and which Candidates persist.

Figure 1 connects this operator process to the study's empirical comparisons. The Phenomenology analysis characterizes gains and regressions through repeated task evaluations. The Generalization analysis examines how these gains vary across original splits and external benchmarks. The Mechanisms analysis examines Parent improvement potential under a common generation process, Component interactions, and admission decisions on frozen pools.

2.2.2 Benchmarks and Evaluation Metrics

Benchmarks. General tasks are represented by τ^3**-Banking** (Shi et al., 2026), with 97 tasks, and coding tasks by **Terminal-Bench 2** (Merrill et al., 2026), with 89 tasks. Both are split into Train/Validation/Test. Evolution uses Train/Validation evidence for search and admission, while Test is reserved for Seed and Champion evaluation.

Performance. Let $\mathcal{D}$ be an evaluation task set with $N = |\mathcal{D}|$ tasks, each executed once per pass over k passes. The outcome $Y_{ni}(A) \in \{0, 1\}$ indicates whether agent A succeeds on task n in pass i. With $\mathbb{I}[\cdot]$ an indicator function, *Reliability* (Avg@k) and *Coverage* (Pass@k) are:

$$\text{Avg@}k = \frac{1}{Nk} \sum_{n=1}^{N} \sum_{i=1}^{k} Y_{ni}(A), \quad \text{Pass@}k = \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}\left[\max_{1 \le i \le k} Y_{ni}(A) = 1\right]. \tag{2}$$

Task turnover counts coverage gains and losses relative to Seed. $\text{Std@}k$ is the population standard deviation of per-pass success rates.

Cost. Each model call q costs $c(q)$. A task execution t costs c_t, the sum of these charges over the agent's calls. Evolution cost C_{evo} sums costs over the set of evolution calls $\mathcal{L}_{\text{evo}}$. For deployment, $\mathcal{T}_k$ contains executions of A on $\mathcal{D}$ with recorded costs in the first k passes. Their mean cost is $\bar{c}^{(k)}_{\text{dep}}$. Multiplying it by k gives the estimated per-task deployment cost $C_{\text{dep}}@k$:

$$C_{\text{evo}} = \sum_{q \in \mathcal{L}_{\text{evo}}} c(q), \quad \bar{c}^{(k)}_{\text{dep}}(A; \mathcal{D}) = \frac{1}{|\mathcal{T}_k|} \sum_{t \in \mathcal{T}_k} c_t, \quad C_{\text{dep}}@k(A; \mathcal{D}) = k\, \bar{c}^{(k)}_{\text{dep}}(A; \mathcal{D}). \tag{3}$$

3 PHENOMENOLOGY: REVISITING WHAT HAS BEEN EVOLVED

In this section, we compare each frozen Champion with its matching Seed in reliability, empirical coverage, task-level transitions, and deployment cost. Table 1 reports eight-pass aggregate results, Figure 2 shows paired task-level success counts, and Figure 3 relates Pass@k to C_{dep}@k for $k = 1, 2, 4, 8$ using the same canonical trials. Appendix C provides detailed comparisons and stability checks.

3.1 OVERALL RELIABILITY AND EMPIRICAL TASK COVERAGE

Table 1: Reliability and empirical coverage across eight passes. Generation 0 is the Seed; later entries give the selected Champion's Generation. Scores are percentages and deltas are percentage points.

Benchmark	Harness	Variant	Generation	Avg@8 ± Std	ΔAvg	Pass@8	ΔPass
τ^3-Banking	MiniSWE	Seed	0	27.4±3.1	–	54.6	–
		AHE	2	32.9±3.9	+5.4	66.0	+11.3
		HyperAgents	9	32.6±4.1	+5.2	57.7	+3.1
		A-Evolve	1	26.3±2.5	−1.2	55.7	+1.0
		GEPA	11	28.2±2.3	+0.8	58.8	+4.1
	Codex	Seed	0	30.3±4.3	–	56.7	–
		AHE	15	36.2±3.4	+5.9	62.9	+6.2
		HyperAgents	18	36.9±2.5	+6.6	66.0	+9.3
		A-Evolve	1	28.1±2.9	−2.2	57.7	+1.0
		GEPA	5	29.5±3.2	−0.8	53.6	−3.1
Terminal-Bench 2	MiniSWE	Seed	0	65.9±4.8	–	86.5	–
		AHE	18	70.2±2.3	+4.4	87.6	+1.1
		HyperAgents	17	67.7±3.2	+1.8	85.4	−1.1
		A-Evolve	4	68.0±2.9	+2.1	86.5	0.0
		GEPA	20	68.5±3.5	+2.7	88.8	+2.2
	Codex	Seed	0	69.9±2.3	–	89.9	–
		AHE	20	72.1±3.4	+2.1	91.0	+1.1
		HyperAgents	18	72.6±3.9	+2.7	86.5	−3.4
		A-Evolve	2	68.0±1.7	−2.0	87.6	−2.2
		GEPA	15	68.1±4.7	−1.8	86.5	−3.4

Aggregate Improvement Is Not a Single Outcome. Table 1 shows reliability and coverage gains in eleven and ten of sixteen lineages, respectively. Four move the metrics in opposite directions across both harnesses, domains, and paradigms; another improves only reliability. Consistency and empirical reach are therefore distinct outcomes that aggregate improvement claims should separate.

3.2 COVERAGE IS A FLOW

Evolution Redistributes Success. All sixteen Champions in Figure 2 gain and lose covered tasks, and increase repeated success on some tasks while decreasing it on others. This pattern spans both harnesses, domains, and paradigms: even when both aggregate metrics improve, coverage does not expand monotonically. Gains and losses should therefore be reported separately. Appendix C.2 gives further analysis.

Behavioral Changes Behind Task-Level Gains and Losses. We inspect contrasting traces for tasks with large success-count changes. Across the sampled harnesses, domains, and paradigms, recurring differences concern state preservation, precise grounding, and task completion. Successful traces preserve recovery files and respect action dependencies, while failures include incorrect arguments, premature termination, and solutions that work locally but fail in the required environment. These contrasts motivate preservation and end-to-end checks rather than equating every outcome flip with acquired or lost task knowledge. Appendix C.3 provides the categories and case evidence.

3.3 RELIABILITY AND DEPLOYMENT EFFICIENCY

Per-Trial Cost and Deployment Efficiency Can Diverge. Eight of eleven reliability-improving Champions cost more per trial than Seed; the three lower-cost gains use the source-editable harness on coding tasks. Yet Figure 3 shows that higher per-trial cost can coexist with fewer attempts and lower cost to reach comparable coverage. MiniSWE with AHE on τ^3-Banking reaches 56.7% coverage in

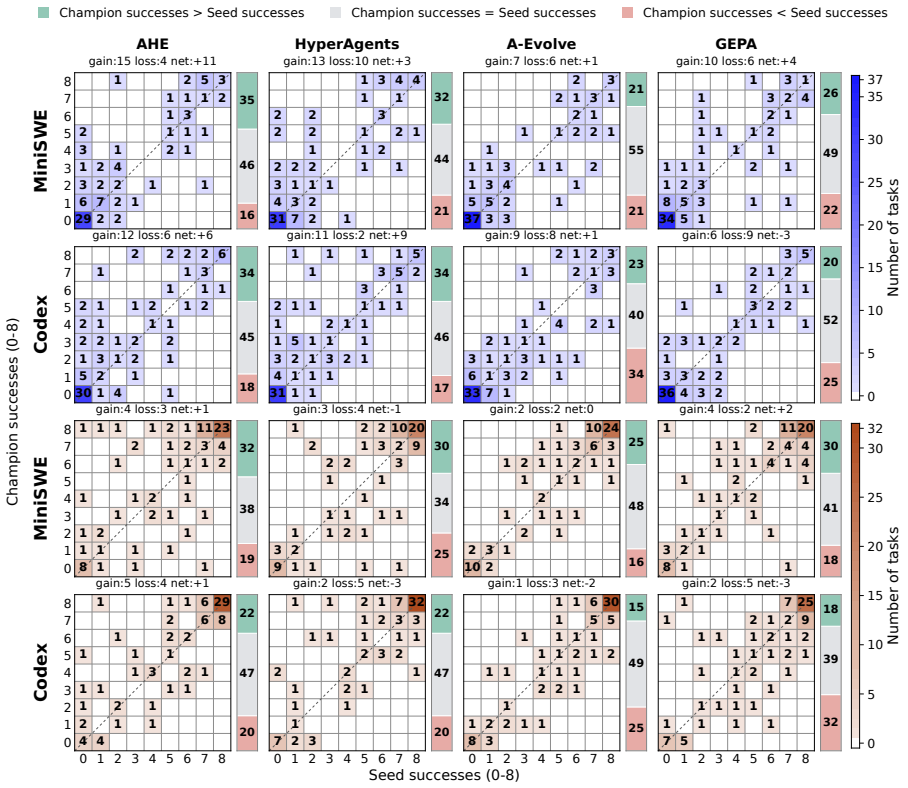


Figure 2: Task-level Seed-to-Champion transitions on τ^3-Banking (blue) and Terminal-Bench 2 (orange). Each cell reports the number of tasks with the corresponding Seed and Champion success counts. Right-hand bars group tasks by whether the Champion succeeds more than, the same as, or less than the Seed. Non-origin cells on the left or bottom edges indicate gained or lost coverage.

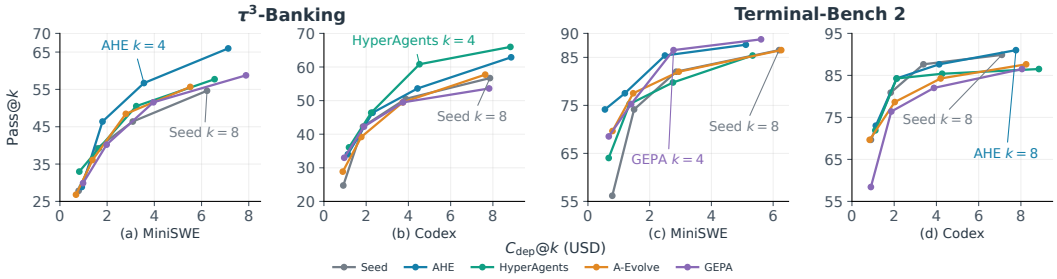


Figure 3: Coverage–cost curves for MiniSWE and Codex on both benchmarks. Points pair $\mathrm{Pass@}k$ with $C_{\mathrm{dep}}@k$ from Equation 3 for $k = 1, 2, 4, 8$.

four passes, exceeding Seed's 54.6% in eight at lower deployment cost. Practical efficiency therefore depends on both cost per attempt and the attempts needed to attain a given coverage level.

4 Generalization: Regularizing The Benefits Of Evolution Within A Controllable Boundary

Table 2 compares frozen Champions with matching Seeds across evolution splits and external transfer. Train, Validation, and Test report contributions to the full source-benchmark gain, summing to the overall change before rounding. Transfer reports gains measured on each external benchmark relative to its matching Seed. Appendix D gives protocols and interpretation.

Table 2: Seed-relative gains across evolution splits and external transfer in percentage points. Darker cells indicate larger gains on a shared linear scale.

Benchmark	Harness	Method	ΔPass@8				ΔAvg@8			
			From evolution data to OOD transfer →				From evolution data to OOD transfer →			
			Train	Validation	Test	Transfer	Train	Validation	Test	Transfer
τ^3-Banking	MiniSWE	AHE	+3.1	+4.1	+4.1	−0.7	+3.4	+0.9	+1.2	+2.7
		HyperAgents	+2.1	+1.0	0.0	−0.4	+3.6	+1.7	−0.1	+0.8
		A-Evolve	0.0	+1.0	0.0	−0.3	−1.7	+0.4	+0.1	+1.4
		GEPA	+1.0	+2.1	+1.0	+1.5	+0.4	+0.5	−0.1	+2.6
	Codex	AHE	+6.2	0.0	0.0	−0.7	+3.2	+1.2	+1.5	−1.7
		HyperAgents	+8.2	+1.0	0.0	−0.7	+4.1	+1.9	+0.5	+0.3
		A-Evolve	+1.0	+2.1	−2.1	−1.0	−1.4	−0.3	−0.5	−2.0
		GEPA	−1.0	+1.0	−3.1	−1.0	−0.9	+0.3	−0.1	−1.4
Terminal-Bench 2	MiniSWE	AHE	+1.1	−1.1	+1.1	+1.8	+2.9	−0.4	+1.8	−2.3
		HyperAgents	−2.2	0.0	+1.1	+3.5	+0.3	+0.8	+0.7	−2.1
		A-Evolve	0.0	−1.1	+1.1	+4.4	+1.5	−0.4	+1.0	−0.2
		GEPA	+1.1	−1.1	+2.2	−2.7	+1.1	+0.1	+1.4	−1.7
	Codex	AHE	+2.2	−2.2	+1.1	+8.0	+2.9	−1.4	+0.6	−0.1
		HyperAgents	0.0	−2.2	−1.1	+6.2	+3.7	−1.7	+0.7	+3.1
		A-Evolve	−1.1	−1.1	0.0	+1.8	0.0	−1.0	−1.0	−2.1
		GEPA	−1.1	−2.2	0.0	+3.5	+0.8	−1.1	−1.5	+1.4

4.1 Generalization Within the Same Benchmarks

Gains Often Survive Beyond Direct Feedback. Test reliability improves in ten of sixteen lineages, including seven of eight implementation-centric cases across both harness types and task domains. Reflection-centric gains are more harness-dependent: most source-editable cases improve, whereas all outer-layer-only cases decline. Improvement therefore frequently reaches feedback-excluded tasks, with systematic differences across harness–paradigm combinations.

4.2 Generalization Cross Benchmarks

We test general-task transfer from τ^3-Banking to the full τ^2-bench (Barres et al., 2026), and coding-task transfer from Terminal-Bench 2 to DeepSWE v1.1 (Huang et al., 2026). Table 2 gives Seed-relative gains and Appendix Table 14 gives absolute scores.

Transfer Selectively Preserves Gains. Coverage falls for seven of eight Champions on the general-task target but rises for seven of eight on the coding target, spanning both harness types and paradigms. Reliability can move in the opposite direction. All source-editable Champions improve general-task transfer reliability but lose coding-task transfer reliability, despite coverage gains in three coding cases. Transfer therefore changes the balance between reliability and coverage, so the persistence of improvement must be assessed separately for each outcome and target.

5 Mechanisms: Tracing How to Discover And Retain The Optimization Gains

5.1 Select: Choosing Which Agent to Improve

We test whether *select* identifies Parents with the greatest improvement potential. Each Champion produces three Candidates through the same A-Evolve analysis and mutation process.

Each benchmark and harness supplies one four-Parent pool, with one Train pass shared by each Parent's three branches. The best Parent maximizes mean Candidate Test Avg@8 minus its own. Table 3 reports its selection probability. Original averages the native HyperAgents,

Table 3: Best Parent selection rates (%). Higher values are bold.

Benchmark	Harness	Original	Random
τ^3-Banking	MiniSWE	17.5	**25.0**
	Codex	8.8	**25.0**
Terminal-Bench 2	MiniSWE	**59.5**	25.0
	Codex	**55.7**	25.0

A-Evolve, and GEPA rules using Train or Validation evidence; Random samples uniformly. Appendix E.1 gives the protocol.

Current Scores Do Not Consistently Identify Improvement Potential. Original falls below the 25% random baseline on general tasks but exceeds 55% on coding tasks, across both harnesses. This reversal shows that selection signals align with further improvement differently across domains. Evaluating *select* therefore requires measuring subsequent gains alongside current performance.

5.2 Analyze: Identifying What the Evidence Supports

Tables 4 and 5 inform *analyze* by comparing eight Champions' *Components*, functionally related changes with their dependencies, against *generic reminders* such as checking tools and verifying results. Appendix E.5 defines each Component, and Appendix E.2 gives the protocol.

Table 4: Isolated Components versus complete Champions. Appendix E.5 defines each row's A and B. Scores are percentages. Best values are bold.

Benchmark	Harness	Method	Avg@8 (%)				Pass@8 (%)			
			Seed	Seed + A	Seed + B	Champion	Seed	Seed + A	Seed + B	Champion
τ^3-Banking	MiniSWE	AHE	27.4	28.2	32.1	**32.9**	54.6	53.6	64.9	**66.0**
		HyperAgents		**34.4**	32.0	32.6		**59.8**	55.7	57.7
	Codex	AHE	30.3	29.6	34.8	**36.2**	56.7	56.7	**63.9**	62.9
		HyperAgents		**37.2**	36.5	36.9		67.0	**72.2**	66.0
Terminal-Bench 2	MiniSWE	AHE	65.9	67.7	69.1	**70.2**	86.5	**88.8**	87.6	87.6
		GEPA	65.9	67.3	64.0	**68.5**	86.5	87.6	86.5	**88.8**
	Codex	AHE	69.9	70.2	69.9	**72.1**	89.9	85.4	86.5	**91.0**
		HyperAgents	69.9	69.1	68.0	**72.6**	**89.9**	**89.9**	86.5	86.5

Table 5: Seed versus Seed with a generic reminder. Scores are percentages.

Benchmark	Harness	Avg@8 (%)		Pass@8 (%)	
		Seed	Generic reminder	Seed	Generic reminder
τ^3-Banking	MiniSWE	27.4	**27.8**	54.6	**55.7**
	Codex	**30.3**	28.6	**56.7**	**56.7**
Terminal-Bench 2	MiniSWE	**65.9**	**65.9**	**86.5**	84.3
	Codex	**69.9**	65.7	**89.9**	88.8

Table 6: Size-normalized best-Candidate retention (%), averaged over four methods.

Benchmark	Harness	Retention (%)	
		Original	Constrained
τ^3-Banking	MiniSWE	**31.3**	25.0
	Codex	18.8	**25.0**
Terminal-Bench 2	MiniSWE	**25.0**	12.5
	Codex	43.8	**50.0**

Specific Changes Outperform Generic Guidance. Fifteen of sixteen Components outperform their matching reminders in reliability, with at least one in every case across both harnesses and domains (Tables 4–5). This motivates trace-grounded diagnoses that identify what to change and preserve.

5.3 Mutate: Preserving Useful Behavior While Changing the Agent

Mutation Quality Depends on How Changes Combine. Six of eight Champions outperform either isolated Component in reliability, including all coding-task cases (Table 4). Two general-task cases, one per harness, favor a subset. In one coding case, only the combination improves over Seed. These patterns motivate *mutate* to preserve dependencies and check harmful interactions and useful complementarities in the complete Candidate. Appendix E.3 reports interaction estimates.

5.4 Gate: Deciding Which Changes to Retain

We compare original *gate* rules with a constraint requiring higher mean success and incumbent-relative mean regression ≤ 0.05. A retained set $\mathcal{R}$ scores $1/|\mathcal{R}|$ if it contains the reporting-panel best

Candidate, and zero otherwise. Scores average four method-specific pools per harness; reporting outcomes remain hidden from decisions.

Admission Constraints Have Harness-Dependent Effects. Constrained admission reduces best-Candidate retention from fifteen to six pools. Yet its size-normalized score falls for the source-editable harness and rises for the outer-layer-only harness on both domains (Table 6). The increase reflects smaller sets retaining a best Seed alone. Admission quality therefore requires inspecting preservation, set size, and Candidate identity together. Appendix E.4 gives per-pool results.

6 Recipes: From Empirical Findings to Operator Design

Table 7 connects the findings to operator design.

Table 7: Cross-experiment findings and the design hypotheses they motivate.

Integrated Finding	Supporting Evidence	Design Implication
Gains Involve Trade-offs.	New coverage accompanies regressions and higher cost (Table 1, Figures 2–3).	Consider gains, regressions, and cost in *select* and *gate*.
Gains Are Context-Dependent.	Benefits vary with exposure and transfer target (Table 2).	Specify applicability in *analyze* and *mutate*.
Generation and Retention Shape Gains.	Current scores, combinations, and admission can mislead search (Tables 3–6).	Check subsequent gains, trace evidence, compatibility, and retained alternatives.

6.1 From Synthesis to Operator Revisions

We propose *EvoAudit*, an agent-assisted workflow that runs diagnostics on archived artifacts and returns operator revisions with evidence and validation records (Figure 4).

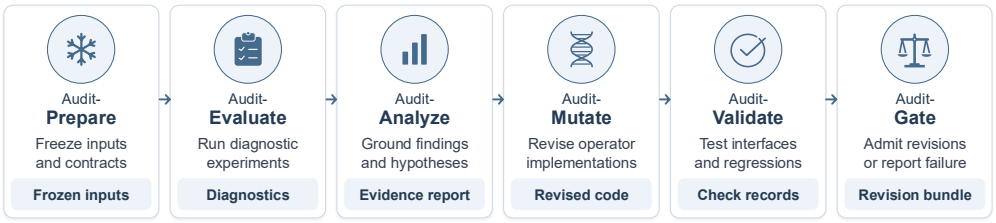


Figure 4: EvoAudit diagnoses archived evolution runs and returns validated operator revisions.

Installing its four revisions in A-Evolve gives *Audit-Guided Evolution* (AGE). Other operators, model and evaluator contracts, and harness editing constraints remain fixed. Appendix F.2 details the implementation.

6.2 Human Evaluation of the Operator Revisions

Eight experts reviewed all four operator revisions. Each chose a direction before seeing the proposal, then judged its evidence, whether it addressed the stated problem, and whether it should enter validation. Table 8 reports the mean number of experts per revision. Overall, expert judgments broadly align with EvoAudit's proposed directions and support validating the resulting revisions. Appendix F.3 details the protocol.

Table 8: Expert review of four revisions.

Measure	Mean / 8
Direction agreement	6/8
Evidence support	6.25/8
Problem addressing	6.5/8
Proposal endorsement	6.75/8

6.3 Comparison with Existing Evolution Methods

Table 9 compares AGE with the four methods across both benchmarks and harnesses.

Table 9: Comparison of AGE with existing evolution methods. Score changes are relative to the Seed. Costs follow Equation 3, with $k = 8$ for deployment and 20 generations for evolution.

Benchmark	Harness	Variant	Reliability		Coverage		Task Turnover		Cost	
			Avg@8	ΔAvg	Pass@8	ΔPass	Gained	Lost	Deployment	Evolution
τ^3-Banking	MiniSWE	Seed	27.4	–	54.6	–	–	–	6.2	–
		AHE	32.9	+5.4	**66.0**	+11.3	**15**	4	7.1	1235.0
		HyperAgents	32.6	+5.2	57.7	+3.1	13	10	6.6	867.0
		A-Evolve	26.3	−1.2	55.7	+1.0	7	6	5.5	1057.6
		GEPA	28.2	+0.8	58.8	+4.1	10	6	7.9	2262.8
		AGE	**34.0**	+6.6	**66.0**	+11.3	13	**2**	6.1	1070.6
	Codex	Seed	30.3	–	56.7	–	–	–	7.9	–
		AHE	36.2	+5.9	62.9	+6.2	12	6	8.9	1385.1
		HyperAgents	36.9	+6.6	66.0	+9.3	11	**2**	8.8	1175.8
		A-Evolve	28.1	−2.2	57.7	+1.0	9	8	7.6	1523.6
		GEPA	29.5	−0.8	53.6	−3.1	6	9	7.8	2372.8
		AGE	**38.1**	+7.9	**67.0**	+10.3	**15**	5	8.9	1480.8
Terminal-Bench 2	MiniSWE	Seed	65.9	–	86.5	–	–	–	6.2	–
		AHE	70.2	+4.4	87.6	+1.1	4	3	5.1	853.4
		HyperAgents	67.7	+1.8	85.4	−1.1	3	4	5.3	723.4
		A-Evolve	68.0	+2.1	86.5	0.0	2	**2**	6.3	983.9
		GEPA	68.5	+2.7	88.8	+2.2	4	**2**	5.6	1629.0
		AGE	**71.8**	+5.9	**89.9**	+3.4	**5**	**2**	5.2	982.7
	Codex	Seed	69.9	–	89.9	–	–	–	7.1	–
		AHE	72.1	+2.1	91.0	+1.1	**5**	4	7.8	1065.5
		HyperAgents	72.6	+2.7	86.5	−3.4	2	5	8.9	987.2
		A-Evolve	68.0	−2.0	87.6	−2.2	1	3	8.2	1254.1
		GEPA	68.1	−1.8	86.5	−3.4	2	5	8.1	1939.2
		AGE	**73.5**	+3.5	**92.1**	+2.2	3	**1**	6.8	1160.5

Combined Revisions Improve Outcomes Across Settings. AGE improves reliability and coverage over A-Evolve in all four settings and loses fewer Seed-covered tasks in three, tying in the fourth. This consistency across domains and harnesses supports the practical value of evidence-guided revision.

7 Discussion

Feedback Can Be Narrower Than a Change's Effects. Harness changes affect many behaviors, while evaluation samples finite executions and compresses their trade-offs. A prompt, skill, or execution routine can help the tasks that supply feedback while disrupting behavior elsewhere. This mismatch offers a common explanation for task regressions, context-dependent change interactions, and selective transfer. It suggests that practical evolution depends on how well feedback represents the consequences of a change, including the resources needed to deploy it.

Evidence Must Match the Decision. Current performance, improvement potential, change compatibility, and retention are distinct objectives. A high-scoring Parent need not yield the greatest subsequent gain, a useful Component can interfere with other changes, and a conservative gate can discard alternatives. Together, the diagnostics show why evidence supporting one decision need not support the next. Evaluation should test the desired consequence of each decision, rather than reuse the same aggregate score throughout evolution.

Toward Decision-Specific Feedback. Evaluate *select* through subsequent gains, ground *analyze* and *mutate* in traces and dependencies, and assess *gate* through both retained quality and preserved alternatives. Deployment cost and external evaluation then determine how broadly these benefits remain useful. EvoAudit and AGE illustrate how these complementary checks can guide a coordinated redesign while preserving the reference method's execution and evaluation contracts.

8 Conclusion

Our study shows that harness evolution redistributes observed task success, bringing both gains and regressions. Across harnesses, task domains, and evolution paradigms, practical improvement depends on productive Parents, compatible changes, preservation through search, and generalization at affordable cost. EvoAudit connects these findings to operator revisions, and AGE improves reliability and coverage over its reference method in all four settings. These results motivate evaluating evolution by the useful behavior it creates and sustains, beyond final benchmark scores.

AI USE STATEMENT

We used AI tools for language polishing, grammar checking, and improving the clarity, conciseness, and overall readability of the manuscript. All AI-assisted revisions were reviewed by the authors.

ETHICS STATEMENT

Experiments run in isolated benchmark environments rather than live user-facing services. Evolving agent instructions and code can introduce regressions and unintended tool use. We retain fixed evaluation contracts and harness-specific editing permissions, and examine regressions and transfer alongside aggregate performance. Participants in the human evaluation provided informed consent. The expert-review protocol evaluates the evidence, reasoning, and proposed approaches for operator revisions, as described in Appendix F.3. Neither benchmark gains nor expert endorsement establishes safety for deployment.

REPRODUCIBILITY STATEMENT

Sections 2.2.1 and 2.2.2 specify the experimental settings, and the appendices detail recipe implementations, Component construction, diagnostic comparisons, and transfer protocols. Archived artifacts link evaluated agent versions to task manifests, split assignments, model and evaluator configurations, pass identifiers, and execution records.

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, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations, ICLR 2026, Rio de Janeiro, Brazil, April 23-27, 2026*. proceedings.iclr.cc, 2026.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik R Narasimhan. $\tauˆ2$-bench: Evaluating conversational agents in a dual-control environment. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=OC2z7iSQKa.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, Yafeng Deng, and Teng Li. Evoagentbench: Benchmarking agent self-evolution via ability transfer. *CoRR*, abs/2607.05202, 2026. doi: 10.48550/ARXIV.2607.05202. URL https://doi.org/10.48550/arXiv.2607.05202.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *CoRR*, abs/2607.07946, 2026. doi: 10.48550/ARXIV.2607.07946. URL https://doi.org/10.48550/arXiv.2607.07946.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Trans. Mach. Learn. Res.*, 2025, 2025. URL https://openreview.net/forum?id=Zy4uFzMviZ.

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 state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak N. Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems. *CoRR*, abs/2512.08296, 2025. doi: 10.48550/ARXIV.2512.08296. URL https://doi.org/10.48550/arXiv.2512.08296.

Jiaqi Li, Xiaobo Wang, Zihao Wang, 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.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huanang Gao, Wenkai Yang, Zhiyuan Liu, and Ning Ding. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *CoRR*, abs/2604.13016, 2026. doi: 10.48550/ARXIV.2604.13016. URL https://doi.org/10.48550/arXiv.2604.13016.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *CoRR*, abs/2604.25850, 2026a. doi: 10.48550/ARXIV.2604.25850. URL https://doi.org/10.48550/arXiv.2604.25850.

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

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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, Estefany Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, 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 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, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *CoRR*, abs/2601.11868, 2026. doi: 10.48550/ARXIV.2601.11868. URL https://doi.org/10.48550/arXiv.2601.11868.

Alexander Novikov, Ngân Vu, 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. *CoRR*, abs/2506.13131, 2025. doi: 10.48550/ARXIV.2506.13131. URL https://doi.org/10.48550/arXiv.2506.13131.

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.

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 *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=mPdmDYIQ7f.

Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik R Narasimhan, and Victor Barres. τ-knowledge: Evaluating conversational agents over unstructured knowledge. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=XHZK5abtw2.

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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xiong-Hui Chen, Jianxin Yang, Zhenru Zhang, Yuqiong Liu, An Yang, Andrew Zhao, Yang Yue, Shiji Song, Bowen Yu, Gao Huang, and Junyang Lin. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for LLM reasoning. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. *CoRR*, abs/2607.12227, 2026. doi: 10.48550/ARXIV.2607.12227. URL https://doi.org/10.48550/arXiv.2607.12227.

Hui Xue and Fan Yang. Rethinking self-evolving agents: Do we still need prescribed optimization pipelines? *CoRR*, abs/2608.09629, 2026. doi: 10.48550/ARXIV.2608.09629. URL https://doi.org/10.48550/arXiv.2608.09629.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. {τ}-bench: A benchmark for \underline{T}ool-\underline{A}gent-\underline{U}ser interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=4OsgYD7em5.

Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nat.*, 639(8055):609–616, 2025. doi: 10.1038/S41586-025-08661-4. URL https://doi.org/10.1038/s41586-025-08661-4.

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

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=pUpzQZTvGY.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob N. Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *CoRR*, abs/2603.19461, 2026c. doi: 10.48550/ARXIV.2603.19461. URL https://doi.org/10.48550/arXiv.2603.19461.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui 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 *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025a. URL https://openreview.net/forum?id=z5uVAKwmjf.

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. *CoRR*, abs/2510.04618, 2025b. doi: 10.48550/ARXIV.2510.04618. URL https://doi.org/10.48550/arXiv.2510.04618.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A Related Work

Agent Optimization and Evolution. Agent improvement spans inference-time refinement, optimization of persistent artifacts, and open-ended self-modification. Reflexion retains linguistic feedback in episodic memory across trials, whereas Self-Refine iteratively revises the current output (Shinn et al., 2023; Madaan et al., 2023). RAM learns from communications to improve an external memory system (Li et al., 2024). DSPy optimizes language-model programs, and TextGrad propagates textual feedback through computational graphs (Khattab et al., 2024; Yüksekgönül et al., 2025). Promptbreeder evolves both task and mutation prompts, ACE maintains evolving context playbooks, and CoEvoSkills co-evolves multi-file skill packages with a separate surrogate verifier (Fernando et al., 2024; Zhang et al., 2025b; 2026a). The four approaches examined here make different choices within this space. AHE exposes editable harness components and uses distilled execution evidence to guide revisions (Lin et al., 2026a). HyperAgents makes both the task agent and its modification procedure editable (Zhang et al., 2026c). A-Evolve frames deployment-time adaptation as goal-directed optimization over persistent system state (Lin et al., 2026b). GEPA combines trajectory reflection for prompt updates with Pareto selection of complementary candidates (Agrawal et al., 2026).

Harness and Program Evolution. Related approaches also optimize system structure and executable code. GPTSwarm optimizes prompt-bearing graph nodes and orchestration edges, while AgentSquare searches combinations of planning, reasoning, tool-use, and memory modules (Zhuge et al., 2024; Shang et al., 2025). ADAS uses a meta agent to generate agent programs (Hu et al., 2025). Self-modification extends to the improvement procedure itself: STOP revises optimizer scaffolding, and the Darwin Gödel Machine evolves its own coding agent (Zelikman et al., 2024; Zhang et al., 2026b). AlphaEvolve instead evolves target algorithm programs using executable evaluation feedback (Novikov et al., 2025). Recent work also explores optimizers that construct their own optimization pipelines (Xue & Yang, 2026). This distinction between improving task solutions, agent implementations, and optimization procedures motivates separating what changes from how those changes are generated and retained.

Capability Boundaries and Mechanisms. Empirical studies of reasoning distinguish improved small-k success from expansion of large-k task coverage (Yue et al., 2025). Token-level analyses identify concentrated learning signals at high-entropy decisions (Wang et al., 2025), while studies of on-policy distillation examine behavioral compatibility and information novelty (Li et al., 2026). We apply a related analytical perspective to persistent agent-system changes rather than model-weight updates. Repeated evaluations distinguish reliability from empirical coverage, common-process Parent expansion tests improvement potential, Component interventions examine change compatibility, and frozen-pool admission replays assess retention.

Evaluation of Agent Systems. Agent evaluation must account for cost, reproducibility, and generalization beyond aggregate success (Kapoor et al., 2025). For harness evolution specifically, Wang et al. (2026) find inconsistent advantages over test-time scaling under comparable feedback and inference budgets, together with limited held-out generalization. Performance also depends on interactions among the model, harness, task structure, and coordination overhead (Kim et al., 2025). The τ-bench family emphasizes reliable tool-agent-user interaction, while Terminal-Bench 2 evaluates long-horizon execution in terminal environments (Yao et al., 2025; Merrill et al., 2026). We study general and coding tasks through these complementary settings, distinguish task exposure during evolution, and assess external transfer without further agent revision. These comparisons connect measured improvement to its deployment cost and scope of applicability.

B Operator Implementations of the Compared Methods

We describe the implementations used in our experiments through the seven operator stages in Table 10. All four methods share fixed target-model and evaluator contracts. MiniSWE permits dependency-closed changes to inner source code as well as prompts, skills, configuration, and orchestration; Codex permits only changes outside its immutable executable and source. HyperAgents additionally permits changes to designated evolution operators.

Evolution Paradigms. Our design-based grouping distinguishes implementation-centric recipes, AHE and HyperAgents, from reflection-centric recipes, A-Evolve and GEPA. The former organize revision around engineering the agent implementation or improvement machinery; the latter organize

it around distilling execution experience and feedback. Both use evidence and may modify code within the harness's legal surface. The grouping follows recipe design, while the harness determines the permitted editing surface. Codex permits changes to external orchestration code while keeping its executable and inner source immutable.

Table 10: Initial operator policies of the four methods. Dashes indicate an omitted optional stage. HyperAgents can subsequently revise its permitted operators.

Operator	AHE	HyperAgents	A-Evolve	GEPA
select	Newest eligible Parent	Score-weighted sampling with a child-count penalty	Highest-scoring eligible Parent	Per-task Pareto-front sampling
rollout	Reuse Parent evaluation traces	Reuse Parent evaluation traces	Fresh Train trajectories	Fresh Train trajectories
analyze	Per-task debugger reports	Metrics and browsable raw traces	Behavior-only trajectory judge	Focused reflective examples
mutate	One testable target change	Edit the target and allowed operators	Distill experience into target changes	Reflection-guided target revision
validate	–	Python syntax checks	–	Strict improvement on the Parent's Train batch
gate	Admit valid completed evaluations	Admit valid completed evaluations	Strict improvement over the matched Parent	Admit valid completed evaluations
record	Outcomes and change evidence	Lineage experience	Outcomes and rationale	Reflection and batch experience

Search scores use Train tasks for AHE and HyperAgents, and Validation tasks for A-Evolve and GEPA. The *validate* operator checks proposals, while the Validation split supplies evaluation tasks. Shared scope and evaluation-eligibility checks apply to all four methods, including those that omit the optional *validate* stage. Test outcomes remain excluded from search.

B.1 AHE

Selection and Diagnosis. AHE emphasizes observability-grounded revision (Lin et al., 2026a). Its *select* operator chooses the newest eligible Parent, not necessarily the highest-scoring one. The *rollout* operator reuses that Parent's evaluation traces. The *analyze* operator produces task-level debugger reports: failure reports identify the failure point, possible cause, and corrective lesson, while successful traces expose reusable strategies and fragile behavior. Mixed outcomes on a task can therefore inform the diagnosis alongside outright failures. An overview links these reports to prior changes and their observed effects.

Revision and Admission. The mutation agent is instructed to make one coherent, testable change, supported by task evidence and an identified execution path. It considers whether to retain, revise, or roll back the preceding change and pursue a different explanation. A change manifest records the proposed mechanism, affected files, predicted fixes, and behaviors at risk. Debugger reports supply hypotheses for revision. There is no separate optional *validate* stage. The *gate* operator admits a completed, eligible evaluation without requiring a higher score, so the next Parent can come from a lower-scoring revision. The *record* operator preserves outcomes and change evidence. Admission and continuation are thus distinct from final Champion reporting.

B.2 HyperAgents

Selection and Evidence. HyperAgents permits revision of the improvement procedure as well as the target agent (Zhang et al., 2026c). Its initial *select* policy samples eligible Parents using a score-dependent weight and a penalty for having many children. Higher scores increase sampling weight, while the child-count penalty reduces repeated expansion of the same Parent. The *rollout* operator reuses Parent evaluation traces. The initial *analyze* operator supplies metrics and browsable raw

traces, leaving their interpretation largely to the mutation agent rather than first producing AHE-style debugger reports.

Revision and Admission. Mutation may change the target and permitted operator implementations while leaving the evaluator and other fixed contracts unchanged. The mutation instructions favor substantive target changes but allow an operator-only revision when the evidence supports a process-level change. An operator revision takes effect at its next invocation, updating the initial policies in Table 10 for subsequent Generations. The initial *validate* stage checks Python syntax in target files and the mutation operator. The initial *gate* admits completed, eligible evaluations without a Parent-relative score threshold. The *record* operator stores lineage experience, including the Parent, outcome, and score. This editable search procedure distinguishes HyperAgents from target-only revision under fixed operators.

B.3 A-EVOLVE

Selection and Behavioral Feedback. Our A-Evolve implementation operationalizes experience-driven adaptation (Lin et al., 2026b). The *select* operator chooses the highest-scoring eligible Parent, and *rollout* collects fresh Train trajectories. The *analyze* operator compresses observable behavior, including tool use, repeated actions, errors, and completion signals, for a trajectory-only judge. This judge evaluates behavior without benchmark reward labels. Its proxy judgments provide feedback for revision, rather than replacing the benchmark score used for admission.

Revision and Admission. The *mutate* operator distills these observations into reusable task-solving guidance and executable target changes. Its instructions emphasize recurrent failure patterns, self-contained skills with their supporting dependencies, and preservation of unrelated behavior. Prompt and skill settings organize revision within the Harness's legal editing boundary. The optional *validate* stage is disabled. After canonical evaluation, the strict *gate* admits a Candidate only if its Validation score exceeds that of the matched Parent. The *record* operator retains outcomes and the revision rationale. Unlike AHE and the initial HyperAgents policy, A-Evolve combines greedy Parent choice with score-improving admission.

B.4 GEPA

Selection and Reflection. GEPA combines trajectory-based reflection with per-task Pareto selection (Agrawal et al., 2026). The *select* operator identifies eligible Candidates attaining the best observed score on at least one task, prunes redundant coverage, and samples according to the remaining per-task memberships. This can preserve access to task specialists without selecting solely by aggregate score. The *rollout* operator collects fresh Train trajectories, and *analyze* assembles task examples, outputs, and feedback for focused reflection.

Revision Scope. The named reflection units differ by benchmark. On τ^3-Banking, MiniSWE focuses on its system-prompt configuration, while Codex alternates between its prompt and task-execution skill. On Terminal-Bench 2, both harnesses use the entire target as the reflection unit. These units guide evidence organization rather than impose text-only filesystem permissions. The frozen Champion from MiniSWE with GEPA on Terminal-Bench 2 retains inner-source changes to task-conditioned guidance and shell execution. Reflection-centric therefore describes the revision process, not the absence of code changes. These reflection units are distinct from the post-hoc Components isolated in our diagnostics.

Validation and Admission. The *validate* operator evaluates the proposal on the same Train task batch used for the Parent rollout and requires a complete, scorable comparison with strict improvement. This feedback-informed proposal filter precedes canonical Validation evaluation. The subsequent *gate* checks completion and eligibility without imposing a second Parent-relative improvement threshold. The *record* operator preserves reflection choices, proposals, and batch comparisons. Thus, GEPA's improvement filter lies in *validate*, whereas A-Evolve's lies in *gate*, and they use different task evidence.

Implications for the Analysis. These implementations differ in how they allocate search, summarize evidence, couple target and operator changes, and filter proposals. Similar final gains can therefore arise through different processes. These differences motivate diagnostics of selection, diagnosis, change compatibility, and admission.

C PHENOMENOLOGY: ADDITIONAL PROTOCOLS AND ANALYSES

This appendix supplements the Phenomenology analysis in Section 3 with case-level comparisons, task-transition stability, deployment-cost accounting, and split-level heatmaps. Section C.4.1 provides the numerical cost table, and Section C.5 contains the split-level heatmaps.

C.1 MATCHED OUTCOME COMPARISONS

Evaluation and Interpretation. Every configuration in Table 1 uses the same benchmark tasks and eight complete passes. Each of the eight Champions per benchmark is paired with its harness Seed. The completed comparison contains 7,760 selected τ^3-Banking executions and 7,120 Terminal-Bench 2 executions, including the two Seeds once per benchmark. Each method–harness–benchmark combination contributes one canonical evolution lineage.

Metric Agreement and Disagreement. Across the sixteen Champions, eleven improve reliability and ten improve coverage. Eight improve both, four move the metrics in opposite directions, one improves reliability with unchanged coverage, and three reduce both. The opposite-sign cases occur in reflection-centric general-task evolution and implementation-centric coding-task evolution, each with both harness types. Thus, the distinction between consistency and reach is not confined to one harness type, task domain, or paradigm, even though agreement is more common than disagreement.

Interpreting Trends Across Dimensions. Reliability improves in all eight implementation-centric lineages across harness types and task domains. In the reflection-centric group, it improves in three of four source-editable cases but none of the four outer-layer-only cases. The paradigm-level pattern is therefore harness-dependent. Recipes also differ in feedback, search, and admission design, suggesting concrete mechanisms for further study.

C.2 TASK-LEVEL TURNOVER AND PASS STABILITY

Transition Construction. Each 9×9 heatmap pairs a task's Seed and Champion success counts from zero through eight, using complete 97×8 evaluations on τ^3-Banking or 89×8 evaluations on Terminal-Bench 2. Panels within each benchmark use the same task set and color scale. The split heatmaps retain the original Train, Validation, and Test task identities.

Coverage Turnover Across All Settings. All sixteen Champions both gain and lose covered tasks. Gains range from one to fifteen tasks and losses from two to ten, with both types present in every harness–task–paradigm combination. A positive net coverage delta therefore represents the balance of newly covered and lost tasks rather than preservation of the Seed's solved set. This shared pattern motivates paired gain/loss accounting without requiring a specific method to have the largest net expansion.

Within-Task Reliability Also Redistributes. Every panel contains tasks on both sides of the success-count diagonal. Evolution can improve repeated success on already covered tasks, reduce it elsewhere, and change which tasks are covered at all. The shared combination of these movements explains why neither a net coverage gain nor a reliability gain summarizes preservation.

Pass-Half Stability. We compare passes 1–4 with passes 5–8 to assess the stability of aggregate trends and task transitions. Some aggregate directions differ between halves on τ^3-Banking, showing sensitivity to execution variability.

C.3 BEHAVIORAL AUDIT OF TASK-LEVEL TRANSITIONS

Case Selection and Evidence. We identify the largest positive and negative changes in eight-pass success count within each lineage, breaking ties lexicographically by task identifier before trace inspection. The reported analysis contains 22 task–lineage comparisons with trace-linked behavioral evidence. For each increase, we inspect the first failed Seed execution and the first successful Champion execution in canonical pass order; for each decrease, we inspect the first successful Seed execution and the first failed Champion execution. The audit uses archived actions, observations, submitted artifacts, and verifier records.

Behavior Categories. We assign one primary category to each reference pair based on the concrete behavioral difference linked to its task outcome. Table 11 summarizes categories in the 22 selected large-change cases, and Table 12 illustrates their trace evidence. The agent-assisted audit examines execution behavior; the human evaluation reviews operator revisions.

Table 11: Primary behavior categories in the 22 selected task–lineage comparisons. Increase and decrease refer to eight-pass success counts.

Behavior category	Observable distinction	Increase	Decrease
State and dependency management	Preserve inputs, respect action dependencies, and retain a self-contained executable result.	7	1
Exact grounding	Match product conditions, dates, or output characters to the evidence and task requirements.	1	4
Completion and handoff	Finish all requested work and deliver the required final state or actual handoff.	2	3
Tool discovery and recovery	Use an available interface and recover from tool errors using the current state.	2	1
Implementation debugging	Resolve implementation errors and deliver a working program within the execution window.	1	0

What Changes Behind a Flip. Three themes recur across the selected cases with both harness types, task domains, and evolution paradigms. First, state and dependency management affects the complete workflow. Successful general-task executions order operations so that one preserves the conditions needed by another, while successful coding executions preserve source data before inspection and deliver self-contained artifacts. Second, exact grounding remains fragile after the main workflow succeeds: filed requests can contain an incorrect date, and reconstructed text can contain a single wrong character. Third, intermediate success can fall short of task completion. An agent can open one requested account and stop too early, announce a transfer without completing it, or pass its own program checks while leaving an unacceptable final artifact. These observations motivate checking preserved state, exact arguments, and the final execution boundary alongside the headline success rate. Tool recovery and implementation debugging provide complementary case evidence.

C.4 Deployment-Cost Accounting and Comparisons

A Shared Execution Ledger. Each configuration has 776 selected logical trials on τ^3-Banking or 712 on Terminal-Bench 2. Coverage and cost use this ledger and the metrics in Section 2.2.2. Prefix curves follow the canonical pass order $1, \ldots, k$. Superseded attempts remain in the audit ledger but are excluded from deployment accounting.

A Frequent Cost Premium, With Structured Exceptions. Eight of the eleven reliability-improving Champions have higher mean deployment cost than Seed. Such premiums occur with both harness types, task domains, and paradigms. All five reliability-improving general-task Champions cost more, whereas three of the six reliability-improving coding-task Champions cost less. The latter comprise two implementation-centric cases and one reflection-centric case, all with the source-editable harness. This harness–task cluster shows how the resource trade-off varies by setting, motivating matched-resource comparisons alongside reliability.

Cost and Performance Need Joint Interpretation. All four outer-layer-only Champions on coding tasks cost more than Seed, although only two improve reliability. Coverage–cost curves relate empirical reach to deployment resources at each pass prefix. Evolution cost complements these curves by accounting for the resources used to obtain the evolved agent.

C.4.1 Coverage and Deployment Cost: Detailed Results

Table 13 provides the numerical results behind Figure 3, using the metrics in Section 2.2.2 and the ledger described in Appendix C.4.

Table 12: Illustrative trace contrasts on τ^3-Banking and Terminal-Bench 2. Counts are Seed → Champion successes out of eight; paired Champion counts follow the listed method order.

Domain	Task and lineage	Count	Observed difference in the reference executions
General	Task 054, MiniSWE with A-Evolve	$1 \rightarrow 4$	Seed handles replacement and dispute first, blocking the requested credit-limit increase. Champion completes the increase before those dependent actions.
General	Task 037, MiniSWE with A-Evolve and GEPA	$7 \rightarrow 3/1$	Both Champions successfully file disputes but use the wrong issue-noticed date. Seed's successful reference uses the date required by the evaluator.
General	Task 056, Codex with AHE	$5 \rightarrow 0$	Champion terminates after opening checking. Seed continues the interaction and completes the additional savings and funding requests.
General	Task 090, Codex with GEPA	$1 \rightarrow 5$	Seed repeatedly fails through a generic tool wrapper. Champion switches to the available direct lookup and completes the investigation.
Coding	`db-wal-recovery`, Codex with GEPA	$0 \rightarrow 7$	Seed loses the original write-ahead log during inspection and reconstructs incorrect values. Champion preserves a copy before repair and verifies the recovered values.
Coding	`polyglot-c-py`, MiniSWE with HyperAgents	$1 \rightarrow 8$	Seed leaves test binaries and a script beside the required single source file. Champion leaves only the required file; the contrast concerns final directory contents.
Coding	`path-tracing`, MiniSWE with A-Evolve	$2 \rightarrow 6$	Seed's wrapper relies on a workspace binary unavailable in the evaluation jail. Champion submits a self-contained renderer.
Coding	`path-tracing`, Codex with GEPA	$6 \rightarrow 1$	The direction reverses: Champion replaces its renderer with a workspace-dependent wrapper that matches the image locally but fails in the jail.
Coding	`gcode-to-text`, Codex with HyperAgents and A-Evolve	$8 \rightarrow 4/5$	Champions recover the text geometry but confuse `o` with `0`, or `1` with `i`. Checking the written file does not correct the transcription.

Table 13: Deployment-side coverage and recorded costs on both primary benchmarks. Scores are percentages and costs are in USD.

Benchmark	Harness	Variant	Pass@1	Pass@2	Pass@4	Pass@8	Mean cost/trial	Recorded cost
τ^3-Banking	MiniSWE	Seed	27.8	39.2	46.4	54.6	\$0.779	\$604.52
		AHE	28.9	46.4	56.7	66.0	\$0.891	\$691.57
		HyperAgents	33.0	39.2	50.5	57.7	\$0.819	\$635.56
		A-Evolve	26.8	36.1	48.5	55.7	\$0.690	\$535.57
		GEPA	29.9	40.2	51.5	58.8	\$0.985	\$764.57
	Codex	Seed	24.7	42.3	50.5	56.7	\$0.983	\$762.45
		AHE	34.0	46.4	53.6	62.9	\$1.108	\$859.82
		HyperAgents	36.1	46.4	60.8	66.0	\$1.104	\$856.43
		A-Evolve	28.9	39.2	49.5	57.7	\$0.955	\$740.79
		GEPA	33.0	42.3	49.5	53.6	\$0.978	\$758.73
Terminal-Bench 2	MiniSWE	Seed	56.2	74.2	82.0	86.5	\$0.774	\$551.17
		AHE	74.2	77.5	85.4	87.6	\$0.639	\$455.27
		HyperAgents	64.0	75.3	79.8	85.4	\$0.667	\$474.72
		A-Evolve	69.7	77.5	82.0	86.5	\$0.783	\$557.29
		GEPA	68.5	75.3	86.5	88.8	\$0.701	\$499.14
	Codex	Seed	69.7	80.9	87.6	89.9	\$0.887	\$631.85
		AHE	73.0	84.3	87.6	91.0	\$0.971	\$689.18
		HyperAgents	71.9	84.3	85.4	86.5	\$1.107	\$785.63
		A-Evolve	69.7	78.7	84.3	87.6	\$1.031	\$734.06
		GEPA	58.4	76.4	82.0	86.5	\$1.007	\$716.64

C.5 Split-Level Task-Transition Heatmaps

Figures 5–7 show eight-pass transitions by split, with τ^3-Banking above Terminal-Bench 2.

C.5.1 Train Split

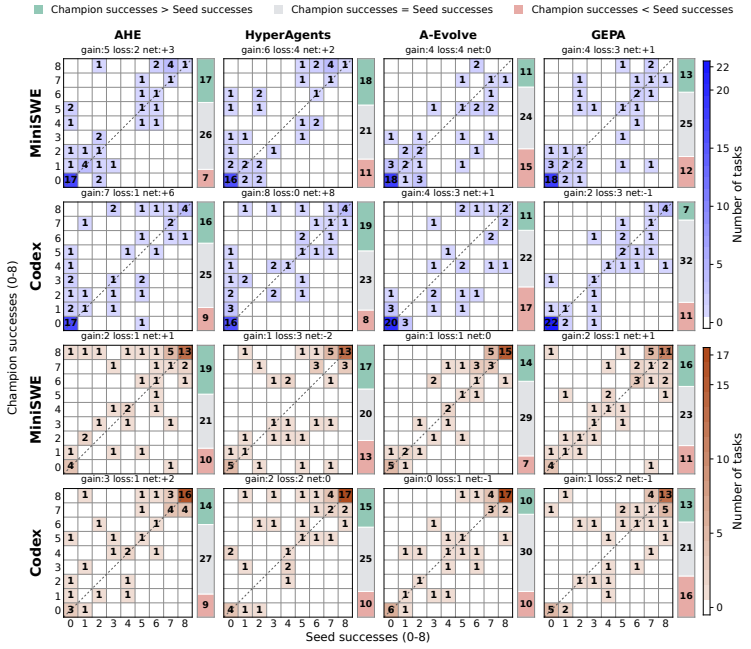


Figure 5: Task-level Seed-to-Champion transitions over eight passes on the Train split. Each panel contains 50 τ^3-Banking tasks (blue) or 50 Terminal-Bench 2 tasks (orange).

C.5.2 Validation Split

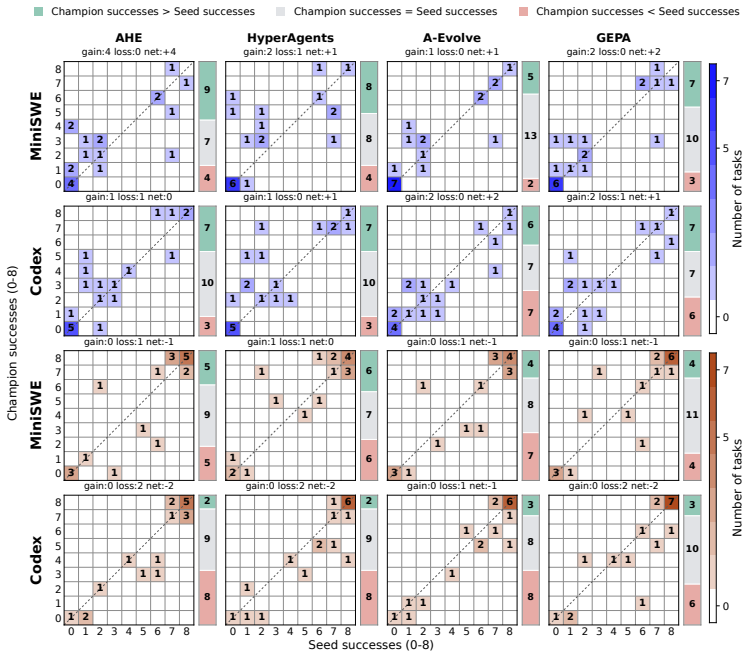


Figure 6: Task-level Seed-to-Champion transitions over eight passes on the Validation split. Each panel contains 20 τ^3-Banking tasks (blue) or 19 Terminal-Bench 2 tasks (orange).

C.5.3 TEST SPLIT

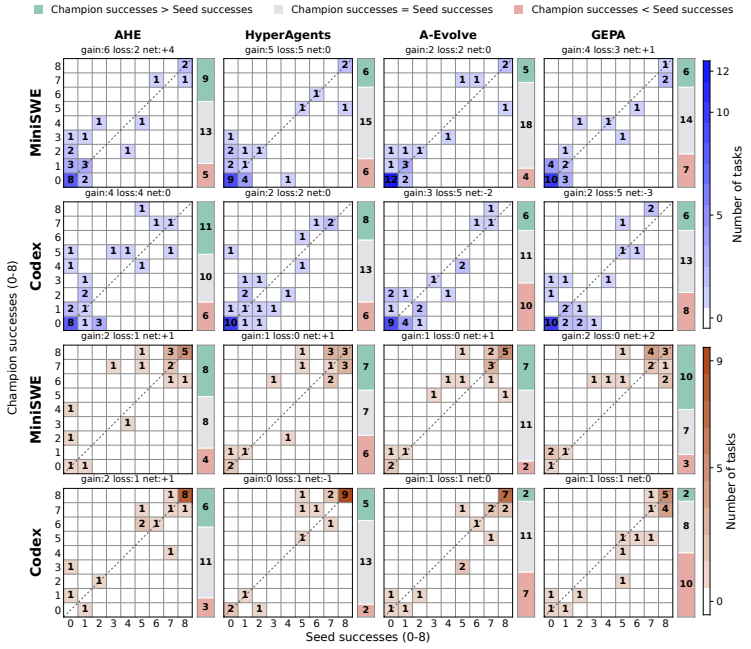


Figure 7: Task-level Seed-to-Champion transitions over eight passes on the Test split. Each panel contains 27 τ^3-Banking tasks (blue) or 20 Terminal-Bench 2 tasks (orange).

D GENERALIZATION: ADDITIONAL PROTOCOLS AND ANALYSES

This appendix supplements the Generalization analysis in Section 4. It separates task exposure during evolution from cross-benchmark transfer, specifies both external evaluation contracts, and collects absolute transfer scores in one table. The two primary benchmarks remain τ^3-Banking and Terminal-Bench 2.

D.1 SEARCH EXPOSURE AND SPLIT ACCOUNTING

Task Exposure and External Transfer. Train trajectories supply evolution feedback, and Validation evidence can influence admission. Test tasks retain the source benchmark's domain and evaluation contract while remaining excluded from mutation feedback and admission decisions. External targets change the benchmark within the corresponding general-task or coding-task domain. The comparison follows these roles to examine gains beyond direct feedback and across benchmarks.

Reuse and Denominators. Table 2 reuses the complete eight-pass measurements reported in Section 3. Split labels retain their roles during evolution. For a split, the change in successful executions is divided by $8N$ and the change in covered tasks by N, where N is the full source-benchmark task count. The three split contributions therefore sum to the overall gain before rounding. External deltas use each target's own task set and matching Seed.

Persistence Across Splits and Harnesses. Ten of sixteen Champions have positive Test reliability contributions. Seven of eight implementation-centric cases improve across both harness types and task domains, with one source-editable general-task exception at -0.1 points. Reflection-centric cases improve in three of four source-editable settings but none of the four outer-layer-only settings. Thus, most implementation-centric gains reach feedback-excluded tasks, while reflection-centric persistence is more harness-dependent. On coding tasks, all outer-layer-only Champions lose Validation reliability, yet the implementation-centric Champions gain Test reliability. Gains therefore vary across splits as well as harness–paradigm combinations.

D.2 General-Task Transfer Protocol

Full-Benchmark Evaluation. The general-task transfer comparison evaluates eight frozen τ^3-Banking Champions and two harness Seeds on the full τ^2-bench (Barres et al., 2026), covering retail, airline, and telecom. Following official benchmark-level aggregation, we compute each metric within each subdomain and take the equal-weight mean across the three subdomains for both Seed and Champion. All 278 tasks have eight valid outcomes per configuration, totaling 22,240 verified evaluations. These measurements supply the absolute scores in Table 14 and matched-Seed gains in Table 2; deltas are computed before rounding. Our $\mathrm{Pass@8}$ measures at least one success in eight attempts, while the benchmark's pass^8 measures consistency across all eight attempts.

Neutral Adaptation and Quality Checks. Within each harness, frozen neutral adapters translate message and tool protocols while preserving target-native user interaction, permissions, state transitions, and verification. All versions share these adapters, with Champion instructions retained unchanged. The evaluation contract fixes task identities, model settings, inference limits, and a score-independent recovery policy. Compatibility checks and smoke tests precede the full evaluation; reporting requires complete valid outcomes under the same contract.

Coverage Contraction With a Metric-Specific Exception. Seven of eight Champions lose coverage on the general-task transfer target. The sole positive case is reflection-centric and source-editable, at $+1.5$ points. The recurring contraction spans both harness types and paradigms, while reliability follows a different pattern: all four source-editable Champions improve, and three of four outer-layer-only Champions decline. Thus, more consistent success on the transfer target can coexist with narrower empirical reach, and coverage contraction should not be described as a uniform loss of all benefits.

D.3 Coding-Task Transfer Protocol

Frozen Evaluation. DeepSWE v1.1 (Huang et al., 2026) tests transfer from terminal-computing tasks to long-horizon repository-level implementation without target-specific mutation or Champion-specific adaptation. The complete comparison contains four frozen Champions and the matching Seed for each harness, each with 113 official tasks and eight passes, or 904 selected trials per configuration. All 9,040 logical outcomes are complete and unique. Task identities, evaluator and model contracts, inference limits, and recovery rules remain fixed across these versions. Table 14 reports the complete scores for both harnesses.

A Target-Dependent Reversal Across Paradigms. Seven of eight Champions gain coding-transfer coverage, compared with seven losing coverage on the general-task target. The coding gains span both harness types and paradigms, with one reflection-centric, source-editable exception at -2.7 points. All four source-editable Champions nevertheless lose coding-transfer reliability, by 0.2–2.3 points, while all four improve it on the general-task target. Outer-layer-only coding-transfer reliability improves in two cases and declines in two. These results separate reach from consistency and show that transfer need not uniformly erode gains.

D.4 Absolute Transfer Scores

Table 14 collects the absolute scores for both transfer targets. Each method row denotes its frozen Champion under the specified harness, Seed is the matched unevolved agent.

Table 14: Absolute performance on external transfer benchmarks. Scores are percentages. τ^2-bench evaluates agents evolved on τ^3-Banking using the official equal-weight aggregation across its three domains; DeepSWE v1.1 evaluates agents evolved on Terminal-Bench 2.

Benchmark	Harness	Variant	Pass@8	Avg@8
τ^2-bench	MiniSWE	Seed	96.2	84.0
		AHE	95.6	86.7
		HyperAgents	95.9	84.8
		A-Evolve	96.0	85.4
		GEPA	97.8	86.6
	Codex	Seed	97.5	87.0
		AHE	96.8	85.3
		HyperAgents	96.8	87.3
		A-Evolve	96.5	85.0
		GEPA	96.5	85.6
DeepSWE v1.1	MiniSWE	Seed	57.5	23.5
		AHE	59.3	21.1
		HyperAgents	61.1	21.3
		A-Evolve	61.9	23.2
		GEPA	54.9	21.8
	Codex	Seed	77.0	45.1
		AHE	85.0	45.0
		HyperAgents	83.2	48.2
		A-Evolve	78.8	43.0
		GEPA	80.5	46.6

E MECHANISMS: ADDITIONAL PROTOCOLS AND ANALYSES

This appendix supplements the Mechanisms analysis in Section 5 with Parent expansion, Component comparisons, and admission replay protocols. Appendix E.5 details Component implementations and case selection.

E.1 SELECT: PARENT CHOICE AND SUBSEQUENT IMPROVEMENT

Parent Pools and Common Generation Process. For each benchmark and harness, we form one pool containing the frozen Champions from AHE, HyperAgents, A-Evolve, and GEPA. The four pools contain 16 Parents. Each Parent initializes three separate branches from the same frozen workspace, producing 48 Candidates in total. Every branch uses the same original A-Evolve *analyze* and *mutate* implementations, prompts, and generation allowance. The four methods provide Parent artifacts; the shared downstream process holds the modification procedure fixed.

Execution Evidence and Evaluation. Each Parent executes the 50 Train tasks once, and its three branches share this evidence while running analysis and mutation independently. Canonical Validation uses one pass over 20 tasks on τ^3-Banking or 19 on Terminal-Bench 2. Each Parent and Candidate receives eight Test passes over 27 or 20 tasks, respectively. Target and mutation models and harness versions follow Section 2.2.1. Test outcomes determine the reporting scores and best-Parent labels. Candidate scores enter this analysis regardless of the original A-Evolve admission verdict, so the diagnostic measures the changes produced from each Parent.

Original Selection Rules. We apply the frozen HyperAgents, A-Evolve, and GEPA *select* rules to each common four-Parent pool. HyperAgents uses its native score-weighted sampling rule with Train evidence. A-Evolve uses the one-pass Validation aggregate, and GEPA uses the corresponding per-task Validation vector. Diagnostic metadata, child-count initialization, and tie handling are fixed before these rules are evaluated. AHE's chronological selector requires a shared lineage order, so AHE contributes a Parent artifact and the other three rules define Original. Neither Test outcomes nor sibling Candidate results enter these selection decisions or the analysis and mutation inputs.

Best Parent and Hit Rate. Let $\mathcal{P}$ be a four-Parent pool, $C_{P,j}$ the Candidate from branch j of Parent P, and $S(A)$ the Test Avg@8 of agent A. We define the mean improvement and the best-Parent set as

$$\Delta(P) = \frac{1}{3}\sum_{j=1}^{3} S(C_{P,j}) - S(P), \qquad \mathcal{B} = \arg\max_{P\in\mathcal{P}} \Delta(P). \tag{4}$$

For the three original rules $\mathcal{S}$, let $\pi_s(P)$ be the probability that rule s selects P. Table 3 reports

$$H_{\text{Original}} = \frac{100}{3}\sum_{s\in\mathcal{S}}\sum_{P\in\mathcal{B}} \pi_s(P), \qquad H_{\text{Random}} = 100\frac{|\mathcal{B}|}{4}. \tag{5}$$

A deterministic rule contributes zero or one before aggregation; a stochastic rule contributes its probability of selecting a best Parent. Each pool has a unique maximizer, giving a 25% random baseline. Improvement remains signed, so the best Parent is the one with the largest gain or smallest decline.

Current Performance and Improvement Potential. In all four pools, the Parent with the highest Test score differs from the Parent with the largest mean improvement. On general tasks, the best-Parent gain is -2.0 points for the source-editable harness and $+19.6$ for the outer-layer-only harness; on coding tasks, it is $+4.0$ and $+1.5$, respectively. The original rules' mean hit rate falls below random on general tasks and exceeds it on coding tasks for both harnesses. These comparisons motivate assessing selection by subsequent improvement under a common generation process.

E.2 Analyze: Component and Reminder Comparisons

Comparison Logic. Applying a Component to the matching Seed measures its standalone effect; comparing that variant with Champion tests how the remaining changes affect its benefit. Generic-reminder controls test whether broad guidance reproduces those effects. These comparisons inform *analyze* through the specificity of useful guidance and *mutate* through the composition of changes.

A Recurring Specificity Advantage. Fifteen of sixteen Component variants have higher reliability than their matching generic-reminder control, and every Champion case has at least one such Component. The exception is the isolated shell-adaptation Component in the reflection-centric, source-editable coding-task case, at 64.0 versus 65.9 for the reminder; its task-guidance Component instead reaches 67.3. Useful specific changes are therefore not reproduced by general advice, even though not every isolated Component helps. Reminders leave source-editable reliability close to Seed on both task domains and lower outer-layer-only reliability on both.

Case Composition. The eight Component cases cover both harness types and task domains. Seven are implementation-centric; the reflection-centric case uses the source-editable harness on coding tasks. Each case pairs concrete Champion changes with a generic-reminder intervention on the matching Seed.

Diagnosis Requirements. The specificity advantage motivates a diagnosis contract that links failure explanations to execution traces, identifies useful behavior to preserve, and states when the proposed correction applies. We assess this contract through the changes and outcomes it produces.

E.3 Mutate: Component Effects and Interactions

Complete Revisions Usually Outperform Subsets. Six of eight Champions have higher reliability than either isolated Component, including two implementation-centric general-task cases and all four coding-task cases. The exceptions are two other implementation-centric general-task cases, one per harness type: the source-editable interface Component reaches 34.4 versus 32.6 for Champion, and the outer-layer-only orchestration Component reaches 37.2 versus 36.9. In the latter case, skills alone also increase coverage to 72.2 versus 66.0, despite slightly lower reliability. Conversely, an implementation-centric coding-task Champion with the outer-layer-only harness reaches 72.6 reliability even though both isolated variants, at 69.1 and 68.0, fall below Seed's 69.9. These results motivate evaluating combinations alongside isolated effects.

Interaction Definition and Estimates. For the four completed τ^3-Banking decompositions, let S_0, S_A, S_B, and S_{AB} denote Avg@8 for Seed, Seed + A, Seed + B, and Champion. We measure

non-additivity by

$$I = (S_{AB} - S_0) - [(S_A - S_0) + (S_B - S_0)].$$

A negative value means that the joint gain is smaller than the sum of the two standalone gains. The Champions from AHE have interactions of 0.0 points for MiniSWE (95% task-paired interval $[-4.5, 4.9]$) and $+2.1$ for Codex ($[-2.2, 6.2]$), with both intervals including zero. The Champions from HyperAgents have negative interactions for MiniSWE (-6.3; $[-11.7, -1.2]$) and Codex (-6.6; $[-12.6, -0.6]$). These cross-harness observations identify interference as a design risk in the sampled implementation-centric general-task cases.

Compatibility Checks. Interaction estimates use the four verified τ^3-Banking decompositions, while Terminal-Bench 2 provides targeted subset comparisons. Negative interactions motivate checking whether a Candidate diff follows its diagnosis, includes necessary dependencies, and preserves useful behavior. Coupled changes remain permissible when their dependencies and rationale are explicit.

E.4 Gate: Size-Normalized Best-Candidate Retention

Candidate Pools. For each benchmark, harness, and method, we freeze four distinct artifacts from the same evolution lineage: the Seed, historical Champion, last legal Candidate, and Candidate nearest 50% of cumulative recorded evolution cost. Predeclared 25% and 75% checkpoints resolve duplicates. This gives eight pools per benchmark. Each artifact has eight-pass measurements on separate decision and reporting panels of 20 and 27 tasks for τ^3-Banking, or 19 and 20 for Terminal-Bench 2. These task panels follow the canonical diagnostic splits.

Original Admission Policies. Original gate uses the initial implementations in Table 10. AHE, HyperAgents, and GEPA accept all eligible Candidates with completed evaluations; A-Evolve requires strict mean improvement over its matched Parent. The comparison supplies four legal, fully evaluated Candidates to these fixed admission policies, with upstream proposal checks held outside the replay.

Rules and Replay Contract. Each pool starts with Seed as incumbent and processes the historical Champion, last legal Candidate, and cost-indexed checkpoint in the frozen order. This is a diagnostic order, not the historical generation order. An accepted challenger becomes the incumbent for subsequent comparisons, while all previously retained Candidates remain in the set. The constrained rule is unchanged from the original replay: it requires strict mean improvement over the incumbent on the same purchased evidence, with ties rejected, and

$$L = \frac{1}{|\mathcal{D}_g|} \sum_{x \in \mathcal{D}_g} \max\{0, \hat{p}_{\text{inc}}(x) - \hat{p}_{\text{cand}}(x)\} \leq 0.05, \tag{6}$$

where $\mathcal{D}_g$ contains 20 decision tasks on τ^3-Banking or 19 on Terminal-Bench 2, and each success fraction uses passes 1–2. All policies receive the same 160 or 152 unique Candidate–task trials, respectively; eligibility-only gates need not use their numeric scores. For A-Evolve's original gate, the matched parent is the current replay incumbent. A separate reporting panel of 27 or 20 tasks, respectively, uses eight passes to identify the empirical best and never supplies decision evidence. Both benchmarks use the same fixed order, strict-improvement rule, regression cap, and retained-set definition.

Metric Definitions. For method m and gate g, let $\mathcal{P}_m$ be its four-Candidate input pool, $\mathcal{R}_{m,g}$ the retained set including Seed, and $\mathcal{B}_m = \arg\max_{c \in \mathcal{P}_m} \text{Avg@8}(c)$ on the reporting panel. The hit indicator is $H_{m,g} = \mathbf{1}[\mathcal{R}_{m,g} \cap \mathcal{B}_m \neq \varnothing]$ and set size is $N_{m,g} = |\mathcal{R}_{m,g}|$. Retaining any tied maximizer counts as a hit. The size-normalized score is $H_{m,g}/N_{m,g}$; Seed ensures a nonempty retained set. Each main-table cell reports $100 \sum_m (H_{m,g}/N_{m,g})/4$, separately for each benchmark and harness. This is the mean of per-pool ratios, not the ratio of mean hit to mean size. Values are rounded to one decimal place only after aggregation. All sixteen pools have a unique empirical best. The score therefore equals the chance of drawing that Candidate uniformly from the retained set; this interpretation does not prescribe the method's actual *select* policy.

A Recurring Preservation Risk. The original policies retain the reporting-panel best in fifteen pools, compared with six under Constrained. The nine newly excluded best Candidates span both harness types, task domains, and paradigms. Size-normalized scores nevertheless improve in four pools, remain unchanged in three, and fall in nine. The improvements include two implementation-centric

Table 15: Per-pool audit of size-normalized best-Candidate retention. Each policy cell gives H/N, the binary best-retention indicator divided by set size, including Seed. The empirical best maximizes reporting-panel $\mathrm{Avg@8}$; Generation numbers identify the frozen artifacts.

Benchmark	Harness	Method	Empirical best	Original gate H/N	Constrained H/N
τ^3-Banking	MiniSWE	AHE	Generation 9	$1/4$	$0/1$
		HyperAgents	Generation 3	$1/4$	$1/2$
		A-Evolve	Generation 1	$1/2$	$1/2$
		GEPA	Generation 9	$1/4$	$0/1$
	Codex	AHE	Generation 10	$1/4$	$0/2$
		HyperAgents	Generation 11	$1/4$	$0/2$
		A-Evolve	Generation 20	$0/2$	$0/2$
		GEPA	Seed	$1/4$	$1/1$
Terminal-Bench 2	MiniSWE	AHE	Generation 18	$1/4$	$0/1$
		HyperAgents	Generation 10	$1/4$	$1/2$
		A-Evolve	Generation 4	$1/4$	$0/1$
		GEPA	Generation 20	$1/4$	$0/2$
	Codex	AHE	Generation 20	$1/4$	$0/1$
		HyperAgents	Generation 18	$1/4$	$0/1$
		A-Evolve	Seed	$1/1$	$1/1$
		GEPA	Seed	$1/4$	$1/1$

source-editable pools retaining the best among fewer alternatives, and two reflection-centric outer-layer-only pools retaining a best Seed alone. Pruning can therefore improve the diagnostic metric while excluding useful Candidates elsewhere. The two outer-layer-only averages increase even while evolved best Candidates are lost in other pools. Jointly reporting hit, set size, and Candidate identity distinguishes concentration on a strong fallback from preservation of evolved gains.

Replay Evidence and Design Implications. The replay applies original and constrained admission rules to the same frozen Candidate pools and measured scores. The constrained rule combines strict score improvement with a regression cap. Comparing retained-set quality, size, and membership motivates admission rules that balance improvement, preservation, and fallback retention.

E.5 Component Definitions and Representative Content

This subsection records the frozen Component definitions for the eight-case matrix of Table 4. All sixteen Component variants and eight Champion comparators have complete evaluations. A and B identify the two groups within each case. The definitions specify retained agent changes, implementation files, execution dependencies, and representative content. Prompt and code excerpts have line wrapping adjusted for display, and code excerpts reproduce the indicated implementation segment. MiniSWE paths are relative to `src/minisweagent/`; Codex paths are relative to the Candidate's target directory. Each Component is evaluated as a complete agent intervention.

Construction and Comparison. Seed + A and Seed + B retain the matching Seed outside the specified Component and its execution dependencies. Components are separated by function: a configuration file can contain both interface settings and task-solving instructions, which we isolate by semantic key. Required compatibility adaptations are identified below. Codex Components remain outside its immutable executable and source. The Terminal-Bench 2 cases evaluate targeted Champion subsets. Interaction estimates use decompositions whose joint Components reproduce the compared Champion.

Case Selection. The matrix contains four Champion cases per benchmark, with two from each harness. Eligibility requires at least two functionally distinct groups of persistent changes that can be isolated with their execution dependencies on the matching Seed while keeping the model and evaluator fixed. The τ^3-Banking cases comprise Champions from AHE and HyperAgents in both harnesses. On Terminal-Bench 2, we retain the previously evaluated Champions from MiniSWE with AHE and Codex with HyperAgents, and add MiniSWE with GEPA and Codex with AHE after auditing all eight Champions. The additions prioritize clear functional boundaries and complementary

mechanisms. Component boundaries were frozen before evaluation, and all completed outcomes are retained.

Selection History and Extension. The initial Terminal-Bench 2 sample selected the largest historical unrounded $\mathrm{Avg@8}$ gain within each harness, yielding MiniSWE with AHE and Codex with HyperAgents. Case identities remain frozen, and source-faithful reevaluations replace the two superseded Codex Champion measurements. MiniSWE with GEPA separates task guidance from shell execution adaptation, complementing AHE's broader validation and submission controls. Codex with AHE separates instruction content from persistent context and skill support, complementing HyperAgents' dependency bootstrap. Functional clarity and mechanism complementarity determine the additions. Their identities and Component boundaries were frozen before evaluation, and all added outcomes are reported.

E.5.1 τ^3-BANKING: MINISWE WITH AHE (G2)

Component A: Interface. This Component changes how shell/tool output is presented to the model, while retaining the Seed's task-solving instructions and bash action interface. It extracts readable text from structured tool-response envelopes, removes duplicate text blocks, and preserves salient information when an observation is long. Rather than retaining only its beginning and end, the observation also includes a bounded selection of middle content containing tool schemas, execution results, required fields, and conversation-control signals. The changed files are `models/utils/actions_toolcall.py`, `models/utils/actions_toolcall_response.py`, the added `models/utils/observation_output.py`, and the `observation_template` key in `config/mini.yaml`.

> *Observation-budget constants from the added helper:*
>
> ```
> DISPLAY_OUTPUT_LIMIT = 10000
> DISPLAY_OUTPUT_HEAD_CHARS = 3000
> DISPLAY_OUTPUT_TAIL_CHARS = 3000
> SALIENT_OUTPUT_MAX_CHARS = 2500
> ```

Component B: Strategy. This Component replaces the system prompt, task prompt, and format-error guidance in `config/mini.yaml`; observation processing stays at the Seed. The instructions distinguish the benchmark's tool-driven general tasks from coding tasks, prioritize live state and tool schemas, require one focused action followed by inspection, prohibit invented arguments or case facts, and distinguish a stopped conversation from a successfully completed backend action. The format-error guidance also specifies concise recovery from token-limit truncation and a tool-only final submission.

> *Task-prompt excerpt:* "Prefer live environment state, exact tool outputs, runtime status, and tool schemas over inference from general KB text."
>
> *Completion-rule excerpt:* "Only treat the task as complete when the required environment state or required action has actually been achieved."

E.5.2 τ^3-BANKING: MINISWE WITH HYPERAGENTS (G9)

Component A: Interface. This Component exposes structured benchmark tools in place of a bash-only interface when it detects tasks from the general-task benchmark. It adds tool schemas, argument parsing (including nested JSON-string normalization), dispatch to the benchmark environment, and a dedicated final-submission action. The dependency-closed implementation spans `agents/interactive.py`, `environments/local.py`, `utils/tau3.py`, both action/tool-response utilities, and seven adapters in `models/`: the chat and Responses implementations for LiteLLM, OpenRouter, and Portkey, plus Requesty. The thirteenth changed file, `config/mini.yaml`, adds neutral transport and format-error instructions so that the Seed prompt does not request an unavailable bash tool. This compatibility text describes how to call the exposed tools, without importing the Champion's task-solving strategy.

> *Exposed tool names:* `tau3_call`, `tau3_list_tools`, `tau3_get_tool_schema`, and `submit_final_output`.
>
> *Compatibility-note excerpt:* "This note describes only the available transport; determine the task policy and actions from the task, tool schemas, and runtime responses."

Component B: Strategy. This Component changes only the task-instance prompt in `config/mini.yaml`. It retains the Champion's service workflow: start the conversation once, consult the latest user message and schema, perform one action at a time, separate user communication from domain-tool calls, respect stop/transfer signals, and ground advice in documented policy and hard user requirements. Because the Seed does not expose Component A's structured tools, the workflow is expressed through the Seed's `tau3-mcp.py` command-line interface. This is the evaluated CLI-translated strategy Component, not an unmodified copy of the native-tool prompt; the translation is an execution dependency, not an independent strategy finding.

> *Translated task-prompt excerpts:* "Perform exactly one focused Tau3 CLI action per response and inspect its output before the next action." "Do not recommend a product or action that misses a user-stated hard requirement."

E.5.3 τ^3-BANKING: CODEX WITH AHE (G15)

Component A: Orchestration. The only changed file is the outer adapter `agent.py`. Before invoking Codex, it detects service-task markers in the instruction and appends a workflow-guidance block, unless that block is already present. The injected text emphasizes exact tool discovery, a per-entity worklist, assistant-side versus user-side actions, date and eligibility checks, canonical write arguments, and completing all requested items before ending. The Seed's prompt and skill files are retained. Here, *orchestration* includes conditional instruction delivery; it does not mean that the Component contains no natural-language guidance.

> *Injected-guidance excerpt:* "Keep a live per-entity worklist for the case. For each named transaction, card, account, product, referral, or other requested item, track the exact entity, the user-described problem, the currently grounded workflow or enum, any missing fields, and the next read or write needed."

Component B: Prompt and Skills. This Component replaces `prompt.md` and `skills/task-execution/SKILL.md`, and adds `skills/tau3-service-workflows/SKILL.md`, while retaining the Seed adapter. The prompt distinguishes coding work from the general-task benchmark's workflows and directs the latter to the specialized skill. That skill specifies a six-part workflow: discover the live tool surface; ground on the exact entity; identify who must perform the next state change; construct exact writes; preserve the selected product or workflow branch; and finish all requested work. It includes explicit checks for relative dates, enums, optional fields, user-side follow-through, and multi-item completion.

> *Specialized-skill excerpt:* "Once the user accepts a supported product or workflow branch, stop re-optimizing and execute that exact branch."
>
> *Prompt excerpt:* "If the task involves multiple requested actions or entities, keep a short checklist and do not finish while any requested item is unresolved."

E.5.4 τ^3-BANKING: CODEX WITH HYPERAGENTS (G18)

Component A: Orchestration. The outer `agent.py` detects the configured service server or service markers in the instruction, then prepends a guidance block unless its header is already present. The same server-detection helper is reused by the existing installation path. The injected guidance prioritizes live account investigation, eligible and actionable recommendations, exact knowledge-base procedures, discoverable-tool checks, literal argument formatting, and completion of accepted workflows. Prompt and skill files remain at the Seed, so the specialized-skill reference in this block does not itself install Component B.

> *Injected-guidance excerpt:* "Do not substitute a neighboring tool or an incidental side effect for a separately documented step. If the KB names a dedicated replacement, approval, closure, report, or submission tool, use that exact tool instead of assuming another mutation already covered it."

Component B: Skills. This Component changes `skills/task-execution/SKILL.md` to route tasks from the general-task benchmark to the added `skills/tau3-banking-runtime/SKILL.md`; the outer adapter and main prompt stay at the Seed. The specialized skill covers identity verification, account-specific lookups, recommendation eligibility, tool unlocking and execution

checks, multi-step discrepancies, transfers, closures, and final-state verification. It also includes implementation-sensitive payload rules: preserve surfaced key order, whitespace, explicit null/false values, and numeric forms in nested argument strings. These are actual retained instructions, not a claim that equivalent JSON representations generally have different semantics.

> *Payload-formatting excerpt:* "Serialize that inner JSON as a single-line object with double-quoted keys, the surfaced key order, and standard spaces after each comma and colon."
> *Tool-check excerpt:* "After `unlock_discoverable_agent_tool`, check that the returned tool name, description, and parameters match the tool you meant to unlock."

E.5.5 TERMINAL-BENCH 2: MINISWE WITH AHE (G18)

Component A: Static Instructions. Only `config/mini.yaml` changes. The retained prompt additions call for early inspection of visible tests and checkers, exact deliverable paths and formats, fresh-shell verification, preservation of original inputs and consumers, and rechecking after any subsequent change. They reject proxy checks, fabricated substitutes, and unresolved failures as evidence of completion. These instructions do not add a programmatic submission guard.

> *Contract-first instruction:* "Before long implementation, downloads, or external research, spend the first 1-2 commands locating the real acceptance path."
> *Reverification instruction:* "Any edit after a passing verification invalidates that verification. If you change the artifact, environment, dependency set, or generated output afterward, rerun the exact end-to-end check against the final state before submitting."

Component B: Dynamic Guard. The dynamic checks span `agents/default.py` and `environments/local.py`; `config/mini.yaml` supplies their hooks and activation flags, without Component A's static instruction additions. It discovers visible verifier files, derives deliverable-boundary hints from the task, supplies fresh-shell compatibility helpers, records test inspection and unresolved check failures, and inspects submission attempts. The guard can replace a completion signal with a warning about uninspected verifiers, unhealthy package state, unresolved failures, missing files, or deliverable-boundary violations. It warns once per warning fingerprint; repeating the same submission can proceed, so this is not an unconditional veto or a replacement for the benchmark evaluator.

> *Activated flags in the configuration:*

```
pause_submit_for_uninspected_verifier: true
pause_submit_for_unhealthy_system_packages: true
pause_submit_for_unresolved_verifier_failures: true
pause_submit_for_deliverable_boundary_violations: true
```

E.5.6 TERMINAL-BENCH 2: MINISWE WITH GEPA (G20)

Component A: Task Guidance. This Component combines static instructions about exact deliverables and evidence preservation with task-conditioned reminders. In `agents/default.py`, `_build_task_guardrails` detects wording associated with exhaustive outputs or fragile evidence and generates the corresponding guidance. The template in `config/mini.yaml` consumes the generated `task_guardrails` value and includes the accompanying static instructions. The generator and template hook stay in the same Component because either alone disconnects the mechanism. The local command-execution implementation remains at the Seed.

Component B: Shell Execution Adaptation. Only `environments/local.py` changes. The helper `_preferred_shell_executable` resolves `MSWEA_LOCAL_SHELL`, defaulting to `bash`, and supplies the available executable to the subprocess launcher. It falls back to the platform default when the preferred shell cannot be resolved. This changes shell semantics at the command-execution boundary without importing Component A's guidance or changing the model configuration.

E.5.7 TERMINAL-BENCH 2: CODEX WITH AHE (G20)

Component A: Prompt and Skills. This Component replaces `prompt.md` and `skills/task-execution/SKILL.md`. The instructions prioritize the verifier-facing contract, a minimal

coherent change on the active execution path, and checks of the final artifact at the consumer-visible boundary. Prompt and skill content form one guidance Component rather than separate Components based on file location. The adapter, compaction settings, model, reasoning effort, and Codex executable remain at the Seed.

Component B: Persistent Context and Skill Support. The outer `agent.py` moves temporary Codex state and skill-upload paths to a persistent Codex directory, while `codex.toml` enables prefix-preserving compaction with `auto_compact_token_limit_scope` set to `body_after_prefix`. These changes support instruction continuity without importing Component A's prompt or skill contents. The evolved adapter changes are semantically applied to the exact evaluation Seed, preserving its authentication, provider routing, session handling, and cancellation cleanup. Codex's executable and source, the model, and reasoning effort are unchanged.

E.5.8 TERMINAL-BENCH 2: CODEX WITH HYPERAGENTS (G18)

Component A: Instructions. The prompt and task-execution skill are replaced in `prompt.md` and `skills/task-execution/SKILL.md`; the outer adapter remains at the matching Seed. The prompt and skill instruct the agent to preserve authentic input artifacts, inspect visible verifiers, reconstruct missing checker dependencies, react to failed checks, match exact output paths and final-state requirements, remove probe-only artifacts, and run a final check from a fresh shell or process. They also ask the agent to diagnose silent long-running commands instead of repeatedly polling without new evidence.

> *Skill excerpt:* "Recreate the verifier runtime before declaring the task unavailable."
> *Failure-handling excerpt:* "Clear red checks before closing. If any command, test, assertion, or custom validation failed after your last meaningful edit, rerun that same path cleanly or keep debugging."

Component B: Orchestration and Bootstrap. Only the outer `agent.py` changes. The Champion's bootstrap changes are merged onto the exact Seed adapter used in the matched evaluation, rather than replacing that adapter with a different baseline wrapper. During setup, the adapter installs or locates `uv` and `uvx`, creates a sourceable environment helper, and exposes stable executable entrypoints in standard locations for fresh-shell verifier commands. Installation first attempts the Python package route and then an installer fallback; setup reports a warning if the bootstrap remains unavailable. The Seed's prompt and skills are retained. This Component prepares the execution environment; it does not alter the verifier's scoring rules or Codex's executable.

> *Bootstrap function's documented purpose:* "Install stable verifier entrypoints for uv-based test harnesses."
> *Setup-hook excerpt:*

```
try:
    await self.exec_as_root(environment,
        command=_verifier_bootstrap_command())
except RuntimeError:
    self.logger.warning(
        "Failed to bootstrap uv verifier entrypoints",
        exc_info=True)
```

Generic-Reminder Controls Are Separate. The reminders in Table 5 are not additional Champion Components. On τ^3-Banking, both harnesses receive: "This is an interactive tool-use task. Inspect the available runtime tool interface, use tools when needed, verify the resulting state, and complete every requested item before ending." The Terminal-Bench 2 MiniSWE control adds: "Before final submission, inspect any available tests or check commands and run the closest exact end-to-end check on the final artifact. Do not submit while a relevant check is still failing or running." These controls add general advice without the case-specific prompt bundle, interface implementation, or dynamic guard above.

Recovery of Interrupted Evaluations. Four τ^3-Banking Codex reminder evaluations were interrupted by API rate limits after their conversations had started. Under a documented recovery-policy amendment, we fixed these four task–pass identities and reran each once from its initial state with unchanged model, simulator, target, and evaluator settings. The other 772 numeric outcomes were

retained, and all four replacement outcomes were accepted, including three zeros. Original attempts remain in the audit record; selected outcomes and their costs enter the reported metrics.

F Operator Redesign: Implementation and Evaluation

F.1 Evidence and Implementation Requirements

The analysis agent combines results from the Phenomenology, Generalization, and Mechanisms analyses with source traces and reference operators. It identifies recurring patterns across harnesses, task domains, and evolution paradigms and links each finding to the relevant comparisons and observed behavior. The resulting evidence report supplies the rationale and applicability conditions for operator revision.

Each proposed revision specifies the original rule, observed problem, implementation change, expected effect, and validation checks. The implementation agent receives this specification together with the evidence report, reference code, input/output contract, and permitted editing surface. It returns versioned code or instructions, evidence links, applicability conditions, fallback behavior, and validation records. Interface checks preserve the reference *rollout*, *validate*, and *record* stages and each harness's editing contract.

Diagnosis and Mutation Requirements. The *analyze* revision asks diagnoses to identify the observed failure and useful behavior to preserve. The *mutate* revision asks Candidate changes to implement that diagnosis coherently and retain the required dependencies. These complementary contracts connect execution evidence to the resulting intervention.

Revision Objectives. The four revisions in Table 7 target complementary decisions. Regression-aware *select* uses preservation evidence to resolve score ties, behavior-grounded *analyze* links diagnoses to execution traces, dependency-aware *mutate* organizes coherent changes, and cost-aware *gate* considers resource use at equal scores. These objectives guide implementation, expert review, and algorithm evaluation.

F.2 AGE Implementation

Recipe and Evidence Flow. AGE installs four fixed operator revisions in the A-Evolve recipe described in Appendix B, shared across both benchmarks and harnesses. Each generation selects a Parent, collects fresh Train trajectories, diagnoses behavior, and proposes one Candidate. Framework scope checks and canonical Validation evaluation precede admission and recording. The reference *rollout* and *record* implementations are unchanged, and the optional *validate* stage is disabled. MiniSWE permits changes to inner source and outer artifacts, whereas Codex permits only outer-artifact changes. The evaluator, task partitions, model settings, resource limits, and evolution operators remain fixed. Online decisions use Train and Validation evidence.

Run Configuration. Each run uses 20 generations, one proposed Candidate per generation, and search random seed 0. The target harnesses are MiniSWE 2.4.5 and Codex 0.143.0, both using GPT-5.4 with high reasoning effort. Diagnosis and mutation use Codex 0.146.0 with GPT-5.4 and xhigh effort through Harbor. Every rollout executes the 50 Train tasks afresh. Canonical scoring executes each Validation task once, using 20 tasks on τ^3-Banking and 19 on Terminal-Bench 2. Frozen Champions receive separate eight-pass reporting evaluations.

Select: Regression-Aware Tie-Breaking. AGE retains the highest-score objective and revises only exact ties. Let $\mathcal{P}$ be the eligible Parents with finite recorded Validation scores $s(C)$, and let $\mathcal{T} = \arg\max_{C\in\mathcal{P}} s(C)$. When comparable task-level evidence is available, selection minimizes Seed-relative regressions within $\mathcal{T}$:

$$R(C) = \sum_{t\in\mathcal{D}_{\text{val}}} \mathbb{I}[\bar{r}_t(C) < \bar{r}_t(A_0)], \qquad P \sim \text{Uniform}\left(\arg\min_{C\in\mathcal{T}} R(C)\right),$$

where $\mathcal{D}_{\text{val}}$ is the Validation task set and $\bar{r}_t$ is the mean recorded reward on task t. Comparable evidence requires a unique eligible Seed evaluation, matching task-set hashes and task identities, equal per-task repetition counts, and completed trials with finite rewards for every tied Candidate and the Seed. Otherwise, the seeded generator samples uniformly from the original highest-score tie set.

A unique maximizer is selected directly, and an empty eligible scored set stops selection. The rule reuses recorded outcomes and preserves the highest-score objective.

Analyze: Behavior-Grounded Diagnosis. The analyzer collects the current rollout and, when available, one preceding rollout along the selected lineage, retaining the last 30 records in their stored order. Each judge input contains chronological tool calls, successful and failed tool results, and the completion signal. Tool arguments and results are capped at 600 and 800 characters, respectively. Each rendered trajectory uses at most 12,000 characters, preserving approximately two-thirds from its beginning and one-third from its end when truncated. Interrupted executions can include a bounded termination message. The behavior bundle excludes benchmark rewards, verifier outputs, hidden tests, raw task text, and raw-case paths.

The judge returns `score`, `category`, `outcome`, and `failure_reason`. Scores 0–3 indicate visible breakdown, 4–6 indicate partial or ambiguous progress, and 7–10 indicate visible completion supported by an appropriate check or comparably decisive result. Verdicts distinguish `Observed:` behavior from `Inference:` explanations and ground completion judgments in trajectory evidence. These behavioral scores guide diagnosis, while benchmark scores drive *select* and *gate*. Each request permits one attempt with a 600-second timeout. Invalid or failed responses return an unknown verdict with sentinel score -1.

The analyzer exports trajectories, behavioral signals, and verdicts in a structured evidence bundle. The selected text view uses at most 30,000 characters for τ^3-Banking and 60,000 for Terminal-Bench 2, omitting trailing whole records beyond these budgets. Mutation consumes at most a 50,000-character prefix of this view. Both truncation steps add explicit omission markers.

Mutate: Mechanism-Linked, Dependency-Closed Changes. The mutation agent receives the selected behavior evidence, Parent workspace, prompt and skill inventory, and editing contract. Before choosing files, its instructions require the observed behavior, inferred mechanism, replacement behavior, activation conditions, and expected trade-off. A recurring concrete pattern or a single severe, visibly supported breakdown can motivate a change, while ambiguous evidence favors preserving current behavior. The agent must inspect affected interfaces and their producers and consumers, make one coherent intervention, preserve useful behavior in successful traces, and check for conflicting or duplicated guidance. Dependency-closed changes may span multiple files and layers.

The agent is instructed to edit the permitted target directly, inspect its final diff, run proportionate structural checks, and propose a validation case for the expected benefit. Outputs retain the patch, changed-file list, surface-check report, agent response, and usage record. Structural checks enforce editing surfaces and interfaces, while the mutation instructions specify evidence grounding, dependency closure, and behavior preservation. Execution errors stop the operator and are recorded with their diagnostic information.

Gate: Score-First, Cost-Aware Admission. AGE retains strict Parent-relative score improvement and adds a cost tie-break. For a Candidate C and its matched Parent P, with finite Validation scores, admission is

$$\mathrm{accept}(C,P) = \mathbb{I}\big[s(C) > s(P) \ \vee \ \big(s(C) = s(P) \wedge Q(C,P) \wedge c(C) < c(P)\big)\big] .$$

Here c is the recorded USD cost of canonical Validation evaluation. The comparability predicate Q requires matching task-set hashes, finite strictly positive costs, and a direct Parent evaluation. Equal-score admission requires a verified, comparable cost reduction. When cost comparability fails, only the score-improvement branch applies. A valid matched Parent and finite scores are required, and all other Candidates are rejected. The operator reuses canonical evaluations and preserves score improvement as its primary objective.

F.3 Human Review Materials and Reporting

Eight experts independently reviewed all four frozen operator revisions, with balanced case and direction-option orders. For each case, they inspected the original rule and evidence and submitted a direction before seeing EvoAudit's proposal. They then reviewed its reasoning, implementation changes, applicability, and validation plan, judging evidence support, problem addressing, and approval to validate. Each question included an insufficient-evidence option and optional comments. Figures 8 and 9 show the direction-selection and proposal-review interfaces for the *select* revision.

Table 8 reports mean positive counts per revision out of eight experts. Direction agreement uses a frozen mapping between the initial choice and EvoAudit's direction. Evidence support and problem addressing count fully positive responses, while endorsement counts approval to validate without further changes. Each measure sums positive counts across the four revisions and divides by four. Assignments, proposal versions, questions, and aggregation rules were frozen before formal review. All formal judgments are retained, and subsequent changes receive new proposal versions.

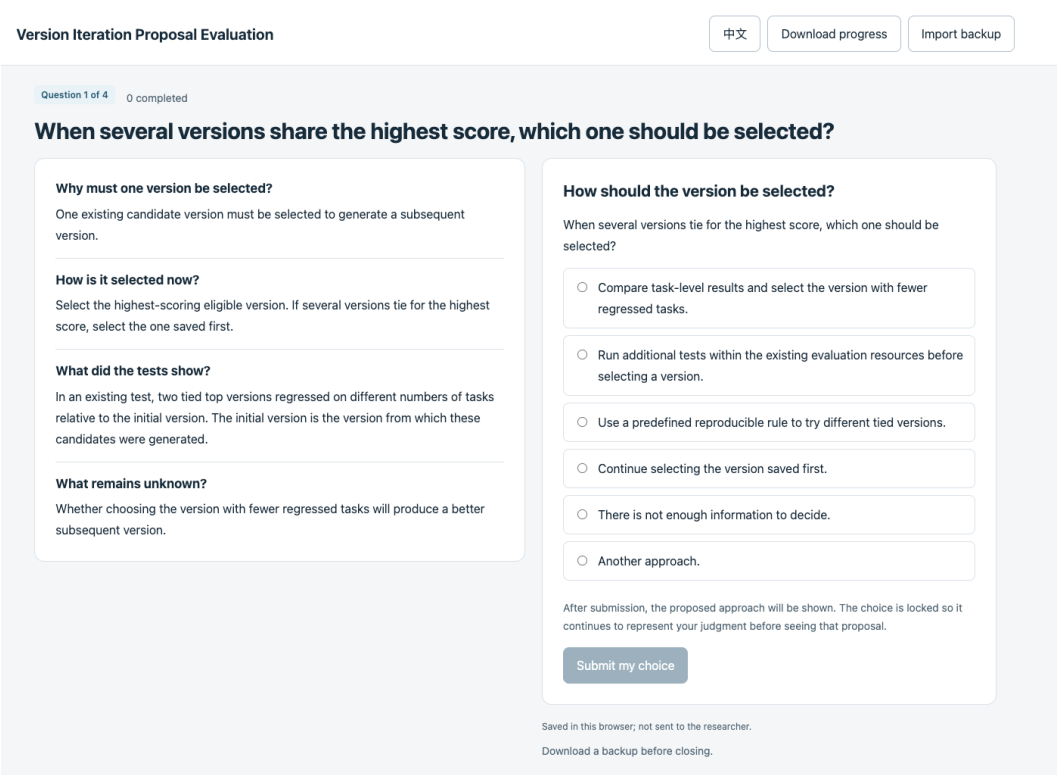


Figure 8: Human review of *select* before proposal disclosure. Experts inspect the original rule and evidence, then submit their preferred revision direction.

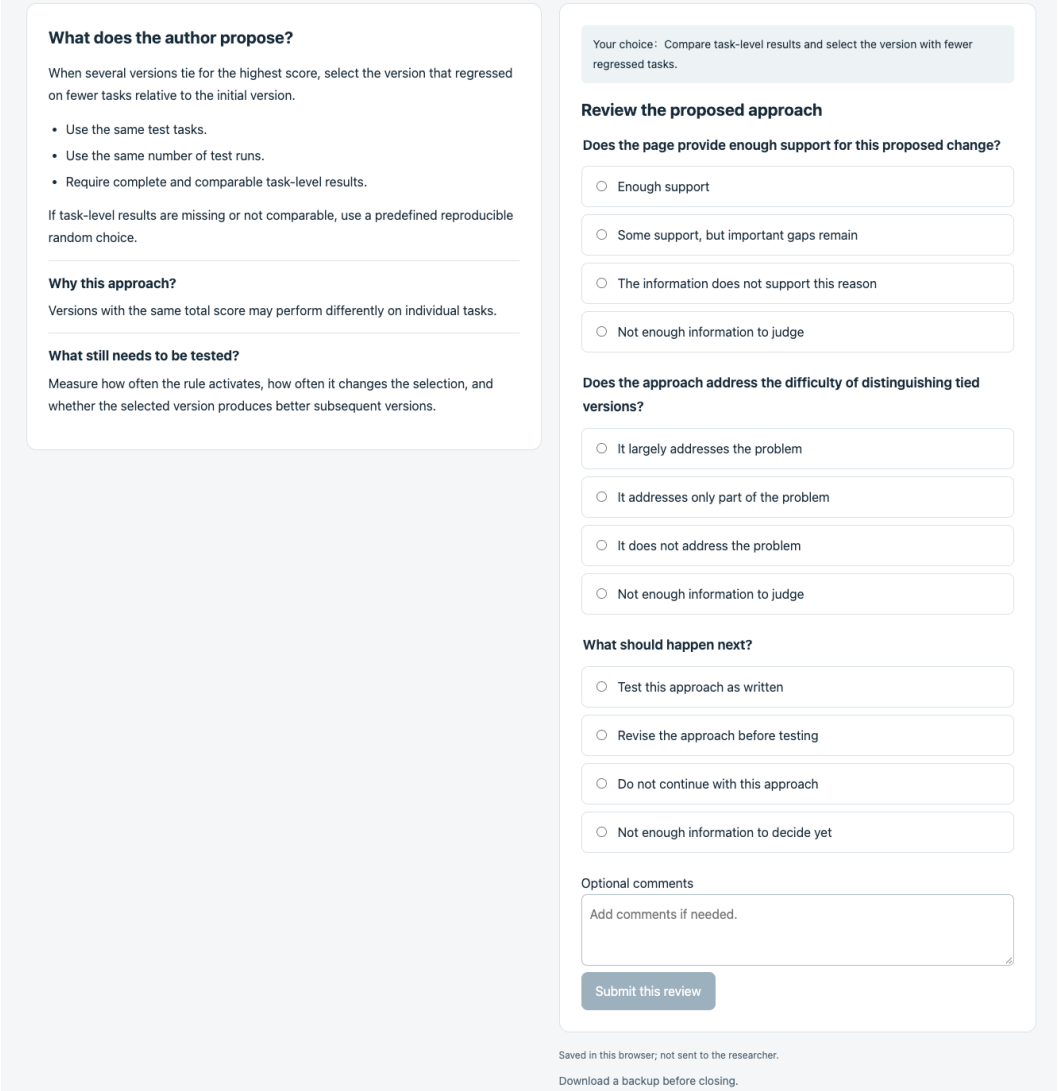


Figure 9: Human review of *select* after proposal disclosure. Experts inspect the revision and its reasoning, then assess evidence support, problem addressing, and endorsement for validation.

F.4 Comparison with Existing Evolution Methods

We compare archived 20-generation runs of AHE, HyperAgents, A-Evolve, and GEPA with separate 20-generation AGE runs on both primary benchmarks and harnesses. Each frozen Champion receives eight benchmark passes against its matching Seed. Table 9 reports reliability, coverage, task turnover, and deployment and evolution costs. Evidence-collection and analysis/design-agent costs are recorded separately from online evolution.

G EVOLUTION TREES

The following figures show all sixteen archived evolution runs, each comprising the Seed and 20 generation attempts. Unsuccessful attempts are retained. Arrows follow the recorded parent-to-child relationships, not chronological adjacency; layout positions and node colors do not indicate performance or acceptance decisions.

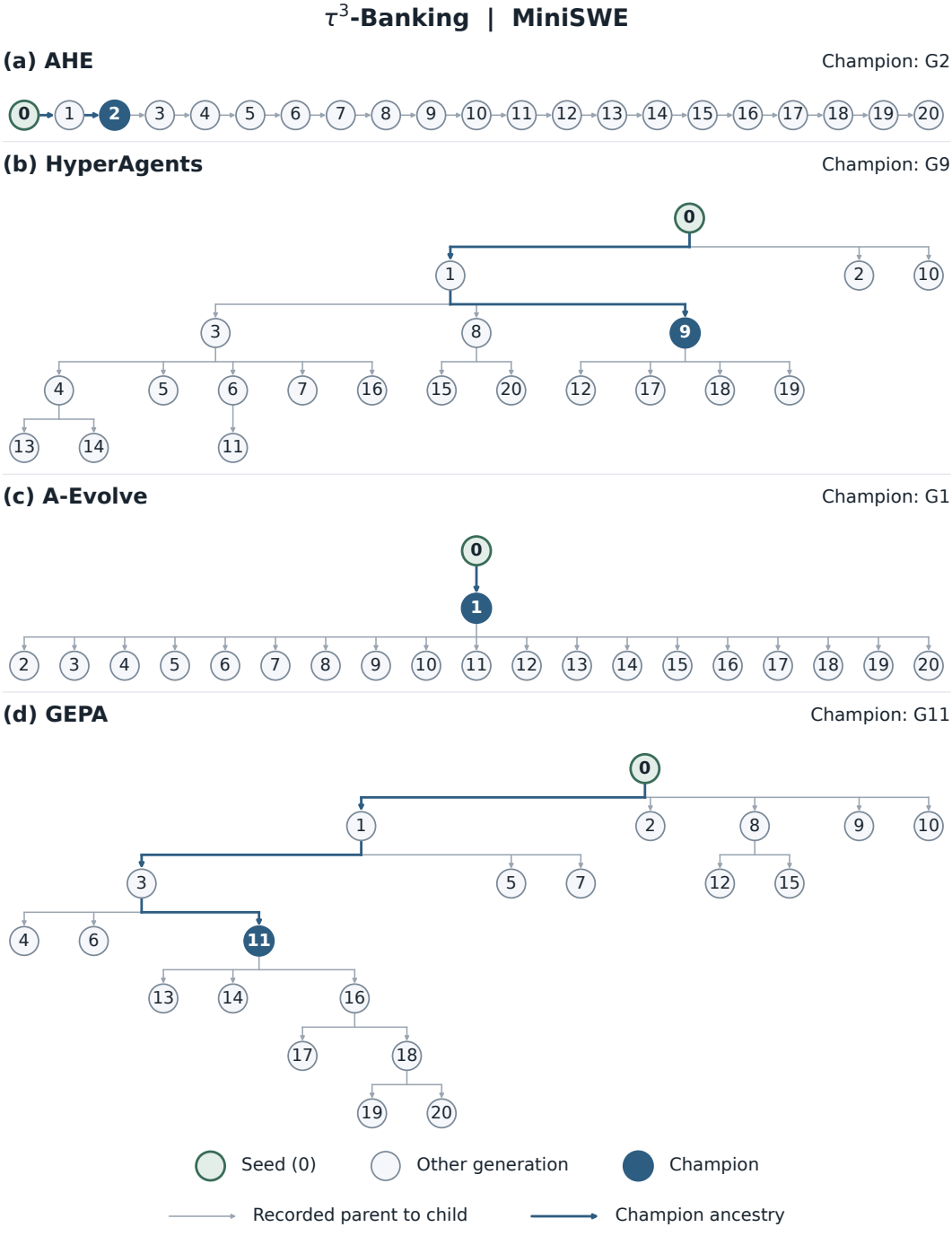


Figure 10: Evolution trees on τ^3-Banking with MiniSWE. Nodes are labeled by generation, with 0 denoting the Seed. Blue marks the Champion used in the paper and its ancestral path.

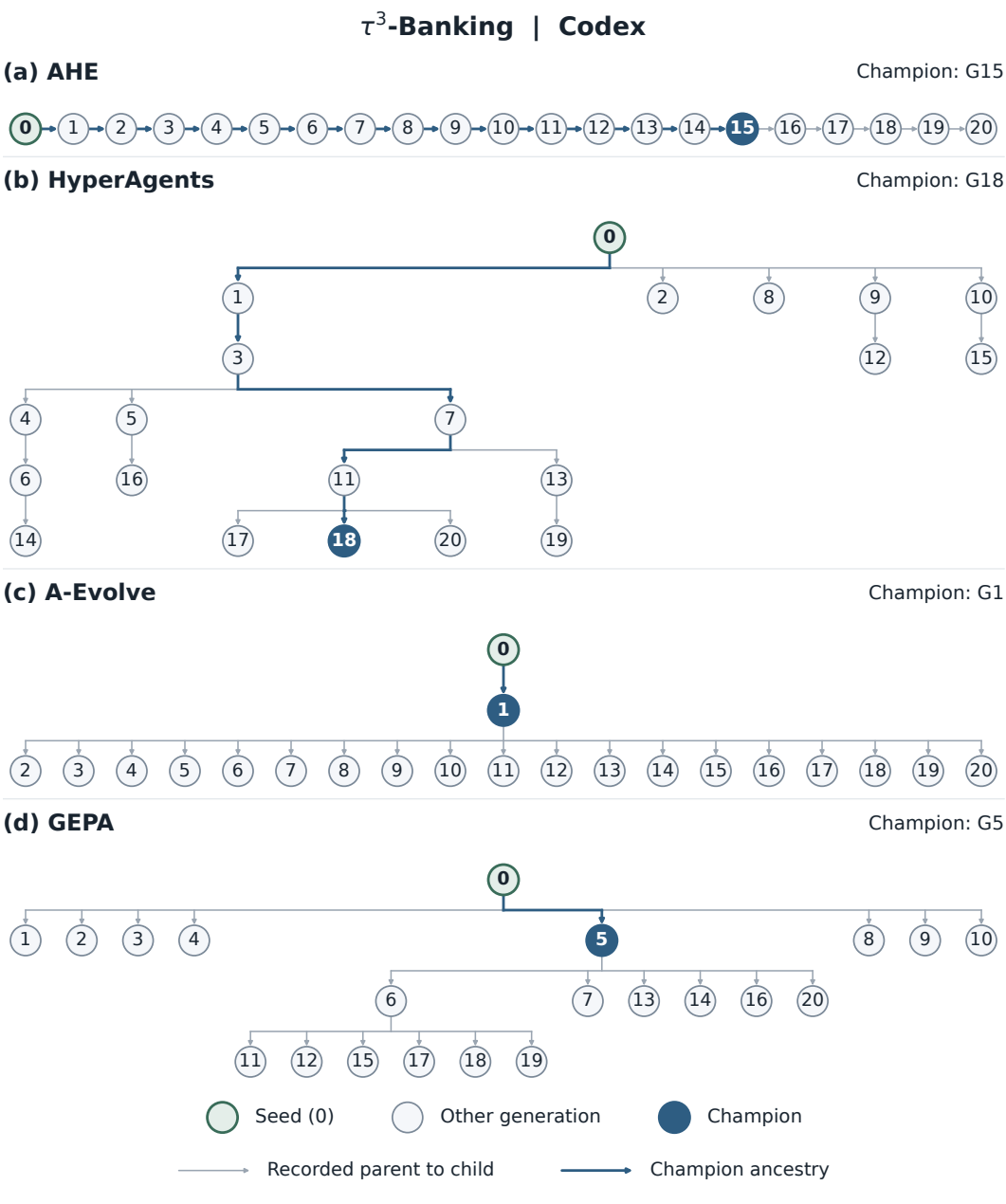


Figure 11: Evolution trees on τ^3-Banking with Codex. Nodes are labeled by generation, with 0 denoting the Seed. Blue marks the Champion used in the paper and its ancestral path.

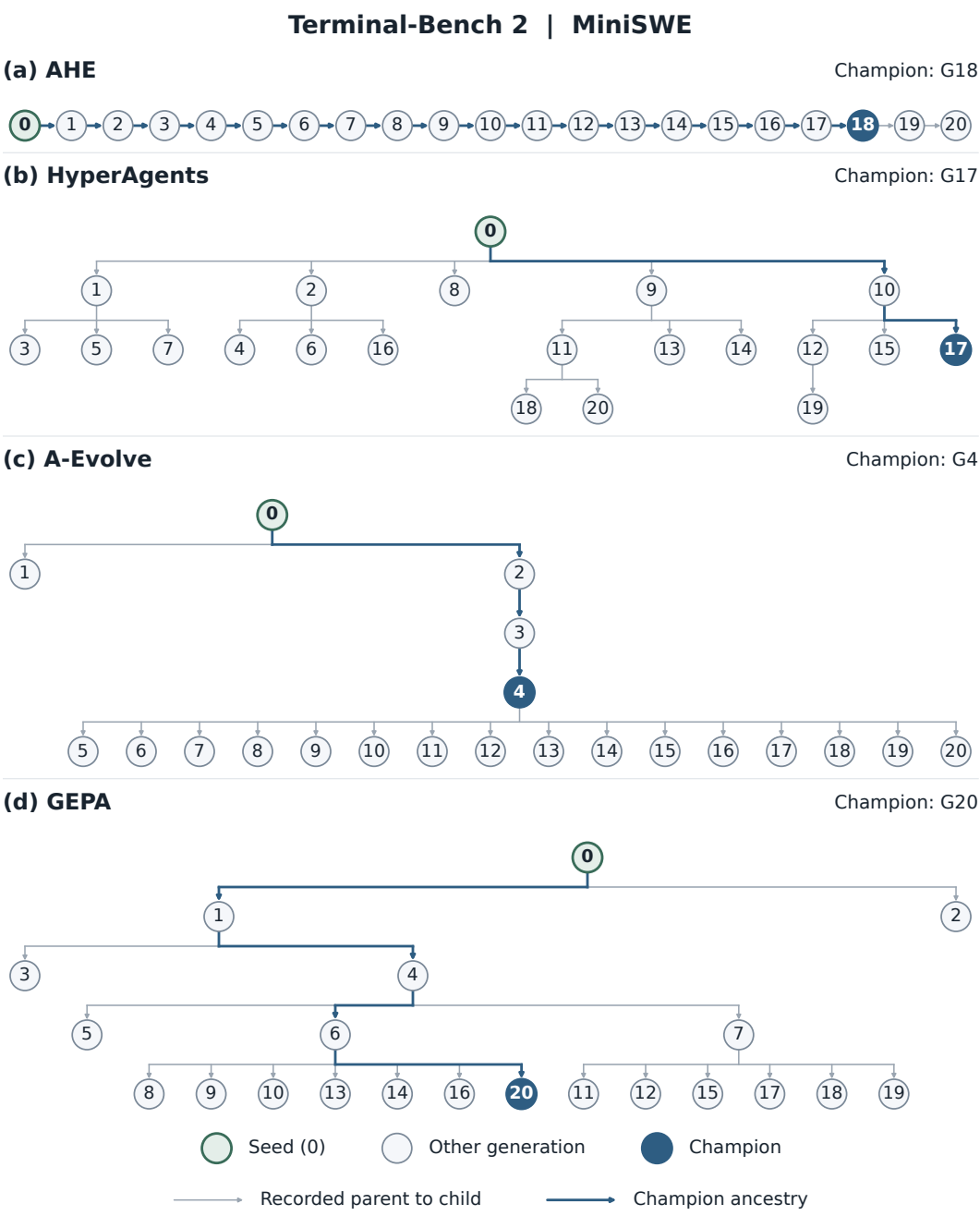


Figure 12: Evolution trees on Terminal-Bench 2 with MiniSWE. Nodes are labeled by generation, with 0 denoting the Seed. Blue marks the Champion used in the paper and its ancestral path.

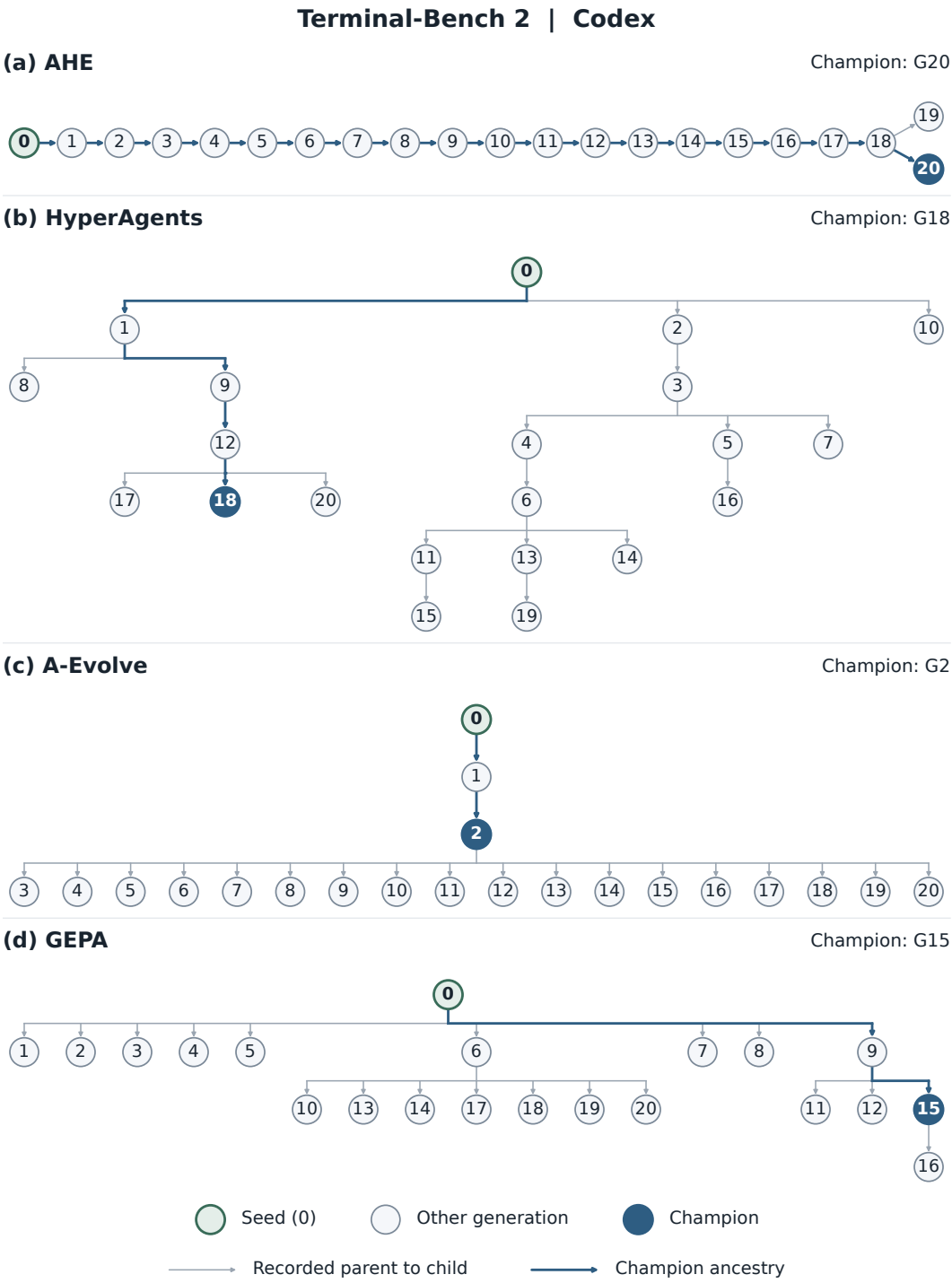


Figure 13: Evolution trees on Terminal-Bench 2 with Codex. Nodes are labeled by generation, with 0 denoting the Seed. Blue marks the Champion used in the paper and its ancestral path.