Harnessing Agentic Evolution

Jiayi Zhang[1,2], **Yongfeng Gu**[2], **Jianhao Ruan**[1,2], **Maojia Song**[3], **Yiran Peng**[2], **Zhiguang Han**[4], **Jinyu Xiang**[1], **Zhitao Wang**[5], **Caiyin Yang**[6], **Bang Liu**[7], **Chenglin Wu**[2,*], **Yuyu Luo**[1,*]
[1]The Hong Kong University of Science and Technology (Guangzhou), [2]DeepWisdom, [3]Singapore University of Technology and Design, [4]Nanyang Technological University, [5]Shanghai Jiao Tong University, [6]Tsinghua University, [7]Université de Montréal & Mila

Abstract

Agentic evolution improves programs, workflows, and agent systems through iterative generation and evaluation. Fixed search procedures can stagnate, while general-purpose agents can lose track of accumulated evidence during long runs. We formulate evolution as an interactive environment whose process state exposes candidates, feedback, traces, failures, and costs. We introduce AEVO, a harnessed meta-editing framework in which an outer meta-agent revises the search procedure or an inner agent's operating context, while the inner optimizer remains responsible for generating evaluated artifacts. A protected harness records candidates and executes budgeted evolution segments. On two agentic and reasoning benchmarks, AEVO improves over five evolution baselines, with an approximately 26% mean relative gain over the strongest baseline on each benchmark. Across three open-ended optimization tasks, it achieves the best or tied-best scores among the evaluated methods under an up-to-100-round protocol. Per-run statistics characterize variability, while intervention audits show both productive and unsuccessful mechanism edits.

1 Introduction

Agentic evolution reframes LLM-based problem solving as a process of constructing and revising solutions [17, 7]. Instead of treating the model only as a generator of candidate answers, these methods use LLMs, agentic workflows, or coding agents to drive iterative improvement: produce candidate artifacts, interpret feedback from evaluation, and influence what the system explores next [23]. This paradigm has been applied to program synthesis [11], scientific discovery [18, 40], systems optimization [26, 6], and agent self-improvement [43, 28, 24]. In this paper, we use **agentic evolution** to broadly refer to evolution processes whose search behavior is driven by either structured agentic procedures or general-purpose agents.

Existing agentic evolution methods typically instantiate this paradigm in two ways. In **procedure-based evolution**, a predefined outer loop controls parent selection, candidate generation, evaluation, and population update [19, 38, 44]. This makes evolution modular and reproducible, but also ties long-horizon search to fixed selection rules, feedback summaries, and update heuristics. In **agent-based evolution**, a general-purpose agent manages the search process by observing feedback, inspecting traces, editing candidates, writing tools, and deciding what to try next [12, 23]. This gives evolution greater flexibility, but the agent can drift as candidates, logs, hypotheses, and intermediate files accumulate. In both cases, long-horizon evolution remains prone to local optima: procedures may repeatedly exploit the same hand-designed search pattern, while agents may overcommit to misleading evidence or stale assumptions in a growing context.

*Yuyu Luo and Chenglin Wu are the corresponding authors (yuyuluo@hkust-gz.edu.cn)

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

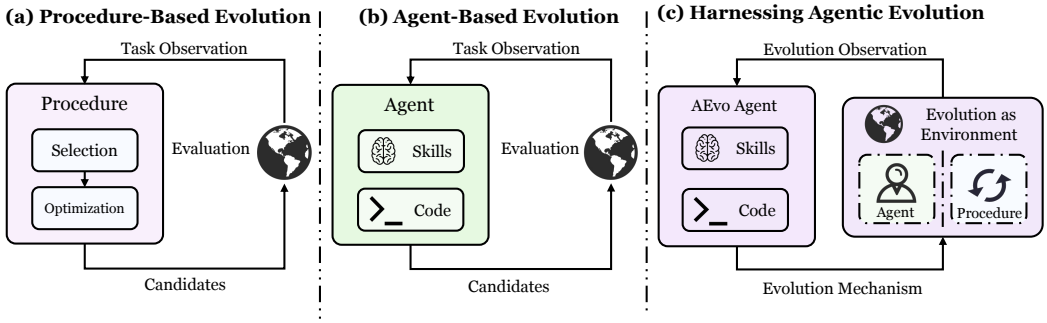


Figure 1: **Harnessing agentic evolution as an interactive environment.** (a) Procedure-based evolution runs a fixed loop for selection, optimization, evaluation, and update. (b) Agent-based evolution lets a general-purpose agent manage search through feedback, tools, skills, and code actions. (c) AEvo treats the evolution process as an interactive environment. The accumulated evolution context becomes process-level state, while a meta-agent edits the underlying procedure or agent operating context that controls future evolution.

Recent work has tried to address these limitations by either broadening exploration with collaborative agents [23] or making the evolution mechanism self-modifying [43]. These directions show that stronger search context and editable improvement mechanisms are useful, but they do not by themselves provide a stable interface for long-horizon evolution. The core challenge is that evolution accumulates candidates, feedback, traces, failures, and intermediate decisions over time, yet lacks a unified way to organize this evidence and revise the mechanism that drives future evolution.

We address this challenge by formulating **agentic evolution as an interactive environment**. As illustrated in Figure 1, this view shifts evolution from an unstructured iterative process into an environment that exposes process-level state and supports external intervention. The state is the accumulated evolution context, including candidates, feedback, traces, failures, costs, and search history. The transition mechanism is the current evolution mechanism: either an explicit search procedure or the operating context that shapes a general-purpose agent's future decisions. A meta-agent acts on this environment not by generating the next candidate, but by editing the mechanism that controls how future evolution proceeds. This makes the same environment view applicable to both hand-designed procedures and general-purpose evolution agents.

Realizing this view requires a harnessed design. The evolution environment is large, noisy, and constantly changing. Without a stable interface, a meta-agent may lose track of reliable evidence, revisit old attempts or make edits whose effects are hard to verify. At the same time, evaluation and candidate records must remain protected from the agents that modify the evolution process. These challenges motivate a harness that makes evolution observable, editable, and externally governed.

We therefore introduce AEvo, a harnessed framework for meta-editing agentic evolution. AEvo standardizes the evolution workspace, protects the evaluator, records every evaluated candidate into a searchable history, and exposes process-level information to the meta-agent. It then runs evolution through a two-phase loop. In the meta-editing phase, the meta-agent edits the current mechanism and specifies how the next segment should run. In the evolution segment, the updated mechanism runs under this plan and produces multiple candidates before the next meta-agent intervention. The same loop can revise both procedure- and agent-based evolution. The environment view defines the problem and intervention interface; numerical gains come from the concrete combination of meta-editing, segment execution, and protected evidence management, rather than from the abstraction alone.

Our contributions are threefold. (1) **Environment Formulation:** We formulate agentic evolution as an interactive environment, where accumulated evolution context becomes process-level state and meta-actions edit the mechanism that drives future evolution. (2) **Harnessed Meta-Editing:** We introduce AEvo, a harnessed framework for meta-editing agentic evolution that protects evaluation, records evaluated candidates, and supports coarse-grained intervention through meta-editing phases and evolution segments. (3) **Cross-Form Instantiation and Evaluation:** We instantiate the same framework on both procedure-based and agent-based evolution, showing that AEvo can revise either explicit procedure components or agent operating contexts. On two standard benchmarks, AEvo

improves over five evolution baselines by approximately 26% on average relative to the strongest baseline on each benchmark. On three open-ended tasks, it achieves best or tied-best scores among the evaluated methods under the same round cap. Additional cost comparisons, per-run statistics, and failure analyses qualify these results.

2 Related Work

Agentic Evolution. A growing line of work uses LLMs and agents to iteratively improve artifacts through generation, feedback, and revision [14, 16, 15, 29]. Prompt methods optimize language-model programs or feedback-driven prompts, including DSPy [13], SPO [31], TextGrad [39], and GEPA [1]. Another line studies the automated design and evolution of agentic systems and workflows, such as ADAS [10], Darwin Gödel Machine [42], Huxley-Gödel Machine [28], AFlow [44], RobustFlow [33], and SkillRL [30]. Recent open-ended discovery systems further apply evolutionary search to scientific and algorithmic discovery, including AlphaEvolve [19], OpenEvolve [25], TTS-Discover [40], CORAL [23], SimpleTES [38], and ASI-Evolve [34]. These systems differ in how much search control is fixed or delegated to agents; several also adapt prompts, memory, or tools. AEVO focuses on a bounded interface separating mechanism editing from artifact generation, rather than claiming that runtime adaptation is unique to our method. GEPA also reuses reflective feedback to improve prompts, so history-aware context construction alone does not establish a distinct causal mechanism.

Agentic Meta-Evolution. Early meta-learning work showed that the learning rule itself can be optimized, for example by learning recurrent reinforcement-learning dynamics [27] or evolving policy-gradient objectives [9]. Recent agentic systems extend this idea to editable agent programs and memory systems. HyperAgents study self-referential agent programs in which both task-solving behavior and the meta-improvement mechanism can be modified [43]. MemEvolve and ALMA similarly explore meta-evolution over agent memory designs [41, 32]. Unlike HyperAgents, which internalize meta-improvement within a self-modifying agent program, AEVO treats agentic evolution as an interactive environment observed and edited through an external harness, covering both hand-designed procedures and general-purpose agents while keeping evaluation and candidate recording externally governed.

Online Adaptation and Meta-Optimization. EvoTest [8] separates an Actor from an Evolver that revises prompts, memory, hyperparameters, and tool-use routines after episodes. AEVO shares this execution/adaptation separation, while exposing explicit procedure edits and multi-round run plans through an external harness. Automated algorithm configuration and adaptive operator control also change search behavior; unlike a predefined parameter or operator space, our editable surface includes executable procedure structure and agent context. The learned systems in Wang et al. [27], Houthooft et al. [9] optimize recurrent learning dynamics or update objectives through meta-training; AEVO instead edits an inspectable mechanism online within a run. These are differences in intervention interface and deployment regime, not a claim of formal dominance.

3 Problem Formulation

3.1 Agentic Evolution

We formulate agentic evolution as a process for optimizing an artifact through repeated improvement rounds. Let $x \in \mathcal{X}$ denote the object being optimized, such as a program, prompt, workflow, skill, tool, or agent component. We use r to index the evolution round. Each round produces a round context c_r, which contains the candidates generated in that round, their evaluation results, execution traces, failures, costs, and any intermediate information produced during optimization. The accumulated evolution context after r rounds is denoted as

$$\mathcal{C}_r = (c_1, c_2, \ldots, c_r).$$

Finally, let Π denote the optimization mechanism that advances evolution:

$$c_r = \Pi(\mathcal{C}_{r-1}), \qquad \mathcal{C}_r = \mathcal{C}_{r-1} \oplus c_r,$$

where $\oplus$ appends the newly produced round context to the accumulated evolution context. Π does not have to be a fixed algorithm; it can also be an agentic process that reads the history, reasons over

feedback, and decides how to generate the next candidate. Thus, Π represents the mechanism by which search is continued from the current evolution context.

Under this formulation, existing agentic evolution methods mainly differ in how Π is instantiated. In **procedure-based evolution**, Π is a predefined outer loop whose behavior is mainly determined by selection and optimization: the selection rule chooses previous candidates or contexts from $\mathcal{C}_{r-1}$, while the optimization operator generates new candidates from the selected information. Evaluation assigns scores, traces, and feedback to the generated candidates, providing signals for future selection and update. In **agent-based evolution**, Π is instead implemented by a general-purpose agent. Rather than following fixed selection-and-optimization rules, the agent reads the accumulated context $\mathcal{C}_{r-1}$ and decides what to do next, such as inspecting feedback, comparing candidates, modifying artifacts, writing tools, or generating new attempts. Thus, procedure-based evolution specifies search control explicitly but rigidly, while agent-based evolution leaves search control implicit in the agent's context-conditioned behavior.

In both cases, evolution proceeds by repeatedly applying Π while accumulating context $\mathcal{C}_r$. This context records not only the candidates produced by evolution, but also how search has unfolded through evaluation results, traces, failures, costs, and intermediate artifacts. The next subsection uses this accumulated context to define an environment view of evolution.

3.2 Evolution as an Interactive Environment

We treat the evolution process itself as an interactive environment for a meta-agent. At round r, the process state includes the round index, accumulated evidence, and current mechanism. Including the mechanism makes explicit that identical histories can induce different future search behavior:

$$s_r = (r, \mathcal{C}_r, \Pi_r).$$

When the optimization mechanism may change across rounds, we write the current mechanism as Π_r. This mechanism specifies the transition of the environment. Without intervention, the next round is produced by applying the current mechanism to the current context:

$$c_{r+1} = \Pi_r(\mathcal{C}_r), \qquad s_{r+1} = (r+1, \mathcal{C}_r \oplus c_{r+1}, \Pi_r).$$

Thus, Π_r is the transition rule that determines how the evolution process continues.

To interact with this environment, we introduce a meta-agent M. The role of Π_r is to continue the candidate search, while the role of M is to act on the evolution process that governs this search. Since the full state s_r can be large and noisy, the meta-agent receives an observation extracted from the state:

$$o_r = \Phi(s_r) = \Phi(r, \mathcal{C}_r, \Pi_r),$$

where Φ is the harness-provided structured observation interface, not a separately trained summarizer. It exposes relevant history and mechanism files, including progress, failures, invalid attempts, costs, and redundant directions. The transition notation suppresses sampling randomness and task-evaluator calls for readability.

Given this observation, the meta-agent produces an edit action:

$$a_r = M(o_r).$$

The action does not directly become the next candidate. Instead, it modifies the transition rule of the evolution environment:

$$\Pi_{r+1} = \mathrm{Edit}(\Pi_r, a_r).$$

The edited mechanism is then used to continue evolution:

$$c_{r+1} = \Pi_{r+1}(\mathcal{C}_r), \qquad \mathcal{C}_{r+1} = \mathcal{C}_r \oplus c_{r+1}.$$

In this sense, we formulate agentic evolution as an environment in which the state is the accumulated evolution context, the observation is a summary of this context, and the action edits the mechanism that controls future search.

This formulation applies to both forms of agentic evolution. For procedure-based evolution , editing Π_r changes explicit components such as selection, optimization, feedback use, budget allocation, or update rules. For agent-based evolution, editing Π_r changes the agentic context that shapes future decisions, such as skills, goals, tools, feedback format, or execution context. In both cases, the meta-agent steers evolution not by proposing one more candidate, but by modifying how subsequent search is carried out. Section 4 describes the system design used to instantiate this formulation.

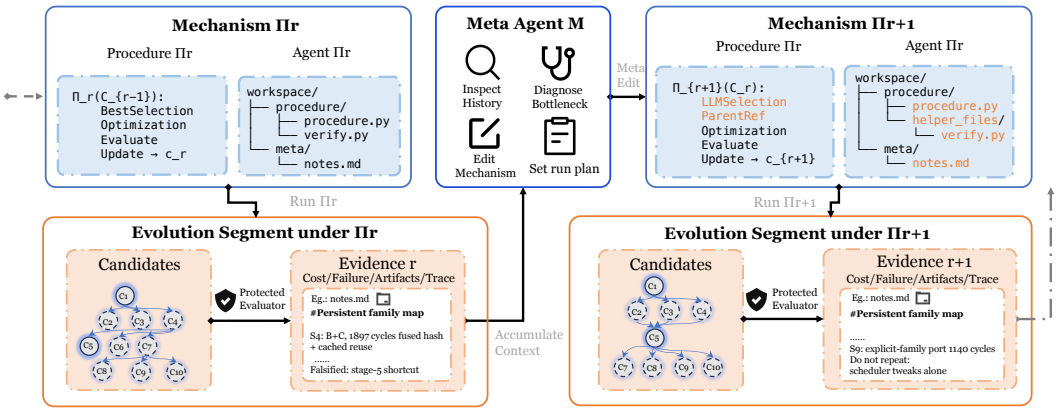


Figure 2: **Architecture of AEvo.** The harness runs evolution segments under the current mechanism Π_r, protects evaluation, and records structured evidence. A meta-agent observes this evidence to edit Π_r into Π_{r+1} and set the next run plan, enabling coarse-grained intervention over both procedures and agent contexts.

4 Methodology

Figure 2 illustrates AEvo. AEvo instantiates the environment view in Section 3 as a harnessed loop that alternates between a *meta-editing phase* and an *evolution segment*. In the meta-editing phase, the meta-agent updates the current evolution mechanism Π_r and specifies how the next segment should run, including its iteration budget and stopping conditions. In the evolution segment, the updated mechanism runs under this plan and may produce multiple evaluated candidates before the next meta-agent intervention. Thus, one meta-edit can govern a segment of future evolution rather than a single candidate.

4.1 Design of AEvo

Meta-editing Phase. The meta-editing phase decides both **what to change** and **how to continue**. The meta-agent can be any coding-capable agent that can inspect a workspace, edit files, execute commands, and follow the AEvo meta-agent skill specification, such as Claude Code [2], Codex [20], or open-source coding agents [36, 21]. Given the current workspace, it inspects the accumulated history, then produces a meta-action consisting of a workspace edit and a run plan. The workspace edit modifies files that define Π_r, such as procedure code, prompts, skills, goals, tools, feedback formats, validators, notes, or execution context. The run plan specifies how the next evolution segment should proceed, including the allowed iterations, budget use, and stopping conditions.

The boundary is operational: the meta-agent may edit mechanism files and the run plan, but may not edit submitted task artifacts or invoke official evaluation directly. For example, changing an inner agent's prompt to govern subsequent search edits the mechanism; changing a prompt that is itself the evaluated artifact generates a candidate. The skill organizes intervention as Read–Attribute–Choose Action–Run–Record, retaining process evidence outside the inner context. When evolution is productive, the meta-agent may allocate more iterations to the current mechanism; when it repeatedly produces invalid candidates, redundant attempts, or irrelevant exploration, the meta-agent may stop the segment and revise Π_r before continuing. A representative operational skill excerpt is provided in Appendix C.2.3.

Harnessed Evolution Segment. An evolution segment is the interval executed after a meta-edit. It runs the current mechanism Π_r under the run plan produced by the meta-agent. Depending on the setting, this segment may consist of several rounds of a procedure, or an inner-agent session that produces multiple candidate attempts. Each candidate submitted for official evaluation passes through the harness-controlled evaluator, and the resulting artifact, score, trace, failure information, cost, and provenance are appended to the candidate history.

The harness provides the stable boundary needed for reliable meta-editing. It organizes candidates, logs, traces, evaluation records, meta-agent instructions, and editable evolution components into a

fixed workspace layout. To reduce evaluator tampering, the evaluator is isolated from both agents: the inner optimizer submits candidates through the gateway, while neither agent may inspect protected evaluator internals, access hidden benchmark artifacts, or directly write official scores. Editable local validators are distinct from the protected official evaluator. This boundary addresses the evaluated threat model; it is not a guarantee against all reward hacking. The harness further exposes a command-line interface for initializing workspaces, launching evolution segments, inspecting recent status and candidate history, and continuing the current process. Thus, the harness does not decide how evolution should improve; it provides the protected and inspectable interface through which evolution can be observed, edited by the meta-agent, recorded, and resumed.

4.2 Instantiating AEvo

AEvo applies the same two-phase loop to both forms of agentic evolution. The outer loop is unchanged: the meta-agent edits the current mechanism Π_r and specifies how the next evolution segment should run. The difference lies in what Π_r consists of.

Procedure-based Evolution. For procedure-based evolution, Π_r is an explicit evolution procedure. It defines how previous candidates or contexts are selected, how new candidates are generated, how evaluation feedback is used, and how candidate history is updated. A meta-action therefore edits the procedure itself, such as revising the selection strategy, changing the optimization operator, altering the feedback summary, adding local filtering or retry logic, adjusting budget use, or repairing candidate management. The edited procedure then controls the next evolution segment, which may run for multiple rounds before the next meta-agent intervention.

Agent-based Evolution. For agent-based evolution, Π_r is the operating context of a general-purpose evolution agent, including goals, skills, tools, memory files, shared notes, validators, and execution setup. A meta-action therefore edits the conditions under which the next inner-agent session will evolve, such as revising a skill, rewriting the session goal, changing how evaluator feedback is presented, or reorganizing shared notes. The inner agent remains responsible for generating candidates, while AEvo revises the context that shapes future evolution.

5 Experiments

5.1 Settings

Tasks. We evaluate AEvo on two standard benchmarks, Terminal-Bench [26] and ARC-AGI-2 [5], and three open-ended optimization tasks, circle_packing_26 (CP26)[22], autocorrelation_second (AC2)[4], and Anthropic's Kernel optimization task [3]. Together, these tasks cover agentic problem solving, abstract reasoning, and code-evolving open-ended optimization. Detailed task definitions and evaluation protocols are given in Appendix C.1.

Baselines. We group baselines into three families: single-agent inference, agent-based evolution, and procedure-based evolution. On Terminal-Bench and ARC-AGI-2, we compare against one-shot ReAct [37] and five procedure-based evolution baselines: ADAS [10], DGM [42], AFlow [44], SPO [31], and GEPA [1]. On the three open-ended tasks, we compare against two agent-based evolution baselines, Codex and Claude Code, and two procedure-based evolution baselines, OpenEvolve [25] and HyperAgents [43]. This setup lets us compare both variants of AEvo against systems that either keep a fixed search procedure or rely on an agent to improve artifacts directly.

Implementation Details. AEvo is instantiated in two forms. In the procedure-based setting, a meta-agent edits the evolution procedure while leaving the task evaluator fixed. In the agent-based setting, the meta-agent steers a coding-agent harness through prompts, notes, and reusable utilities. We use Claude Code and Codex as the meta-agent interfaces, backed by Claude-Opus-4.7 and GPT-5.4 as the optimization models. For Terminal-Bench and ARC-AGI-2, candidate execution uses Gemini-3-Flash. All models are accessed through APIs. Within each comparison, methods use the same task evaluator and raw feedback, matched optimization/execution models, and a common round cap while retaining their native evolution logic. Full settings, stopping criteria, and initialization details are in Appendix B. Round matching alone does not equalize total cost.

Metrics. For Terminal-Bench and ARC-AGI-2, we report task score and the first optimization round that reaches the best score. Results are summarized with Avg@3 over three independent

Table 1: Open-ended optimization results. Task 1/2/3 correspond to circle_packing_26, autocorrelation_second, and Kernel optimization task. Scores denote Best@3 over three runs using the model shown in the Model column, Best R. denotes the first round reaching the reported best score, and $/R denotes the average cost per optimization round. Arrows indicate optimization direction; **bold** and underline mark the best and second-best scores.

Category	Method	Model	Task 1 ↑			Task 2 ↑			Task 3 ↓		
			Score	Best R.	$/R	Score	Best R.	$/R	Score	Best R.	$/R
Agent-Based Evolution	Codex	GPT-5.4	**2.6359**	3	0.82	0.9176	96	0.04	1667	4	0.96
	Claude Code	Claude-Opus-4.7	2.6305	50	0.78	0.9438	44	0.81	1615	97	0.51
Procedure-Based Evolution	OpenEvolve	Claude-Opus-4.7	2.6303	80	0.42	0.9186	99	0.67	2411	99	0.62
	HyperAgents	Claude-Opus-4.7	**2.6359**	32	9.50	0.9245	48	2.83	7086	86	1.56
	OpenEvolve	GPT-5.4	2.6341	19	0.23	0.9118	74	0.54	2464	100	0.57
	HyperAgents	GPT-5.4	2.6359	47	3.19	0.9237	61	1.46	3015	98	1.03
Ours	AEvo `Procedure`	Claude-Opus-4.7	**2.6359**	4	1.47	0.9278	29	0.70	1803	55	1.37
	AEvo `Agent`	Claude-Opus-4.7	**2.6359**	2	0.34	**0.9459**	99	1.40	1519	55	1.27
	AEvo `Agent`	GPT-5.4	**2.6359**	17	0.32	0.9398	100	1.31	**1138**	99	1.23

Table 2: Standard benchmark results. Scores denote Avg@3 over three runs with Gemini-3-Flash as the execution model. For AEvo, $\pm$ denotes sample SD across runs; baseline SDs are not reported here. Best R. follows the original first-best-round reporting. The **bold** and underline mark the best and second-best scores.

Category	Method	Model	Terminal-Bench ↑		ARC-AGI-2 ↑	
			Score	Best R.	Score	Best R.
Single-Agent Inference	ReAct `Pass@1`	Gemini-3-Flash	28.6	–	21.8	–
Procedure-Based Evolution	ADAS	Gemini-3-Flash	38.6	7	36.0	3
	DGM	Gemini-3-Flash	44.3	19	29.8	5
	AFlow	Gemini-3-Flash	44.3	11	31.8	14
	SPO	Gemini-3-Flash	42.9	19	25.0	6
	GEPA	Gemini-3-Flash	41.4	15	22.5	13
Ours	AEvo `Procedure`	Gemini-3-Flash	**53.8 ± 2.5**	7	**47.0 ± 1.18**	12

runs. Appendix D additionally reports every available AEvo run and sample standard deviations; Best@3 measures discovery performance, whereas Avg@3 characterizes single-run outcomes. For the open-ended tasks, we report the task-native objective with Best@3 over three runs, together with the first round that reaches the best result (Best R.) and the average dollar cost per optimization round ($/R). Exact task-specific objectives and cost computation are provided in Appendix C.1.

5.2 Main Results

Overall performance. Tables 1 and 2 show best or tied-best open-ended scores among the evaluated methods. On Kernel, AEvo reaches 1138 cycles within 100 rounds. This is a within-suite result: CORAL [23] reports 1350 cycles with one agent and 1103 with four under a different protocol, so we do not claim an unconditional state of the art. On the standard benchmarks, the relative gains over each task's strongest baseline are $53.8/44.3 - 1 = 21.4\%$ and $47.0/36.0 - 1 = 30.6\%$, averaging approximately 26%. This unweighted mean of task-wise relative gains is distinct from the ARC-only cost comparison below.

Cost-matched evidence. Both methods in Table 3 use Opus-4.7 for optimization and Gemini-3-Flash with high reasoning for execution. At 20 rounds, AEvo uses $3.1\times$ the optimization cost but only 20.6% higher total cost; evaluation accounts for 91.4% of its total. ADAS's nearest checkpoint costs 3.2% more than AEvo and remains at 36.0; extending it to 40 rounds also leaves its best

Table 3: Supplementary ARC-AGI-2 cost comparison. Costs are USD; optimization and evaluation are reported separately. These checkpoints provide a near dollar-cost match, not an exact token or FLOP match.

Method	Rounds	Optim.	Eval.	Total	Score
ADAS	20	7.32	210.14	217.46	36.0
AEvo	20	22.57	239.60	262.17	47.0
ADAS, nearest cost	25	10.63	259.83	270.46	36.0
ADAS, extended	40	16.71	430.14	446.85	36.0

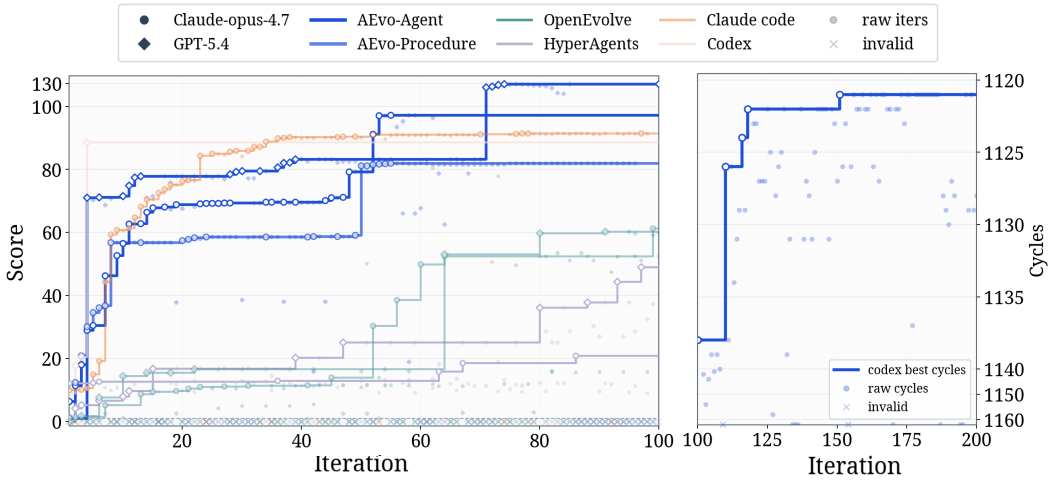


Figure 3: Evolution trajectories on the *Kernel optimization* task. The left panel compares eight methods over the first 100 iterations, where blue curves denote AEvo variants. The y-axis reports the normalized score induced by cycle reduction, so higher is better; raw iterations and invalid evaluations are shown as scattered markers. The right panel extends the AEvo run from 100 to 200 iterations and reports raw cycles, where lower is better.

test score unchanged. This evidence weakens the additional-fixed-procedure-rounds explanation on ARC-AGI-2. It does not establish cost superiority on every task or isolate which AEvo component causes the gain.

Cost analysis and agentic behavior. The open-ended tasks reveal that cost is not solely determined by whether a method is agent-based or procedure-based. Agent-based evolution can remain cost-competitive when implemented through coding-agent interfaces with prompt caching and persistent contexts. AEvo `Agent` maintains low per-round cost: 0.34–0.32 on *circle_packing_26*, 1.40–1.31 on *autocorrelation_second*, and 1.27–1.23 on Kernel optimization. By contrast, procedure-based methods can become expensive in long-horizon optimization by repeatedly constructing large prompts over an expanding search history without comparable caching, as visible in HyperAgents' higher per-round cost on Task 1 and Task 2.

At the same time, direct coding agents show why agent freedom alone is insufficient for reliable evolution. Even with prompts encouraging long-horizon search, coding agents often stop early once local improvements become difficult, as seen in the early best rounds of Codex on Task 1 and Task 3. An early best round alone does not establish when a run stopped; the ablation completion counts provide direct evidence of incomplete budget use. AEvo addresses this by placing the coding agent inside an explicit evolution harness, where rounds, candidate records, and evaluation feedback are maintained outside the agent's local decision loop.

5.3 Evolution Dynamics

Revising evolution after plateaus. As shown in Figure 3, procedure-based methods such as OpenEvolve and HyperAgents tend to flatten once their current selection or mutation strategy stops

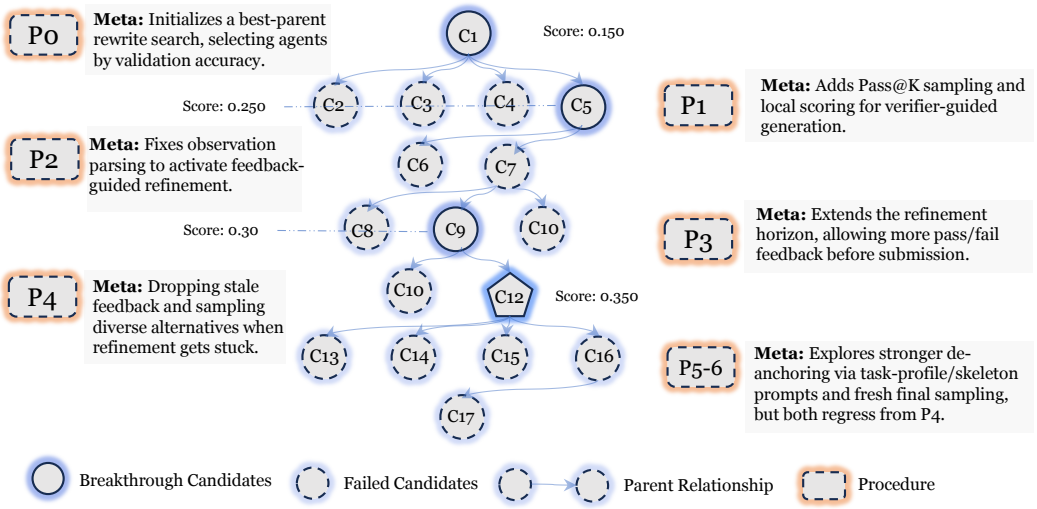


Figure 4: Case study of procedure evolution on an ARC-AGI-2 task. Each P denotes a procedure produced or revised by the meta-agent, and each C denotes a candidate agent generated by the current procedure. Solid nodes are breakthrough candidates, dashed nodes are failed candidates, and arrows indicate parent relationships. The meta-agent improves the search process by changing the procedure across stages, while failed candidates provide feedback for later interventions.

producing useful candidates. In contrast, AEvo can revise the mechanism that drives subsequent evolution. When progress stalls or repeated failures appear, these signals become process-level feedback: the meta-agent can adjust the procedure, directive, or reusable search context, producing step-wise improvements after plateaus. This is visible in the late-stage jump that leads to the best 100-round result.

Using the evolution budget effectively. Direct coding agents can obtain strong early gains through internal simulation, execution, and debugging, but they may stop early once local progress becomes difficult. AEvo avoids tying progress to the agent's local stopping decision by maintaining explicit rounds, candidate records, and evaluation feedback outside the agent context. This allows the external evolution budget to be used more consistently.

Scaling beyond early gains. The right panel extends Codex-based AEvo from 100 to 200 iterations. The best result improves from 1138 to 1121 cycles, showing that AEvo continues to benefit from additional rounds rather than saturating after an early strong candidate. This extended trajectory is additional evidence of improvement beyond 100 rounds, not a matched-budget comparison with the 100-round baseline table. The strong-seed experiment starts a fresh 100-round run from the 1121-cycle artifact and reaches 1108; this uses a prior AEvo artifact and must not be ranked as a from-scratch 100-round result.

5.4 Ablation Study

Table 4 ablates meta-agent skills and the evolution harness on Kernel. Full AEvo reaches 1138 cycles in 100 rounds. Without the skill, runs end at 37/99/65 rounds and the best valid result is 1407. Without the harness, one run reaches 1167 but two of three exhibit reward hacking. These bundled removals support the implementation's usefulness and evaluator integrity; they do not causally separate persistent state, evaluator protection, segment control, and meta-editing.

Sensitivity and initialization. Keeping the Opus-4.7 outer meta-agent and 20-round ARC budget fixed, replacing only the inner optimizer with Sonnet-4.6 yields 43.3 ± 1.35, versus 47.0 ± 1.18 with Opus. Both exceed ADAS's 36.0, although the comparison does not isolate all context-engineering effects. Starting AEvo from ADAS's discovered structure and procedure yields 42.9 ± 0.98 over three additional 10-round runs. Without matched-budget native ADAS continuation from the same checkpoint, this is strong-initialization evidence rather than causal attribution. Full statistics are in

Appendix D. Segment length is an adaptive run-plan action; its benefit has not been independently established against fixed segment schedules.

Intervention outcomes. An audit of a 100-evaluation Kernel run finds 5/8 interventions improve the global best, 2/8 do not, and 1/8 is regressive or invalid. In a 16-round ARC run, the corresponding counts are 3/7, 2/7, and 2/7. These selected traces describe unsuccessful interventions as well as improvements; they do not establish that every diagnosis is causally correct. The harness retains the best valid artifact, but unsuccessful edits can still waste budget. Appendix D reports examples and the audit's scope.

5.5 Case Study

Figure 4 illustrates how AEVO performs meta-intervention during procedure-based evolution on an ARC-AGI-2 task. Starting from P_0, the meta-agent initializes a best-parent rewrite procedure that selects candidate agents by validation accuracy. This first produces an initial breakthrough candidate C_1, but subsequent variants expose several failure modes in observation parsing and refinement. The meta-agent then revises the procedure rather than continuing the same search blindly: P_1 adds Pass@K sampling and local scoring for verifier-guided generation, P_2 fixes observation parsing to activate feedback-guided refinement, and P_3 extends the refinement horizon to use more pass/fail feedback before submission. When the search becomes stuck, P_4 drops stale feedback and samples more diverse alternatives, leading to a stronger candidate. Later interventions P_5–P_6 explore stronger de-anchoring through task-profile and skeleton prompts, but these regress from P_4. This example shows that failed candidates are not merely discarded; they become process-level evidence that helps the meta-agent decide how to revise the future evolution procedure. Additional evolved procedures, optimized agent harnesses, and task-level analyses are provided in Appendix C.3.1 and Appendix C.3.2.

6 Conclusion

We presented AEVO, a harnessed framework for editing the mechanism that drives agentic evolution. An outer meta-agent revises explicit procedures or an inner agent's operating context, while the harness preserves process evidence and protects official evaluation. Results across agentic, reasoning, and open-ended tasks support this concrete implementation, with an additional near-cost-matched comparison on ARC-AGI-2. Per-run statistics and failure audits also expose variability and unsuccessful edits. Future work should independently test component contributions and extend evaluation to cheaper agents, longer runs, and noisier feedback.

7 Limitations and Broader Impacts

Our evaluation covers a small task suite and three runs per main setting; baseline variances and broad statistical significance are not established. Best@3 emphasizes discovery rather than expected reliability. Dollar costs depend on model pricing, caching, and interface accounting; the near-cost-matched result covers ARC-AGI-2 only. The skill/harness ablations remove bundles of capabilities and do not separately identify the contributions of mechanism editing, persistent state, segment control, or evaluator protection. Strong-seed experiments lack native continuation controls. Model-stack comparisons also change the agent interface, so they are not isolated outer-model ablations.

The implemented workflows rely mainly on frontier proprietary agents; complete runs with lightweight open-source harnesses remain untested. Most tasks have automated candidate-boundary evaluation. PaperReview's supplementary outcome-prediction result does not establish scientific-review quality, creative discovery, or robustness to human feedback. One 200-round extension does not characterize arbitrary horizons. Protected evaluation reduces the observed tampering risk but does not guarantee that generated code is safe or that every meta-edit helps.

The framework may improve software optimization and evidence reuse, but autonomous code execution can introduce vulnerabilities or consume substantial resources. Deployment should retain sandboxing, access controls, audit records, budget limits, and human review before using generated artifacts in consequential settings. Paper-review outcome prediction should not replace expert assessment or be used as an automated acceptance decision.

8 Acknowledgements

This paper was supported by the NSF of China (62402409); Youth S&T Talent Support Programme of Guangdong Provincial Association for Science and Technology (SKXRC2025461); the Young Talent Support Project of Guangzhou Association for Science and Technology (QT-2025-001); Guangzhou Basic and Applied Basic Research Foundation (2026A1515010269, 2025A04J3935); and Guangzhou-HKUST(GZ) Joint Funding Program (2025A03J3714).

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[2] Anthropic. Claude Code, 2025. `https://docs.anthropic.com/en/docs/claude-code/overview`.

[3] Anthropic PBC. Anthropic's Original Performance Take-Home. `https://github.com/anthropics/original_performance_takehome`, January 2026. GitHub repository, commit 5452f74. Accessed: 2026-05-06.

[4] Christopher Boyer and Zane Kun Li. An improved example for an autoconvolution inequality. *Experimental Mathematics*, pages 1–7, 2026.

[5] Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

[6] Mingyi Deng, Lijun Huang, Yani Fan, Jiayi Zhang, Fashen Ren, Jinyi Bai, Fuzhen Yang, Dayi Miao, Zhaoyang Yu, Yifan Wu, et al. Interactcomp: Evaluating search agents with ambiguous queries. *arXiv preprint arXiv:2510.24668*, 2025.

[7] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

[8] Yufei He, Juncheng Liu, Yue Liu, Yibo Li, Tri Cao, Zhiyuan Hu, Xinxing Xu, and Bryan Hooi. EvoTest: Evolutionary test-time learning for self-improving agentic systems. *arXiv preprint arXiv:2510.13220*, 2025. URL `https://arxiv.org/abs/2510.13220`.

[9] Rein Houthooft, Yuhua Chen, Phillip Isola, Bradly Stadie, Filip Wolski, OpenAI Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. *Advances in Neural Information Processing Systems*, 31, 2018.

[10] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

[11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[12] Andrej Karpathy. autoresearch: Ai agents running research on single-gpu nanochat training automatically. GitHub repository, 2026. Accessed: 2026-05-06.

[13] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. 2024.

[14] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. In *ICML*. OpenReview.net, 2025.

[15] Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. Deepeye-sql: A software-engineering-inspired text-to-sql framework. *Proc. ACM Manag. Data*, 4(3):158:1–158:28, 2026. doi: 10.1145/3802035. URL `https://doi.org/10.1145/3802035`.

[16] Boyan Li, Yiran Peng, Yupeng Xie, Sirong Lu, Yizhang Zhu, Xing Mu, Xinyu Liu, and Yuyu Luo. Deepeye: A steerable self-driving data agent system. In *Companion of the 2026 International Conference on Management of Data*, SIGMOD Companion '26, Bengaluru, India, 2026. ACM. doi: 10.1145/3788853.3801612.

[17] Bang Liu, Xinfeng Li, Jiayi Zhang, Jinlin Wang, Tanjin He, Sirui Hong, Hongzhang Liu, Shaokun Zhang, Kaitao Song, Kunlun Zhu, et al. Advances and challenges in foundation agents: From brain-inspired intelligence to evolutionary, collaborative, and safe systems. *arXiv preprint arXiv:2504.01990*, 2025.

[18] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

[19] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[20] OpenAI. Codex, 2025. `https://openai.com/index/introducing-codex/`.

[21] OpenCode. OpenCode: The open source AI coding agent, 2025. `https://opencode.ai`.

[22] Ronald Peikert, Diethelm Würtz, Michael Monagan, and Claas de Groot. Packing circles in a square: A review and new results. In *System Modelling and Optimization: Proceedings of the 15th IFIP Conference Zurich, Switzerland, September 2–6, 1991*, pages 45–54. Springer, 2007.

[23] Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, et al. Coral: Towards autonomous multi-agent evolution for open-ended discovery. *arXiv preprint arXiv:2604.01658*, 2026.

[24] Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, et al. Aorchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026.

[25] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL `https://github.com/algorithmicsuperintelligence/openevolve`.

[26] The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025. URL `https://github.com/laude-institute/terminal-bench`.

[27] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[28] Wenyi Wang, Piotr Piekos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-g\" odel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. *arXiv preprint arXiv:2510.21614*, 2025.

[29] Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, Liang Chen, Yuyao Zhai, Bang Liu, Chenglin Wu, and Yuyu Luo. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines, 2026. URL `https://arxiv.org/abs/2602.14296`.

[30] Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

[31] Jinyu Xiang, Jiayi Zhang, Zhaoyang Yu, Fengwei Teng, Jinhao Tu, Xinbing Liang, Sirui Hong, Chenglin Wu, and Yuyu Luo. Self-supervised prompt optimization. *arXiv preprint arXiv:2502.06855*, 2025.

[32] Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026.

[33] Shengxiang Xu, Jiayi Zhang, Shimin Di, Yuyu Luo, Liang Yao, Hanmo Liu, Jia Zhu, Fan Liu, and Min-Ling Zhang. Robustflow: Towards robust agentic workflow generation. *arXiv preprint arXiv:2509.21834*, 2025.

[34] Weixian Xu, Tiantian Mi, Yixiu Liu, Yang Nan, Zhimeng Zhou, Lyumanshan Ye, Lin Zhang, Yu Qiao, and Pengfei Liu. Asi-evolve: Ai accelerates ai. *arXiv preprint arXiv:2603.29640*, 2026.

[35] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025. URL https://arxiv.org/abs/2504.08066.

[36] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

[37] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[38] Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Caiyin Yang, Chang Su, Rahul Thapa, Rui Yang, Ruihua Liu, Zeyu Li, et al. Evaluation-driven scaling for scientific discovery. *arXiv preprint arXiv:2604.19341*, 2026.

[39] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[40] Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

[41] Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025.

[42] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

[43] Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026.

[44] 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, 2025. URL https://openreview.net/forum?id=z5uVAKwmjf.

[45] Bingchen Zhao, Jenny Zhang, Chenxi Whitehouse, Minqi Jiang, Michael Shvartsman, Abhishek Charnalia, Despoina Magka, Tatiana Shavrina, Derek Dunfield, Oisin Mac Aodha, and Yoram Bachrach. APRES: An agentic paper revision and evaluation system. *arXiv preprint arXiv:2603.03142*, 2026. URL https://arxiv.org/abs/2603.03142.

A Ablation Study Details

Table 4: Ablation study on the Kernel optimization task. Full reports the main AEVO $_{\texttt{Agent}}$ setting, while each ablation reports three independent runs. Total R. denotes completed evolution rounds, and Best R. denotes the first round reaching the reported best cycles.

Method	Run	Reward Hack	Total R.	Cycles ↓	Best R.	Invalid / Total
Full	–	No	100	**1138**	99	16/100
w/o Meta-Agent Skills	1	No	37	2379	18	2/37
	2	No	99	1536	99	28/99
	3	No	65	1407	53	21/65
w/o Evolution Harness	1	No	100	1167	81	19/100
	2	Yes	100	N/A	N/A	47/100
	3	Yes	57	N/A	N/A	22/57

B Implementation Details

When temperature is exposed, we set it to 1; when reasoning-effort control is available, we use the high setting; and we use a maximum context budget of 128k tokens. We run 20 optimization rounds on Terminal-Bench and ARC-AGI-2. On open-ended optimization tasks, we allow up to 100 rounds and early-stop when a run reaches the known target score or fails to improve for 25 consecutive rounds. The initial procedure in the procedure-based setting uses best-valid-candidate selection plus a heuristic LLM optimizer; details are given in Appendix C.2.1. The initial agent used on the standard agentic and reasoning tasks is a ReAct-style [37] agent; details are given in Appendix C.2.2.

C Additional Experimental Details

C.1 Task and Metric Details

Our main experiments cover two standard benchmarks and three open-ended optimization tasks. Terminal-Bench evaluates end-to-end task completion in terminal environments, while ARC-AGI-2 measures abstract reasoning under fixed evaluation rules. The three open-ended tasks use hidden or fixed external evaluators and require the optimizer to improve executable code rather than only produce one-shot answers.

Circle Packing. In *circle_packing_26*, the goal is to pack 26 circles into a unit square and maximize the sum of their radii. The evaluator returns a validity bit and the achieved packing score. Higher is better.

Autocorrelation-Second. In *autocorrelation_second*, the goal is to construct a non-negative function on $[-1/4, 1/4]$ that maximizes

$$R(f) = \frac{\|f * f\|_2^2}{\|f * f\|_1 \cdot \|f * f\|_\infty}.$$

The evaluator scores the submitted construction directly by this ratio. Higher is better.

Performance Engineering. In the performance-engineering take-home, the goal is to optimize a kernel for a simulated VLIW SIMD machine while preserving correctness on the hidden tests. We report raw cycle count in the main table, so lower is better, although the evaluator also exposes a normalized score derived from the cycle count.

Reporting Protocol. For Terminal-Bench and ARC-AGI-2, we report Avg@3 across three independent runs, together with the first optimization round that reaches the best score. For open-ended optimization, we report Best@3 across three runs under a fixed evaluation budget, together with the

first round that reaches the best result and the average dollar cost per optimization round. Open-ended runs are capped at 100 rounds and are early-stopped if they reach the known target or plateau for 25 consecutive rounds.

Cost Computation. For a run with R optimization rounds, the average dollar cost per round is

$$\$/R = \frac{1}{R}\sum_{r=1}^{R}\left(p_{\text{in}}n_{\text{in}}^{(r)} + p_{\text{cache}}n_{\text{cache}}^{(r)} + p_{\text{out}}n_{\text{out}}^{(r)}\right),$$

where $n_{\text{in}}^{(r)}$, $n_{\text{cache}}^{(r)}$, and $n_{\text{out}}^{(r)}$ denote the input, cached-input, and output token counts in round r, and p_{in}, p_{cache}, and p_{out} are the corresponding provider prices.

C.2 Initialization Details

C.2.1 Initial Procedure in Procedure-Based AEVO

The initial procedure used by procedure-based AEVO is intentionally minimal. It selects the current best valid candidate as the parent, applies a single LLM rewrite step, and then invokes the fixed evaluator. The excerpt below shows the corresponding editable surface.

```
class BestByScoreSelection(Selection):
    async def select(self, harness: Harness) -> InfoBundle:
        latest = harness.latest()
        if latest is None:
            seed = harness.new_candidate(parent_round=None)
            harness.write_artifact(
                seed, SOLUTION_FILE, _read(harness.workspace / SEED_FILE)
            )
            seed_metrics = await ReadEvalEvaluation().evaluate(seed)
            harness.record_metrics(seed.round, seed_metrics)
            latest = seed

        best = harness.best(metric="combined_score") or latest
        return InfoBundle(parent=best, notes=f"best round={best.round}")


class LLMRewriteOptimization(Optimization):
    SYSTEM_PROMPT = (
        "You are an expert Python programmer optimizing a circle-packing "
        "solution. Respond with ONLY a single ```python ...``` block."
    )

    async def optimize(self, info, harness, new_candidate):
        parent_code = _read(info.parent.artifacts_dir / SOLUTION_FILE)
        parent_metrics = harness.read_metrics(info.parent.round)
        prompt = (
            f"Current program (score={parent_metrics.get('combined_score', 'n/a')}):\n"
            f"```python\n{parent_code}\n```"
        )
        raw = await llm(prompt, max_tokens=128000)
        harness.write_artifact(
            new_candidate, SOLUTION_FILE, _extract_python_block(str(raw))
        )
```

This initialization exposes a limited search surface: it uses score-based selection, a single-parent rewrite step, and no richer failure analysis. This makes later meta-edits easier to interpret because they modify these exposed handles rather than the evaluator.

C.2.2 Initial Agent in Agentic and Reasoning Tasks

For Terminal-Bench and ARC-AGI-2, the initial agent is a minimal ReAct-style agent. The prompt surface and the `step()` loop are both editable, but the seed version contains only basic reasoning, short-horizon memory, and a strict JSON+`bash` action protocol.

```
REACT_PROMPT = """
==== Instruction ====
{instruction}

==== Action Space ====
{action_space}
```

```
==== Memory ====
Recent memory:
{memory}

==== Current Observation ====
{obs}

==== Thinking ====
You should think step by step before you output an action.
"""

async def step(self, observation: Observation, history: Any):
    act_prompt = REACT_PROMPT.format(
        instruction=self.current_env_instruction,
        action_space=self.current_action_space,
        obs=observation,
        memory=self._get_memory(),
    )
    resp = await self.llm(act_prompt)
    memory = parse_llm_output(resp, "memory")
    action = self.parse_action(resp)

    if isinstance(action, dict) and action.get("action") == "execute":
        params = action.setdefault("params", {})
        if not params.get("command"):
            bash_cmd = self._extract_bash_command(resp)
            if bash_cmd:
                params["command"] = bash_cmd

    await self.memory.add_memory(obs=agent_obs, action=action, thinking=memory)
    return action, resp, act_prompt
```

This weak initialization leaves room for improvement, but does not by itself isolate the contribution of meta-editing from additional search or context engineering. Strong-initialization results and their control limitations are reported in Appendix D.

C.2.3 Meta-Agent Skill Excerpt

The meta-agent is governed by a compact operational skill that constrains how it reads the workspace, attributes failure, and chooses one causal intervention at a time. A representative excerpt is shown below.

```
## 1. Core Loop

Read -> Attribute -> Choose Action -> Run Inner-Agent -> Record

## 4. Choose One Action

At each inter-session boundary, choose exactly one action.

### A. goal change
Change only ‘sessions/_next_goal.md‘.

### B. Harness change
Change durable support files such as:
- ‘skill/evolve_skill.md‘
- ‘shared/validators/‘
- ‘shared/tools/‘
- ‘shared/notes/‘

## 8. Hard Rules

Allowed:
- write ‘sessions/_next_goal.md‘
- edit ‘skill/evolve_skill.md‘
- launch inner-agent through ‘python -m evolver.cli run-inner-agent‘

Forbidden:
- edit ‘candidates/‘
- call ‘oer-eval eval‘ directly
- bypass the gateway or evaluator
```

The meta-agent therefore acts as a controller over future search rather than as an additional task-facing worker. The skill keeps that role boundary explicit.

C.3 Representative Evolved Artifacts and Outcomes

C.3.1 Representative Evolved Procedures

The excerpt below is taken from a high-performing ARC-AGI-2 procedure. In later rounds, the evolved procedure forwards not only the current best artifact but also per-task slices from alternative references, enabling the optimizer to formulate a single causal hypothesis from those diagnostics.

```
class CrossCandidateSelection(Selection):
    REF_K = 2

    async def select(self, harness: Harness) -> InfoBundle:
        ...
        scored.sort(key=lambda t: (t[1], t[0].round), reverse=True)

        best_cand, best_acc, _ = scored[0]
        refs = []
        ref_notes = []
        for c, acc, _metrics in scored[1:]:
            if len(refs) >= self.REF_K:
                break
            refs.append(c)
            tasks_block = _per_task_results_block(
                _read(c.folder / "eval_output.txt")
            )
            ref_notes.append(
                f"  Round {c.round}: accuracy={acc:.4f}\n"
                f"{_indent(tasks_block, '    ')}"
            )

        best_tasks = _per_task_results_block(
            _read(best_cand.folder / "eval_output.txt")
        )

        notes_lines = [
            f"Selection: best parent round={best_cand.round} accuracy={best_acc:.4f}",
            f"Total candidates so far: {len(all_rounds)}",
            "",
            "[Best per-task results]",
            _indent(best_tasks or "(no per-task block)", "  "),
        ]
        return InfoBundle(
            parent=best_cand,
            references=refs,
            notes="\n".join(notes_lines),
        )
```

```
class LLMRewriteAgentOptimization(Optimization):
    async def optimize(self, info, harness, new_candidate):
        parent_code = _read(info.parent.artifacts_dir / ARTIFACT_FILE)
        parent_metrics = harness.read_metrics(info.parent.round)
        parent_eval_log = _read(info.parent.folder / "eval_output.txt")

        parent_acc = float(parent_metrics.get("accuracy", 0.0) or 0.0)
        ref_blocks = []
        for ref in info.references or []:
            ref_code = _read(ref.artifacts_dir / ARTIFACT_FILE)
            ref_metrics = harness.read_metrics(ref.round) or {}
            ref_acc = float(ref_metrics.get("accuracy", 0.0) or 0.0)
            label = "ANTI-EXAMPLE" if ref_acc < parent_acc else "REFERENCE"
            ref_blocks.append(
                f"### {label} round {ref.round} "
                f"(accuracy={ref_acc:.4f})\n"
                f"```python\n{ref_code[:6000]}\n```"
            )
        ref_section = "\n\n".join(ref_blocks) if ref_blocks else "(none yet)"

        base_prompt = (
            f"## Selection notes\n{info.notes}\n\n"
            f"## Current agent.py "
            f"(round {info.parent.round}, accuracy={parent_metrics.get('accuracy', 'n/a')})\n"
            f"```python\n{parent_code}\n```\n\n"
            f"## Recent eval output (tail)\n"
            f"```\n{parent_eval_log[-_EVAL_LOG_TAIL:]}\n```\n\n"
            f"## Alternative references\n{ref_section}\n\n"
            f"## Task\n"
            f"Produce an improved agent.py. Form ONE specific theory "
            f"about the current failures and test the smallest edit "
```

```
            f"that directly probes that theory."
        )
```

This excerpt illustrates the main leverage of procedure-mode AEvo: evolution changes the evidence presented to the optimizer and the way that evidence is structured, not only the candidate artifact being scored.

C.3.2 Representative Evolved Agent Harness

In the agent-based setting, the evolved object is the harness seen by future inner-agent sessions rather than a single submitted artifact. In the performance-engineering run, this harness contained at least five durable layers: a task skill, a session-specific goal, a persistent family map, support utilities for evaluation accounting, and structured session notes written back into the workspace.

Task skill.

```
# Evolve Skill - Performance Engineering (VLIW Kernel)

Optimize a kernel on a simulated VLIW SIMD architecture to minimize
clock cycles. Combined Score = ‘147734 / cycles‘ (higher is better;
the goal at the top of your
prompt pins this session’s focus - read it before reading anything
else.

## Eval

$ oer-eval eval --program attempts/v1.py
Combined Score: 6.38
Validity:       1.0
Remaining Evals: 95
Session Evals Remaining: 9

## Don’t stop early

While ‘Session Evals Remaining > 0‘ you do not get to exit. "Current
cycle count looks good" is not a stop condition; an unspent eval is a
planned experiment you failed to run.

If you feel done, that is the signal to read
‘candidates/candidate_<best>/program.py‘ (or your own best vN.py),
name the specific structural reason it’s stuck, and submit a candidate
from a different family.

## CRITICAL - eval DB and the session-2 finding

1. Stage the local DB once at session start:
   ‘cp shared/notes/oer_eval_local_template.db ./.oer_eval.local.db‘
2. Run all evals with an explicit local DB:
   ‘oer-eval eval --program attempts/vN.py --db-path ./.oer_eval.local.db‘
3. The meta agent will replay the rows from
   ‘./.oer_eval.local.db‘ into the workspace ‘.oer_eval.db‘ after
   your session ends.
4. Do NOT spend evals re-confirming the readonly issue.
5. NEVER copy or mutate ‘../../../.oer_eval.db‘ directly from
   inside the sandbox.

## SESSION_NOTES.md (required on exit)

Write ‘SESSION_NOTES.md‘ at your cwd root before finishing.
```

Session goal.

```
# goal for next inner-agent session (session 7)

## Status
- Current best: 1774 cycles, score 83.28.
- 61 evals remaining in the global quota; this session has MAX_EVALS = 15.
- load is now the dominant bottleneck again.
- Session 6 only used 6 of 15 evals before exiting.

## Setup commands
cp shared/notes/oer_eval_local_template.db ./.oer_eval.local.db
mkdir -p attempts
cp shared/notes/best_v59_session6.py attempts/v59_parent.py
oer-eval eval --program attempts/v59_parent.py --db-path ./.oer_eval.local.db

## Hypotheses (in test order - DO NOT skip later ones if you finish early)
```

```
Test 1: family D'' - depth-3 cache.
Test 2: family C - software-pipelined inner loop.
Test 3: hash-chain cross-stage algebra.
Test 4: scheduler tie-breaks aware of engine occupancy.
Test 5: aggressive scratch reclaim + bundle merging.

## Required behavior
- MAX_EVALS = 15. Submit 15 attempts.
- Use `--db-path ./.oer_eval.local.db` on every eval.
- Save attempts as `attempts/vN.py`.
- Use offline `Engine` simulator for sanity before eval.

## Required output (`SESSION_NOTES.md`)
- Best Combined Score and corresponding `attempts/vN.py`.
- v59 starting profile and the v_best ending profile.
- For each attempt: score, validity, structural change, which engine shifted.
- Final local-DB `Remaining Evals`.
- One specific, falsifiable hypothesis for session 8.
```

Persistent family map.

```
# Family Map - VLIW Kernel

## Architecture cheatsheet
- VLEN=8, batch_size=256 -> 32 SIMD groups
- rounds=16
- SLOT_LIMITS: alu 12 / valu 6 / load 2 / store 2 / flow 1
- 6 hash stages, each pure valu (`+`, `^`, `<<`, `>>`)
- Per element: parity -> next_idx = 2*idx + (1 if even else 2)

## Sessions
### session 4 - family B+C: fused hash + ping-pong + cached depth-1 reuse
- best 77.88 / 1897 cycles (`v31`)
- v27: collapse hash stages 0/2/4 into `multiply_add`
- v31: cached depth-1 node `vselect` across rounds

### session 5 - family D: idx-update structural reductions
- best 79.60 / 1856 cycles (`v47`)
- stage-5 short-circuit hypothesis was FALSIFIED
- v43: depth-1 base preselection
- v47: generic non-root precompute of `2*idx + 1`

### session 9 - breakthrough: 1140 cycles via explicit-family port
- best 129.59 / 1140 cycles (`v91`)
- v88 explicit-family port (-597)
- v89 depth-1 madd (-10)
- v90 depth-3 2-madd (-19)
- v91 scalar alu parity + per-lane xor (-8)

## Do not repeat
- Scheduler tie-break / priority tweaks alone.
- Stage-5 short-circuit `(a^const)^(a>>16)`.
- Stage-4 -> stage-5 fusion via `multiply_add`.

## Workflow gotchas
- Use `--db-path ./.oer_eval.local.db` after copying
  `shared/notes/oer_eval_local_template.db`.
- Meta agent replays after session ends.
```

Replay utility.

```
#!/usr/bin/env python3
"""Replay rows from a session's local oer-eval DB into the workspace DB.

Used by the meta agent ONLY (after a session ends) to credit evals that the
codex sandbox forced into a session-local DB. Inner agents must not run this.
"""
session_id = int(sys.argv[1])
ws = Path(__file__).resolve().parent.parent.parent
session_dir = ws / "sessions" / f"session_{session_id}"
local_db = session_dir / "agent_workspace" / ".oer_eval.local.db"
ws_db = ws / ".oer_eval.db"

src_rows = src.execute(
    "SELECT agent_name, problem_name, program_path, program_content, "
    "validity, eval_time, combined_score, error, raw_result "
    "FROM evaluations ORDER BY id"
).fetchall()
```

```
for r in src_rows:
    dst.execute(
        "INSERT INTO evaluations "
        "(agent_name, problem_name, program_path, program_content, "
        " validity, eval_time, combined_score, error, raw_result) "
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)",
        (agent_name, r[1], r[2], r[3], r[4], r[5], r[6], r[7], r[8]),
    )
```

Session memory written back by the inner agent.

```
# Session Notes - session 9

## Best result
- Best Combined Score: 129.59
- Best cycles: 1140
- Best file: ‘attempts/v91.py‘

## Attempt log
- ‘attempts/v88.py‘ - explicit-family port
- ‘attempts/v89.py‘ - ‘depth1‘ node select ‘vselect -> madd‘
- ‘attempts/v90.py‘ - reduced-flow ‘depth3‘ selector
- ‘attempts/v91.py‘ - scalar root parity + per-lane alu xor
- ‘attempts/v97.py‘ - full depth-2 two-‘madd‘ rewrite
- ‘attempts/v98.py‘ - scalarize generic-round post-hash parity extraction

## What worked
- ‘v88‘ cut 597 cycles from ‘v59_parent‘.
- ‘depth1‘ binary-select to ‘madd‘ paid.
- ‘depth3‘ reduced-flow selector paid more.

## What did not work
- naive generic-round selector rewrite was wrong for this schedule shape.
- fully flow-free depth-2 rewrite also hurt.
- scalarizing generic-round parity was catastrophic.

## Hypothesis for session 10
- a partial depth-2 rewrite that replaces only one of the two
  ‘flow.vselect‘s with ‘madd‘ should beat 1140.
```

Together, these excerpts show that the evolved agent harness is not a single prompt edit. It is a layered control structure consisting of persistent instructions, hypothesis-carrying goals, accumulated family-level memory, support code for evaluator interaction, and structured records that are promoted into future sessions.

C.3.3 Representative Optimization Outcomes

ARC-AGI-2 best artifact. The best ARC-AGI-2 artifact in our run reaches accuracy 0.35 $(7/20)$. Structurally, the final agent has three persistent components: a prompt surface that distinguishes normal feedback refinement from "fresh exploration," a local Pass@K scorer over cached training pairs, and a validate-versus-submit controller that uses the best local score and plateau state.

```
FEEDBACK_SUFFIX = """
==== Previous Attempt Feedback ====
A previous attempt produced this code:
‘‘‘python
{prev_code}
‘‘‘
It passed {passed}/{total} training examples.

FAILING examples (must fix these):
{fail_details}
"""

FRESH_EXPLORE_SUFFIX = """
==== Fresh Exploration Required ====
Previous attempts have been stuck at a partial solution.
IGNORE any previous approach and propose a completely
different interpretation of the transformation rule.
"""

DIVERSITY_HINTS = [
    "",
    "Consider symmetry, rotation, reflection, or tiling patterns.",
    "Consider connected components, object counting, or shape detection.",
    "Consider color mapping, replacement, or color-dependent rules.",
```

```
]

class ArcAGIAgent(BaseAgent):
    """Pass@K with fresh-exploration escape when stuck at local optimum."""

    num_samples: int = Field(default=3)
    cached_train_pairs: List[Dict[str, Any]] = Field(default_factory=list)
    best_score_so_far: int = Field(default=-1)
    best_response_so_far: str = Field(default="")
    best_code_so_far: str = Field(default="")
    best_fails: List[Tuple[int, Any]] = Field(default_factory=list)
    hint_rotation_offset: int = Field(default=0)
    stuck_counter: int = Field(default=0)
```

```
class ArcAGIAgent(BaseAgent):
    async def step(self, observation: Observation, history: Any):
        ...
        base_prompt = ARCAGI_PROMPT.format(
            instruction=instruction_text,
            action_space=self.current_action_space,
            memory=self.memory.as_text() if self.memory else "None",
            obs=observation,
        )

        train_pairs = _extract_train_pairs(observation)
        if train_pairs:
            self.cached_train_pairs = train_pairs
        elif self.cached_train_pairs:
            train_pairs = self.cached_train_pairs
        n_train = len(train_pairs)

        use_fresh_explore = (
            self.stuck_counter >= 2
            and n_train > 0
            and 0 <= self.best_score_so_far < n_train
        )

        if not use_fresh_explore and (
            n_train > 0
            and self.best_score_so_far >= 0
            and self.best_score_so_far < n_train
            and self.best_code_so_far
            and self.best_fails
        ):
            fail_details = _format_fail_details(self.best_fails, train_pairs)
            feedback_suffix = FEEDBACK_SUFFIX.format(
                prev_code=self.best_code_so_far[:3000],
                passed=self.best_score_so_far,
                total=n_train,
                fail_details=fail_details,
            )
        elif use_fresh_explore:
            feedback_suffix = FRESH_EXPLORE_SUFFIX

        prompts = []
        for i in range(self.num_samples):
            hint_idx = (i + self.hint_rotation_offset) % len(DIVERSITY_HINTS)
            prompts.append(base_prompt + feedback_suffix + DIVERSITY_HINTS[hint_idx])
        responses = await asyncio.gather(*[self._sample_once(p) for p in prompts])

        for r in valid_responses:
            code = _extract_python_block(r)
            score, fails = _score_candidate(code, train_pairs)
            if score > round_best_score:
                round_best_score = score
                round_best_response = r
                round_best_code = code
                round_best_fails = fails

        if round_best_score > self.best_score_so_far:
            self.best_score_so_far = round_best_score
            self.best_response_so_far = round_best_response
            self.best_code_so_far = round_best_code
            self.best_fails = round_best_fails
            self.stuck_counter = 0
        elif 0 <= self.best_score_so_far < n_train:
            self.stuck_counter += 1

        action = self.parse_action(self.best_response_so_far or round_best_response)
        if self.best_score_so_far >= n_train:
```

```
            action["action"] = "submit"
        elif step_num >= self.max_refine_steps:
            action["action"] = "submit"
        else:
            action["action"] = "validate"
```

Analysis. The improvement over the seed agent comes from a tighter coupling between search control and task-local verification. First, caching the training pairs removes a brittle dependency on the observation format at later steps, so local verification remains available throughout the interaction. Second, Pass@K sampling with prompt diversification converts a single-sample ReAct loop into a small search procedure over candidate solvers, with selection driven by observed agreement on the training pairs rather than by the raw LLM output alone. Third, the separation between `FEEDBACK_SUFFIX` and `FRESH_EXPLORE_SUFFIX` makes the agent alternate explicitly between exploitation and hypothesis reset: partial but improving candidates are refined through concrete failure-conditioned feedback, whereas persistent plateaus trigger a prompt regime that suppresses anchoring to the current local optimum. Finally, the validate-versus-submit controller ties action choice to the best verified score rather than to the latest response, which reduces premature submission of partially correct programs.

Performance-engineering best artifact. The best performance-engineering artifact is a two-file submission. The top-level program is only a wrapper; the schedule-level optimization resides in a benchmark-specialized base that exposes a small set of round-family control points. The final validated artifact reaches 1138 cycles by combining evaluator-compatible packaging, explicit specialization of the benchmark rounds, and a non-uniform assignment of selector logic across engines.

Submitted wrapper.

```
from importlib.util import module_from_spec, spec_from_file_location
from pathlib import Path

_SPEC = spec_from_file_location("variant_base", Path(__file__).with_name("variant_base.py"))
_MOD = module_from_spec(_SPEC)
assert _SPEC.loader is not None
_SPEC.loader.exec_module(_MOD)
VariantKernelBuilderBase = _MOD.VariantKernelBuilderBase

class KernelBuilder(VariantKernelBuilderBase):
    DEPTH1_NODE_STYLE_CALLS = ("madd", "vselect")
```

Parameterized benchmark family.

```
BENCH_FOREST_HEIGHT = 10
BENCH_N_NODES = 2 ** (BENCH_FOREST_HEIGHT + 1) - 1
BENCH_BATCH_SIZE = 256
BENCH_ROUNDS = 16
BENCH_FOREST_BASE = 7
BENCH_INP_INDICES_P = BENCH_FOREST_BASE + BENCH_N_NODES
BENCH_INP_VALUES_P = BENCH_INP_INDICES_P + BENCH_BATCH_SIZE
BENCH_TILES = BENCH_BATCH_SIZE // VLEN

class VariantKernelBuilderBase:
    ROOT_SCALAR_AND_CALLS = (True, False)
    DEPTH1_NODE_STYLE_CALLS = ("madd", "madd")
    DEPTH1_POSTHASH_SCALAR_AND_CALLS = (False, False)
    DEPTH2_STYLE_CALLS = ("base", "base")
    DEPTH2_POSTHASH_SCALAR_AND_CALLS = (False, False)
    DEPTH3_STYLE_CALLS = ("madd", "madd")
    DEPTH3_POSTHASH_SCALAR_AND_CALLS = (False, False)
```

Core benchmark schedule.

```
root_scalar_and_calls = self.ROOT_SCALAR_AND_CALLS
depth1_node_style_calls = self.DEPTH1_NODE_STYLE_CALLS
depth1_posthash_scalar_and_calls = self.DEPTH1_POSTHASH_SCALAR_AND_CALLS
depth2_style_calls = self.DEPTH2_STYLE_CALLS
depth2_posthash_scalar_and_calls = self.DEPTH2_POSTHASH_SCALAR_AND_CALLS
depth3_style_calls = self.DEPTH3_STYLE_CALLS
depth3_posthash_scalar_and_calls = self.DEPTH3_POSTHASH_SCALAR_AND_CALLS

def emit_hash(tile_ids):
    for tile in tile_ids:
        self.emit_madd(vals[tile], vals[tile], mul4097_v, addc1_v)
```

```
    for tile in tile_ids:
        self.emit_valu(">>", tmp1s[tile], vals[tile], shift19_v)
    for tile in tile_ids:
        for lane in range(VLEN):
            self.emit_alu("^", vals[tile] + lane, vals[tile] + lane, c2_s)
    for tile in tile_ids:
        self.emit_valu("^", vals[tile], vals[tile], tmp1s[tile])
    for tile in tile_ids:
        self.emit_madd(vals[tile], vals[tile], mul33_v, addc3_v)
    for tile in tile_ids:
        self.emit_valu("<<", tmp1s[tile], vals[tile], shift9_v)
    for tile in tile_ids:
        for lane in range(VLEN):
            self.emit_alu("+", vals[tile] + lane, vals[tile] + lane, c4_s)
    for tile in tile_ids:
        self.emit_valu("^", vals[tile], vals[tile], tmp1s[tile])
    for tile in tile_ids:
        self.emit_madd(vals[tile], vals[tile], mul9_v, addc5_v)
    for tile in tile_ids:
        self.emit_valu(">>", tmp1s[tile], vals[tile], shift16_v)
    for tile in tile_ids:
        for lane in range(VLEN):
            self.emit_alu("^", vals[tile] + lane, vals[tile] + lane, c6_s)
    for tile in tile_ids:
        self.emit_valu("^", vals[tile], vals[tile], tmp1s[tile])

def emit_parity_and(dest_vec: int, src_vec: int, use_scalar_and: bool):
    if use_scalar_and:
        for lane in range(VLEN):
            self.emit_alu("&", dest_vec + lane, src_vec + lane, one_v + lane)
    else:
        self.emit_valu("&", dest_vec, src_vec, one_v)

def round_root(use_scalar_and: bool):
    for tile in tiles:
        self.emit_valu("^", vals[tile], vals[tile], root_v)
    emit_hash(tiles)
    for tile in tiles:
        emit_parity_and(idxs[tile], vals[tile], use_scalar_and)

def round_depth1(node_style: str, use_scalar_and: bool):
    for tile in tiles:
        if node_style == "madd":
            self.emit_madd(tmp0s[tile], idxs[tile], node12_diff_v, node1_v)
        elif node_style == "vselect":
            self.emit_vselect(tmp0s[tile], idxs[tile], node2_v, node1_v)
        else:
            raise ValueError(f"unknown depth1 node style: {node_style}")
    for tile in tiles:
        self.emit_valu("^", vals[tile], vals[tile], tmp0s[tile])
    emit_hash(tiles)
    for tile in tiles:
        emit_parity_and(tmp0s[tile], vals[tile], use_scalar_and)
        self.emit_madd(idxs[tile], idxs[tile], two_v, tmp0s[tile])

def round_depth2(style: str, use_scalar_and: bool):
    for tile in tiles:
        self.emit_valu("&", d2_bit0, idxs[tile], one_v)
        self.emit_valu(">>", d2_bit1, idxs[tile], one_v)
        if style == "base":
            self.emit_vselect(tmp0s[tile], d2_bit0, node4_v, node3_v)
            self.emit_vselect(d2_mix, d2_bit0, node41_diff_v, node30_diff_v)
        elif style == "madd_first":
            self.emit_madd(tmp0s[tile], d2_bit0, node43_diff_v, node3_v)
            self.emit_vselect(d2_mix, d2_bit0, node41_diff_v, node30_diff_v)
        elif style == "madd_second":
            self.emit_vselect(tmp0s[tile], d2_bit0, node4_v, node3_v)
            self.emit_madd(d2_mix, d2_bit0, d2_mix_delta_v, node30_diff_v)
        else:
            raise ValueError(f"unknown depth2 style: {style}")
        self.emit_madd(tmp0s[tile], d2_bit1, d2_mix, tmp0s[tile])
        self.emit_valu("^", vals[tile], vals[tile], tmp0s[tile])
    emit_hash(tiles)
    for tile in tiles:
        emit_parity_and(tmp0s[tile], vals[tile], use_scalar_and)
        self.emit_madd(idxs[tile], idxs[tile], two_v, tmp0s[tile])
```

```
def round_depth3(style: str, use_scalar_and: bool):
    for tile in tiles:
        self.emit_valu("&", d3_bit0, idxs[tile], one_v)
```

```
        self.emit_valu(">>", d3_bit1, idxs[tile], one_v)
        self.emit_valu("&", d3_bit2, d3_bit1, one_v)
        self.emit_valu(">>", d3_bit1, idxs[tile], two_v)
        if style == "madd":
            self.emit_vselect(d3_pair0, d3_bit0, depth3_nodes[1], depth3_nodes[0])
            self.emit_vselect(d3_pair1, d3_bit0, depth3_diff_lo1_v, depth3_diff_lo0_v)
            self.emit_madd(d3_pair0, d3_bit2, d3_pair1, d3_pair0)
            self.emit_vselect(d3_pair1, d3_bit0, depth3_nodes[5], depth3_nodes[4])
            self.emit_vselect(tmp0s[tile], d3_bit0, depth3_diff_hi1_v, depth3_diff_hi0_v)
            self.emit_madd(d3_pair1, d3_bit2, tmp0s[tile], d3_pair1)
            self.emit_vselect(tmp0s[tile], d3_bit1, d3_pair1, d3_pair0)
        elif style == "vselect":
            self.emit_vselect(d3_pair0, d3_bit0, depth3_nodes[1], depth3_nodes[0])
            self.emit_vselect(d3_pair1, d3_bit0, depth3_nodes[3], depth3_nodes[2])
            self.emit_vselect(d3_pair2, d3_bit0, depth3_nodes[5], depth3_nodes[4])
            self.emit_vselect(d3_pair3, d3_bit0, depth3_nodes[7], depth3_nodes[6])
            self.emit_vselect(d3_pair0, d3_bit2, d3_pair1, d3_pair0)
            self.emit_vselect(d3_pair2, d3_bit2, d3_pair3, d3_pair2)
            self.emit_vselect(tmp0s[tile], d3_bit1, d3_pair2, d3_pair0)
        else:
            raise ValueError(f"unknown depth3 style: {style}")
        for lane in range(VLEN):
            self.emit_alu("^", vals[tile] + lane, vals[tile] + lane, tmp0s[tile] + lane)
    emit_hash(tiles)
    for tile in tiles:
        emit_parity_and(tmp0s[tile], vals[tile], use_scalar_and)
        self.emit_madd(idxs[tile], idxs[tile], two_v, tmp0s[tile])
        self.emit_valu("+", idxs[tile], idxs[tile], depth4_base_v)

def round_gather(update_idx: bool):
    for tile in tiles:
        for lane in range(VLEN):
            self.emit_load_offset(tmp0s[tile], idxs[tile], lane)
    for tile in tiles:
        self.emit_valu("^", vals[tile], vals[tile], tmp0s[tile])
    emit_hash(tiles)
    if update_idx:
        for tile in tiles:
            self.emit_valu("&", tmp0s[tile], vals[tile], one_v)
            self.emit_vselect(tmp1s[tile], tmp0s[tile], add_odd_v, add_even_v)
            self.emit_madd(idxs[tile], idxs[tile], two_v, tmp1s[tile])

round_root(use_scalar_and=root_scalar_and_calls[0])
round_depth1(depth1_node_style_calls[0], depth1_posthash_scalar_and_calls[0])
round_depth2(depth2_style_calls[0], depth2_posthash_scalar_and_calls[0])
round_depth3(depth3_style_calls[0], depth3_posthash_scalar_and_calls[0])
round_gather(update_idx=True)
round_gather(update_idx=True)
round_gather(update_idx=True)
round_gather(update_idx=True)
round_gather(update_idx=True)
round_gather(update_idx=True)
round_gather(update_idx=False)
round_root(use_scalar_and=root_scalar_and_calls[1])
round_depth1(depth1_node_style_calls[1], depth1_posthash_scalar_and_calls[1])
round_depth2(depth2_style_calls[1], depth2_posthash_scalar_and_calls[1])
round_depth3(depth3_style_calls[1], depth3_posthash_scalar_and_calls[1])
round_gather(update_idx=True)
```

```
Evaluation:      valid=True, cycles=1138, speedup=129.82x
Status:          success
Problem:         perf_engineering
Combined Score:  129.8189806678383
Validity:        1.0
Eval Time:       31.66170883178711s
```

Analysis. The low cycle count comes from three coupled changes. First, the file-local `importlib` wrapper is not cosmetic: without it, the evaluator rejects the whole family because sibling modules are not imported through the workspace package path. The wrapper therefore preserves a modular implementation while keeping the submission evaluator-compatible. Second, the benchmark-specialized base restructures the kernel around the exact round pattern of the benchmark rather than a generic loop. Its control flow is emitted as four specialized top-of-tree rounds, followed by seven gather rounds, followed by the same four specialized rounds and a final gather; this removes generic control overhead where the traversal repeatedly revisits the upper levels of the tree and makes the period-11 reuse pattern directly schedulable. Within each specialized round, `emit_hash` is emitted phase-major

across tiles, which exposes many independent chains to the scheduler and improves overlap between `alu`, `valu`, and memory operations. Third, the remaining gains come from engine balancing rather than from further structural refactoring. On this VLIW benchmark, `flow` has only one slot per cycle, so replacing a binary selector by `multiply_add` is profitable only when the induced `valu` pressure stays below the new bottleneck. The progression recorded in the run data is consistent with this view: the explicit-family port yields the dominant improvement (-597 cycles), additional `madd`-based selectors at depth 1 and depth 3 save another 29 cycles, an `alu`-based parity/XOR rebalance saves 8, and the final artifact gains the last 2 cycles by reverting only the second specialized depth-1 selector back to `vselect`. The best artifact is therefore not simply a shorter program; it is a schedule in which specialization, phase ordering, and per-round engine placement are co-tuned to the simulator's slot limits.

Taken together, these artifacts illustrate two modes of durable improvement in AEvo: benchmark tasks improve through changes in how the optimizer reasons over failures, whereas open-ended optimization improves through the preservation and recombination of low-level implementation knowledge across many sessions.

D Additional Results and Reporting Scope

D.1 Per-run results

Table 5 reports the three recorded AEvo runs. For scores x_1, x_2, x_3, we compute $\bar{x} = \sum_i x_i/3$ and sample SD $s = \sqrt{\sum_i (x_i - \bar{x})^2/(3-1)}$. These describe run-to-run variability, not standard errors or confidence intervals. We do not assume normality or claim statistical significance. Baseline per-run outcomes are not available here. CP26 configurations reach the same reported ceiling of 2.6359; time-to-target is reported through Best R. in the main table, without a per-run time-to-target variance estimate.

Table 5: AEvo run outcomes; parentheses identify the optimization stack for open-ended tasks. Standard-benchmark execution uses Gemini-3-Flash.

Task	Variant / stack	Run 1	Run 2	Run 3	Mean $\pm$ SD
Terminal-Bench 2	Procedure	56.25	53.75	51.25	53.75 ± 2.50
ARC-AGI-2	Procedure	46.0	46.7	48.3	47.00 ± 1.18
AC2	Procedure / Opus	.9278	.9273	.9261	$.9271 \pm .0009$
AC2	Agent / Opus	.9459	.9430	.9452	$.9447 \pm .0015$
AC2	Agent / Codex	.9398	.9376	.9384	$.9386 \pm .0011$
Kernel	Procedure / Opus	1803	1842	1911	1852.0 ± 54.69
Kernel	Agent / Opus	1519	1567	1549	1545.0 ± 24.25
Kernel	Agent / Codex	1138	1157	1230	1175.0 ± 48.57

D.2 Inner optimizer and strong initialization

For ARC-AGI-2, the outer meta-agent remains Opus-4.7 and the execution model remains Gemini-3-Flash. Keeping the evaluator, prompt, and 20-round budget fixed, the Opus inner optimizer obtains 46.0/46.7/48.3 (47.0 ± 1.18); Sonnet-4.6 obtains 44.7/43.3/42.0 (43.3 ± 1.35). Starting from ADAS's agent structure and learning procedure (score 36.0), three additional 10-round AEvo runs obtain 42.3/44.0/42.3 (42.9 ± 0.98). The evolved procedure selects informative successes and failures from history when constructing optimization context, overlapping with reflective context optimization. There is no same-seed, matched-budget ADAS continuation or fixed history-selection control, so neither extra compute nor context construction is independently ruled out. A separate Kernel run initialized from the prior 1121-cycle AEvo artifact reaches 1108 in a further 100 rounds. This differs from extending the original 1138-cycle run to 200 rounds (1121 cycles); neither result is a from-scratch 100-round comparison.

D.3 Intervention audit and motivation

The audit covers two representative trajectories with evaluation ledgers, mechanism versions, transcripts, and directives: a 100-evaluation Kernel run and a 16-round ARC procedure run. We classify

an intervention by whether subsequent candidates improve the global best, fail to improve it, or are regressive/invalid. Kernel has 5/8, 2/8, and 1/8 interventions in these categories; ARC has 3/7, 2/7, and 2/7. These fractions are descriptive and selected-run-specific. In Kernel, two sessions spend 30 evaluations without improvement; a later import refactor produces 14 invalid candidates. In ARC, an initial solver-prompt rewrite reduces the audit score from .15 to .00; freezing that prompt and editing orchestration is followed by .25. Repairing a JSON/fenced-grid parsing mismatch activates local verification. Task-profile prompting reduces a .35 score to .20/.25 and is reverted. Increasing refinement to 18 inner calls per task does not break a .30 plateau. These within-trajectory observations are not controlled causal tests, and the audit scores are not the aggregate test scores in Table 2. A separate analyzed ADAS trajectory peaks at round 3 and does not improve over the remaining 17 rounds; reported summaries are 55% regressive transitions, 33% zero-score rounds, and 84% of budget spent after the peak. These motivate plateau-aware control but do not estimate a population-level plateau frequency.

D.4 Supplementary PaperReview result

A supplementary task derived from APRES [45] predicts real accept/reject outcomes for papers from ICLR 2024/2025 and NeurIPS 2023/2024. This experiment uses Opus-4.7 as the outer meta-agent, Claude-4.5-Sonnet for optimization, and GPT-4o for execution. Accuracy is 52% for the initial procedure and 67% after 10 evolution rounds, compared with 63% for an AI-Scientist-v2 static reviewer [35]. These are preliminary aggregate results: sample counts, split construction, class balance, repeated runs, uncertainty, and contamination controls are not specified here. Consequently, this result is not used to support a quantitative generalization or review-quality claim; conference outcomes are imperfect labels for scientific quality.

D.5 Baseline scope and literature comparison

OpenEvolve is an open-source implementation inspired by AlphaEvolve rather than its official implementation. CORAL is cited as related work, not a reproduced baseline: its paper reports 1350 Kernel cycles for one agent and 1103 for four agents with its own models and budgets. Its shared memory, heartbeat interventions, and evaluator separation overlap with parts of our harness. We therefore limit numerical ranking claims to Table 1; comparing counts of agents or rounds alone does not establish equal resource use.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state three contributions with quantitative results that match the experimental findings in Section 5.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 7 discusses model dependence, cost, task scope, small sample sizes, bundled ablations, strong-seed controls, and evaluator limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not include formal theoretical results; it is a systems and empirical contribution.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 5.1 specifies the evaluated tasks, baselines, models, metrics, and reporting protocol. Appendix B provides implementation details, including budgets, stopping criteria, and initialization settings. Appendix C.1 provides task definitions, metric definitions, and cost computation details, while Appendix C.2.1 gives initialization details.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: An anonymized code supplement was submitted for review. A public de-anonymized repository with complete reproduction instructions is not linked in this manuscript; release is planned upon publication.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5.1 details the main experimental settings, including tasks, baselines, models, metrics, and reporting protocol. Appendix B provides implementation details, and Appendix C.1 provides task-specific definitions and metrics.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Appendix D reports AEvo per-run results and sample SD over three runs. Baseline variances and broad significance tests are unavailable, so this checklist item remains No for the complete comparison.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: We report API models, per-round costs, and the ARC optimization/evaluation cost breakdown. Complete worker, memory, runtime, and total-project resource accounting is not provided.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work uses benchmarks and commercial LLM APIs without recruiting human participants. Section 7 discusses execution risks and responsible use; this is not a claim that all dual-use risks are absent.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section 7 discusses potential optimization benefits, unsafe code execution, resource consumption, and limits on using paper-outcome prediction.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release pre-trained models or scraped datasets. The released code is an optimization framework that relies on existing commercial APIs.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: The benchmarks and baseline methods are cited, but this manuscript does not enumerate the licenses and terms for every asset.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [No]

    Justification: The manuscript includes mechanism and skill excerpts, but complete public code documentation and reproduction scripts are not linked here.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This work does not involve crowdsourcing or human subjects.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: This work does not involve human subjects research.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are the core component of both the meta-agent and the inner optimization loop. Section 4 and Section 5.1 fully describe all LLM models used and their roles.