MILO: AUTOMATED HARNESS DISCOVERY VIA ORCHESTRATED MULTI-AGENT EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern agentic systems typically consist of an artificial intelligence (AI) model and a harness, a software layer that manages its control flow and interactions with the environment. Agent performance on long-horizon tasks is often strongly influenced by its harness design. Yet building effective harnesses requires significant human effort due to the combinatorial design space, and this effort must be repeated as models are updated. To automate harness design, existing methods provide limited exploration of this search space. Most optimize only parts of the harness, such as prompts or skills, while others struggle to escape local optima due to their fixed search strategies and exploitative bias in LLM-driven search.

In this paper, we present MILO *(Meta-evolutionary Island Orchestration)*, a framework for automated harness discovery that co-evolves the harness and its own search strategy. Three components drive the search: (i) a *hierarchical lineage memory* of island-based trees that uses rejected mutations as negative evidence to prune unpromising paths and steer toward promising lineages; (ii) per-island *mutator agents* that combine global search history with feedback on parent weaknesses to rewrite entire harnesses; and (iii) an *orchestrator agent* that adapts the search by grafting and speciating lineages, reassigning mutators, and revising the curriculum. Together, they make MILO a meta-evolutionary harness-discovery framework that self-adapts its memory, mutators, and curriculum to balance exploration and exploitation. Across three long-horizon benchmarks, Terminal-Bench 2.1, PaperBench, and DeepSWE, MILO-discovered harnesses outperform eight state-of-the-art harnesses and six search methods with frontier (Opus 4.8) and open-weight (gpt-oss-120b) backbones. With Opus 4.8, it improves resolution rate over its initial harness by $+12.0\%$, $+28.3\%$ and $+10.3\%$, respectively, versus the best prior search gains $+4.5\%$ (GEPA), $+18.3\%$ (Meta-Harness) and 0% (no improvement observed). On Terminal-Bench 2.1, it reaches $86.1\pm2.0\%$, above the official leaderboard's top entry ($83.8\pm2.3\%$), while consuming 26% fewer tokens than its initial harness. On EinsteinArena's open problems, MILO-evolved harnesses tighten the best-known upper bounds for Erdős minimum-overlap ($0.3808586\rightarrow0.3808568$) and the first and third autocorrelation inequalities ($1.50274365\rightarrow1.50274360$, $1.45081\rightarrow1.44889$).

1 INTRODUCTION

Over the last few years, generative AI has transformed software engineering. What began as code-completion tools like Copilot (GitHub, 2021) has grown into *AI agents* such as Claude Code (Anthropic, 2025a) and Codex (OpenAI, 2025), which act autonomously on *long-horizon tasks*, from resolving GitHub issues (Yang et al., 2024a) to running end-to-end machine learning engineering (Chan et al., 2025). These agents typically pair a large language *model* (LLM), with a *harness*. The harness is the software layer that executes the model's proposed actions in a stateful environment and governs its *scaffolding* (prompts, tools, context, and memory) and *control flow*, such as spawning sub-agents, intercepting tool calls, verifying outputs, and deciding to stop (Ning et al., 2026). A growing body of evidence (Anthropic, 2025b; Tian et al., 2026) shows that the harness shapes agent performance as much as the model. On Terminal-Bench, for example, GPT-5 solves 35.2% with Terminus 2 but 49.6% with Codex, consuming 35% fewer tokens (Merrill et al., 2026).

Designing a strong harness is therefore a *low-cost, high-impact* way to improve an agent. It requires far less compute than model training and can compensate for weaknesses of the underlying

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

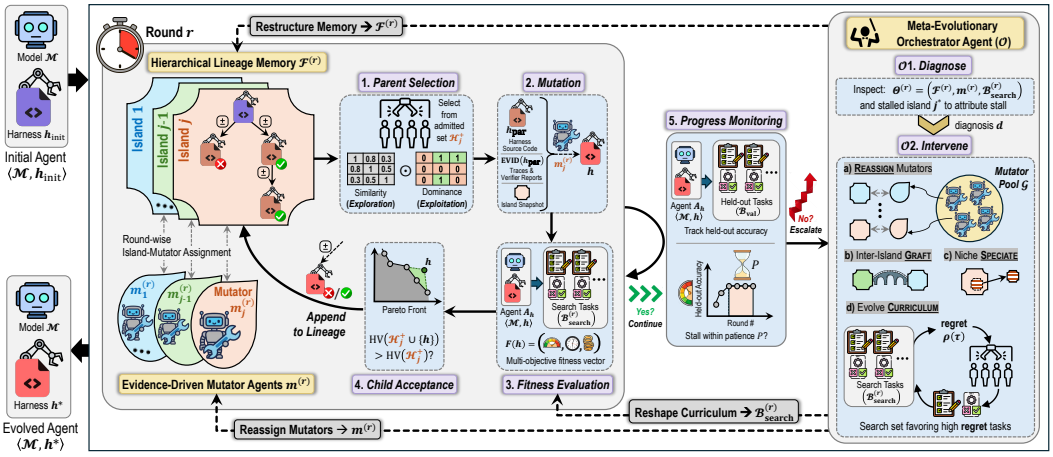
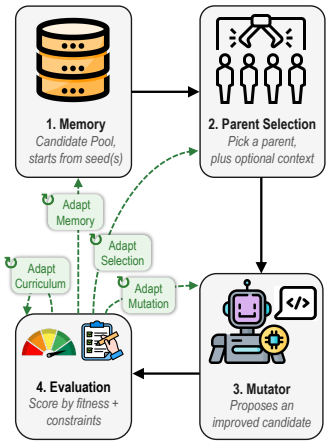


Figure 1: **Overview of MILO, a meta-evolutionary harness discovery framework.** Given an agent, MILO discovers a stronger harness for its model via two coupled loops: an inner loop that evolves harnesses and an outer loop that evolves the search strategy. In the inner loop, (1) a parent is drawn from the island's admitted candidates, (2) its *mutator agent* rewrites the whole harness from the parent's failure traces and the island's *lineage memory*, including rejected candidates, and (3–4) the child is scored and admitted on multi-objective Pareto gain. (5) When progress stalls, the outer loop's *orchestrator agent* diagnoses the island and revises its mutator, memory, and curriculum.

model (Niklaus, 2026; Ben Sghaier et al., 2026). Yet, while model weights are learned against explicit objectives, *harness engineering* remains artisanal: manual, ad hoc, and expert-driven. This also applies to production harnesses like Claude Code and Codex, which take teams months to build (Cognition, 2026), and open-source ones like mini-SWE-agent (Yang et al., 2024b) and DeepAgents (LangChain, 2025). Engineers inspect failures, adjust heuristics, and iterate over a handful of designs (Lee et al., 2026) by manual exploration of a combinatorial space. The effort is perpetual: harnesses are model-specific, as models differ in tool use, error modes, and prompt sensitivity (Sclar et al., 2024), so a harness tuned for one can be suboptimal for another and must be re-tuned.

This motivates **automated harness discovery** (AHD), which, given an agent, discovers a stronger harness for its model. AHD is typically cast as LLM-driven *evolutionary search* (ES): a loop in which an LLM-based mutator proposes harnesses, an oracle scores them, and the results guide the next round. Because the loop is model-agnostic, rerunning it per model automates the perpetual re-tuning. More broadly, AHD is a step toward agents that improve their own harness, or *recursive self-improvement* (Chen et al., 2026). Depending on what they optimize, ES methods fall into two categories. **Instance-level discovery** super-optimizes a *solution* for a single task. FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025), OpenEvolve (Sharma, 2025), and EvoX (Liu et al., 2026) work at this level. **Strategy-level discovery** optimizes the *solver* itself (for AHD, the harness), which must generalize across a task distribution. AHD is therefore strategy-level, as are GEPA (Agrawal et al., 2025), A-Evolve (Lin et al., 2026b), Meta-Harness (Lee et al., 2026), and Self-Harness (Zhang et al., 2026a). Yet existing ES methods suffer from four shortcomings when applied to harness discovery. **First**, *search space exploration*: by nature, an LLM-based mutator *exploits* far more than it *explores*. It repeats the same family of edits (Si et al., 2026), so the search suffers *diversity decay*, collapsing onto variants of a few designs. For strategy-level discovery like AHD, exploration is crucial, as gains on long-horizon tasks arise largely from global structural mutations rather than local refinement (Lin et al., 2026a). Existing methods either confine mutation to prompts or skills (GEPA, A-Evolve), leaving most of the search space unexplored, or rewrite the whole harness (Self-Harness, Meta-Harness) but inherit the exploitation bias of their fixed LLM mutators. **Second**, *sample-efficient search*: such exploration is also costly. Fast oracles let instance-level methods like FunSearch evaluate millions of candidates, whereas scoring one harness takes hours. Every evaluation, failures included, must therefore inform the search. Yet most prior methods are *failure-blind*, retaining only survivors in the form of a single candidate (A-Evolve), a Pareto frontier (GEPA), or the fittest per MAP-Elites cell (OpenEvolve, AlphaEvolve), so failed directions are prone to being retried. **Third**, *self-adaptive search*: the effectiveness of AHD hinges on its search strategy, such as which parents are selected, how they are mutated, and which candidates are kept. Yet most ES methods, including OpenEvolve, Self-Harness, and Meta-Harness,



(a) General skeleton for evolutionary search

Method	Level	Domain	Memory	Parent selection	Mutator
(0) Open-loop test-time scaling. *No executed outcome feeds the next attempt, so gains cannot compound.*					feedback ✗
Best-of-N (Cobbe et al., 2021)	INSTANCE	Math	—	Best of N i.i.d. draws	Single LLM
Chain-of-Thought (Wei et al., 2022)	INSTANCE	Reasoning	—	—	Single LLM
Tree-of-Thoughts (Yao et al., 2023a)	INSTANCE	Planning	Transient thought tree	LLM-scored states	Single LLM
(I) Iterative refinement. *Executed feedback steers one greedy lineage; no diversity-preserving population, so it stalls at local optima.*					+ **feedback** ✓
Reflexion (Shinn et al., 2023)	INSTANCE	Code, QA	Verbal reflections	Latest attempt	Single LLM
Eureka (Ma et al., 2024)	INSTANCE	RL rewards	Best-so-far reward	Iteration best	Single LLM
AIDE (Jiang et al., 2025)	INSTANCE	ML tasks	Solution tree	Fixed tree rule	Single LLM
Self-Harness (Zhang et al., 2026a)	STRATEGY	Harnesses	Single harness	Current harness	Single LLM
A-Evolve (Liu et al., 2026b)	STRATEGY	Skills	Single workspace	Current workspace	Single Agent
AHE (Lin et al., 2026a)	STRATEGY	Harnesses	Single git workspace	Current harness	Single Agent
(II) Population search. *An archive preserves diversity, but every search knob follows a preset schedule, not live progress.*					+ **population** ✓
ELM (Lehman et al., 2022)	INSTANCE	Robotics	MAP-Elites grid	Random-niche elite	Single LLM
FunSearch (Romera-Paredes et al., 2024)	INSTANCE	Algorithms	Island archive	Island exemplars	Single LLM
AlphaEvolve (Novikov et al., 2025)	INSTANCE	Algorithms	MAP-Elites islands	Island + inspirations	Multi-LLM ensemble
OpenEvolve (Sharma, 2025)	INSTANCE	Algorithms	MAP-Elites islands	Island + inspirations	Fixed-mix multi-LLM
AutoHarness (Lou et al., 2026)	INSTANCE	Text games	Hypothesis tree	Thompson sampling	Single LLM
GEPA (Agrawal et al., 2025)	STRATEGY	Prompts	Pareto front	Pareto sampling	Single LLM
Meta-Harness (Lee et al., 2026)	STRATEGY	Harnesses	Population + Pareto	Seed harness (fixed)	Single Agent
(III) Adaptive strategy. *One or more components are retuned online from search progress, but by a fixed, hand-coded rule.*					+ **adaptive** ✓
ShinkaEvolve (Lange et al., 2025)	INSTANCE	Algorithms	Island program database	Fitness + novelty	Multi-LLM ↻ *bandit picks model*
EvoControl (Hu et al., 2026)	INSTANCE	Code	Population + memory ↻ *memory carries across tasks*	Fitness-proportional	Single LLM
AdaEvolve (Cemri et al., 2026)	INSTANCE	Algorithms	Async islands ↻ *UCB shifts island budget*	UCB across islands	Single LLM
CORAL (Qu et al., 2026)	INSTANCE	Algorithms	Shared file-system	Agent's choice ↻ *agent re-picks parent*	Async multi-agent
DarwinX (Zhang et al., 2026b)	STRATEGY	Harnesses	Archive of all variants	β-greedy on lineage gain	Single proposer ↻ *task regime picks evidence*
(IV) Self-adaptive strategy. *The adaptation rule is no longer fixed: the search rewrites its own strategy when progress stalls.*					+ **self-adaptive** ✓
EvoX (Liu et al., 2026)	INSTANCE	Algorithms	Population + strategy archive	Strategy chooses parent ↻ *LLM rewrites the strategy*	Single LLM ↻ *LLM rewrites prompt*
MILO (ours)	STRATEGY	Harnesses	Failure-aware per-island lineage trees ↻ *inter-island graft & speciation*	Softmax on dominance × similarity ↻ *Dom, Sim self-adapt based on memory state*	Multi-Agent ↻ *Reassign mutators across islands*

(b) Methods by discovery level (strategy in blue) and adaptiveness (0–IV)

Figure 2: **Categorizing LLM-driven evolutionary search. (a)** The loop shared by all methods. **(b)** Most methods fix their search strategy in advance (stages 0–II); those that adapt it online (III–IV) are mostly instance-level. ↻: adaptation by fixed rule (*adaptive*) or meta-evolution (*self-adaptive*).

fix the strategy apriori and thus cannot escape local optima. Some adapt one component online by a hard-coded rule (Sec. 2). Even EvoX, the most adaptive, evolves only how parents are selected and mutated, while its mutator, curriculum, and all else stay fixed. **Fourth**, *multi-objective fitness*: the harness sets not only whether an agent succeeds but also its token consumption and latency. For instance, rewriting a tool description cut completion time by 40% (Anthropic, 2025b). Yet almost all ES methods optimize accuracy alone and thus drift to costly harnesses (Zhang et al., 2025b). Only EvoFlow (Zhang et al., 2025a) and Meta-Harness optimize token cost; none optimizes latency.

Key Insight. We address automated harness discovery with **MILO** (*Meta-evolutionary Island Orchestration*), which couples an island-based memory of explored harnesses, multiple mutator agents, and an orchestrator agent (Fig. 1; Secs. 4.1–4.2). Our key insight is to evolve the complete search strategy alongside the harness, through four design choices. *i) Search space exploration*: each *island* (lineage) grows in isolation from a distinct initial harness (seed) with an assigned mutator. Islands exchange successful ideas but never compete for survival, confining a mutator's exploitative bias to its island. In fact, this design lets even the weakest seed yield the best harness (Tab. 2(b)). *ii) Sample-efficient search*: our *hierarchical memory* keeps each seed-to-candidate path, including rejected candidates. Successes show which edits paid off, while failures prune later mutations and push exploration to bolder structural changes. *iii) Self-adaptive search*: at a local optimum, our orchestrator diagnoses the island, revising its mutator, memory, and curriculum. *iv) Multi-objective fitness*: a candidate is admitted only if it advances the Pareto frontier in accuracy, tokens, and latency.

Contributions. The main contributions of this paper are summarized as follows:

- **Automated harness discovery (AHD) via diversity-preserving, prune-inducing, exploratory search.** We formalize harness engineering as strategy-level discovery of a solver (a harness) generalizing over tasks (Sec. 3) and propose MILO, an AHD framework. Its memory is partitioned into lineages (*islands*) sharing ideas but never competing, so distinct harnesses survive the exploitative bias of mutator agents (*diversity-preserving*). It retains rejected candidates, so mutators avoid failed patterns (*prune-inducing*) and pursue bolder structural changes (*exploratory*, Sec. 4).
- **Self-adaptive search via a meta-evolutionary orchestrator agent.** We introduce an orchestrator in evolutionary search (ES) that detects and escapes local optima by evolving the search strategy (mutators, memory, and curriculum). To our knowledge, no prior ES adapts as broadly (Sec. 4).
- **State-of-the-art results on software engineering and open mathematical problems.** MILO outperforms eight SoTA harnesses and six evolutionary search methods on Terminal-Bench 2.1, PaperBench, and DeepSWE with Opus 4.8 and gpt-oss-120b. With gpt-oss-120b, it reaches 15.6% on DeepSWE, $20\times$ its initial harness, while prior search stays below 1%. Its harnesses generalize: the Terminal-Bench 2.1 harness scores $2.7\times$ Mini-SWE-Agent on the harder Frontier-Bench without further search. On EinsteinArena, MILO sets new records on three open problems, surpassing the best-known bounds, including those of AlphaEvolve, TTT-Discover, and EvoX (Sec. 5.4).

2 Related Work

Instance-level discovery. Most LLM-driven search is instance-level. We abstract it into four components in Fig. 2(a): i) memory, ii) parent selection, iii) mutation, and iv) evaluation; methods differ in how much they *add*, and how much they *adapt* online (Fig. 2(b)). Stages 0-I cannot leave a greedy lineage: *open-loop test-time scaling* does not feed outcomes back (Cobbe et al., 2021; Yao et al., 2023a), and *iterative refinement* closes the loop but hits local optima (Shinn et al., 2023). *Population search* (Stage II) adds a diversity-preserving archive: FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025), and OpenEvolve (Sharma, 2025) spread candidates across islands, migrating on a fixed schedule. Yet their memory keeps only survivors, not failed candidates, and their hand-set strategy cannot react to stalls. Borrowing Eiben et al. (1999)'s terminology, *adaptive strategies* (Stage III) retune one component by a fixed rule: ShinkaEvolve (Lange et al., 2025) bandits the mutating model, and AdaEvolve (Cemri et al., 2026) shifts compute across islands. Only *self-adaptive* EvoX (Liu et al., 2026) (Stage IV) rewrites the strategy itself when progress stalls.

Strategy-level discovery. Strategy-level methods are far fewer, and vary in how much of the harness they evolve. APE (Zhou et al., 2022), OPRO (Yang et al., 2023), TextGrad (Yuksekgonul et al., 2024), DSPy (Khattab et al., 2024), and GEPA (Agrawal et al., 2025) tune only prompts, while A-Evolve (Lin et al., 2026b) and SkillOpt (Yang et al., 2026) add skills and memory. Those that evolve the whole harness itself stay on the lower stages of Fig. 2(b): Self-Harness (Zhang et al., 2026a) refines a single harness (Stage I), as does AIDE^2 (Srikanth et al., 2026). Meta-Harness (Lee et al., 2026) keeps a population but no parent selection: every candidate edits the seed (Stage II). DarwinX (Zhang et al., 2026b) selects over a harness archive, adapting by hard-coded rules (Stage III).

Overall, methods at both levels suffer *diversity decay*, and strategy-level ones sit at low adaptiveness stages. MILO addresses both, self-adapting the search more broadly than any prior ES we know.

3 Preliminaries and Problem Statement

Definition 3.1 (*Agents and harnesses*). An agent $A := \langle \mathcal{M}, h \rangle$ pairs a model $\mathcal{M}$ with a *harness* $h \in \mathcal{H}$, the layer governing its interaction with the environment: $\mathcal{M}$ proposes actions, h executes them. With $\mathcal{M}$ fixed, MILO identifies each agent with its harness, and we write A_h for $A = \langle \mathcal{M}, h \rangle$. It is *representation-agnostic*, accepting any harness with a runnable entry point (`build_agent`, App. B.1) and editable source. Hence h may be a Python program, a YAML file, or a tool/skill repository, combining ReAct (Yao et al., 2023b), CodeAct (Wang et al., 2024), and sub-agents (LangChain, 2025). As each h is finite but arbitrarily long, $\mathcal{H}$ is countably infinite.

Definition 3.2 (*Agent evaluation*). We evaluate an agent A_h on long-horizon tasks by its *accuracy* and *cost*. Let $\mathcal{T}$ be a task distribution (e.g., terminal-based software engineering tasks), from which a *benchmark* $\mathcal{B} = \{\tau_i\}_{i=1}^n$ is an i.i.d. sample. Each *task* $\tau = \langle s_\tau, V_\tau \rangle$ pairs an *initial state* s_τ (e.g., a repository and instructions) with a task-specific *verifier* $V_\tau : \mathcal{S} \to [0,1] \times \Sigma^*$ over *environment states* $\mathcal{S}$. Its criteria are hidden and may be deterministic (e.g., a test suite) or stochastic (e.g., an LLM-judged rubric). Executing agent A_h on task τ yields $(s', \xi, \mathbf{cost})$: a *final state* $s' \in \mathcal{S}$, reached along an *execution trace* ξ (its turn-by-turn record), and a vector $\mathbf{cost} \in \mathbb{R}^q_{\geq 0}$ (e.g., tokens, latency). The verifier returns $(\mathrm{acc}, \mathrm{rpt}) := V_\tau(s')$, an *accuracy* and a textual *report* (e.g., failing tests). As agent execution is stochastic, $(s', \xi, \mathbf{cost})$ and hence $(\mathrm{acc}, \mathrm{rpt})$ vary across runs. We thus run k rollouts per task, record per-task *accuracy*, *cost*, and *evidence*, and aggregate all three over $\mathcal{B}$:

$$\begin{array}{ll} \textit{task } \tau \in \mathcal{B}\text{:} & \textsc{Acc}(A_h,\tau) := \frac{1}{k}\sum_{j=1}^{k} \mathrm{acc}_j \in [0,1], \quad \textbf{Cost}(A_h,\tau) := \frac{1}{k}\sum_{j=1}^{k} \mathbf{cost}_j \in \mathbb{R}^q_{\geq 0}, \quad \textsc{Evid}(A_h,\tau) := \{(\xi_j, \mathrm{rpt}_j)\}_{j=1}^{k}, \\ \textit{benchmark } \mathcal{B}\text{:} & \textsc{Acc}(A_h,\mathcal{B}) := \frac{1}{|\mathcal{B}|}\sum_{\tau\in\mathcal{B}} \textsc{Acc}(A_h,\tau), \quad \textbf{Cost}(A_h,\mathcal{B}) := \frac{1}{|\mathcal{B}|}\sum_{\tau\in\mathcal{B}} \textbf{Cost}(A_h,\tau), \quad \textsc{Evid}(A_h,\mathcal{B}) := \{\textsc{Evid}(A_h,\tau)\}_{\tau\in\mathcal{B}}. \end{array} \tag{1}$$

We henceforth write h for A_h, e.g., $\textsc{Acc}(h, \mathcal{B})$. By convention, high Acc and low **Cost** are better.

Problem statement (*Automated Harness Discovery, AHD*). Given an agent $A_h = \langle \mathcal{M}, h \rangle$ and a target task distribution $\mathcal{T}$, our goal is to transform A_h into $A_{h^\star} = \langle \mathcal{M}, h^\star \rangle$. That is, with the model $\mathcal{M}$ fixed, we discover a new harness $h^\star$ that advances the agent's accuracy–cost frontier over $\mathcal{T}$.

Concretely, the *inputs* are (i) the agent A_h (optionally with more seed harnesses $\mathcal{H}_{\mathrm{init}} \ni h$), whose model $\mathcal{M}$ is a *black box* with inference access only, (ii) the benchmark $\mathcal{B}$, sampled from $\mathcal{T}$, split into $\mathcal{B}_{\mathrm{search}} \sqcup \mathcal{B}_{\mathrm{val}}$, (iii) constraints $\mathcal{C}$ requiring *validity* (a runnable agent), *generality* (no logic tailored to individual tasks), and optional *user* restrictions, and (iv) a budget R (e.g., number of rounds, time limit). Within budget R, AHD *searches* the countably infinite set $\mathcal{H}_{\mathcal{C}} := \{h \in \mathcal{H} : h \models \mathcal{C}\}$, guided by $\mathcal{B}_{\mathrm{search}}$, and *outputs* the harness $h^\star$ that most advances the accuracy–cost frontier on $\mathcal{B}_{\mathrm{val}}$. The *goal* is for $A_{h^\star}$ to surpass A_h and even the strongest hand-built agents $\mathcal{A}_{\mathrm{SOTA}}$ on this frontier.

4 PROPOSED METHODOLOGY

We introduce MILO, a strategy-level evolutionary framework (Fig. 1) that discovers harnesses to advance an agent's accuracy–cost frontier. Three components drive the search (Sec. 4.1): *i) hierarchical lineage memory* of all generated harnesses, *ii) evidence-driven mutator agents* that generate new candidates, and *iii) an orchestrator agent* that adapts the search strategy. An evolution loop (Sec. 4.2, Algo. 1) couples them. Sec. 4.3 generalizes to other domains and instance-level discovery.

4.1 CORE INTERACTING COMPONENTS

4.1.1 HIERARCHICAL LINEAGE MEMORY

The memory holds every harness the search generates. At round r, it consists of $K^{(r)} \in \mathbb{N}$ *islands*, each evolving its own *lineage* in isolation, so lineages never compete with each other. Island j stores its lineage *hierarchically* as a node- and edge-labeled tree $\mathcal{L}_j^{(r)}$: its nodes $\mathcal{H}_j^{(r)}$ are the harnesses in the island so far, its directed edges $E_j^{(r)}$ run from parent to child, and its labelings $\ell_{\mathcal{H}}$ and ℓ_E record each harness's evaluation and the edit that produced it. The memory is thus a forest

$$\mathcal{F}^{(r)} \;:=\; \big\{\mathcal{L}_j^{(r)}\big\}_{j=1}^{K^{(r)}}, \quad \text{where} \quad \mathcal{L}_j^{(r)} := \big(\mathcal{H}_j^{(r)},\; E_j^{(r)},\; \ell_{\mathcal{H}},\; \ell_E\big), \tag{2}$$

$$\ell_{\mathcal{H}}(h) := \big(\textsc{Acc}(h, \mathcal{B}_{\text{search}}),\; \textbf{Cost}(h, \mathcal{B}_{\text{search}}),\; \textsc{Evid}(h, \mathcal{B}_{\text{search}}),\; \textsc{Adm}(h)\big), \quad \ell_E(e) := \big(\delta_e,\; \Delta_e^{\text{acc}},\; \boldsymbol{\Delta}_e^{\text{cost}}\big). \tag{3}$$

That is, a node stores its accuracy, cost, and evidence on $\mathcal{B}_{\text{search}}$ (Eq. 1) and its admission verdict $\textsc{Adm}(h) \in \{0, 1\}$ (Sec. 4.2). An edge stores the edit δ_e, the source-code patch turning the parent harness h_{par} into the child harness h, and the resulting gains in search-set accuracy and cost, $\Delta_e^{\text{acc}} := \textsc{Acc}(h, \mathcal{B}_{\text{search}}) - \textsc{Acc}(h_{\text{par}}, \mathcal{B}_{\text{search}})$ and $\boldsymbol{\Delta}_e^{\text{cost}} := \textbf{Cost}(h_{\text{par}}, \mathcal{B}_{\text{search}}) - \textbf{Cost}(h, \mathcal{B}_{\text{search}})$.

Initialization. At round $r = 1$, we pick from $\mathcal{H}_{\text{init}}$ the $K^{(1)}$ harnesses that are most dissimilar in behavior (Sec. 4.2) and seed one island with each of them. Every tree $\mathcal{L}_j^{(1)}$ is thus a single node.

Round-wise growth. In each round r, island j's mutator agent (Sec. 4.1.2) mutates a selected parent $h_{\text{par}} \in \mathcal{H}_j^{(r)}$ into a child h (Sec. 4.2, stages 1–2). The child grows $\mathcal{L}_j^{(r)}$ into $\mathcal{L}_j^{(r+1)}$ by adding node h, edge $(h_{\text{par}} \to h)$, and their labels. Crucially, the memory is *append-only*: it keeps both admitted and rejected children (Sec. 4.2, stage 4) with their verdict $\textsc{Adm}(h)$. Mutators thus see which edits paid off and which failed, so they avoid repeating failures and pursue bolder structural changes.

Self-adaptive nature. The orchestrator (Sec. 4.1.3) restructures the memory in two ways. GRAFT lets an island borrow from a donor island: its next child h is bred with a *co-parent* h_{co} from the donor, kept as a *cross-island edge* $(h_{\text{co}} \to h)$. These edges form a set $E_\times$, so each $\mathcal{L}_j^{(r)}$ stays a tree while the whole memory is a *directed acyclic graph* once $E_\times$ is non-empty. SPECIATE moves a *crowded-out* subtree rooted at h to a *new island* with its own mutator, so $K^{(r)}$ grows by one.

4.1.2 EVIDENCE-DRIVEN MUTATOR AGENTS

A mutator introduces new candidate harnesses into the population by modifying existing ones. At round r, each island j's assigned mutator $m_j^{(r)}$ turns a selected parent h_{par} into a child harness h (Sec. 4.2, stages 1–2). In our framework, mutators are *coding agents* that reason over many turns with coding tools (read, write, edit, search, bash). Each receives three inputs: (a) the *source code* of h_{par}; (b) its *failure evidence* $\textsc{Evid}(h_{\text{par}}, \mathcal{B}_{\text{search}}^{(r)})$ (Eq. 1); and (c) the full *island snapshot* of $\mathcal{L}_j^{(r)}$:

$$h \;\leftarrow\; m_j^{(r)}\Big(h_{\text{par}},\; \textsc{Evid}(h_{\text{par}}, \mathcal{B}_{\text{search}}^{(r)}),\; \mathcal{L}_j^{(r)}\Big). \tag{4}$$

In the mutator prompt (App. C.1, Listings 6), inputs (a) and (b) are given as on-disk paths. This keeps the prompt uncluttered and lets the agent open only what its diagnosis needs. Input (c) renders $\mathcal{L}_j^{(r)}$ as an indented tree, like a UNIX directory listing. Each harness $h' \in \mathcal{L}_j^{(r)}$ occupies one line under its parent, giving (i) its depth, (ii) the on-disk path of h', (iii) the label $\ell_E(e)$ of its incoming edge $e = (h'_{\text{par}} \to h')$, and (iv) its own label $\ell_{\mathcal{H}}(h')$ (Eq. 3). The prompt also prescribes a *diagnose-first workflow*. The agent finds from the evidence why the parent fails, then applies anything from *targeted refinement* (e.g., a local fix) to *structural redesign* (e.g., a control-flow rewrite).

The mutator must decide which aspect of the parent harness to change and which changes are worth exploring. MILO grounds this decision in two sources of evidence. The parent's traces and reports give a *focused view* of its weaknesses and thus what to fix; the island snapshot gives a *global view* of which edits succeeded or failed per lineage and thus whether *targeted refinement* suffices or *structural redesign* is due. Prior ES mutators get far less evidence: FunSearch, AlphaEvolve, and

EvoX expose only the parent's source and a few top candidates; OpenEvolve, ShinkaEvolve, GEPA, A-Evolve, and Self-Harness add the parent's execution feedback; Meta-Harness has no parent selection: every candidate edits the root, so no tree can even form. To our knowledge, none combines lineage and execution history. Most also mutate via a one-shot LLM call (Fig. 2(b)) that sees only its context, while MILO's coding agent fetches what it needs to iterate on the child harness.

Island-mutator assignment. We maintain a pool $\mathcal{G} := \{\mu_1, \ldots, \mu_n\}$ of mutators, comprising off-the-shelf frontier coding agents and custom agents pairing a frontier LLM with an open-source harness (App. A.1). Different model–harness pairings induce different mutation tendencies, so islands with different mutators explore different mutation strategies. Entering round r, $m^{(r)} : \{1, \ldots, K^{(r)}\} \to \mathcal{G}$ is the *island-mutator assignment*, and $\mu_j := m_j^{(r)}$ is island j's mutator.

Initialization. Before round $r = 1$, $m^{(1)}$ assigns each island a uniformly random mutator from $\mathcal{G}$.

Self-adaptive nature. Unlike the memory, the assignment carries over unchanged between rounds, $m^{(r+1)} = m^{(r)}$, until the orchestrator's REASSIGN action (Sec. 4.1.3) revises it for a stalled island j to fit the island's diagnosed need. It then sets $m_j^{(r+1)} \leftarrow \mu$, choosing $\mu \in \mathcal{G}$, $\mu \neq m_j^{(r)}$.

4.1.3 META-EVOLUTIONARY ORCHESTRATOR AGENT

We introduce an *orchestrator* agent $\mathcal{O}$ that helps the search escape local optima by adapting its strategy; unlike the per-island mutators, it observes all islands. Let the *search configuration* entering round r be $\Theta^{(r)} = \big(\mathcal{F}^{(r)}, m^{(r)}, \mathcal{B}_{\text{search}}^{(r)}\big)$: the memory, the island-mutator assignment, and the *curriculum* of evaluation tasks. In a normal round, only the memory changes, $\mathcal{F}^{(r)} \to \mathcal{F}^{(r+1)}$ (Sec. 4.1.1); the other two carry over. When the stall of an island $j^\star$ (Sec. 4.2) triggers $\mathcal{O}$ before round r, it *diagnoses* ($\mathcal{O}_1$) island $j^\star$ against $\Theta^{(r)}$, then *intervenes* ($\mathcal{O}_2$) with one or a few actions:

$$\underbrace{d \;\leftarrow\; \mathcal{O}_1\Big(\Theta^{(r)},\, j^\star\Big)}_{\text{diagnose}}, \qquad \underbrace{\omega \leftarrow \mathcal{O}_2(d), \quad \Theta^{(r)} \;\leftarrow\; \text{apply}\Big(\Theta^{(r)};\, \omega\Big)}_{\text{intervene}}. \tag{5}$$

Diagnose ($\mathcal{O}_1$). $\mathcal{O}_1$ compares the stalled island $j^\star$ with the rest of $\mathcal{F}^{(r)}$ (prompt in App. C.2) and identifies the bottleneck d as one of four sources: the *mutator*, when its children are repeatedly rejected, inert, or minor variations of failed edits (*dry mutator*); the *lineage*, when it produces no admitted child despite another island solving tasks on which $j^\star$ fails (*exhausted lineage*); *selection*, when a dominant lineage crowds out a distinct minority subtree solving different tasks (*crowded-out niche*); or the *population*, when all islands approach the same ceiling (*whole-population plateau*).

Intervene ($\mathcal{O}_2$). Given diagnosis d, $\mathcal{O}_2$ composes an intervention $\omega = (\omega_1, \ldots, \omega_l)$ with $l \leq 2$:

$$\omega_i \in \big\{\textsc{Reassign}(j, \mu),\ \textsc{Graft}(j_{\text{donor}} \to j_{\text{dst}}),\ \textsc{Speciate}(h, \mu),\ \textsc{Curriculum}(\mathcal{B}')\big\}. \tag{6}$$

The four actions ω_i correspond to the four diagnoses (App. C.2). $\textsc{Reassign}(j, \mu)$ gives island j a new mutator $\mu \in \mathcal{G}$, $m_j^{(r)} \leftarrow \mu$. $\textsc{Graft}(j_{\text{donor}} \to j_{\text{dst}})$ breeds both islands' best harnesses as co-parents, so the destination inherits the donor's capability. $\textsc{Speciate}(h, \mu)$ promotes the subtree rooted at h to a new island under mutator μ, preserving a niche that would otherwise be suppressed. $\textsc{Curriculum}(\mathcal{B}')$ replaces the search set, $\mathcal{B}_{\text{search}}^{(r)} \leftarrow \mathcal{B}'$, favoring high-*regret* tasks. Over the admitted harnesses $\mathcal{H}^+$ (Sec. 4.2), regret is the difference between the best attainable and the population's mean accuracy, $\rho(\tau) = \max_{h \in \mathcal{H}^+} \textsc{Acc}(h, \tau) - \text{mean}_{h \in \mathcal{H}^+} \textsc{Acc}(h, \tau)$. Both terms coincide when every harness solves the task (trivial) or none does (impossible), so regret vanishes at either extreme. It peaks on *frontier* tasks that a few harnesses solve and most fail, where search helps most. The loop then applies ω mechanically, except GRAFT, whose child must clear admission (Eq. 7) or is rejected. Overall, by rewriting $\Theta^{(r)}$, the search self-adapts its memory, mutators, and curriculum.

4.2 EVOLUTION LOOP

We represent a harness h by two vectors. Its **multi-objective fitness vector** $\mathbf{F}(h) \in [0, 1]^{1+q}$ stores the accuracy $\textsc{Acc}(h, \mathcal{B}_{\text{search}})$ as entry $F_0(h)$ and the economy $1 - \textbf{Cost}_i(h, \mathcal{B}_{\text{search}})/c_i$ on each cost i as entry $F_{i \geq 1}(h)$, where c_i is the agent's fixed per-task cap. Harness h *dominates* h' ($h \succ h'$) if $F_0(h) > F_0(h') + \varepsilon$ for a tolerance ε, or if both lie within ε and h wins on every cost axis. In island j's *admitted set* $\mathcal{H}_j^+ := \{h \in \mathcal{H}_j^{(r)} : \textsc{Adm}(h) = 1\}$ (stage 4), the non-dominated members form its *Pareto front* $\Pi(\mathcal{H}_j^+) := \{h \in \mathcal{H}_j^+ : \nexists\, h' \in \mathcal{H}_j^+,\ h' \succ h\}$, and $\mathcal{H}^+ := \bigcup_j \mathcal{H}_j^+$ spans all islands. The **behavior vector** $\mathbf{b}(h) \in [0, 1]^{|\mathcal{B}_{\text{search}}|}$ collects the per-task pass rates $\textsc{Acc}(h, \tau)$ and defines, for any two harnesses h_1, h_2, a *similarity* matrix $\text{Sim}_{h_1 h_2} := 1 - \|\mathbf{b}(h_1) - \mathbf{b}(h_2)\|_1 / |\mathcal{B}_{\text{search}}| \in [0, 1]$; with the *dominance* matrix $\text{Dom}_{h_1 h_2} := \mathbf{1}[h_2 \succ h_1]$, it drives parent selection (stage 1). Each round, every island j advances one generation through five stages (Algo. 1), which we detail next.

1. Parent selection. We score each admitted candidate $h_1 \in \mathcal{H}_j^+$ by $\eta(h_1) = -\sum_{h_2} \mathrm{Dom}_{h_1 h_2} \mathrm{Sim}_{h_1 h_2}$ over the other admitted harnesses $h_2 \in \mathcal{H}_j^+$. The parent is sampled at temperature T, using $h_{\mathrm{par}} \sim \mathrm{softmax}\big(\eta(h_1)/T\big)$. This favors harnesses with fewer dominators (*exploitation*) and distinct behaviors (*exploration*).

2. Mutation. Island j's mutator $m_j^{(r)}$ edits h_{par}'s source into a child h (Eq. 4) satisfying $\mathcal{C}$, given the parent's failure evidence and the snapshot of $\mathcal{L}_j^{(r)}$ (Sec. 4.1.2).

Algorithm 1: **MILO's meta-evolutionary loop.** Candidate harnesses and their search strategy evolve jointly.

Input: agent $A_h = \langle \mathcal{M}, h \rangle$, seeds $\mathcal{H}_{\mathrm{init}} \ni h$; benchmark $\mathcal{B} = \mathcal{B}_{\mathrm{search}} \sqcup \mathcal{B}_{\mathrm{val}}$; constraints $\mathcal{C}$; budget R; *hyperparameters:* mutator pool $\mathcal{G}$, initial island count K, patience P, rollouts k
Output: harness $h^\star$ that most advances the agent's accuracy–cost frontier

$\mathcal{F} \leftarrow$ SEED-MEMORY$(\mathcal{H}_{\mathrm{init}}, K)$ ▷ **initialize:** K most dissimilar seeds, one island each (Sec. 4.1.1)
$m \leftarrow$ ASSIGN-MUTATORS$(\mathcal{G}, K)$ ▷ **initialize:** random island-to-mutator map (Sec. 4.1.2)
$\Theta \leftarrow (\mathcal{F}, m, \mathcal{B}_{\mathrm{search}})$ ▷ initial search configuration (Sec. 4.1.3)
$\mathrm{stall}_j \leftarrow 0$ for all islands j ▷ stall counters (Sec. 4.2)
for $r = 1$ **to** R **do**
  for all islands j, i.e., lineage trees $\mathcal{L}_j \in \mathcal{F}$, **in parallel do**
    $h_{\mathrm{par}} \leftarrow$ SELECT-PARENT$(\mathcal{H}_j^+)$ ▷ **stage 1:** parent selection from $\mathcal{L}_j$'s admitted set (Sec. 4.2)
    $h \leftarrow m_j\big(h_{\mathrm{par}}, \mathrm{EVID}(h_{\mathrm{par}}, \mathcal{B}_{\mathrm{search}}), \mathcal{L}_j\big)$ with $h \models \mathcal{C}$ ▷ **stage 2:** mutation (Sec. 4.2)
    $(\mathrm{ACC}, \mathbf{COST}, \mathrm{EVID})(h) \leftarrow$ EVALUATE$(h, \mathcal{B}_{\mathrm{search}}, k)$ ▷ **stage 3:** fitness and evidence (Sec. 4.2)
    $\mathrm{ADM}(h) \leftarrow$ ADMIT$(h, \mathcal{H}_j^+)$ ▷ **stage 4:** child acceptance (Sec. 4.2)
    $\mathcal{L}_j \leftarrow$ ADD-CHILD$(\mathcal{L}_j, h_{\mathrm{par}} \to h, \ell_{\mathcal{H}}(h), \ell_E(h_{\mathrm{par}} \to h))$ ▷ record in memory (Eq. 3)
    if $\mathrm{ADM}(h)$: EVALUATE$(h, \mathcal{B}_{\mathrm{val}}, k)$ ▷ **stage 5:** progress monitoring on held-out tasks (Sec. 4.2)
    $\mathrm{stall}_j \leftarrow 0$ **if** h raised island j's best held-out accuracy, **else** $\mathrm{stall}_j + 1$ ▷ stall counter
  end for
  $\Theta \leftarrow (\mathcal{F}, m, \mathcal{B}_{\mathrm{search}})$ ▷ refresh search configuration: $\mathcal{F}$ grew (line 11)
  if $\max_j(\mathrm{stall}_j) \geq P$ **then** ▷ an island stalled for P rounds: orchestrate (Sec. 4.1.3)
    $j^\star \leftarrow \arg\max_j(\mathrm{stall}_j)$; $d \leftarrow \mathcal{O}_1(\Theta, j^\star)$ ▷ diagnose (Eq. 5)
    $\Theta \leftarrow \mathrm{apply}(\Theta; \mathcal{O}_2(d))$ ▷ intervene: modify search configuration Θ (Eq. 6)
    $\mathrm{stall}_j \leftarrow 0$ for all islands j ▷ reset counters
  end if
end for
return $h^\star \in \mathcal{H}^+$ that most advances the accuracy–cost frontier on $\mathcal{B}_{\mathrm{val}}$ ▷ AHD objective (Sec. 3)

3. Fitness evaluation. The child h runs k rollouts on every task in $\mathcal{B}_{\mathrm{search}}^{(r)}$, recording per-task accuracy, cost, and traces $\big(\mathrm{ACC}(h,\tau), \mathbf{COST}(h,\tau), \mathrm{EVID}(h,\tau)\big)$ in $\ell_{\mathcal{H}}(h)$ (Eq. 3). From these we form its multi-objective fitness vector $\mathbf{F}(h)$, as described above.

4. Child acceptance. Child h enters $\mathcal{L}_j^{(r+1)}$ with verdict $\mathrm{ADM}(h)$ (Eq. 3). It is accepted iff it enlarges $\mathcal{H}_j^+$'s dominated region from the origin, i.e., positive *hypervolume contribution* $\Delta\mathrm{HV}_j(h)$:

$$\mathrm{ADM}(h) = \mathbf{1}\big[\nexists\, h' \in \mathcal{H}_j^+ : h' \succ h\big] = \mathbf{1}\big[\Delta\mathrm{HV}_j(h) > 0\big], \quad \Delta\mathrm{HV}_j(h) = \mathrm{HV}(\mathcal{H}_j^+ \cup \{h\}) - \mathrm{HV}(\mathcal{H}_j^+) \geq 0. \tag{7}$$

5. Progress monitoring and Strategy orchestration. Admitted harnesses are re-scored on the held-out split $\mathcal{B}_{\mathrm{val}}$ to guard against overfitting. Island j's counter stall_j increments each round and resets on a held-out gain; once it reaches patience P, control escalates to the orchestrator (Sec. 4.1.3).

4.3 Generalization to Other Domains

MILO is *model-agnostic*: it can improve any agent irrespective of its underlying model, as shown for frontier and open-weight models (Tabs. 1–2). It is *domain-agnostic*: it only scores and edits source code, so it can improve agents for any task distribution, shown on four benchmarks (Sec. 5.4) and applicable to others like autoformalization (Jana et al., 2026b) and time-series engineering (Cai et al., 2025). As a strategy-level discovery framework, it can also generalize to other solver families like preference-optimization algorithms (Lu et al., 2024) and design heuristics (Dat et al., 2024).

Extension to instance-level discovery. Instance-level ES searches for better solution-generating programs. We propose applying AHD to the agent that writes them (Fig. 3). To evaluate this agent across different starting conditions, we pair starting solutions to the same scientific problem with optimizer programs for it to edit. Each pair forms a workspace s_τ for a task $\tau = \langle s_\tau, V_\tau \rangle$ in $\mathcal{B}_{\mathrm{search}}$. In stage 2 (Sec. 4.2), the mutator edits h_{par} into h. Stage 3 then runs A_h in each workspace to write a revised optimizer π. The verifier V_τ runs π from the supplied solution to obtain solution x and scores its normalized gain over the start. Mean accuracy $\mathrm{ACC}(h, \mathcal{B}_{\mathrm{search}})$ is harness fitness.

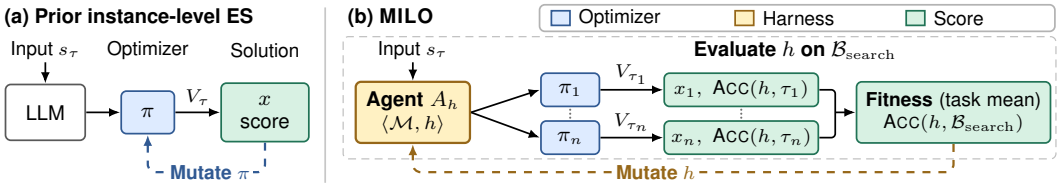


Figure 3: **MILO for instance-level ES.** (a) Prior ES evolves π; (b) MILO evolves A_h, which generates π.

5 Experimental Evaluation

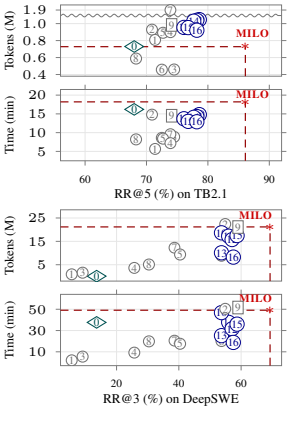
5.1 Benchmarks and Tasks

We evaluate *strategy-level discovery* on three long-horizon benchmarks: **Terminal-Bench 2.1** (Merrill et al., 2026) for command-line tasks, with the harder **Frontier-Bench** (Marten et al., 2026) to test transfer without search, **PaperBench-CodeDev** (Starace et al., 2025) for paper replication, and **DeepSWE** (Huang et al., 2026) for multi-file software engineering. For *instance-level discovery*, we use **EinsteinArena** (Bianchi et al., 2026), a leaderboard of open mathematical problems (App. A.2).

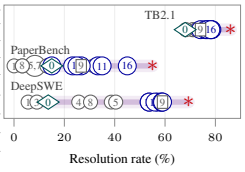
Table 1: **Performance and cost comparison of harnesses, with a frontier backbone.** All rows use Opus 4.8 as model $\mathcal{M}$. **Bold**/underline = column best/second. *Gray*: no fitness gain over Best-of-3 during search; all other entries report the best-fitness harness evaluated on the full benchmark.

Harness	Design	Terminal-Bench 2.1 pass@5↑	Terminal-Bench 2.1 PR@5↑	Terminal-Bench 2.1 RR@5↑	PaperBench PR@3↑	PaperBench RR@3↑	DeepSWE pass@3↑	DeepSWE PR@3↑	DeepSWE RR@3↑
◇ Minimal harness	HAND	85.2	74.9 ±2.7	68.0 ±3.1	65.7 ±1.8	15.0 ±0.0	23.9	20.3 ±3.1	13.6 ±2.7
State-of-the-art harnesses									
① Cline	EXPERT	87.5	80.0 ±2.1	71.4 ±3.0	7.0 ±0.9	1.7 ±1.0	16.8	14.0 ±2.9	5.6 ±2.2
② DeepAgents	EXPERT	83.0	76.8 ±2.2	70.9 ±2.5	75.1 ±0.5	31.7 ±3.3	77.9	85.7 ±2.9	54.9 ±4.1
③ Goose	EXPERT	89.8	83.0 ±1.9	74.5 ±2.7	66.4 ±1.8	8.3 ±1.7	27.4	24.2 ±3.1	9.1 ±3.2
④ Mini-SWE-Agent	EXPERT	89.9	83.2 ±2.0	73.9 ±2.7	57.2 ±0.5	3.3 ±0.9	43.4	65.8 ±2.9	25.7 ±3.4
⑤ OpenCode	EXPERT	87.5	80.9 ±1.8	72.7 ±2.6	64.8 ±2.5	8.3 ±1.4	61.9	84.8 ±2.1	40.4 ±4.1
⑥ OpenHands	EXPERT	88.6	79.4 ±2.2	72.5 ±2.7	66.0 ±0.5	13.3 ±1.9	76.1	91.6 ±1.9	53.7 ±4.0
⑦ Qwen-Code	EXPERT	88.6	81.9 ±2.0	73.9 ±2.9	62.4 ±0.4	8.3 ±1.7	65.5	86.3 ±2.5	38.6 ±4.2
⑧ Terminus-2	EXPERT	86.5	80.0 ±2.3	68.3 ±3.1	61.4 ±0.1	3.3 ±1.7	49.6	83.5 ±2.3	30.4 ±3.6
Automatically-discovered harnesses *(SoTA evolutionary search vs.* MILO*)*									
⑨ Best-of-3	HAND	87.5	79.1 ±2.0	74.1 ±2.6	74.5 ±0.5	26.7 ±3.0	77.0	90.6 ±2.2	59.0 ±3.9
⑩ GEPA *(prompt)*	AUTO	92.0	82.8 ±2.2	78.6 ±2.7	79.3 ±0.5	33.3 ±3.4	73.5	83.6 ±3.3	53.7 ±4.1
⑪ GEPA *(optimize anything)*	AUTO	*87.5*	*79.1* ±4.0	*74.1* ±4.3	82.3 ±0.3	35.0 ±2.9	77.9	89.4 ±2.0	58.7 ±2.8
⑫ A-Evolve *(skill, memory)*	AUTO	92.0	84.4 ±2.2	78.2 ±2.7	*74.5* ±0.8	*26.7* ±6.0	66.4	90.9 ±3.6	57.1 ±4.0
⑬ OpenEvolve	AUTO	88.6	81.9 ±2.0	76.1 ±2.6	71.1 ±0.7	23.3 ±2.9	77.9	91.8 ±2.1	53.7 ±4.1
⑭ ShinkaEvolve	AUTO	89.8	84.1 ±2.0	77.7 ±2.6	71.4 ±0.7	15.0 ±2.9	78.8	89.8 ±2.1	56.0 ±4.1
⑮ EvoX	AUTO	89.8	83.0 ±2.0	76.8 ±2.5	73.7 ±0.5	25.0 ±2.9	78.8	90.0 ±2.2	58.7 ±4.1
⑯ Meta-Harness	AUTO	92.0	82.9 ±2.2	78.2 ±2.7	85.6 ±0.3	45.0 ±3.6	*77.0*	*90.6* ±2.2	*59.0* ±4.0
✳ **MILO** *(proposed)*	AUTO	**93.2**	**90.4** ±1.3	**86.1** ±2.0	**88.6** ±0.5	**55.0** ±5.0	**86.7**	**96.4** ±0.9	**69.3** ±3.4

(a) Performance on three benchmarks



(b) Cost per attempt



(c) Agent performance range

	(A) LLM mutator	(B) +mutator agent	(C) +lineage memory	(D) +multi island	(E) +orch. (ours)
Mutator agent	✗	✓	✓	✓	✓
Lineage memory	✗	✗	✓	✓	✓
Multiple islands	✗	✗	✗	✓	✓
Orchestrator	✗	✗	✗	✗	✓
pass@5↑	88.6	89.8	90.9	92.0	**93.2**
PR@5↑	82.5 ±2.0	86.6 ±0.8	86.2 ±0.8	86.2 ±0.8	**90.4** ±0.8
RR@5↑	76.4 ±2.5	79.1 ±2.1	79.3 ±2.1	80.5 ±2.1	**86.1** ±2.0

(d) Our ablation study

5.2 Evaluation Metrics

To evaluate an agent (Eqn. 1), we run all N tasks of a benchmark k times (k=5 or 3 per leaderboard convention). Let $s_{ij} \in [0,1]$ score attempt j on task i (a *full solve* when $s_{ij} = 1$). We report pass@$k = \frac{1}{N}\sum_i \mathbf{1}[\max_j s_{ij} = 1]$, PR@$k = \frac{1}{Nk}\sum_{i,j} s_{ij}$, and RR@$k = \frac{1}{Nk}\sum_{i,j} \mathbf{1}[s_{ij} = 1]$. **pass@k** measures a solve in any attempt, **pass rate (PR@k)** gives per-attempt partial credit, and **resolution rate (RR@k)** counts full solves. On Terminal-Bench, Frontier-Bench, and DeepSWE, s_{ij} is the fraction of tests passed; error bars are task-clustered 95% confidence intervals. On PaperBench, s_{ij} is the official *replication score*, a rubric-weighted score from a GPT-5.5 judge. An attempt with $s_{ij} \geq 0.9$ counts as resolved for PaperBench's RR, whose PR and RR are the mean $\pm$ one standard error over the k runs. We also compare mean **token consumption** and **latency**.

5.3 State-of-the-Art Baselines and Experimental Setup

We compare three baseline tiers. A **minimal harness** (App. A.4), with only read/write/bash tools, measures the model without harness engineering. Next are eight **State-of-the-art harnesses**: Cline (ClineBot, 2025), DeepAgents (LangChain, 2025), Goose (Block, 2025), Mini-SWE-Agent (Yang et al., 2024b), OpenCode (SST, 2025), OpenHands (Wang et al., 2025), Qwen-Code (Alibaba, 2025), and Terminus-2 (Krafton-AI, 2026). Last are **automatically-discovered harnesses** from six search methods (Fig. 2): three at *instance-level*, OpenEvolve, ShinkaEvolve, and EvoX, and three at *strategy-level*, A-Evolve, GEPA, and Meta-Harness (details in App. A.3). We could not evaluate Self-Harness, DarwinX, and AIDE[2], as their code is not publicly available.

All search methods start from the same three expert-designed DeepAgents harnesses (B10–B12), the *seeds* (App. B.3). *Best-of-3* is the best seed's score without search. Further, each search is given the same 72-hour wall-clock limit, and we report its best candidate. The full setup is in App. A.

5.4 Experimental Results

With Opus 4.8 (Tab. 1) or gpt-oss-120b (Tab. 2) as the model $\mathcal{M}$, we compare harnesses and observe:

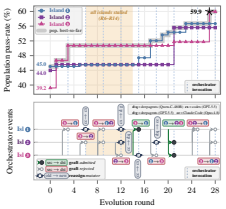
Off-the-shelf SoTA harnesses do not consistently outperform the minimal harness. First, *they can be counterproductive on some task distributions*: with Opus 4.8, seven of eight SoTA harnesses underperform the minimal harness on PaperBench RR@3, whereas six exceed it on DeepSWE by up to 41.3% (Tab. 1a). Second, *harness gains need not transfer across backbones*: on TB2.1, all eight improve RR@5 over the minimal harness with Opus 4.8, but four underperform it with gpt-oss-120b (Tab. 2a). Third, *higher token use does not imply higher accuracy*: on TB2.1 with Opus 4.8, Mini-SWE-Agent uses nearly 2× Goose's tokens for comparable RR@5 (73.9% vs. 74.5%; Tab. 1b).

MILO discovers higher-performing harnesses than SoTA evolutionary search. MILO achieves the highest accuracy across all three benchmarks. With Opus 4.8 (Tab. 1a), it improves RR@k over Best-of-3 by +12.0%, +28.3%, and +10.3% on TB2.1, PB, and DeepSWE, respectively, versus +4.5% (GEPA), +18.3% (Meta-Harness), and 0% (no improvement) for prior search. With gpt-oss-120b (Tab. 2a), it improves PR@k by +11.8%, +8.6%, and +14.8%, versus +8.2% (ShinkaEvolve), +4.8% (Meta-Harness), and +0.1% (OpenEvolve). Overall, we make the following observations:

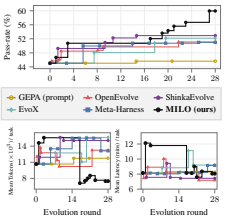
Table 2: **Performance of harnesses, with an open-source backbone.** Using gpt-oss-120b as the model $\mathcal{M}$. **Bold**/underline = column best/second; other conventions are as in Tab. 1.

Harness	Design	Terminal-Bench 2.1 pass@5↑	Terminal-Bench 2.1 PR@5↑	Terminal-Bench 2.1 RR@5↑	PaperBench PR@3↑	PaperBench RR@3↑	DeepSWE pass@3↑	DeepSWE PR@3↑	DeepSWE RR@3↑
◇ Minimal harness	HAND	35.2	38.6 ±2.1	18.6 ±2.5	14.4 ±2.4	0.0 ±0.0	0.0	0.0 ±0.0	0.0 ±0.0
State-of-the-art harnesses									
① Cline	EXPERT	36.4	47.9 ±1.8	23.4 ±2.4	7.0 ±0.6	0.0 ±0.0	0.0	0.0 ±0.0	0.0 ±0.0
② DeepAgents	EXPERT	29.5	36.8 ±2.3	15.5 ±2.4	13.8 ±0.3	0.0 ±0.0	0.0	1.2 ±0.8	0.0 ±0.0
③ Goose	EXPERT	40.9	47.6 ±2.1	24.1 ±2.6	15.0 ±0.6	0.0 ±0.0	0.0	0.5 ±0.5	0.0 ±0.0
④ Mini-SWE-Agent	EXPERT	44.3	53.6 ±2.0	29.8 ±2.5	8.4 ±0.5	0.0 ±0.0	0.0	2.9 ±1.1	0.0 ±0.0
⑤ OpenCode	EXPERT	34.1	40.0 ±2.1	16.8 ±2.6	11.1 ±0.8	0.0 ±0.0	0.0	2.0 ±1.0	0.0 ±0.0
⑥ OpenHands	EXPERT	**52.3**	48.6 ±2.4	31.2 ±2.8	17.1 ±0.9	0.0 ±0.0	0.9	11.0 ±2.4	0.3 ±0.0
⑦ Qwen-Code	EXPERT	26.1	37.2 ±1.9	16.6 ±2.2	9.7 ±0.6	0.0 ±0.0	0.9	3.3 ±1.4	0.3 ±0.6
⑧ Terminus-2	EXPERT	29.5	42.4 ±2.0	17.5 ±2.4	6.2 ±0.4	0.0 ±0.0	0.9	1.6 ±0.9	0.3 ±0.0
Automatically-discovered harnesses *(SoTA evolutionary search vs. MILO)*									
⑨ Best-of-3	HAND	39.8	45.3 ±2.9	23.4 ±2.5	13.0 ±0.7	0.0 ±0.0	0.0	0.8 ±0.6	0.0 ±0.0
⑩ GEPA *(prompt)*	AUTO	39.8	44.2 ±2.5	22.7 ±2.6	14.7 ±1.0	0.0 ±0.0	*0.0*	*0.8 ±0.6*	*0.0 ±0.0*
⑪ GEPA *(optimize anything)*	AUTO	*39.8*	*45.3 ±2.5*	*23.4 ±2.5*	15.5 ±0.7	0.0 ±0.0	*0.0*	*0.8 ±0.6*	*0.0 ±0.0*
⑫ A-Evolve *(skill, memory)*	AUTO	43.2	44.0 ±2.4	25.9 ±2.5	*13.0 ±0.7*	*0.0 ±0.0*	*0.0*	*0.8 ±0.6*	*0.0 ±0.0*
⑬ OpenEvolve	AUTO	39.8	51.1 ±2.2	25.2 ±2.5	13.1 ±1.1	0.0 ±0.0	0.0	0.9 ±0.7	0.0 ±0.0
⑭ ShinkaEvolve	AUTO	44.3	53.5 ±2.1	28.9 ±2.7	12.7 ±0.5	0.0 ±0.0	0.0	0.4 ±0.5	0.0 ±0.0
⑮ EvoX	AUTO	44.3	51.5 ±2.2	29.5 ±2.5	*13.0 ±0.7*	*0.0 ±0.0*	0.0	0.6 ±0.5	0.0 ±0.0
⑯ Meta-Harness	AUTO	40.9	51.0 ±1.9	26.8 ±2.5	17.8 ±1.2	0.0 ±0.0	*0.0*	*0.8 ±0.6*	*0.0 ±0.0*
MILO *(proposed)*	AUTO	**60.2**	**57.1** ±2.0	**34.1** ±3.1	**21.6** ±0.6	0.0 ±0.0	**1.8**	**15.6** ±1.7	**0.6** ±0.8

(a) Performance comparison on TB2.1, PaperBench, and DeepSWE



(b) Our evolution trajectory on TB2.1



(c) Ours vs SoTA evolution on TB2.1

First, MILO *explores more of the harness space than existing methods*. On PaperBench with Opus 4.8 (Tab. 1a), methods that mutate only prompts (GEPA) or skills/memory (A-Evolve) gain at most +6.6% RR@3 over Best-of-3. Existing whole-harness search methods often settle for targeted edits: on TB2.1 with Opus 4.8, the best harnesses of ShinkaEvolve and EvoX differ from the seed only in prompts and rubrics (Lsts. 11, 12), gaining at most +3.6% RR@5 (Tab. 1a). In contrast, MILO's best harness rewrites the control flow (Lst. 14): the agent first produces a minimal valid solution, triages its work as the deadline nears, and stops only after independent verification, gaining +12.0% RR@5. ***Second***, MILO *adapts its search when progress stalls*. With gpt-oss-120b on TB2.1, every prior method makes a few early gains before plateauing (Tab. 2c). MILO stalls as well, with all three islands flat in rounds 6–14 (Tab. 2b). However, inter-island GRAFTs at rounds 15 and 21 and mutator REASSIGNs resume progress, and the weakest seed (39.2%) produces the run's best harness (59.9% at round 27). ***Third***, MILO *raises accuracy while lowering inference cost*. Its TB2.1 harness with Opus 4.8 uses 0.74× the tokens of Best-of-3 while gaining +12.0% RR@5 (Tab. 1b). With gpt-oss-120b, every prior method's best harness costs more than Best-of-3: EvoX spends 1.46× tokens for a +6.1% RR@5 gain, whereas MILO gains +10.7% with 0.70× tokens (Tab. 2c). MILO also withstands a tight output cap of a backbone LLM (Tab. 5). Tab. 1d ablates each component (detailed in App. A.5). Apps. D.1-D.2 list the evolved harnesses.

MILO yields reusable harnesses. A harness evolved on one benchmark should not overfit to it. The generality constraint in $\mathcal{C}$ (Sec. 3) rejects task-specific hard-coding during search, and a manual inspection of the evolved harnesses (App. D.2) found none. As a stronger test, we ran the TB2.1-evolved harness (Lst. 14) unchanged on the harder Frontier-Bench. In RR@3, it beats Best-of-3 by +3.8% and Mini-SWE-Agent, the strongest SoTA harness on TB2.1 by PR@5, by +8.6% (Tab. 3).

MILO advances open mathematical problems. We run the instance-level variant of MILO (Sec. 4.3) on EinsteinArena, evolving harnesses ($\mathcal{M}$ = Opus 5) whose agents write optimizers. We set new records on three problems, surpassing best-known bounds, including those of AlphaEvolve, TTT-Discover, and EvoX (Tab. 4). Each record is confirmed by the arena's official verifier (App. E).

Table 3: **Reusability**. Evaluation of TB2.1-evolved harnesses of MILO on Frontier-Bench, without further search.

Harness	pass@3↑	PR@3↑	RR@3↑
Mini-SWE-Agent	12.9	44.0 ±3.3	5.2 ±2.8
Best-of-3	21.4	48.4 ±2.7	10.0 ±3.4
MILO *(TB2.1-evolved)*	**24.3**	**51.4** ±2.0	**13.8** ±3.2

Table 4: **Scientific discovery: MILO vs. SoTA instance-level evolution.** Evaluation on open mathematical problems from EinsteinArena (↓ = lower is better).

Problem	AlphaEvolve (Georgiev et al., 2025)	TTT-Discover (Yuksekgonul et al., 2026)	EvoX (Liu et al., 2026)	Prior best (live arena leader, Sep '26)	MILO (ours)
Erdős min. overlap ↓	0.380924	0.3808753	–	0.3808586	**0.3808568**
1st autocorrelation ↓	1.5032	1.5028629	–	1.50274365	**1.50274360**
3rd autocorrelation ↓	1.4557	–	1.4558	1.4508066	**1.4488860**

Reproducibility. Apps. A–E give setup, implementation details. Code provided as supplementary.

6 CONCLUSION

We present MILO, a self-adaptive evolutionary framework for automated harness discovery. Its island-partitioned lineage memory preserves diversity and retains rejected mutations to guide structural rewrites by mutator agents. When progress stalls, an orchestrator adapts memory, mutators, and curriculum, evolving the search alongside the harnesses. On Terminal-Bench 2.1, PaperBench, and DeepSWE, MILO shows that structural harness redesign and self-adaptive search yield gains beyond prompt or skill refinement alone. Its harnesses transfer to Frontier-Bench without further search and tighten three best-known bounds on EinsteinArena. These results show that self-adaptive search yields stronger, more efficient, and reusable agents without retraining the underlying model.

DISCLOSURE STATEMENTS

AI USE STATEMENT

LLMs are the object of study in this work: harnesses are the software layer around them. The mutator and orchestrator agents that drive MILO's evolutionary search are themselves LLM-based coding agents (Secs. 4 and 5). Beyond this, LLMs did *not* play a significant role in research ideation or writing. Their use was limited to correcting minor grammatical and typographical errors.

REPRODUCIBILITY STATEMENT

We provide our code as supplementary material: the MILO implementation with its the seed harnesses, the best harness discovered for every benchmark and backbone, the best harness found by each baseline search method, and the evolved optimizers, constructions, and verifier for the open mathematical problems. App. A.1 describes the machine configuration and MILO's configuration used in all experiments; Apps. A–E give the remaining benchmark and evaluation details.

ETHICS STATEMENT

This work involves no human subjects or personal data; all experiments use public benchmarks under their licenses. Like any method that automates agent design, harness search could be applied to objectives beyond the software engineering and scientific discovery tasks studied here. We are not aware of risks specific to this work beyond those associated with the underlying AI models.

SOCIETAL AND ENVIRONMENTAL IMPACT STATEMENTS

Societal impact. This work studies how the harness around a fixed language model can be discovered automatically rather than by hand. Its main benefit is that a given model, frontier or open-weight, becomes more capable on long-horizon tasks without retraining: with gpt-oss-120b, the evolved harness reaches 15.6% on DeepSWE, $20\times$ its initial harness. The discovered harness is a reusable artifact that transfers across tasks and can be inspected or edited like any other program, so its gains are auditable and accessible to anyone who can run the model, not only those able to train.

Environmental impact. MILO has two effects on resource use. The *immediate effect* is that it makes an agent more capable without training a model. Training is the most resource-intensive step: Llama 3.1 405B was trained on up to 16K H100 GPUs (Grattafiori et al., 2024), consuming 30.84M GPU-hours and an estimated 8,930 tons of CO_2eq (Meta AI, 2024). MILO instead treats the model as a black box with inference access only (Sec. 3), so the only resources behind its gains are the inference calls of the search, run from a CPU-only host, a small fraction of what training requires. The *long-term effect* comes from what the search optimizes: MILO admits a harness only if it advances the accuracy–cost frontier, where cost is the tokens the agent processes. On Terminal-Bench 2.1 the evolved harness uses 26% fewer tokens than its starting harness while improving accuracy, and the gains carry over to Frontier-Bench. Because the energy an agent consumes scales with the tokens it processes, this saving lowers the energy use and carbon footprint of every subsequent deployment of the agent. We encourage future work to report token and latency costs alongside accuracy, as we do here, so that capability gains are weighed against their resource use.

Risks. Automated harness discovery can increase the capability and autonomy of agentic systems, which could amplify harm if such systems are deployed in consequential settings without adequate oversight. The harnesses evolved here are selected for accuracy and cost on public benchmarks, so they should not be treated as ready for safety-critical use without further evaluation. The framework, however, is not limited to these objectives: its fitness vector and admission constraints (Sec. 3) accept any measurable property, so safety, robustness, or reliability scores can be added as fitness axes or hard constraints and optimized for directly instead of audited afterwards.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab.

GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *arXiv preprint arXiv:2507.19457*, 2025.

Alibaba. Qwen Code: A Command-Line AI Coding Agent. https://github.com/QwenLM/qwen-code, 2025. Open-source agentic coding tool adapted from Gemini CLI.

Anthropic. Claude Code. https://www.anthropic.com/claude-code, 2025a. Command-line coding agent.

Anthropic. How We Built Our Multi-Agent Research System, 2025b. Anthropic Engineering blog. https://www.anthropic.com/engineering/multi-agent-research-system.

Oussama Ben Sghaier, Hao Li, Bram Adams, and Ahmed E. Hassan. Don't Blame the Large Language Model: How Agent Harness Evolution Shapes Coding Agent Quality. *arXiv preprint arXiv:2607.03691*, 2026.

Federico Bianchi, Yongchan Kwon, Aneesh Pappu, and James Zou. Harnessing the Collective Intelligence of AI Agents in the Wild for New Discoveries. *arXiv preprint arXiv:2606.10402*, 2026. Einstein Arena, https://einsteinarena.com.

Block. Goose: An Open-Source, Extensible AI Agent. https://github.com/block/goose, 2025. Open-source AI agent (desktop app, CLI, and API).

Yifu Cai, Xinyu Li, Mononito Goswami, Michał Wiliński, Gus Welter, and Artur Dubrawski. TimeSeriesGym: A Scalable Benchmark for (Time Series) Machine Learning Engineering Agents. *arXiv:2505.13291*, 2025.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. AdaEvolve: Adaptive LLM-Driven Zeroth-Order Optimization. *arXiv:2602.20133*, 2026.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. MLEbench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *International Conference on Learning Representations (ICLR)*, 2025.

Mingguang Chen, Licheng Wang, and Bo Qu. Recursive Self-Improvement in AI: From Bounded Self-Refinement to Autonomous Research Loops. *arXiv preprint arXiv:2607.07663*, 2026.

ClineBot. Cline: Autonomous Coding Agent. https://github.com/cline/cline, 2025. Open-source coding agent (IDE extension, CLI, and SDK).

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems. *arXiv:2110.14168*, 2021.

Cognition. What We Learned Building Cloud Agents. Cognition blog, 2026. https://cognition.com/blog/what-we-learned-building-cloud-agents.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. HSEvo: Elevating Automatic Heuristic Design with Diversity-Driven Harmony Search and Genetic Algorithm Using LLMs. *arXiv:2412.14995*, 2024.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, et al. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Ágoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical Exploration and Discovery at Scale. *arXiv preprint arXiv:2511.02864*, 2025.

GitHub. Introducing GitHub Copilot: Your AI Pair Programmer. https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/, 2021.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*, 2024.

Tu Hu, Ronghao Chen, Shuo Zhang, Jianghao Yin, Mou Xiao Feng, Jingping Liu, Shaolei Zhang, Wenqi Jiang, Yuqi Fang, Sen Hu, Huacan Wang, and Yi Xu. Controlled Self-Evolution for Algorithmic Code Optimization. *arXiv:2601.07348*, 2026.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring Frontier Coding Agents on Original, Long-Horizon Engineering Tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution. In *27th European Conference on Artificial Intelligence (ECAI)*, pp. 4011–4018. IOS Press, 2024.

Prithwish Jana, Sam Davidson, Bhavana Bhasker, Andrey Kan, Anoop Deoras, and Laurent Callot. TerraFormer: Automated Infrastructure-as-Code with LLMs Fine-Tuned via Policy-Guided Verifier Feedback. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, pp. 578–589, 2026a.

Prithwish Jana, Kaan Kale, Ahmet Ege Tanriverdi, Cruise Song, Sriram Vishwanath, and Vijay Ganesh. ProofBridge: Auto-Formalization of Natural Language Proofs in Lean via Joint Embeddings. In *14th International Conference on Learning Representations (ICLR)*, 2026b.

Piyush Jha, Prithwish Jana, Pranavkrishna Suresh, Arnav Arora, and Vijay Ganesh. RLSF: Fine-Tuning LLMs via Symbolic Feedback. In *28th European Conference on Artificial Intelligence (ECAI)*, pp. 1687–1694. IOS Press, 2025.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-Driven Exploration in the Space of Code. *arXiv:2502.13138*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.

Krafton-AI. Terminus-KIRA: Boosting Frontier Model Performance on Terminal-Bench with Minimal Harness, 2026. URL https://github.com/krafton-ai/kira.

LangChain. DeepAgents. https://github.com/langchain-ai/deepagents, 2025.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards Open-Ended and Sample-Efficient Program Evolution. *arXiv:2509.19349*, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution Through Large Models. *arXiv:2206.08896*, 2022. Also as a chapter in *Handbook of Evolutionary Machine Learning*, Springer 2023.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses. *arXiv preprint arXiv:2604.25850*, 2026a.

Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, Xiang Zhang, Suhang Wang, Benoit Dumoulin, and Jian Pei. Position: Agentic Evolution Is the Path to Evolving LLMs. *arXiv preprint arXiv:2602.00359*, 2026b.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. EvoX: Meta-Evolution for Automated Discovery. *arXiv:2602.23413*, 2026.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: Improving LLM Agents by Automatically Synthesizing a Code Harness. *arXiv preprint arXiv:2603.03329*, 2026.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex J. Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. Discovering Preference Optimization Algorithms with and for Large Language Models. *arXiv:2406.08414*, 2024.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. In *ICLR*, 2024.

Ryan Marten, Alexander G. Shaw, and Andy Konwinski. Frontier-Bench: Measuring Agent Abilities at the Frontier. `https://www.frontierbench.ai/`, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Meta AI. Llama 3.1 Model Card. `https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md`, 2024. Accessed 2026-09-23.

Joel Niklaus. Don't Train the Model, Evolve the Harness, 2026. Harvey LAB benchmark. `https://huggingface.co/spaces/joelniklaus/harness-optimization`.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, Lingjie Chen, Yanjun Zhao, Ke Yang, Bingxuan Li, Cheng Qian, Gaotang Li, Xiao Lin, Zhichen Zeng, Ruizhong Qiu, Sirui Chen, et al. Code as Agent Harness, 2026. arXiv:2605.18747.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. *arXiv:2506.13131*, 2025.

OpenAI. OpenAI Codex. `https://openai.com/codex`, 2025.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. CORAL: Towards Autonomous Multi-Agent Evolution for Open-Ended Discovery. *arXiv:2604.01658*, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical Discoveries from Program Search with Large Language Models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I Learned to Start Worrying About Prompt Formatting. In *International Conference on Learning Representations (ICLR)*, 2024.

Asankhaya Sharma. OpenEvolve: Open-Source Evolutionary Coding Agent. https://github.com/codelion/openevolve, 2025.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *NeurIPS*, 2023.

Chenglei Si, Zitong Yang, Yejin Choi, Emmanuel Candès, Diyi Yang, and Tatsunori Hashimoto. Towards Execution-Grounded Automated AI Research. *arXiv:2601.14525*, 2026.

Dhruv Srikanth, Bingchen Zhao, Dixing Xu, Yuxiang Wu, and Zhengyao Jiang. Recursive Self-Improvement of AI Research Agents, 2026. URL https://arxiv.org/abs/2609.26457.

SST. OpenCode. https://github.com/sst/opencode, 2025.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's Ability to Replicate AI Research. *arXiv preprint arXiv:2504.01848*, 2025.

Muxin Tian, Zhe Wang, Blair Yang, Zhenwei Tang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. SWE-Bench Mobile: Can Large Language Model Agents Develop Industry-Level Mobile Applications? In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 8077–8087, 2026. arXiv:2602.09540.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable Code Actions Elicit Better LLM Agents. In *International Conference on Machine Learning (ICML)*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*, 2022.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers, 2023. arXiv:2309.03409.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024a.

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*, 2024b. URL https://arxiv.org/abs/2405.15793.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. *arXiv preprint arXiv:2605.23904*, 2026.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *NeurIPS*, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023b.

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. arXiv:2406.07496.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to Discover at Test Time. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2601.16175.

Guibin Zhang, Kaijie Chen, Guancheng Wan, Heng Chang, Hong Cheng, Kun Wang, Shuyue Hu, and Lei Bai. EvoFlow: Evolving Diverse Agentic Workflows on the Fly. *arXiv preprint arXiv:2502.07373*, 2025a.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that Improve Themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. *arXiv preprint arXiv:2505.22954*, 2025b.

Yifan Zhang, Yutong Dai, Juntao Tan, Luyu Yang, Rishi Mullur, Thai Hoang, Zhiyuan Hu, James Zhu, Phil Mui, Silvio Savarese, Ran Xu, and Zeyuan Chen. DarwinX: Evolving Agent Harnesses Through Natural Selection. *arXiv preprint arXiv:2608.07545*, 2026b.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large Language Models Are Human-Level Prompt Engineers. *arXiv preprint arXiv:2211.01910*, 2022. arXiv:2211.01910.

APPENDIX

A Experimental Setup 16
A.1 Reproducibility and Implementation Details 16
A.2 Benchmark Details 17
A.3 Setup of the State-of-the-Art Evolutionary Search Baselines 17
A.4 Minimal-Harness Baseline 18
A.5 Search-Mechanism Ablation 19
B Initial (Seed) Harnesses 19
B.1 Base DeepAgents Harness 19
B.2 Expert-Designed Harnesses (B1–B12) 20
B.3 The Three Initial Harnesses for Evolutionary Search (B10–B12) 20
C Prompts Used by MILO 26
C.1 Mutator-Agent Prompt 26
C.2 Orchestrator-Agent Prompt 27
D Strategy-Level Discovery of Harnesses 28
D.1 Best Harness Evolved by Each SoTA Evolutionary Search 28
D.2 Best Harness Evolved by MILO 41
E Instance-Level Scientific Discovery on Open Mathematical Problems 45
E.1 Records against the Best Known Bounds 45
E.2 Best Optimizers Discovered by MILO 46

A EXPERIMENTAL SETUP

A.1 REPRODUCIBILITY AND IMPLEMENTATION DETAILS

Machine configuration. All our harness discovery and agent evaluation experiments run on a single CPU-only x86-64 instance with 192 vCPUs (96 physical cores across two sockets, Intel Xeon Sapphire Rapids at 3.4 GHz, two threads per core), 1.5 TiB of memory across two NUMA nodes, and a 1 TB SSD-backed root volume. The instance runs UNIX (kernel 6.1), Docker 25, and Python 3.12, and hosts only the task containers and search loop; all model inference is remote.

Model access. Every language model in this paper is accessed through Amazon Bedrock: the closed-source backbone Opus 4.8, the open-weight backbone gpt-oss-120b, and the frontier models behind the mutator and orchestrator agents. Tools that expect an OpenAI-compatible endpoint, such as the OpenEvolve and EvoX baselines, reach Bedrock through a local LiteLLM proxy that exposes a `/v1` interface on the host. Our harnesses themselves are built on the DeepAgents framework (LangChain, 2025) and accept any LangChain chat model, so they run unchanged against another API provider or a vLLM server on a local GPU.

Task execution. Every task runs in its own Docker container, launched by the Harbor runner, and requires the agent to act in that environment (running commands, editing code, building software), which makes solutions hard to recover from the web or to memorize. The harnesses we evolve are host-side: the agent loop and its model calls run on the instance, and only tool actions enter the container, so a task's network regime (App. A.2) is enforced as the benchmark specifies, including the fully sealed DeepSWE containers. During harness evolution, we run up to 8 concurrent tasks per island. During agent evaluation with the deployed harness, we run 16 concurrent tasks, with no other task running on the machine so as to capture agent wall-clock time consistently.

Choice of the harness framework. We instantiate the harness space $\mathcal{H}$ of Sec. 3 on DeepAgents (LangChain, 2025), whose LangGraph runtime provides stable model invocation, tool dispatch, and state propagation, while everything above it (system prompt, tools, skill and memory stores, middleware, completion and verification gates, and sub-agent roles and topology) lives in one Python file that calls `create_deep_agent` (App. B.1 shows the seed in full). We chose Python over declarative YAML deliberately: code exposes a far larger mutation surface, so the search can rewrite control flow rather than only select among preset options. Nothing in MILO depends on this choice. It requires only a runnable `build_agent` entry point and editable source (Def. 3.1), so it applies equally to YAML-configured agents, to tool and skill repositories, and to other frameworks

such as the OpenHands SDK (Wang et al., 2025) or Mini-SWE-Agent (Yang et al., 2024b), whatever interaction scheme they implement (ReAct, CodeAct, or sub-agent delegation).

MILO configuration. Each mutation and orchestration call is capped at 45 minutes. The search runs $K{=}3$ islands with stall patience $P{=}3$ and selection temperature $T{=}1$. Fitness evaluates $k{=}3$ attempts per task on the search split, with costs (output tokens, agent seconds) recorded against the agent's fixed per-task cap; the final harness of every method is then re-evaluated at the benchmark's leaderboard k. Every automated search method (SoTA, App. A.3, and MILO) in Tab. 2(b,c) is initialized from the same three expert-designed DeepAgents harnesses (B10–B12, App. B.3). MILO is given a 72-hour wall-clock limit as every other search method we compare against (App. A.3).

Mutator pool and orchestrator. The pool $\mathcal{G}$ is identical in every run and holds six coding agents: two off-the-shelf CLIs, Claude Code (Anthropic, 2025a) with Opus 4.8 and Codex (OpenAI, 2025) with GPT-5.5, and four custom agents that pair the DeepAgents harness with Opus 4.8, GPT-5.5, gpt-oss-120b, or Qwen3-Coder-480B, all served through AWS Bedrock. Every island starts with Claude Code as its mutator, which the orchestrator may reassign. The orchestrator is itself a Claude Code agent (with Opus 4.8) that applies at most two moves per intervention. Dominance is pass-primary with tolerance $\varepsilon{=}0.02$: a child whose accuracy exceeds its parent's by more than ε is admitted regardless of cost, one falling short by $> \varepsilon$ is rejected, and within the band the cost axes decide, with token count and agent seconds normalized by fixed per-task limits (20M tokens, 7,200 s).

A.2 Benchmark Details

This section expands the benchmark summary of Sec. 5.1 with the per-benchmark specifics. In every case the stated time limit is a strict cutoff: when it elapses, the attempt is graded in whatever state it has reached, so a harness that leaves partial progress is scored on that partial progress.

Terminal-Bench 2.1 (Merrill et al., 2026) contains 89 tasks from real command-line workflows (system administration, software installation, data wrangling, and security), 85 of which are rated medium or hard. Each task is allotted 15 minutes to 3 hours and scored by hidden tests.

Frontier-Bench (Marten et al., 2026) is the harder successor of Terminal-Bench 2.1, targeting the same distribution with 74 disjoint tasks whose time limits run from 30 minutes to 8 hours. In this paper, we evaluate on the 70 tasks that require no GPU.

PaperBench-CodeDev (Starace et al., 2025) gives the agent 12 hours to reimplement each of 20 ICML 2024 Spotlight and Oral papers. Each paper is scored by a rubric, co-developed with its authors, whose 8,316 binary leaves grade the components of a faithful replication.

DeepSWE (Huang et al., 2026) poses 113 hand-authored tasks across 91 open-source repositories in five languages, each allotted 90 minutes and scored by hidden test suites. Its reference solutions modify $5.5\times$ as many lines as those of SWE-bench Pro (Deng et al., 2025), typically spread across several files, making it the most edit-heavy benchmark of the four.

Internet-access regimes. The benchmarks also differ in what network access a harness may use, which lets us test harnesses under distinct conditions: Terminal-Bench and Frontier-Bench permit full internet access, PaperBench blocks each target paper's own code repository (access is checked after the run, and any violation zeroes the score), and DeepSWE seals the container off entirely.

A.3 Setup of the State-of-the-Art Evolutionary Search Baselines

Each baseline runs its publicly released code, pinned to the version or commit given below, with its own search loop, prompts, and hyperparameters. To make the comparison with MILO fair, every method is given the same 72-hour wall-clock limit, and we change only three things. (a) *Seeds*: every method starts from the same three expert-designed harnesses B10–B12 (App. B.2); a method that accepts a single seed takes *Best-of-3*. (b) *Evaluator*: every candidate is scored by the benchmark's official test cases or rubric through one shared scorer. (c) *Mutator*: wherever a method calls a single LLM to propose edits, that LLM is Opus 4.8; methods whose search relies on several models keep that machinery, anchored on Opus 4.8; Meta-Harness, whose proposer is a coding agent, uses Claude Code (with Opus 4.8). The per-method settings are as follows:

OpenEvolve (Sharma, 2025) (v0.3.2): three islands with MAP-Elites archives and ring migration. Its migration interval is shortened from 50 to 5 generations so that migration fires within our round

budget, and mutations use its full-rewrite operator because its diff anchors fail on harness-sized files. Its native two-model ensemble is kept, weighted to Opus 4.8 (0.7) and GPT-5.5 (0.3).

ShinkaEvolve (Lange et al., 2025) (v0.0.7): three islands with migration, and its native four-arm UCB bandit over Opus 4.8, GPT-5.5, GPT-5.4, and Sonnet 4.5, the closest four-model pool available on Bedrock. Novelty rejection stays on, with Amazon Titan embeddings in place of OpenAI's.

GEPA (Agrawal et al., 2025) (v0.1.1): both native APIs, `optimize`, which edits the system prompt only, and `optimize_anything`, which edits the whole harness file, each with an Opus 4.8 reflection LLM and GEPA's own Pareto-frontier memory. With gpt-oss-120b, the reflection minibatch is raised from GEPA's default of 3 tasks to 12: on this backbone a 3-task minibatch is all-fail most of the time, so GEPA's acceptance gate never fires and the search cannot start.

EvoX (Liu et al., 2026) (commit `4734f32`): single-population loop with strategy meta-evolution; one Opus 4.8 model serves mutation, strategy rewrites, and guidance (`share_llm`).

A-Evolve (Lin et al., 2026b) (commit `c9d4789`): evolves skill and memory files with the code frozen and, as in its Terminal-Bench recipe, shows its Opus 4.8 evolver only agent trajectories.

Meta-Harness (Lee et al., 2026) (commit `44b9942`): we run its released loop and proposer invocation unchanged. The proposer is Claude Code with Opus 4.8, its proposer skill is retargeted from a Terminus-2 subclass to our DeepAgents `build_agent` genome, and its optimization frontier keeps the best harness after each proposal.

A.4 Minimal-Harness Baseline

The *Minimal harness* rows of Tabs. 1 and 2 run the backbone model inside the smallest possible agent loop. The model receives the benchmark's task instruction as a single user message, with no system prompt; the loop invokes the model, executes its tool calls, appends the results, and repeats until the model stops calling tools. A cap of 100 steps exists only to bound a runaway loop. There are no planning tools, sub-agents, skills, memory, middleware, or context summarization.

Design philosophy. The tool set is the smallest that lets the model perform what the benchmarks of Sec. 5.1 grade. Every task provides files to read (code, data, or a paper) and is graded on files the agent produces, which requires `read_file` and `write_file`, and every benchmark grades *executed* state (Terminal-Bench and Frontier-Bench tasks change the live container, PaperBench replications must run, and DeepSWE grades the committed diff `git diff base..HEAD`), which requires `bash`. Nothing here shapes *how* the model works. Listing 1 gives the source.

Listing 1 · The minimal-harness baseline (`minimal_harness.py`)

The minimal-harness baseline (`minimal_harness.py`): a bare tool-calling loop over the backbone model with only `read_file`, `write_file`, and `bash`. It exposes the same `build_agent(model, backend)` entry point as every harness; `[...]` marks elided error handling.

```python
from langchain_core.messages import ToolMessage
from langchain_core.tools import tool

def _make_tools(backend):
    """The three tools, closed over the benchmark sandbox backend."""

    @tool
    async def read_file(file_path: str, offset: int = 0, limit: int = 2000) -> str:
        """Read a file from the environment. Returns up to `limit` lines
        starting after line `offset` (use offset to page through long files)."""
        r = await backend.aread(file_path, offset=offset, limit=limit)
        if r.error:
            return f"Error: {r.error}"
        return (r.file_data or {}).get("content", "")  # [...] 80K-char cap

    @tool
    async def write_file(file_path: str, content: str) -> str:
        """Write `content` to a file in the environment, creating it (and
        overwriting any existing file) at `file_path`."""
        w = await backend.awrite(file_path, content)
        return f"Error: {w.error}" if w.error else f"Wrote {w.path or file_path}"

    @tool
    async def bash(command: str, timeout: int = 120) -> str:
        """Run a bash command in the environment and return its output
        (stdout+stderr). Use `timeout` (seconds) for long-running commands."""
        r = await backend.aexecute(command, timeout=timeout)
        return r.output or ""  # [...] output cap + nonzero-exit annotation

    return [read_file, write_file, bash]

class ModelOnlyAgent:
    """Bare model + read/write tools; mimics the harness ainvoke contract."""

    def __init__(self, model, backend):
```

```
        self._tools = {t.name: t for t in _make_tools(backend)}
        self._model = model.bind_tools(list(self._tools.values()))
        self._max_steps = 100  # runaway guard only, not scaffolding

    async def ainvoke(self, payload: dict) -> dict:
        messages = list(payload["messages"])  # the task instruction, verbatim
        for _ in range(self._max_steps):
            ai = await self._model.ainvoke(messages)
            messages.append(ai)
            calls = list(getattr(ai, "tool_calls", None) or [])
            if not calls:
                break  # the model decided it is done
            for tc in calls:
                messages.append(ToolMessage(
                    content=await self._run_tool(tc),  # [...] error guards
                    tool_call_id=tc.get("id") or "",
                    name=tc.get("name") or "",
                ))
        return {"messages": messages}

def build_agent(model, backend):
    return ModelOnlyAgent(model, backend)
```

A.5 Search-Mechanism Ablation

Tab. 1d ablates MILO's search machinery along an additive ladder: each configuration turns on one more mechanism than the one to its left, so the difference between neighbouring columns isolates that mechanism. (A) *LLM mutator over flat memory*: a single population without an orchestrator, in which one LLM call proposes each child from the current best over an unstructured archive. (B) +*mutator agent*: the LLM mutator is replaced by an agentic mutator (App. C.1) that inspects the parent's failure evidence and rewrites the harness over multiple tool-calling steps. (C) +*lineage memory*: the flat archive is replaced by the hierarchical lineage memory, so each edit is conditioned on its parent lineage. (D) +*multiple islands*: the population is partitioned into islands that evolve competing strategies in parallel. (E) +*orchestrator*: the full system of Sec. 4.2. Every configuration seeds from B10–B12, fixes the backbone to Opus 4.8, and searches Terminal-Bench 2.1. RR@5 rises at every step, $76.4 \rightarrow 79.1 \rightarrow 79.3 \rightarrow 80.5 \rightarrow 86.1$, with the largest gain from the orchestrator.

Output-cap robustness. Under a 4K per-call output cap (Tab. 5), Best-of-3 achieves only 34.4% RR@5 while MILO still evolves an 81.1% harness.

B Initial (Seed) Harnesses

B.1 Base DeepAgents Harness

Listing 2 shows the stock DeepAgents harness with nothing added: it runs on the framework's built-in defaults and therefore measures out-of-the-box behavior. The agent loop itself lives inside LangGraph and is *not* part of the harness. What the file exposes are the configuration surfaces a harness may edit (system prompt, extra tools, sub-agents, skills, and long-term memory), which `build_agent` assembles into a runnable agent.

Listing 2 · The base DeepAgents harness

The base DeepAgents harness. Every editable surface (`SYSTEM_PROMPT`, `EXTRA_TOOLS`, `SUBAGENTS`, `SKILLS`, `MEMORY`) is left at its built-in default.

```
from __future__ import annotations
from typing import Any
from deepagents import create_deep_agent

# ---- editable harness surfaces (the mutable surface the search rewrites) ----

# Front-of-prompt instructions. None => use deepagents' own BASE_AGENT_PROMPT
# alone (plus any per-model profile suffix): the stock, unmodified baseline.
SYSTEM_PROMPT: str | None = None

# Extra tools beyond deepagents' built-ins (write_todos, file ops, execute, task).
# Each entry is a LangChain BaseTool / callable / tool dict. Empty => built-ins only.
EXTRA_TOOLS: list[Any] = []

# Sub-agents to delegate to (isolated context windows). Empty => none.
SUBAGENTS: list[Any] = []

# Skill source paths (each a folder with SKILL.md + optional scripts/resources).
# deepagents loads metadata up front and the body on demand. Empty => none.
SKILLS: list[str] = []

# Long-term memory source paths (markdown preloaded as memory). Empty => none.
MEMORY: list[str] = []


# ---- factory: assemble the harness around an injected model and backend -----
```

```python
def build_agent(model, backend):
    """Assemble the deepagents harness around an injected model and backend.

    model:   a provider model id ("bedrock/...", "anthropic:...") or a
             pre-initialized LangChain BaseChatModel.
    backend: the execution backend (e.g. LocalShellBackend) enabling the
             'execute' shell tool and file operations.

    Returns a compiled LangGraph agent, invoked with:
        agent.invoke({"messages": [{"role": "user", "content": task}]})
    """
    return create_deep_agent(
        model=model,
        tools=EXTRA_TOOLS,
        system_prompt=SYSTEM_PROMPT,       subagents=SUBAGENTS,
        skills=SKILLS or None,
        memory=MEMORY or None,
        backend=backend,
    )
```

B.2 Expert-Designed Harnesses (B1–B12)

Starting from the base harness (Listing 2), we asked practitioners to design harnesses of increasing sophistication. Each of the resulting twelve, B1–B12, adds one component to the base harness. Tabs. 5 and 6 report their accuracy on Terminal-Bench 2.1 with Opus 4.8 and gpt-oss-120b as the backbone, together with the harness MILO discovers in each of the three settings: Opus 4.8 under a 4K output cap per LLM call, Opus 4.8 under a 128K cap, and gpt-oss-120b under a 128K cap.

The 4K setting is the most telling. Capping every model call at 4K output tokens is an artificial handicap, and it hurts all twelve expert harnesses badly: their RR@5 falls from 70.5–75.0 at 128K to 20.2–34.4 at 4K. The harness MILO evolves under the same cap reaches 81.1 RR@5, within 5.0% of the 86.1 it reaches at 128K, and matches its pass@5 of 93.2 exactly. A stronger harness thus compensates for the weakness of the model, even one imposed artificially.

Table 5: **Performance of expert-designed harnesses vs. MILO on Terminal-Bench 2.1, Opus 4.8** (k=5). Each id B1–B12 adds one component to base harness; we compare truncating 4K and non-truncating 128K output caps. **Bold**/underline = best/second per column.

Harness	Design	Opus-4.8 (cap 4K)			Opus-4.8 (cap 128K)		
		pass@5↑	PR@5↑	RR@5↑	pass@5↑	PR@5↑	RR@5↑
Baseline							
Stock DeepAgents	HAND	33.7	30.1 ±2.0	21.1 ±2.2	83.0	76.8 ±2.2	70.9 ±2.6
Expert-engineered harnesses							
① system prompt	HAND	28.1	30.4 ±1.7	20.2 ±1.8	87.5	78.1 ±2.1	72.0 ±2.7
② filesystem-write permissions	HAND	32.6	30.1 ±2.0	21.3 ±2.2	89.8	79.1 ±2.2	73.9 ±2.8
③ context-clamp middleware	HAND	33.7	30.9 ±2.0	21.8 ±2.1	87.5	79.6 ±2.0	73.9 ±2.6
④ code-exec tool	HAND	34.8	30.7 ±1.9	21.8 ±2.2	89.8	78.5 ±2.3	73.4 ±2.8
⑤ edit tool & skills	HAND	30.3	31.5 ±1.9	22.0 ±1.9	85.2	79.8 ±1.9	75.0 ±2.4
⑥ windowed editor & search	HAND	34.8	31.7 ±1.8	22.9 ±2.2	85.2	79.4 ±2.0	73.0 ±2.5
⑦ skills library	HAND	32.6	32.2 ±1.7	22.9 ±1.8	87.5	79.2 ±2.1	73.6 ±2.6
⑧ self-reflection memory	HAND	33.7	32.4 ±2.0	23.1 ±2.1	85.2	77.7 ±1.8	73.0 ±2.3
⑨ deadline awareness	HAND	38.2	33.1 ±2.0	24.3 ±2.3	86.4	80.1 ±2.1	74.8 ±2.5
⑩ sub-agent roles	HAND	46.1	36.7 ±2.2	28.1 ±2.4	83.0	75.8 ±2.1	70.5 ±2.6
⑪ plan-and-solve gate	HAND	46.1	38.9 ±2.1	30.8 ±2.5	87.5	76.0 ±2.3	70.5 ±2.7
⑫ reproduce-first rubric gate	HAND	49.4	43.6 ±2.0	34.4 ±2.4	87.5	79.1 ±2.0	74.1 ±2.4
Automatically-discovered harness							
MILO (init ⑩⑪⑫)	AUTO	**93.2**	**87.4** ±1.5	**81.1** ±2.2	**93.2**	**90.4** ±1.3	**86.1** ±2.0

Table 6: **Performance of expert-designed harnesses vs. MILO on Terminal-Bench 2.1, gpt-oss-120b** (128K cap, k=5). Rows and formatting as in Tab. 5.

Harness	Design	gpt-oss-120b (cap 128K)		
		pass@5↑	PR@5↑	RR@5↑
Baseline				
Stock DeepAgents	HAND	29.5	36.8 ±2.3	15.5 ±2.4
Expert-engineered harnesses				
① system prompt	HAND	35.2	42.1 ±2.5	21.1 ±2.5
② filesystem-write permissions	HAND	29.5	40.2 ±2.2	17.3 ±2.3
③ context-clamp middleware	HAND	40.9	44.4 ±2.4	22.5 ±2.6
④ code-exec tool	HAND	34.1	43.0 ±2.2	22.3 ±2.2
⑤ edit tool & skills	HAND	38.6	47.1 ±2.2	24.5 ±2.4
⑥ windowed editor & search	HAND	42.0	43.5 ±2.4	22.3 ±2.7
⑦ skills library	HAND	36.4	42.7 ±2.5	20.9 ±2.5
⑧ self-reflection memory	HAND	35.2	41.4 ±2.2	19.5 ±2.5
⑨ deadline awareness	HAND	36.4	43.4 ±2.0	20.5 ±2.4
⑩ sub-agent roles	HAND	36.4	43.6 ±2.3	20.9 ±2.5
⑪ plan-and-solve gate	HAND	39.8	45.3 ±2.5	23.4 ±2.5
⑫ reproduce-first rubric gate	HAND	36.4	43.2 ±2.5	20.9 ±2.5
Automatically-discovered harness				
MILO (init ⑩⑪⑫)	AUTO	**60.2**	**57.1** ±2.0	**34.1** ±3.1

B.3 The Three Initial Harnesses for Evolutionary Search (B10–B12)

The three most mature designs in Tabs. 5 and 6, B10 (sub-agent roles), B11 (plan-and-solve gate), and B12 (reproduce-first rubric gate), are the shared seeds from which every evolutionary search in this paper starts. MILO seeds one island with each. A method that accepts a single seed takes *Best-of-3*, the strongest of the three for that benchmark and backbone by resolution rate, with pass rate as the tiebreak. We present the source code of harnesses B10-B12 below.

Listing 3 · Seed B10: sub-agent roles **RR@5 = 70.5**

Seed B10 (`deepagents_orchestrator.py`): a lead agent delegates to three sub-agents with isolated contexts and cannot stop until the `verifier` sub-agent has been dispatched. Full source.

```
"""
deepagents_orchestrator.py -- philosophy: DIVISION OF LABOR.

Thesis: a single agent juggling exploration, implementation, and verification in one
context does all three worse. Splitting the work across focused subagents -- each with
its own isolated context and instructions -- keeps the lead agent's context clean (it
sees conclusions, not the noise of every sub-investigation) and lets each role be good
at one thing. This is the multi-agent strategy that several top leaderboard harnesses use.

Real logic (not prose): three real subagents with distinct system prompts and roles,
wired via deepagents' `subagents=` (each runs in its own context window and reports back
through the `task` tool). The verifier subagent independently inspects the final state --
a structural check the lead agent cannot fake by asserting success. A CompletionGate
keeps the lead from stopping before it has delegated that verification.

This file is fully self-contained -- the gate middleware is defined inline, nothing is
imported from a sibling module -- so the evolutionary loop can mutate it as one unit.

Mutable surfaces: the roster of subagents, each subagent's prompt/tools/model, and the
orchestrator prompt. The evolutionary loop can add/remove roles or re-scope them.
"""

from __future__ import annotations

from typing import Any

from langchain.agents.middleware import AgentMiddleware
from langchain.agents.middleware.types import hook_config
from langchain_core.messages import AIMessage
from deepagents import create_deep_agent


class CompletionGateMiddleware(AgentMiddleware):
    """Refuse to let the lead agent stop until it has ACTUALLY delegated verification.

    A naive gate that nudges once and then allows the next stop is hollow: the model can
    ignore the nudge and stop unverified on its immediately-following turn. This gate is
    EVIDENCE-BASED. When the model tries to end (an AIMessage with no tool calls), it
    checks whether the `verifier` subagent has been dispatched since the last gate. If not,
    it re-injects the directive and jumps back to the model -- repeatedly, up to a hard
    `max_gates` ceiling (so a genuinely-stuck run can't loop forever). The gate is only
    "satisfied" once verifier evidence is present in the transcript.
    """

    def __init__(self, max_gates: int = 3, verifier_name: str = "verifier") -> None:
        super().__init__()
        self.max_gates = max_gates            # hard ceiling on re-prompts (anti-infinite-loop)
        self.verifier_name = verifier_name
        self._gated = 0
        self._last_seen_msgs = 0              # message count at the last gate (to scope "since")

    @hook_config(can_jump_to=["model"])
    def after_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._gate(state)

    @hook_config(can_jump_to=["model"])
    async def aafter_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._gate(state)

    def _verifier_dispatched(self, msgs) -> bool:
        """True if the `task` tool was called with the verifier subagent anywhere in the
        transcript (a structural signal the lead actually delegated verification)."""
        for m in msgs:
            for tc in (getattr(m, "tool_calls", None) or []):
                args = tc.get("args", {}) if isinstance(tc, dict) else {}
                blob = (str(tc.get("name", "")) + " " + str(args)).lower()
                if self.verifier_name in blob:
                    return True
        return False

    def _gate(self, state) -> dict[str, Any] | None:
        msgs = state.get("messages") or []
        if not msgs:
            return None
        last = msgs[-1]
        if not isinstance(last, AIMessage) or getattr(last, "tool_calls", None):
            return None
        # Satisfied: the verifier was actually dispatched at some point -> allow the stop.
        if self._verifier_dispatched(msgs):
            return None
        if self._gated >= self.max_gates:
            return None  # ceiling reached; don't loop forever on an uncooperative model
        self._gated += 1
        directive = (
            "You have NOT yet had the work independently verified. Before you finish, you "
            "MUST dispatch the `verifier` subagent (via the `task` tool) to confirm the "
            "task's end state, and act on its verdict. Do not stop until the verifier has "
            "actually run and reports the task is satisfied."
        )
        return {"messages": [{"role": "user", "content": directive}], "jump_to": "model"}

ORCHESTRATOR_PROMPT = """\
You are the lead agent on a sandboxed terminal task (work in /app unless told otherwise),
coordinating specialist subagents via the `task` tool. The task is judged by the final
state of the environment.

Keep YOUR context for the plan; push detail to subagents:
- First, understand the task and extract the concrete required end state (exact files,
  outputs, services). Lay out the steps with `write_todos` so you don't silently skip one
  -- a missing step is the most common failure. Give each subagent a SPECIFIC, well-scoped
  instruction (objective, what to report back, boundaries); vague delegation causes
  duplicated or missed work.
- Use the `explorer` subagent to investigate the environment and report findings
  (structure, relevant files, how things are wired) -- so you don't fill your context
  with raw exploration output.
- Do the core implementation yourself, or hand well-scoped pieces to the `coder` subagent.
```

```
- Before you finish, ALWAYS dispatch the `verifier` subagent to independently confirm the
  task's end state. Trust its verdict over your own assumptions; if it reports the task
  is not satisfied, fix the issue and verify again.
"""

EXPLORER_PROMPT = """\
You are an exploration specialist. Investigate the environment to answer the lead agent's
question: inspect directory structure, read the relevant files, and determine how the
system is wired. Run read-only commands. Report a concise, concrete summary of what you
found (paths, key contents, configurations, anything that affects how the task is solved).
Do not make changes -- your job is to inform, not to act.
"""

CODER_PROMPT = """\
You are an implementation specialist. Carry out the specific, well-scoped change the lead
agent asked for: write/edit the files or run the commands needed, then confirm your change
applied cleanly. Stay within the scope you were given; report what you changed and any
issue you hit. Do not redesign the overall approach -- that's the lead agent's job.
"""

VERIFIER_PROMPT = """\
You are an independent verification specialist. Do NOT take the lead agent's word that the
task is done. Inspect the real end state yourself: read the produced files back, run the
test suite, hit the running service, check exit codes. Report a clear verdict -- SATISFIED
or NOT SATISFIED -- with the concrete evidence you observed. If not satisfied, say exactly
what is wrong.
"""


def _subagents():
    # tools omitted on each => they inherit the lead agent's built-in shell/file tools,
    # so every subagent can actually inspect and act in the container.
    return [
        {"name": "explorer", "description": "Investigate the environment and report findings (read-only).", "system_prompt": EXPLORER_PROMPT},
        {"name": "coder", "description": "Implement a specific, well-scoped change.", "system_prompt": CODER_PROMPT},
        {"name": "verifier", "description": "Independently verify the task's final state.", "system_prompt": VERIFIER_PROMPT},
    ]


def build_agent(model, backend):
    return create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=ORCHESTRATOR_PROMPT,
        subagents=_subagents(),
        middleware=[CompletionGateMiddleware(max_gates=1)],
    )
```

Listing 4 · Seed B11: plan-and-solve gate **RR@5 = 70.5**

Seed B11 (`deepagents_planner.py`): plan-and-solve discipline, with a plan nudge before the first action and a plan-completion review at stop. The island that produced MILO's best harness (Listing 14) was founded on this seed. Full source.

```
"""
deepagents_planner.py -- philosophy: PLAN FIRST, THEN EXECUTE (Plan-and-Solve).

Thesis (Wang et al., "Plan-and-Solve Prompting", ACL 2023, arXiv:2305.04091): the most
common failure on multi-step tasks is the *missing-step* error -- the model dives into
execution and silently skips a required step. Forcing it to FIRST extract the concrete
requirements and devise an explicit plan, THEN carry the plan out step by step, measurably
cuts both missing-step and calculation errors (their Table 6: missing-step 12%->7%,
calculation 7%->5% vs. plain chain-of-thought; +6.3 avg accuracy across arithmetic
benchmarks). The mechanism is almost free: it is a prompt discipline, here reinforced
with deepagents' real planning surface (the `write_todos` tool) and a single re-plan gate.

Why this fits a terminal agent: TB2 tasks are exactly multi-step ("set up X, configure Y,
then make 2 pass"). Skipping the configure step quietly is the classic missing-step error.

Real logic (not prose):
  - SYSTEM_PROMPT bakes in the PS+ discipline ("understand the requirements, extract the
    concrete inputs/constraints, write a complete plan, THEN execute it step by step,
    checking intermediate results"), wired to `write_todos` so the plan is a living
    artifact the agent maintains, not a one-off paragraph that scrolls away.
  - PlanGateMiddleware (inline, concurrency-safe -- no globals): on the FIRST turn it
    nudges the model to lay down a todo plan before acting; and when the model tries to
    finish, it forces ONE pass back over the plan to confirm every step is actually done
    (catching the silently-skipped step), then lets it stop.

Mutable surfaces: the PS+ prompt wording, whether the first-turn plan nudge fires, and the
re-plan/verify gate count. Invoke with the standard messages payload.
"""

from __future__ import annotations

from typing import Any

from langchain.agents.middleware import AgentMiddleware
from langchain.agents.middleware.types import hook_config
from langchain_core.messages import AIMessage
from deepagents import create_deep_agent


class PlanGateMiddleware(AgentMiddleware):
    """Nudge a plan up front, and force one plan-completion review before stopping.

    Two cheap interventions, both bounded so a finished run is never looped forever:
      - before the first model call, inject a one-time directive to write the todo plan
        BEFORE touching the environment (the Plan-and-Solve "devise a plan first" step);
      - when the model tries to end, inject a one-time directive to walk the plan and
        confirm each step's end state with a concrete check (catch the missing step),
        then jump back to the model.
    """

    def __init__(self, plan_nudge: bool = True, max_gates: int = 1) -> None:
        super().__init__()
        self.plan_nudge = plan_nudge
        self.max_gates = max_gates
        self._nudged = False
```

```
        self._gated = 0

    def before_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._maybe_plan(state)

    async def abefore_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._maybe_plan(state)

    def _maybe_plan(self, state) -> dict[str, Any] | None:
        if not self.plan_nudge or self._nudged:
            return None
        msgs = state.get("messages") or []
        # Only fire at the very start (before the agent has taken any action).
        if any(isinstance(m, AIMessage) for m in msgs):
            self._nudged = True
            return None
        self._nudged = True
        directive = (
            "Before running anything: understand the task, extract the concrete "
            "requirements (exact files, inputs, constraints, and the verifiable end "
            "state), and use `write_todos` to lay down a complete step-by-step plan. "
            "Then carry out the plan one step at a time, checking each step's result "
            "before moving on."
        )
        return {"messages": [{"role": "user", "content": directive}]}

    @hook_config(can_jump_to=["model"])
    def after_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._gate(state)

    @hook_config(can_jump_to=["model"])
    async def aafter_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._gate(state)

    def _gate(self, state) -> dict[str, Any] | None:
        msgs = state.get("messages") or []
        if not msgs:
            return None
        last = msgs[-1]
        if not isinstance(last, AIMessage) or getattr(last, "tool_calls", None):
            return None
        if self._gated >= self.max_gates:
            return None
        self._gated += 1
        directive = (
            "Before you finish: go back through your plan (the todos) and, for EACH step, "
            "confirm with a concrete command that it is actually done and correct -- not "
            "that you intended to do it. Pay special attention to any step you might have "
            "skipped. If anything is incomplete or wrong, fix it. Only stop once every "
            "step is verified against the real environment."
        )
        return {"messages": [{"role": "user", "content": directive}], "jump_to": "model"}


SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment.

Work in two phases, the Plan-and-Solve way:

1. UNDERSTAND & PLAN. First read the task carefully and extract the concrete
   requirements: which exact files must exist or change, what inputs/parameters and
   constraints apply, and what the verifiable end state is. Inspect the working directory
   once. Then write a complete, ordered plan with `write_todos` -- one todo per required
   step. A missing step is the most common way these tasks fail, so make the plan
   exhaustive.

2. EXECUTE THE PLAN. Carry out the plan one step at a time. After each step, check the
   intermediate result with a concrete command before moving to the next; keep the todo
   list updated. If reality contradicts the plan, revise the plan rather than pushing on.

Before finishing, walk the whole plan again and verify each step's end state for real
(run the test, read the file back, hit the service). Do not rely on your assumption that
a step worked.
"""


def build_agent(model, backend):
    return create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[PlanGateMiddleware(plan_nudge=True, max_gates=1)],
    )
```

Listing 5 · Seed B12: reproduce-first rubric gate **RR@5 = 74.1**

Seed B12 (`deepagents_scientist.py`): the reproduce-first rubric gate. Each time the agent tries to finish, deepagents' built-in `RubricMiddleware` has a grader sub-agent re-read the transcript against a rubric, which the `_RubricAgent` wrapper injects on every call. The strongest seed under Opus 4.8 on Terminal-Bench 2.1 and the Best-of-3 seed for DeepSWE. Full source.

```
"""
deepagents_scientist.py -- philosophy: REPRODUCE FIRST, TEST-DRIVEN, EVIDENCE-GATED.

Thesis: the agent that wins debugging/fixing tasks does not start by editing. It first
REPRODUCES the failure (turns the bug into a concrete failing command), then forms a
hypothesis, makes the smallest change, and re-runs the SAME command to confirm -- the
scientific method applied to terminal tasks. This is the discipline behind SWE-agent's
reproduce-script practice and Terminus/KIRA's verify-before-submit, made the spine of the
loop instead of an afterthought. It directly attacks the most expensive failure mode in
our own results: stopping at a plausible-but-wrong state (the "all-green by assertion"
trap), and the timeouts caused by editing blindly and thrashing.

Mechanism -- two reinforcing layers:
  1. SYSTEM_PROMPT: a strict reproduce -> hypothesize -> minimal-change -> re-verify
     protocol, plus "write or find a check that fails now and must pass at the end."
  2. RubricMiddleware (deepagents' built-in evaluator-optimizer loop): a grader subagent
     checks the transcript against an explicit "what done looks like" rubric each time
     the agent would finish, and bounces it back with specifics if a criterion is unmet.
     This is Anthropic's evaluator-optimizer pattern as native middleware -- an
     independent reviewer the lead cannot satisfy by merely asserting success. The rubric
     is injected per-invocation, so build_agent wraps invoke to attach it automatically.

deepagents' RubricMiddleware only activates when a `rubric` is present in the invocation
state; to keep the standard `build_agent(model, backend)` + `agent.ainvoke({"messages":
...})` contract, we return a thin wrapper whose `ainvoke`/`invoke` inject the default
rubric if the caller didn't supply one. Grader uses the same injected model (no globals).

Mutable surfaces: the rubric criteria, the grader's max_iterations, and the prompt.
"""

from __future__ import annotations

from deepagents import create_deep_agent
from deepagents.middleware.rubric import RubricMiddleware


# The default "what done looks like" rubric. RubricMiddleware grades the transcript
# against these each time the agent tries to finish, and continues it with specifics if
# any criterion is unmet. Phrased as end-state checks, not intentions.
DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The failure or requirement was reproduced or pinned down with a concrete command
   before any fix was attempted (a failing test, a reproducing command, or an explicit
   inspection of the current state).
2. The change made is minimal and targeted at the identified cause -- no unrelated files
   or configuration were altered as a side effect.
3. The required end state was verified by RE-RUNNING the same concrete check that
   defined success (the test now passes / the command now succeeds / the file now has the
   required content / the service actually responds), and its successful output is shown.
4. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
"""


SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
   in one sentence -- before editing. Do not edit on a hunch.
3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
   what the task requires; leave the rest of the system identical (no stray files or
   config edits).
4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
   output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

Treat errors as evidence, not noise. Never declare success on assumption; success is the
re-run check passing in front of you.
"""


class _RubricAgent:
    """Thin wrapper that injects the default rubric into invocations.

    RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
    preserves the seed's standard call contract by adding the rubric when the caller
    omitted it, while still allowing a caller (or the evolutionary loop) to override it.
    """

    def __init__(self, agent, rubric: str) -> None:
        self._agent = agent
        self._rubric = rubric

    def _with_rubric(self, payload):
        if isinstance(payload, dict) and "rubric" not in payload:
            payload = {**payload, "rubric": self._rubric}
        return payload

    async def ainvoke(self, payload, *args, **kwargs):
        return await self._agent.ainvoke(self._with_rubric(payload), *args, **kwargs)

    def invoke(self, payload, *args, **kwargs):
        return self._agent.invoke(self._with_rubric(payload), *args, **kwargs)

    def __getattr__(self, name):
        # Delegate everything else (get_state, stream, etc.) to the compiled graph.
        return getattr(self._agent, name)
```

```python
# The grader re-reads the transcript and reasons each time the agent would finish, so it
# is the seed's main cost driver: with max_iterations=3 a hard task can pay up to 3 extra
# full-model reasoning passes right at the end -- exactly when wall-clock is scarcest and a
# timeout auto-zeros the trial. We cap it at 2 (one re-grade after a fix). If a lighter
# grader model is available, the evolutionary loop should pass GRADER_MODEL to cut cost
# further; by default the grader inherits the injected solver model.
GRADER_MAX_ITERATIONS = 2
GRADER_MODEL = None  # None -> use the injected solver model


def _thinking_free(model):
    """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
    can't be copied / has no thinking fields.

    Why the grader must not think: the grader is a structured-output classifier (it must emit
    a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
    with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
    the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
    Converse API rejects ("The final block in an assistant message cannot be `thinking`"),
    making the grader fail and silently disabling the rubric gate (the seed's whole point).
    Extended thinking buys nothing for a short verdict classification, so we strip it on the
    grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
    touch neither deepagents nor the shared model builder; the original model is unmutated.
    """
    amrf = getattr(model, "additional_model_request_fields", None)
    if not amrf or "thinking" not in amrf:
        return model
    cleaned = {k: v for k, v in amrf.items() if k not in ("thinking", "output_config")}
    try:
        return model.model_copy(update={"additional_model_request_fields": cleaned})
    except Exception:
        return model


def build_agent(model, backend):
    grader_model = _thinking_free(GRADER_MODEL or model)
    agent = create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
```

C PROMPTS USED BY MILO

The prompts below are reproduced verbatim from the implementation of MILO.

C.1 MUTATOR-AGENT PROMPT

The instruction below is handed to island j's mutator $m_j^{(r)}$ each round, verbatim up to the elisions marked `[...]`. Angle-bracket placeholders are filled per round. The three inputs of Eq. 4 enter as *absolute file paths* rather than inlined content: the source of the parent h_{par}, its failure evidence $\text{EVID}(h_{\text{par}}, \mathcal{B}_{\text{search}}^{(r)})$ (Eq. 1), and the island snapshot of $\mathcal{L}_j^{(r)}$, which lists the source path of every harness in the lineage. The workflow forces diagnosis before editing. The failure evidence plays the role that compiler, symbolic-execution, and verifier feedback play in training code models (Jana et al., 2024; Jha et al., 2025; Jana et al., 2026a). MILO applies that signal at search time to the harness rather than to the model weights.

Listing 6 · The mutation prompt (handed to the mutator agent each round)

The mutation prompt. Angle-bracket placeholders are filled each round; `[...]` marks two elided reference blocks.

```
# Role
You are a harness optimizer in an evolutionary search on Terminal-Bench 2.
Each round you produce ONE improved version of a parent harness. Objective:
raise the task pass-rate; secondarily keep cost low (output_tokens,
wall_clock_s; lower is better). Pass-rate is primary.

A harness is a single self-contained Python file exposing:
    def build_agent(model, backend):
        # returns a compiled deepagents agent (create_deep_agent(...))
It must NOT construct the model or backend (both are injected). It builds a
deepagents agent over a FROZEN solver LLM; you are tuning the HARNESS around
that model, not the model.
[... deepagents customization reference; seed-harness + SoTA catalogs ...]

# Your inputs (on disk -- open them with your tools; nothing is inlined here)
- Parent harness -- the file you are improving:
    <parent_path>
- Failure evidence -- WHY the parent loses tasks (read this to find the
  root cause):
    <evidence_dir>/report.md        per-task results: pass/fail, tests, cost
    <evidence_dir>/traces/<task>/   for each FAILED task --
        trajectory.json  what the agent actually did, step by step
        ctrf.json        which tests failed and their messages
        result.json      the verifier reward
- Search history -- what has already been tried across the whole population:
    <tree_md>
    every harness so far with its pass-rate + costs, its lineage, the diff
    that produced it, and a "what moved the needle" ranking (which changes
    helped vs. were rejected). It also lists the ABSOLUTE ON-DISK PATH of
    every node (its harness.py, with its diff.patch alongside), and you are
    FREE to open and read ANY of them with your tools -- pull up a promising
    ancestor's or sibling's FULL source to build on it, or a rejected node's
    source+diff to see exactly what failed and avoid it.

# Workflow (do these in order)
1. Read the search history (<tree_md>) -- learn which changes already helped
   and which were rejected, so you neither repeat a dead end nor undo a win.
2. Read the parent harness (<parent_path>).
3. Investigate failures: list <evidence_dir>/traces, then read
   trajectory.json for 3-5 representative failed tasks. Identify the
   concrete, RECURRING reason the parent loses -- e.g. stops before
   verifying, loops on an error, misuses a tool, exhausts its turns,
   misreads the task.
4. Decide ONE hypothesis: a specific failure mode + the harness change that
   fixes it.
5. CHOOSE THE SCOPE OF YOUR CHANGE -- match the boldness of the edit to what
   the evidence shows. Read the parent's recent lineage in the history and
   decide which situation you are in:
   - REFINEMENT (make a TARGETED edit) -- the design is fundamentally
     working and is close. Use this when: the parent has a healthy
     pass-rate, recent edges still moved pass-rate, and the failures are
     specific and fixable in place. Then make ONE focused change: sharpen a
     prompt, add or fix a middleware, add a tool/skill.
   - STRUCTURAL REDESIGN (make a BOLD, architectural change) -- the current
     design is STUCK and incremental edits will not free it. Switch to this
     when the evidence shows the approach itself is the ceiling, e.g.: the
     parent's last 2-3 children were REJECTED, or pass-rate has been FLAT
     across several edges; or the failures are not one bug but a WHOLE
     CLASS the architecture can't address. Then RE-ARCHITECT the harness
     around a different operating principle -- e.g. a multi-subagent
     division of labor, a different control strategy (verification-gated
     stopping, persistence-on-giveup, context discipline), or new skills
     distilled from the LEARNINGS in the history. A bold change is a
     HYPOTHESIS about the architectural cause, not a random redesign.
   Bias rule: do NOT default to timid. If the parent is stuck, a targeted
   edit is the WRONG choice -- escalate. If the parent is improving and
   close, do NOT gratuitously rewrite a working design -- refine it.
6. Write the COMPLETE improved harness to:
    <output_path>

# Constraints
- Exactly ONE coherent change -- but "one change" scales with scope: at
  REFINEMENT scope it is a single targeted edit; at STRUCTURAL scope it is
  one coherent re-architecture, NOT a grab-bag of unrelated edits.
- Build on what the history shows worked; do not re-try a change already
  rejected.
```

```
- Keep `build_agent(model, backend)` intact; keep the file self-contained
  and importable.
- Write ONLY <output_path>. Everything else is a read-only reference.

# Report (end your turn with a short note)
- The failure mode you found, with the specific task + trajectory step as
  evidence.
- The SCOPE you chose (REFINEMENT or STRUCTURAL REDESIGN) and the snapshot
  signal that decided it.
- The change you made and why it should raise pass-rate.
```

C.2 ORCHESTRATOR-AGENT PROMPT

The instruction below is handed to the orchestrator $\mathcal{O}$ once an island's stall counter reaches the patience P (Sec. 4.2, stage 5), abridged, with elisions marked `[...]`; angle-bracket placeholders are filled per invocation. It covers both phases of Eq. 5: the agent inspects the search configuration $\Theta^{(r)}$ and the stalled island $j^\star$, and its only output is a structured decision file holding the diagnosis d and the intervention ω, at most two actions from Eq. 6, which the loop then applies. For each GRAFT the agent is re-invoked with this diagnosis as context to write the merged harness.

Listing 7 · The orchestration prompt (handed to the orchestrator agent at a stall)

The orchestration prompt (diagnose phase, abridged).

```
# Role
You are the ORCHESTRATOR of an evolutionary harness search on
Terminal-Bench 2.

# How the search works (so your decision fits the mechanism)
- The population is split into ISLANDS. Each island is a small Pareto front
  of harnesses that evolves INDEPENDENTLY: every round it selects a parent
  and an OPTIMIZER AGENT mutates it into a child, which is admitted only if
  it is not dominated (better pass-rate, or a cheaper niche).
- Islands started from DIFFERENT seed harnesses, so each explores a
  different design philosophy. They never interact on their own -- crossing
  ideas between islands is YOUR job.
- The search has STAGNATED: at least one island has not improved its best
  pass-rate for several consecutive rounds. You are called to diagnose why
  and restructure the search to unstick it.

# This is PHASE 1 of 2 -- DECIDE (you do not edit harnesses here)
You only output a DECISION. The loop then EXECUTES it: reassignment and
speciation are applied directly; for each recombination YOU will be
re-invoked (phase 2) with this diagnosis in hand to actually write the
merged harness.

# What you can see (open it with your tools; nothing is inlined)
- Whole-population snapshot -- every island, every harness (admitted and
  rejected) with its pass-rate and costs, its lineage, the diff edge that
  produced it, and a "what moved the needle" ranking. Same evidence the
  mutators see, but across ALL islands at once:
    <snapshot_md>
  The snapshot lists the ON-DISK PATH of every node, and you are FREE to
  open and read ANY file you want with your tools. Read whatever nodes
  across whatever islands help you diagnose the stall.

# First, figure out WHICH KIND of stall this is
  A. DRY OPTIMIZER (-> variation). An island's optimizer keeps proposing
     children that get REJECTED, or that change the prose but not the
     behavior, or it retries a change the ranking already shows failed.
     The agent itself is the bottleneck -> reassign that island a
     different optimizer from the pool.
  B. LOCAL OPTIMUM / CONVERGED ISLAND (-> structure: recombine). An island
     has plateaued BUT a DIFFERENT island clearly solves tasks this one
     fails. The missing capability already exists elsewhere -> graft it in
     by recombining the two islands.
  C. CROWDED-OUT NICHE (-> structure: speciate). Inside one island, a
     behaviorally-distinct minority harness is dominated/ignored because
     selection favors the majority lineage. It deserves its own search ->
     speciate it into a new island.
  D. WHOLE-POPULATION PLATEAU. Every island is near the same ceiling. The
     best levers are still B and C -- apply them across the islands that
     differ most.

# The optimizer-agent pool you can assign from (WHAT each spec is)
[... one line per pool spec: model + character ...]

# The moves you can choose (each addresses specific hypotheses above)
  (i)  VARIATION -- REASSIGN optimizer agents from the pool. You may re-map
       ANY SUBSET of islands in a single decision; the entire reassignment
       counts as ONE move. Addresses hypothesis A.
  (ii) STRUCTURE --
       - RECOMBINE two DIFFERENT islands: mate the destination island's
         best harness with a donor island's best harness -> a child
         (admitted to the destination) fusing the donor's distinct
         mechanism into the destination lineage. Addresses B (and D).
         [... seed/SoTA catalog references ...]
       - SPECIATE: promote a specific, behaviorally-distinct harness to
         its OWN new island with its own optimizer. Addresses C (and D).

# Decompose into DISCRETE, WELL-DEFINED decisions
Every move is atomic and must be FULLY SPECIFIED so the loop can execute it
without guessing -- an under-specified or compound move is skipped. [...]

# Choose ONLY the 1-2 MOST PRESSING moves -- do NOT propose everything
Stagnation is unstuck by the SMALLEST effective intervention. Emit AT MOST
2 MOVES TOTAL. Prefer one decisive move over a scattershot of speculative
```

```
ones; the search runs many more rounds and you will be called again if the
stall persists.

# Current island -> optimizer-agent assignment (reason about THIS before
# reassigning)
[... island -> spec mapping; existing island ids; next new island id ...]

# Output -- write ONLY this file, valid JSON, then stop:
    <decision_path>
{
  "diagnosis": "<2-4 sentences: the cause, grounded in specific
                harnesses/edges in the snapshot>",
  "loci": [<any of "variation", "structure">],
  "reassign":  [{"island": <iid>, "optimizer": "<pool spec>", "why": "..."}],
  "recombine": [{"dest_island": <iid>, "donor_island": <iid>, "why": "..."}],
  "speciate":  [{"from_island": <iid>, "harness_hid": "<hid>",
                 "optimizer": "<pool spec>", "why": "..."}]
}

# Rules
- Ground EVERY choice in the snapshot (name the islands / harnesses / edges
  you reason from).
- Only reference islands that exist and optimizers in the pool.
- For recombine, dest_island and donor_island must be DIFFERENT existing
  islands.
- For speciate, harness_hid must exist in from_island.
- AT MOST 2 moves total (reassignment of any #islands = one move).
```

In this second invocation, executed once per $\textsc{Graft}(j_{\text{donor}} \to j_{\text{dst}})$, the agent receives the sources of both parents, the destination island's h_{par} and the donor's co-parent h_{co} (Sec. 4.1.1), the failure evidence $\textsc{Evid}(h_{\text{par}}, \mathcal{B}^{(r)}_{\text{search}})$ staged in its workspace, and its own diagnosis d with the move's rationale as context. It is instructed to merge the donor's distinct mechanism into the destination harness; the merged child h then passes through the standard admission test (Eq. 7).

D Strategy-Level Discovery of Harnesses

D.1 Best Harness Evolved by Each SoTA Evolutionary Search

All methods start from the same three seeds (B10–B12, App. B.3); the single-candidate baselines take the Best-of-3 for this configuration, B12 (RR@5 74.1), and MILO seeds one island with each. Tab. 7 summarizes the mechanisms in each method's best Terminal-Bench 2.1 harness with Opus 4.8; the listings give those harnesses as diffs against B12 (Listing 5), in the order of Tab. 1. GEPA (optimize-anything) returned the seed unchanged and has no listing.

Four failure modes dominate Terminal-Bench 2.1: *premature completion*, *deadline overrun*, *lax verification*, and *missing know-how*. Every baseline rewords or lightly extends the seed: GEPA edits the prompt, A-Evolve adds skills, Meta-Harness adds a deadline middleware, and OpenEvolve, ShinkaEvolve, and EvoX return prompt rewrites of B12 despite being free to edit code. MILO alone rebuilds the assembly, replacing the seed's plan gate with an independent verifier gate, a scaffold floor, and a deadline governor, and leads by $+7.5$ RR@5 (Tab. 1).

Table 7: **Mechanisms in each method's best harness, by failure mode addressed** (Terminal-Bench 2.1, Opus 4.8). *Search reach* = the surface the method may edit. Failure modes: *premature completion* (success declared at a wrong state), *deadline overrun* (timeout zeros the trial), *lax verification* (self-check weaker than the hidden grader), *missing know-how*. Marks: ✓ code-level mechanism (named), ∼ prompt-level mitigation, ✗ unaddressed.

		Search reach				Failure mode addressed (mechanism)				
Method	**Design**	Prompt	Skills/ mem	Tools	Middle-ware	Premature completion	Deadline overrun	Lax verification	Missing know-how	**RR@5↑**
reproduce-first rubric gate seed [B12]	EXPERT	✓	✗	✗	✓	✓ *rubric gate*	✗	✗	✗	74.1
GEPA *(prompt)*	AUTO	✓	✗	✗	✗	✓ *rubric gate*	∼	∼	✗	78.6
GEPA *(optimize-anything)*	AUTO	✓	✗	✓	✓	✓ *rubric gate*	∼	∼	✗	74.1
A-Evolve *(skill, memory)*	AUTO	✓	✓	✗	✗	✓ *rubric gate*	∼	✓ *criteria skill*	✓ *skills lib*	78.2
OpenEvolve	AUTO	✓	✓	✓	✓	✓ *rubric gate*	✗	✗	✗	76.1
ShinkaEvolve	AUTO	✓	✓	✓	✓	✓ *rubric gate*	∼	∼	∼ *playbooks*	77.7
EvoX	AUTO	✓	✓	✓	✓	✓ *rubric gate*	∼	∼	✗	76.8
Meta-Harness	AUTO	✓	✓	✓	✓	✓ *rubric gate*	✓ *pacing mw*	✗	✗	78.2
MILO *(ours)*	AUTO	✓	✓	✓	✓	✓ *verifier gate*	✓ *floor+governor*	✓ *adversarial verifier*	✗	**86.1**

Listing 8 · GEPA *(prompt-only)*, diff vs. Best-of-3 (B12, Lst. 5) **RR@5 = 78.6**

GEPA (prompt-only). With the code frozen, GEPA rewrites `SYSTEM_PROMPT` alone: the seed's four-step protocol becomes a longer instruction to reach a valid finished state early and refine it in place, with economy rules against repeated expensive operations and speculative detours. Rubric gate and code are inherited unchanged.

```
    """
    deepagents_scientist.py -- philosophy: REPRODUCE FIRST, TEST-DRIVEN, EVIDENCE-GATED.

    Thesis: the agent that wins debugging/fixing tasks does not start by editing. It first
    REPRODUCES the failure (turns the bug into a concrete failing command), then forms a
    hypothesis, makes the smallest change, and re-runs the SAME command to confirm -- the
    scientific method applied to terminal tasks. This is the discipline behind SWE-agent's
    reproduce-script practice and Terminus/KIRA's verify-before-submit, made the spine of the
    loop instead of an afterthought. It directly attacks the most expensive failure mode in
    our own results: stopping at a plausible-but-wrong state (the "all-green by assertion"
    trap), and the timeouts caused by editing blindly and thrashing.

    Mechanism -- two reinforcing layers:
      1. SYSTEM_PROMPT: a strict reproduce -> hypothesize -> minimal-change -> re-verify
         protocol, plus "write or find a check that fails now and must pass at the end."
      2. RubricMiddleware (deepagents' built-in evaluator-optimizer loop): a grader subagent
         checks the transcript against an explicit "what done looks like" rubric each time
         the agent would finish, and bounces it back with specifics if a criterion is unmet.
         This is Anthropic's evaluator-optimizer pattern as native middleware -- an
         independent reviewer the lead cannot satisfy by merely asserting success. The rubric
         is injected per-invocation, so build_agent wraps invoke to attach it automatically.

    deepagents' RubricMiddleware only activates when a `rubric` is present in the invocation
    state; to keep the standard `build_agent(model, backend)` + `agent.ainvoke({"messages":
    ...})` contract, we return a thin wrapper whose `ainvoke`/`invoke` inject the default
    rubric if the caller didn't supply one. Grader uses the same injected model (no globals).

    Mutable surfaces: the rubric criteria, the grader's max_iterations, and the prompt.
    """

    DEFAULT_RUBRIC = """\
    The task is DONE only if every criterion below is satisfied, with evidence visible in the
    transcript (a command was actually run and its output shown -- not merely asserted):
    1. The failure or requirement was reproduced or pinned down with a concrete command
       before any fix was attempted (a failing test, a reproducing command, or an explicit
       inspection of the current state).
    2. The change made is minimal and targeted at the identified cause -- no unrelated files
       or configuration were altered as a side effect.
    3. The required end state was verified by RE-RUNNING the same concrete check that
       defined success (the test now passes / the command now succeeds / the file now has the
       required content / the service actually responds), and its successful output is shown.
    4. If the task specified particular outputs, file paths, formats, or values, each was
       checked against the requirement rather than assumed.
    """
    # [...]
    SYSTEM_PROMPT = """\
    You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
    unless told otherwise). The task is judged by the final state of the environment. Work
    like a scientist, not a guesser.

    PROTOCOL:
    1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
       you can run now: reproduce the failure, run the failing test, or inspect the exact
       current state. If no check exists, create the smallest one (a short script or command)
       that fails now and must succeed at the end. Inspect the working directory once first.
    2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
       in one sentence -- before editing. Do not edit on a hunch.
    3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
       what the task requires; leave the rest of the system identical (no stray files or
       config edits).
    4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
       output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

    Treat errors as evidence, not noise. Never declare success on assumption; success is the
    re-run check passing in front of you.
    """
    SYSTEM_PROMPT = 'You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app\nunless told otherwise). The task is judged by the
            FINAL STATE of the environment, not by your\nexplanation of it. Work like a scientist, not a guesser: trust checks you actually run, not\nreasoning you
            believe. Prefer executing and observing over narrating what you expect.\n\nOVERARCHING PRIORITY -- REACH A FINISHED, VALID STATE, THEN IMPROVE IT.\nThe
            run can end at any moment (time, steps, or an unexpected error). A correct-shaped artifact\nalready saved in the right place beats a better answer you
            never got to write. So: get to a\ncomplete, contract-satisfying answer as early as you can, keep the judged artifact in a valid\nstate throughout, and
            refine its content in place. Be economical -- avoid repeating expensive\noperations, avoid long speculative detours, and never run commands likely to
            hang, flood\noutput, or damage the environment or the thing being judged. If an approach isn\'t working,\nchange something informed by the last
            observation; never blindly retry the identical action.\n\nPROTOCOL:\n\n1. INSPECT & PIN DOWN THE CONTRACT. Inspect the working directory once first.
            Then restate\n   the goal as a precise, checkable success condition: what artifact must exist, at what\n   location, in exactly what form (structure,
            format, units, encoding, ordering, single vs.\n   multiple values, whitespace/newlines). Actively hunt for clues about the expected shape of\n   the
            answer -- the task wording, existing files, provided examples, schemas, or the test\n   command itself. If the requirement is ambiguous, take the most
            literal, minimal\n   interpretation, and do NOT emit extra or "bonus" output that an exact-match verifier could\n   reject.\n\n2. REPRODUCE / BUILD THE
            CHECK. Before changing anything, create the smallest thing that\n   fails now and must pass at the end: reproduce the failure, run the failing test,
            or\n   build/run the real system. Make this check mirror how the task will actually be judged as\n   closely as you can. When the environment offers a
            real verification path (compile it, run\n   the program, run the exact test command referenced by the task), use it -- do not substitute\n   an argument
            that it "would" work.\n\n3. VERIFY AGAINST GROUND TRUTH, NOT YOUR OWN ASSUMPTIONS. Ruthlessly separate "internally\n   consistent / well-formed / legal
            / self-agreeing" from "actually correct for the\n   requirement." A check that only confirms your own interpretation proves nothing. When your\n
            result rests on an uncertain step -- perception, parsing, reconstruction, inference, or a\n   tool\'s output you cannot directly see -- treat that step
            as your primary risk and confirm it\n   an INDEPENDENT way against the original source. If the task supplies any oracle (an expected\n   prefix/sample,
            a checksum, a validator, a reference), use it to discriminate between\n   competing interpretations rather than guessing. Distrust degraded signals: if
            a tool returns\n   something stale, empty, truncated, or placeholder-like, do not build on it -- re-establish the\n   raw facts through a second
            independent path before proceeding. Do not declare a stronger or\n   more direct verification method "unavailable" until you have genuinely attempted
            it.\n\n4. HYPOTHESIZE. State the specific cause, or the specific change the requirement implies, in\n   one sentence -- before editing. Do not edit on a
            hunch.\n\n5. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only what\n   the task requires; leave the rest of the
            system identical (no stray files, no unrelated\n   config edits). Remove scratch/test artifacts you created for your own use, and confirm any\n   input
            you were given remains untouched.\n\n6. RE-VERIFY. Re-run the EXACT check from step 2 AND confirm the output contract from step 1\n   (right location,
            exact form, no extra content). If it does not pass, read the actual output,\n   refine the hypothesis, and iterate -- never retry the same thing
            blindly.\n\nTreat errors as evidence, not noise. Never declare success on assumption or on a\nself-satisfying proxy check; success is the real,
            judged-style check passing in front of you.\nIf you genuinely cannot fully verify, spend your remaining effort attacking your riskiest\nassumption
            rather than polishing what already looks right.'
    # [...]
        """Thin wrapper that injects the default rubric into invocations.

        RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
        preserves the seed's standard call contract by adding the rubric when the caller
        omitted it, while still allowing a caller (or the evolutionary loop) to override it.
        """
    # [...]
        """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
```

```
    can't be copied / has no thinking fields.

    Why the grader must not think: the grader is a structured-output classifier (it must emit
    a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
    with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
    the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
    Converse API rejects ("The final block in an assistant message cannot be `thinking`"),
    making the grader fail and silently disabling the rubric gate (the seed's whole point).
    Extended thinking buys nothing for a short verdict classification, so we strip it on the
    grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
    touch neither deepagents nor the shared model builder; the original model is unmutated.
    """
# [...]
def build_agent(model, backend):
    grader_model = _thinking_free(GRADER_MODEL or model)
    agent = create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
```

Listing 9 · A-Evolve *(skill & memory)*, diff vs. Best-of-3 (B12, Lst. 5) RR@5 = 78.2

A-Evolve (skill and memory). Prompt and code stay fixed; A-Evolve grows a git-tracked workspace of six `SKILL.md` files, which the bridge shown here loads into the seed: two process skills (find and verify acceptance criteria, finish and submit) and four per-ecosystem skills (Cython extensions, gate-level circuit files, long-running ML training, XSS-filter bypass). Skill bodies are omitted.

```
"""
harness_bridge.py -- the ONE glue point between A-Evolve's evolving workspace and our shared
deepagents seed harnesses.

A-Evolve's genome is an `AgentWorkspace` directory (`prompts/system.md`, `skills/*/SKILL.md`,
`memory/*.jsonl`). Our benchmarks evaluate a deepagents seed via `build_agent(model, backend)`
run through `benchmarks/<b>/run.sh` (harbor). This module bridges the two so the SAME evaluator
every SOTA method uses (GEPA's `scorers.score`) can score an A-Evolve-evolved workspace.

HOW: this file IS a harness -- `run.sh`'s `HARNESS=<path>` accepts any `.py` exposing
`build_agent(model, backend)`, and harbor loads it verbatim. When called it:

  1. reads the workspace path + chosen seed + which layers to inject from the environment
     (AEVOLVE_WORKSPACE / AEVOLVE_SEED / AEVOLVE_INJECT -- set by the launcher; propagated by
     `scorers.score` because it forwards `os.environ`);
  2. materializes the workspace's evolved `skills/` INTO the harbor container and loads them with
     deepagents' native `SkillsMiddleware` -- the EXACT materialize-then-load pattern proven in
     `seed_harnesses/deepagents_native_skills.py` (incl. the `_FullPathLsBackend` als-fix);
  3. optionally overrides the seed's `system_prompt` with the workspace's `prompts/system.md`
     (only if the "prompt" layer is being evolved -- the TB recipe is skills-only, so off);
  4. calls the chosen seed's OWN `build_agent`, preserving all of its own middleware.

Injection is done by monkeypatching `deepagents.create_deep_agent` for the duration of the seed's
`build_agent` call, so we add our middleware to whatever the seed already builds -- generic across
every shared seed (orchestrator / planner / scientist), no per-seed code.

This keeps A-Evolve's algorithm + loop UNCHANGED (imported, not reimplemented) and holds the solver
+ evaluator fixed to the shared apples-to-apples contract (`common/README.md`).
"""

from deepagents import create_deep_agent
from deepagents.middleware.rubric import RubricMiddleware
import base64
import hashlib
import importlib
import os
from pathlib import Path
# [...]
# The default "what done looks like" rubric. RubricMiddleware grades the transcript
# against these each time the agent tries to finish, and continues it with specifics if
# any criterion is unmet. Phrased as end-state checks, not intentions.
DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The failure or requirement was reproduced or pinned down with a concrete command
   before any fix was attempted (a failing test, a reproducing command, or an explicit
   inspection of the current state).
2. The change made is minimal and targeted at the identified cause -- no unrelated files
   or configuration were altered as a side effect.
3. The required end state was verified by RE-RUNNING the same concrete check that
   defined success (the test now passes / the command now succeeds / the file now has the
   required content / the service actually responds), and its successful output is shown.
4. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
"""
def workspace_genome_hash(workspace: Path, inject: str) -> str:
    """Content hash of the injected genome (skills [+ prompt]) for THIS workspace.

    CRITICAL for correct eval dedup. The shared evaluator (gepa/scorers.py) keys its score cache
    AND the harbor job on sha256(harness_file_bytes) -- it never hashes the workspace dir. Our bridge
    FILE is byte-constant across evolution cycles (the evolving skills live in the workspace, passed
    by env), so without this the SAME task re-solved in a later cycle would hit a STALE cache /
    harbor-resume and the evolver would never see the effect of its own skill changes. We therefore
    generate a per-eval harness FILE that embeds this hash (see materialize_harness), so identical
    genome => identical bytes => correct reuse, changed genome => new bytes => fresh eval."""
    parts: list[str] = []
    ws = Path(workspace)
    if "skills" in inject:
        for sk in sorted((ws / "skills").rglob("*")):
            if sk.is_file() and not sk.name.startswith("_"):
                try:
                    parts.append(str(sk.relative_to(ws)) + "\0" + sk.read_text())
                except (UnicodeDecodeError, OSError):
                    continue
    if "prompt" in inject:
        p = ws / "prompts" / "system.md"
        if p.exists():
            parts.append("prompt\0" + p.read_text())
    return hashlib.sha256("\1".join(parts).encode()).hexdigest()[:16]
```

```
# [...]
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
   in one sentence -- before editing. Do not edit on a hunch.
3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
   what the task requires; leave the rest of the system identical (no stray files or
   config edits).
4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
   output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

Treat errors as evidence, not noise. Never declare success on assumption; success is the
re-run check passing in front of you.
"""
def materialize_harness(workspace: Path, seed: str, inject: str, dest: Path) -> Path:
    """Write a per-eval harness file = this bridge's source + a trailing genome-hash comment.

    The comment makes the file's bytes change iff the injected genome (skills/prompt) changes, so
    the shared scorer's content-hash cache/job keying is correct across evolution cycles (see
    workspace_genome_hash). build_agent still reads the live workspace via env at run time; the
    embedded hash is purely a cache-discriminator. Also pins the workspace/seed/inject as comments
    for provenance."""
    dest = Path(dest)
    dest.parent.mkdir(parents=True, exist_ok=True)
    src = Path(__file__).read_text()
    ghash = workspace_genome_hash(workspace, inject)
    footer = (
        f"\n\n# -- per-eval provenance (do not edit; discriminates the shared scorer's cache) --\n"
        f"# AEVOLVE_WORKSPACE = {workspace}\n"
        f"# AEVOLVE_SEED      = {seed}\n"
        f"# AEVOLVE_INJECT    = {inject}\n"
        f"# WORKSPACE_GENOME_HASH = {ghash}\n"
    )
    dest.write_text(src + footer)
    return dest

# NOTE on imports: deepagents + langchain are needed ONLY by build_agent, which runs inside harbor
# (the repo .venv, where they're installed). materialize_harness / workspace_genome_hash are called
# from the a-evolve .venv (which has NO deepagents), so all deepagents/langchain imports are LAZY
# (inside build_agent). harbor_agent.py puts repo-root seed_harnesses/ on sys.path before loading a
# harness file, so `import deepagents...` and `import deepagents_<seed>` both resolve at run time.

# Where the evolved skills are materialized inside the container (same convention as native_skills).
_CONTAINER_SKILLS_ROOT = "/tmp/aevolve_skills"
# [...]
class _RubricAgent:
    """Thin wrapper that injects the default rubric into invocations.

    RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
    preserves the seed's standard call contract by adding the rubric when the caller
    omitted it, while still allowing a caller (or the evolutionary loop) to override it.
    """
# ---------- read the evolved workspace off the host FS ----------
# [...]
    def __init__(self, agent, rubric: str) -> None:
        self._agent = agent
        self._rubric = rubric
def _read_workspace_skills(workspace: Path) -> dict[str, dict[str, str]]:
    """Return {skill_name: {relpath: file_text}} for every skill dir in workspace/skills/.

    Mirrors what A-Evolve's `AgentWorkspace.list_skills` considers a skill: a subdir of `skills/`
    (excluding `_`-prefixed like `_drafts`) containing a `SKILL.md`. We copy ALL files in the dir
    (SKILL.md + any helper scripts the evolver wrote), so the materialized skill is complete."""
    skills_dir = workspace / "skills"
    out: dict[str, dict[str, str]] = {}
    if not skills_dir.is_dir():
        return out
    for d in sorted(skills_dir.iterdir()):
        if not d.is_dir() or d.name.startswith("_"):
            continue
        if not (d / "SKILL.md").exists():
            continue
        files: dict[str, str] = {}
        for f in sorted(d.rglob("*")):
            if f.is_file():
                try:
                    files[str(f.relative_to(d))] = f.read_text()
                except (UnicodeDecodeError, OSError):
                    continue  # skip binaries / unreadable files -- SKILL.md is what matters
        if files:
            out[d.name] = files
    return out
# [...]
    def _with_rubric(self, payload):
        if isinstance(payload, dict) and "rubric" not in payload:
            payload = {**payload, "rubric": self._rubric}
        return payload
# [...]
    async def ainvoke(self, payload, *args, **kwargs):
        return await self._agent.ainvoke(self._with_rubric(payload), *args, **kwargs)
def _read_workspace_prompt(workspace: Path) -> str | None:
    p = workspace / "prompts" / "system.md"
    return p.read_text() if p.exists() else None
# [...]
    def invoke(self, payload, *args, **kwargs):
        return self._agent.invoke(self._with_rubric(payload), *args, **kwargs)

# ---------- container materialization (from native_skills) ----------

async def _materialize(backend, root: str, skills: dict[str, dict[str, str]]) -> None:
    """Write the evolved SKILL.md library into the container in one round-trip (base64 heredocs).
    Byte-for-byte the approach in deepagents_native_skills._materialize, reading the evolved
    workspace instead of a hardcoded dict."""
    parts = ["set -e"]
    for name, files in skills.items():
```

```
        d = f"{root}/{name}"
        parts.append(f"mkdir -p '{d}'")
        for rel, content in files.items():
            # allow skills to reference their own dir via the SKILLS_ROOT token, like native_skills
            body = content.replace("SKILLS_ROOT", root)
            sub = f"{d}/{rel}"
            parent = str(Path(sub).parent)
            parts.append(f"mkdir -p '{parent}'")
            b64 = base64.b64encode(body.encode()).decode()
            parts.append(f"printf %s '{b64}' | base64 -d > '{sub}'")
    try:
        await backend.aexecute("\n".join(parts))
    except Exception:  # never fail a trial over skill materialization
        pass


def _make_materialize_mw(backend, root: str, skills: dict[str, dict[str, str]]):
    """Build the 'materialize skills into the container first' middleware. Defined as a FACTORY
    (not a module-level class) because it subclasses deepagents' AgentMiddleware, which is only
    importable inside harbor's repo venv -- see the import note at the top of this file."""
    from langchain.agents.middleware import AgentMiddleware

    class _MaterializeSkillsFirst(AgentMiddleware):
        """Write the evolved skill folders into the container BEFORE SkillsMiddleware's loader runs.
        Ordering within the user middleware list guarantees this before_agent runs first."""

        def __init__(self, backend, root: str, skills: dict[str, dict[str, str]]) -> None:
            super().__init__()
            self._backend = backend
            self._root = root
            self._skills = skills
            self._done = False

        async def abefore_agent(self, state, runtime):
            if not self._done and self._backend is not None and self._skills:
                await _materialize(self._backend, self._root, self._skills)
                self._done = True
            return None

    return _MaterializeSkillsFirst(backend, root, skills)


class _FullPathLsBackend:
    """Backend shim that makes `ls` return FULL paths, required by SkillsMiddleware discovery on
    the vendored HarborSandbox (whose ls returns bare basenames). Verbatim from
    deepagents_native_skills._FullPathLsBackend -- wraps the backend for THIS SkillsMiddleware only,
    leaving the agent-facing `ls` tool and every other seed untouched."""

    def __init__(self, inner) -> None:
        self._inner = inner

    async def als(self, path: str):
        result = await self._inner.als(path)
        entries = getattr(result, "entries", None)
        if entries:
            base = path.rstrip("/")
            for e in entries:
                p = e.get("path", "")
                if p and "/" not in p:
                    e["path"] = f"{base}/{p}"
        return result
# [...]
        # Delegate everything else (get_state, stream, etc.) to the compiled graph.
        return getattr(self._agent, name)
        return getattr(self._inner, name)
# [...]
# The grader re-reads the transcript and reasons each time the agent would finish, so it
# is the seed's main cost driver: with max_iterations=3 a hard task can pay up to 3 extra
# full-model reasoning passes right at the end -- exactly when wall-clock is scarcest and a
# timeout auto-zeros the trial. We cap it at 2 (one re-grade after a fix). If a lighter
# grader model is available, the evolutionary loop should pass GRADER_MODEL to cut cost
# further; by default the grader inherits the injected solver model.
GRADER_MAX_ITERATIONS = 2
GRADER_MODEL = None  # None -> use the injected solver model


def _thinking_free(model):
    """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
    can't be copied / has no thinking fields.

    Why the grader must not think: the grader is a structured-output classifier (it must emit
    a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
    with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
    the grader then sometimes ends a turn on a 'thinking' block with no tool call -- which the
    Converse API rejects ("The final block in an assistant message cannot be 'thinking'"),
    making the grader fail and silently disabling the rubric gate (the seed's whole point).
    Extended thinking buys nothing for a short verdict classification, so we strip it on the
    grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
    touch neither deepagents nor the shared model builder; the original model is unmutated.
    """
    amrf = getattr(model, "additional_model_request_fields", None)
    if not amrf or "thinking" not in amrf:
        return model
    cleaned = {k: v for k, v in amrf.items() if k not in ("thinking", "output_config")}
    try:
        return model.model_copy(update={"additional_model_request_fields": cleaned})
    except Exception:
        return model

# ---------------------- the build_agent harbor calls ----------------------
# [...]
def build_agent(model, backend):
    grader_model = _thinking_free(GRADER_MODEL or model)
    agent = create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
    """Assemble the chosen seed's agent with the evolved workspace injected.

    Env (set by run_aevolve.py, forwarded through scorers.score -> run.sh -> harbor):
```

```
    AEVOLVE_WORKSPACE  path to the A-Evolve workspace dir (the genome)          [required]
    AEVOLVE_SEED       seed module name, e.g. "deepagents_scientist"            [required]
    AEVOLVE_INJECT     comma list of layers to inject: any of skills,prompt     [default: skills]
    """
    workspace = Path(os.environ["AEVOLVE_WORKSPACE"]).resolve()
    seed_name = os.environ["AEVOLVE_SEED"]
    inject = {s.strip() for s in os.environ.get("AEVOLVE_INJECT", "skills").split(",") if s.strip()}

    seed_mod = importlib.import_module(seed_name)

    skills = _read_workspace_skills(workspace) if "skills" in inject else {}
    ws_prompt = _read_workspace_prompt(workspace) if "prompt" in inject else None

    import deepagents
    from deepagents.middleware.skills import SkillsMiddleware
    _orig_create = deepagents.create_deep_agent

    def _patched_create(*args, **kwargs):
        # 1) prompt layer (only if being evolved): override the seed's system_prompt.
        if ws_prompt is not None:
            kwargs["system_prompt"] = ws_prompt
        # 2) skills layer: append materialize + native SkillsMiddleware AFTER the seed's own
        #    middleware, so ordering (materialize -> load) holds and the seed's mw is preserved.
        if skills:
            mw = list(kwargs.get("middleware") or [])
            mw.append(_make_materialize_mw(backend, _CONTAINER_SKILLS_ROOT, skills))
            mw.append(SkillsMiddleware(backend=_FullPathLsBackend(backend),
                                       sources=[_CONTAINER_SKILLS_ROOT]))
            kwargs["middleware"] = mw
        return _orig_create(*args, **kwargs)

    # Some seeds import create_deep_agent by-name into their own module namespace; patch BOTH the
    # package attribute and the seed module's binding so the override is seen regardless.
    deepagents.create_deep_agent = _patched_create
    seed_had = hasattr(seed_mod, "create_deep_agent")
    if seed_had:
        seed_orig = seed_mod.create_deep_agent
        seed_mod.create_deep_agent = _patched_create
    try:
        return seed_mod.build_agent(model, backend)
    finally:
        deepagents.create_deep_agent = _orig_create
        if seed_had:
            seed_mod.create_deep_agent = seed_orig
```

Listing 10 · OpenEvolve, diff vs. Best-of-3 (B12, Lst. 5) **RR@5 = 76.1**

OpenEvolve. Free to edit code, it returns a compressed B12: a shorter four-step protocol and rubric, a grader that runs on a thinking-free copy of the model (`_without_thinking`) for at most two iterations, and a lightly refactored `_RubricAgent` wrapper. The rubric-gate assembly is kept.

```
SYSTEM_PROMPT = """\
You are an autonomous software-fixing agent in a Linux sandbox. Work in /app unless the
task says otherwise. The final environment state is what matters.

Use this loop:
1. Inspect first: pwd, list the repo, read the relevant files or error output.
2. Define a concrete success check before editing. Prefer the user's failing command/test.
   If none exists, use the smallest command that pins the required state.
3. State the likely cause/change in one sentence, then make the smallest targeted edit.
   Avoid unrelated rewrites, broad formatting, stray files, or package installs unless needed.
4. Re-run the same success check. If it fails, use the new output as evidence and iterate.
5. When done, report only what changed and the verification command/output.

Never claim success from intuition. Show that the command, test, file content, or service
state requested by the task was actually verified.
"""
# [...]
# The default "what done looks like" rubric. RubricMiddleware grades the transcript
# against these each time the agent tries to finish, and continues it with specifics if
# any criterion is unmet. Phrased as end-state checks, not intentions.
DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The failure or requirement was reproduced or pinned down with a concrete command
   before any fix was attempted (a failing test, a reproducing command, or an explicit
   inspection of the current state).
2. The change made is minimal and targeted at the identified cause -- no unrelated files
   or configuration were altered as a side effect.
3. The required end state was verified by RE-RUNNING the same concrete check that
   defined success (the test now passes / the command now succeeds / the file now has the
   required content / the service actually responds), and its successful output is shown.
4. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
"""
DEFAULT_RUBRIC = """\
The task is complete only when the transcript shows evidence for all of these:
1. The agent inspected or reproduced the relevant current state before editing.
2. A concrete success check was identified and actually run or inspected.
3. The edit was minimal and targeted to the task; no unrelated files/configuration changed.
4. The same concrete check, plus any task-specific required paths/values/formats, was
   verified after the edit with visible successful output.
If any item is missing, tell the agent exactly what evidence or fix is still required.
"""

GRADER_MAX_ITERATIONS = 2
GRADER_MODEL = None
# [...]
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
```

```
    2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
       in one sentence -- before editing. Do not edit on a hunch.
    3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
       what the task requires; leave the rest of the system identical (no stray files or
       config edits).
    4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
       output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

    Treat errors as evidence, not noise. Never declare success on assumption; success is the
    re-run check passing in front of you.
    """


    class _RubricAgent:
        """Thin wrapper that injects the default rubric into invocations.

        RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
        preserves the seed's standard call contract by adding the rubric when the caller
        omitted it, while still allowing a caller (or the evolutionary loop) to override it.
        """

        def __init__(self, agent, rubric: str) -> None:
            self._agent = agent
            self._rubric = rubric

        def _with_rubric(self, payload):
            if isinstance(payload, dict) and "rubric" not in payload:
                payload = {**payload, "rubric": self._rubric}
            return payload

        async def ainvoke(self, payload, *args, **kwargs):
            return await self._agent.ainvoke(self._with_rubric(payload), *args, **kwargs)

        def invoke(self, payload, *args, **kwargs):
            return self._agent.invoke(self._with_rubric(payload), *args, **kwargs)

        def __getattr__(self, name):
            # Delegate everything else (get_state, stream, etc.) to the compiled graph.
            return getattr(self._agent, name)


    # The grader re-reads the transcript and reasons each time the agent would finish, so it
    # is the seed's main cost driver: with max_iterations=3 a hard task can pay up to 3 extra
    # full-model reasoning passes right at the end -- exactly when wall-clock is scarcest and a
    # timeout auto-zeros the trial. We cap it at 2 (one re-grade after a fix). If a lighter
    # grader model is available, the evolutionary loop should pass GRADER_MODEL to cut cost
    # further; by default the grader inherits the injected solver model.
    GRADER_MAX_ITERATIONS = 2
    GRADER_MODEL = None  # None -> use the injected solver model


    def _thinking_free(model):
        """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
        can't be copied / has no thinking fields.

        Why the grader must not think: the grader is a structured-output classifier (it must emit
        a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
        with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
        the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
        Converse API rejects ("The final block in an assistant message cannot be 'thinking'"),
        making the grader fail and silently disabling the rubric gate (the seed's whole point).
        Extended thinking buys nothing for a short verdict classification, so we strip it on the
        grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
        touch neither deepagents nor the shared model builder; the original model is unmutated.
        """
        amrf = getattr(model, "additional_model_request_fields", None)
        if not amrf or "thinking" not in amrf:
    def _without_thinking(model):
        """Use a non-thinking copy for rubric grading so structured verdicts are reliable."""
        updates = {}
        for attr in ("additional_model_request_fields", "model_kwargs", "extra_body"):
            value = getattr(model, attr, None)
            if isinstance(value, dict) and any(k in value for k in ("thinking", "output_config")):
                updates[attr] = {k: v for k, v in value.items() if k not in ("thinking", "output_config")}
        if not updates:
    # [...]
            cleaned = {k: v for k, v in amrf.items() if k not in ("thinking", "output_config")}
    # [...]
                return model.model_copy(update={"additional_model_request_fields": cleaned})
            return model.model_copy(update=updates)
    # [...]
    class _RubricAgent:
        def __init__(self, agent, rubric):
            self._agent = agent
            self._rubric = rubric

        def _add(self, payload):
            if isinstance(payload, dict) and "rubric" not in payload:
                return {**payload, "rubric": self._rubric}
            return payload

        def _add_many(self, payloads):
            return [self._add(p) for p in payloads] if isinstance(payloads, list) else payloads

        async def ainvoke(self, payload, *args, **kwargs):
            return await self._agent.ainvoke(self._add(payload), *args, **kwargs)

        def invoke(self, payload, *args, **kwargs):
            return self._agent.invoke(self._add(payload), *args, **kwargs)

        async def abatch(self, payloads, *args, **kwargs):
            return await self._agent.abatch(self._add_many(payloads), *args, **kwargs)

        def batch(self, payloads, *args, **kwargs):
            return self._agent.batch(self._add_many(payloads), *args, **kwargs)

        async def astream(self, payload, *args, **kwargs):
            async for item in self._agent.astream(self._add(payload), *args, **kwargs):
                yield item

        def stream(self, payload, *args, **kwargs):
            return self._agent.stream(self._add(payload), *args, **kwargs)
```

```
    def __getattr__(self, name):
        return getattr(self._agent, name)


def build_agent(model, backend):
    grader_model = _thinking_free(GRADER_MODEL or model)
    grader = _without_thinking(GRADER_MODEL or model)
    agent = create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
        middleware=[RubricMiddleware(model=grader, max_iterations=GRADER_MAX_ITERATIONS)],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
```

Listing 11 · ShinkaEvolve, diff vs. Best-of-3 (B12, Lst. 5) RR@5 = 77.7

ShinkaEvolve. The opposite move in prompt space: B12's four-step protocol grows into a six-step playbook (orient cheaply first, error-recovery rules, per-ecosystem playbooks, a final audit) and the rubric to five criteria, with an instruction not to bounce the agent for process or style alone. Code is inherited unchanged.

```
"""
deepagents_scientist.py -- philosophy: REPRODUCE FIRST, TEST-DRIVEN, EVIDENCE-GATED.

Thesis: the agent that wins debugging/fixing tasks does not start by editing. It first
REPRODUCES the failure (turns the bug into a concrete failing command), then forms a
hypothesis, makes the smallest change, and re-runs the SAME command to confirm -- the
scientific method applied to terminal tasks. This is the discipline behind SWE-agent's
reproduce-script practice and Terminus/KIRA's verify-before-submit, made the spine of the
loop instead of an afterthought. It directly attacks the most expensive failure mode in
our own results: stopping at a plausible-but-wrong state (the "all-green by assertion"
trap), and the timeouts caused by editing blindly and thrashing.

Mechanism -- two reinforcing layers:
  1. SYSTEM_PROMPT: a strict reproduce -> hypothesize -> minimal-change -> re-verify
     protocol, plus "write or find a check that fails now and must pass at the end."
  2. RubricMiddleware (deepagents' built-in evaluator-optimizer loop): a grader subagent
     checks the transcript against an explicit "what done looks like" rubric each time
     the agent would finish, and bounces it back with specifics if a criterion is unmet.
     This is Anthropic's evaluator-optimizer pattern as native middleware -- an
     independent reviewer the lead cannot satisfy by merely asserting success. The rubric
     is injected per-invocation, so build_agent wraps invoke to attach it automatically.

deepagents' RubricMiddleware only activates when a `rubric` is present in the invocation
state; to keep the standard `build_agent(model, backend)` + `agent.ainvoke({"messages":
...})` contract, we return a thin wrapper whose `ainvoke`/`invoke` inject the default
rubric if the caller didn't supply one. Grader uses the same injected model (no globals).

Mutable surfaces: the rubric criteria, the grader's max_iterations, and the prompt.
"""

DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The failure or requirement was reproduced or pinned down with a concrete command
   before any fix was attempted (a failing test, a reproducing command, or an explicit
   inspection of the current state).
2. The change made is minimal and targeted at the identified cause -- no unrelated files
   or configuration were altered as a side effect.
3. The required end state was verified by RE-RUNNING the same concrete check that
   defined success (the test now passes / the command now succeeds / the file now has the
   required content / the service actually responds), and its successful output is shown.
4. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
"""
DEFAULT_RUBRIC = """\
Judge the END STATE, not process style. The task is DONE only if every criterion below is
satisfied with evidence visible in the transcript (a command was run and output shown, or
file content/state was directly inspected -- not merely asserted):
1. The requirement was pinned down with a concrete verifier before any fix: a failing test,
   reproducing command, explicit state inspection, service probe, or small ad-hoc verifier.
   For artifact creation tasks, direct inspection of the initial state counts.
2. The change made targets the identified requirement/cause. No unrelated refactors,
   dependency churn, config changes, or stray debug artifacts unless clearly required.
3. The required end state was verified by RE-RUNNING the same verifier that defined success
   (or the closest exact equivalent), and the successful result is visible in the transcript.
4. If the task specified outputs, paths, formats, values, metrics, plots, files, APIs, or
   commands, each explicit requirement was checked against evidence rather than assumed.
5. The final filesystem state appears intentional: no obvious temporary debug files remain,
   and no syntax/diff hygiene issue is visible that would invalidate the patch.

Do NOT bounce for process/style concerns alone if explicit requirements are verifiably met.
If only a cheap specific check of an explicit requirement is missing, ask for that check;
otherwise pass the completed task.
"""
# [...]
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
   in one sentence -- before editing. Do not edit on a hunch.
3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
   what the task requires; leave the rest of the system identical (no stray files or
   config edits).
4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
   output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

Treat errors as evidence, not noise. Never declare success on assumption; success is the
re-run check passing in front of you.
"""
```

```
SYSTEM_PROMPT = """\
You are an autonomous software-engineering agent in a sandboxed Linux environment. The
task is judged by the final filesystem/process state, not explanation. Work in /app unless
told otherwise. Work like a scientist, not a guesser.

PROTOCOL:

1. ORIENT CHEAPLY FIRST
   Inspect the working directory once before doing anything else:
   - pwd
   - ls -la
   - git status --short if available
   - inspect project shape (README, pyproject.toml, package.json, setup.py, tests/,
     Makefile, Cargo.toml, etc.)
   Do not begin with a full build, broad test suite, or long-running command.

2. REPRODUCE / PIN DOWN
   Before editing, turn the goal into a concrete check you can run now:
   - If there's a bug, reproduce it with the smallest failing command/test
   - If there's no existing test, create the smallest temporary verifier (ideally a
     one-liner or tiny script) that fails now and must pass at the end
   - For artifact tasks, inspect the exact current state
   Identify the exact success conditions from the user request: required behavior, files,
   outputs, tests, commands, metrics, plots, APIs, or paths. Note which are explicit vs
   inferred, and the cheapest concrete verifier for each.

3. HYPOTHESIZE BEFORE PATCHING
   Read/search relevant files before editing. Prefer rg, small file views, targeted commands.
   State the specific cause or required change in one sentence from observed code/output.
   Do not patch from a vague guess.

4. MINIMAL CHANGE
   Make the smallest coherent change that satisfies the requirement.
   - Avoid unrelated refactors/formatting churn/dependency upgrades
   - Preserve public APIs unless the task asks for change
   - Do not mask failures by weakening tests or swallowing errors broadly
   - If multiple independent requirements exist, handle them ONE AT A TIME with verification
     between each

5. RE-VERIFY
   After each patch, run:
   a. The exact reproducer/verifier that defined success
   b. Adjacent targeted checks for touched code
   c. Only then any broader cheap suite if worthwhile
   Passing some other command is not enough; re-run the original success check or the
   closest exact equivalent. If it does not pass, read the actual error message carefully --
   every detail matters. Refine the hypothesis based on the specific error and iterate.
   Do not retry the same thing blindly.

6. FINAL AUDIT
   - Re-check every explicit requirement against evidence
   - If files were edited, inspect git diff --stat and preferably git diff --check
   - Verify generated files by exact path plus small content/header sample when relevant
   - Ensure no temporary debug artifacts remain
   - Keep the final response concise: what changed and what passed

ERROR-RECOVERY RULES:
- Treat every error message as evidence; read the first meaningful traceback/error.
- If the same command fails twice, change the hypothesis before trying again.
- Use timeouts for commands that may hang (servers, tests, builds, notebooks, long scripts).
- Prefer existing environment/tooling over installing packages; install only if clearly
  necessary and lightweight.
- Once the targeted verifier passes and explicit requirements are checked, stop exploring;
  do not burn time on unnecessary broad commands.

TASK PLAYBOOKS:
- Python: prefer a single failing test or test file before full pytest.
- JS/TS: inspect package.json scripts; prefer targeted test commands before broad ones.
- CLI/text transformation: use tiny inputs and verify exact stdout/file output, including
  whitespace/newlines/JSON validity when relevant.
- Research/reproduction: identify the claimed artifact (metric/table/plot/file), run the
  smallest command that produces it, and verify the artifact exists with plausible content.
- Web/service: start services only when needed and verify with curl or the documented
  client; clean up any background processes you started.

Never declare success because a patch looks plausible. Finish only when contract verifiers
pass or when you have a clearly documented unavoidable blocker.
"""
```

```
    # [...]
        """Thin wrapper that injects the default rubric into invocations.

        RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
        preserves the seed's standard call contract by adding the rubric when the caller
        omitted it, while still allowing a caller (or the evolutionary loop) to override it.
        """
    # [...]
```

```
# timeout auto-zeros the trial. We cap it at 2 (one re-grade after a fix). If a lighter
# grader model is available, the evolutionary loop should pass GRADER_MODEL to cut cost
# further; by default the grader inherits the injected solver model.
# timeout auto-zeros the trial. We cap it at 2 (one re-grade after a fix): the 0.85-scoring
# ancestor used 2 and beat the 3-iteration variant on both score and cost, since a third
# grading pass mostly nitpicks a task that is already correct and risks a near-timeout
# bounce. If a lighter grader model is available, the evolutionary loop should pass
# GRADER_MODEL to cut cost further; by default the grader inherits the injected solver model.
```

```
    # [...]
        """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
        can't be copied / has no thinking fields.

        Why the grader must not think: the grader is a structured-output classifier (it must emit
        a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
        with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
        the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
        Converse API rejects ("The final block in an assistant message cannot be 'thinking'"),
        making the grader fail and silently disabling the rubric gate (the seed's whole point).
        Extended thinking buys nothing for a short verdict classification, so we strip it on the
        grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
        touch neither deepagents nor the shared model builder; the original model is unmutated.
        """
    # [...]
    def build_agent(model, backend):
        grader_model = _thinking_free(GRADER_MODEL or model)
        agent = create_deep_agent(
            model=model,
```

```
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
```

Listing 12 · EvoX, diff vs. Best-of-3 (B12, Lst. 5) **RR@5 = 76.8**

EvoX. A prompt-space rewrite of B12 with the code untouched: the rubric becomes four end-state criteria that pass once a concrete check has run and every explicit sub-requirement is addressed, a one-bounce budget names the single most important missing action, and the system prompt gains a cover-the-whole-task rule.

```
"""
deepagents_scientist.py -- philosophy: REPRODUCE FIRST, TEST-DRIVEN, EVIDENCE-GATED.

Thesis: the agent that wins debugging/fixing tasks does not start by editing. It first
REPRODUCES the failure (turns the bug into a concrete failing command), then forms a
hypothesis, makes the smallest change, and re-runs the SAME command to confirm -- the
scientific method applied to terminal tasks. This is the discipline behind SWE-agent's
reproduce-script practice and Terminus/KIRA's verify-before-submit, made the spine of the
loop instead of an afterthought. It directly attacks the most expensive failure mode in
our own results: stopping at a plausible-but-wrong state (the "all-green by assertion"
trap), and the timeouts caused by editing blindly and thrashing.

Mechanism -- two reinforcing layers:
  1. SYSTEM_PROMPT: a strict reproduce -> hypothesize -> minimal-change -> re-verify
     protocol, plus "write or find a check that fails now and must pass at the end."
  2. RubricMiddleware (deepagents' built-in evaluator-optimizer loop): a grader subagent
     checks the transcript against an explicit "what done looks like" rubric each time
     the agent would finish, and bounces it back with specifics if a criterion is unmet.
     This is Anthropic's evaluator-optimizer pattern as native middleware -- an
     independent reviewer the lead cannot satisfy by merely asserting success. The rubric
     is injected per-invocation, so build_agent wraps invoke to attach it automatically.

deepagents' RubricMiddleware only activates when a `rubric` is present in the invocation
state; to keep the standard `build_agent(model, backend).invoke({"messages": ...})`
contract, we return a thin wrapper whose `ainvoke`/`invoke` inject the default
rubric if the caller didn't supply one. Grader uses the same injected model (no globals).

Mutable surfaces: the rubric criteria, the grader's max_iterations, and the prompt.
"""

DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The failure or requirement was reproduced or pinned down with a concrete command
   before any fix was attempted (a failing test, a reproducing command, or an explicit
   inspection of the current state).
2. The change made is minimal and targeted at the identified cause -- no unrelated files
   or configuration were altered as a side effect.
3. The required end state was verified by RE-RUNNING the same concrete check that
   defined success (the test now passes / the command now succeeds / the file now has the
   required content / the service actually responds), and its successful output is shown.
4. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
"""
DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The required end state was verified by RUNNING a concrete check that defines success
   (the test now passes / the command now succeeds / the file now has the required
   content / the service actually responds), and its successful output is shown in the
   transcript. This is the single most important criterion.
2. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
3. The change is targeted at the requirement -- no clearly unrelated files or
   configuration were destroyed or broken as a side effect.

4. EVERY explicit sub-requirement in the task was addressed -- not just the first or
   easiest one. If the task lists multiple things to do, check that each was completed
   and verified, not partially done. Incomplete completion is the top cause of failure.

Grade PASS if criteria 1-4 are met. IMPORTANT: do NOT bounce the agent back merely
because it did not "reproduce first" or its change was not "minimal" -- those are
preferred process, not requirements. If the end state is genuinely verified, PASS. Only
FAIL when the transcript shows the verification was NOT run, FAILED, the required
output/value is provably wrong, or a stated sub-requirement was clearly left unaddressed.
If no automated check is possible for this task, accept a direct inspection of the final
state as verification. When unsure, PASS rather than risk an unproductive loop -- but if a
listed sub-requirement is visibly missing, FAIL with a one-line note naming exactly what
is missing so the agent can finish it.

When you FAIL, you get only ONE more agent cycle before the run ends, so make the bounce
maximally efficient: name the SINGLE most important concrete action still needed (e.g.
"run test X -- it was never executed" or "sub-requirement Y not done"), not a broad
critique. Never demand extra polish, refactoring, or re-verification of things already
shown passing -- that wastes the last cycle and risks a timeout that scores zero.
"""
# [...]
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
   in one sentence -- before editing. Do not edit on a hunch.
3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
   what the task requires; leave the rest of the system identical (no stray files or
   config edits).
4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
   output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

Treat errors as evidence, not noise. Never declare success on assumption; success is the
```

```
re-run check passing in front of you.
"""
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
   in one sentence -- before editing. Do not edit on a hunch.
3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
   what the task requires; leave the rest of the system identical (no stray files or
   config edits).
4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
   output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

Treat errors as evidence, not noise. Never declare success on assumption; success is the
re-run check passing in front of you.

COVER THE WHOLE TASK:
- Re-read the task statement before finishing. If it lists multiple deliverables, sub-
  tasks, or conditions, address EVERY one -- partial completion scores zero. Enumerate the
  requirements to yourself and tick each off with evidence.
- MANDATORY FINAL SELF-CHECK before you conclude: list every explicit requirement from
  the task and, next to each, cite the exact command output or file inspection ALREADY IN
  the transcript that proves it is done. If a requirement already has evidence, do NOT
  re-run it -- just cite it. Only if a requirement lacks concrete evidence, go address it
  now rather than concluding. Doing this yourself avoids being bounced back at the end
  when the time budget is tightest.
- After your fix passes its targeted check, run the broader/existing test suite once if
  one exists, to confirm you did not break something else. Do not skip this when it is
  cheap.

HANDLING FRICTION (to avoid dead ends / errored runs):
- Environment setup may be needed: if a command fails because a package, dependency, or
  build step is missing, install/build it and continue -- do not abandon the task.
- Read tool output carefully. A non-zero exit, traceback, or "command not found" is a
  clue, not a stop sign. Adapt: try an alternate tool, path, flag, or approach.
- When a command produces very long output that buries the signal, re-run it filtered
  (grep for the error/keyword, pipe to `tail`, use `-q`/`--tb=short`, etc.) so the actual
  failure line is visible rather than scrolled away -- do not conclude from truncated noise.
- Never leave the environment in a worse or half-edited state than you found it. If an
  edit made things worse, revert it before trying another approach.
- If you are stuck after a few genuinely different attempts, fall back to the simplest
  approach that satisfies the requirement rather than an elegant one that does not work.

EFFICIENCY & STOPPING RULES:
- You have a limited time/step budget. Act decisively; avoid long detours and repeated
  identical commands. If a command fails the same way twice, change your approach rather
  than retrying it.
- Prefer running the existing test suite or the check the task implies over inventing an
  elaborate new harness. Keep reproduction lightweight.
- Once ALL required outputs pass their checks and are confirmed, STOP. Do not keep
  polishing, refactoring, or exploring -- extra edits risk regressions and wasted budget.
- LAND THE PLANE: a timeout scores ZERO even if the task was essentially done. Run your
  decisive success check EARLY once you have a candidate fix -- not only at the very end --
  so you are never caught mid-verification when time runs out. The moment the core
  solution is verified and every stated requirement has evidence, conclude immediately;
  do not open new investigations "just to be safe."
- If a check is impossible to run in this environment, verify by direct inspection of the
  final state (file contents, command output) and clearly state what you confirmed.
"""
# [...]
    """Thin wrapper that injects the default rubric into invocations.

    RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
    preserves the seed's standard call contract by adding the rubric when the caller
    omitted it, while still allowing a caller (or the evolutionary loop) to override it.
    """
# [...]
    """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
    can't be copied / has no thinking fields.

    Why the grader must not think: the grader is a structured-output classifier (it must emit
    a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
    with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
    the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
    Converse API rejects ("The final block in an assistant message cannot be `thinking`"),
    making the grader fail and silently disabling the rubric gate (the seed's whole point).
    Extended thinking buys nothing for a short verdict classification, so we strip it on the
    grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
    touch neither deepagents nor the shared model builder; the original model is unmutated.
    """
# [...]
def build_agent(model, backend):
    grader_model = _thinking_free(GRADER_MODEL or model)
    agent = create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
```

Listing 13 · Meta-Harness, diff vs. Best-of-3 (B12, Lst. 5) **RR@5 = 78.2**

Meta-Harness. One targeted change on B12: a `WallClockPacingMiddleware` appends the elapsed wall-clock time and escalating convergence guidance to every model call, so the agent banks a verified passing state before the deadline. Protocol, rubric gate, and grader are inherited unchanged.

```
"""
deepagents_wallclock_pacing.py -- parent: deepagents_scientist (the frontier seed).

FAILURE MODE THIS TARGETS (from the frontier leader's own evidence):
Of the leader's 25 failed/errored tasks, EVERY ONE is a `harbor.trial.errors.AgentTimeoutError`
-- the agent was killed at its per-task wall-clock deadline. 20 of those were killed BEFORE the
container reached a passing state (reward 0, reported as "the run errored before completion", no
transcript); only 5 of the total failures were genuine wrong answers. So ~80% of all failures are
the solver running out of wall-clock, not reasoning errors -- and a timeout auto-zeros the trial
with no partial credit. The seed's docstring already names this as its worst cost driver.

WHY IT HAPPENS: the solver gets no sense of elapsed time. Each step it sees only the message
history -- never the clock -- so on long tasks it keeps exploring / re-verifying / polishing and is
cut off before it has banked a verified-passing state. The seed's reproduce-first + rubric-gate
discipline is verification-heavy, which spends exactly the scarce resource.

THE ONE CHANGE: add `WallClockPacingMiddleware`. On every model call it appends to the system
prompt (a) a live elapsed-wall-clock readout and (b) escalating, budget-agnostic convergence
guidance. This gives the frozen solver the missing signal to PACE ITSELF: bank a verified-passing
state early, then refine, instead of leaving verification for the end. Everything else about the
scientist seed -- the reproduce->hypothesize->minimal-change->re-verify protocol, the RubricMiddleware
"done" gate, and the thinking-free grader -- is preserved UNCHANGED, so this is a clean A/B on the
single lever "does time-awareness convert timeout-losses into passes without hurting the passes?"

Design notes making the guidance safe and general (no task-specific knowledge):
  * The guidance is BEHAVIORAL, not a hard stop. Every escalation is conditional on "if the primary
    requirement is NOT yet verified as passing" -- it reprioritizes toward the success condition, it
    never tells the agent to abandon still-needed work. So it cannot make the many tasks that
    already finish comfortably give up early.
  * Escalation keys off ABSOLUTE elapsed minutes, which is self-scaling: a task with a small budget
    dies before the strongest tier ever fires, while a task that has survived longer necessarily has
    a larger budget (and more runway), so it is exactly the one that should feel more urgency. If a
    real per-task budget is ever exposed via DEEPAGENTS_TASK_BUDGET_S, the middleware switches to
    budget-relative thresholds automatically.
  * The clock starts at the first model call (the true start of agent work), read via time.monotonic
    so it is immune to system-clock changes.

Mutable surfaces: the pacing thresholds/copy, the rubric criteria, the grader's max_iterations,
and the system prompt.
"""

import os
import time
# [...]
# AgentMiddleware is langchain's middleware base that every deepagents middleware subclasses
# (deepagents' own summarization/todo middleware import it from here); SystemMessage is langchain
# core. Both live in the same solver venv as deepagents.
from langchain.agents.middleware import AgentMiddleware
from langchain_core.messages import SystemMessage
# [...]
# The default "what done looks like" rubric. RubricMiddleware grades the transcript
# against these each time the agent tries to finish, and continues it with specifics if
# any criterion is unmet. Phrased as end-state checks, not intentions.

# The default "what done looks like" rubric. RubricMiddleware grades the transcript against these
# each time the agent tries to finish, and continues it with specifics if any criterion is unmet.
DEFAULT_RUBRIC = """\
The task is DONE only if every criterion below is satisfied, with evidence visible in the
transcript (a command was actually run and its output shown -- not merely asserted):
1. The failure or requirement was reproduced or pinned down with a concrete command
   before any fix was attempted (a failing test, a reproducing command, or an explicit
   inspection of the current state).
2. The change made is minimal and targeted at the identified cause -- no unrelated files
   or configuration were altered as a side effect.
3. The required end state was verified by RE-RUNNING the same concrete check that
   defined success (the test now passes / the command now succeeds / the file now has the
   required content / the service actually responds), and its successful output is shown.
4. If the task specified particular outputs, file paths, formats, or values, each was
   checked against the requirement rather than assumed.
"""
# [...]
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment. Work
like a scientist, not a guesser.

PROTOCOL:
1. REPRODUCE / PIN DOWN. Before changing anything, turn the goal into a concrete check
   you can run now: reproduce the failure, run the failing test, or inspect the exact
   current state. If no check exists, create the smallest one (a short script or command)
   that fails now and must succeed at the end. Inspect the working directory once first.
2. HYPOTHESIZE. State the specific cause or the specific change the requirement implies --
   in one sentence -- before editing. Do not edit on a hunch.
3. MINIMAL CHANGE. Make the smallest change that addresses the hypothesis. Change only
   what the task requires; leave the rest of the system identical (no stray files or
   config edits).
4. RE-VERIFY. Re-run the EXACT check from step 1. If it does not pass, read the actual
   output, refine the hypothesis, and iterate -- do not retry the same thing blindly.

Treat errors as evidence, not noise. Never declare success on assumption; success is the
re-run check passing in front of you.
"""

# -- Wall-clock pacing (the one new mechanism) ------------------------------------------
# Always-on preamble: states the hard-deadline invariant and the resulting strategy. This alone
# gives the solver the framing it currently lacks; the elapsed readout + tiers add live pressure.
_PACING_ALWAYS = (
    "WALL-CLOCK DISCIPLINE: this task runs under a hard wall-clock deadline enforced by the "
    "harness. If that deadline passes before the required end state exists in the environment, the "
    "task scores ZERO -- no partial credit, no matter how much progress was made or how close you "
    "were. So treat time as your scarcest resource: reach a VERIFIED-passing state on the primary "
    "requirement as EARLY as you can, then refine it. Prefer a simple solution you can confirm now "
    "over an elaborate one you might not finish. Verify incrementally as you go, not all at once at "
```

```
    "the end, so a working state is always banked."
)

# Escalation copy. Each tier is CONDITIONAL on not-yet-passing, so it reprioritizes rather than
# forcing a stop. Absolute-minute thresholds are self-scaling across the suite's budget tiers.
_PACING_LATE = (
    "PACING -- you are past the early phase. If you do NOT yet have the primary requirement in a "
    "verified-passing state, stop broad exploration and refactoring now and take the shortest path "
    "to a confirmed-passing state: make the smallest change that could satisfy the requirement and "
    "re-run the exact check that defines success rather than assuming it works."
)
_PACING_FINAL = (
    "PACING -- you are deep into the budget and later than most tasks run. If the required end state "
    "is NOT yet verified as passing, make that your SOLE objective: write the required outputs to "
    "their exact expected paths, apply your best known-working solution, and re-run ONLY the "
    "specific success check. Do not open new investigations or optional improvements -- a banked "
    "passing state beats an unfinished better one."
)

_LATE_S = 11 * 60      # absolute-elapsed fallback: begin converging
_FINAL_S = 20 * 60     # absolute-elapsed fallback: pass-or-nothing focus
_REL_LATE = 0.55       # budget-relative thresholds when a real budget is known
_REL_FINAL = 0.80


def _fmt_mmss(seconds: float) -> str:
    m, s = divmod(int(seconds), 60)
    return f"{m}m{s:02d}s"


class WallClockPacingMiddleware(AgentMiddleware):
    """Inject live elapsed-time + escalating convergence guidance into every model call.

    The frozen solver otherwise has no sense of wall-clock time, which is the leader's dominant
    failure mode (killed by the per-task deadline before banking a passing state). This does not
    add tools or change control flow; it only augments the system prompt per call, exactly like
    deepagents' own todo/summarization prompt-injection middleware.
    """

    def __init__(self, budget_s: float | None = None) -> None:
        super().__init__()
        self._start: float | None = None
        env = os.environ.get("DEEPAGENTS_TASK_BUDGET_S", "").strip()
        if budget_s is not None:
            self._budget_s: float | None = budget_s
        elif env:
            try:
                self._budget_s = float(env)
            except ValueError:
                self._budget_s = None
        else:
            self._budget_s = None

    def _elapsed(self) -> float:
        # Lazy start: the clock begins at the first model call (true start of agent work).
        if self._start is None:
            self._start = time.monotonic()
        return time.monotonic() - self._start

    def _note(self) -> str:
        elapsed = self._elapsed()
        parts = [_PACING_ALWAYS, f"Elapsed wall-clock so far: {_fmt_mmss(elapsed)}."]
        if self._budget_s:
            frac = elapsed / self._budget_s
            parts.append(
                f"That is about {frac:.0%} of this task's wall-clock budget "
                f"({_fmt_mmss(self._budget_s)})."
            )
            late, final = frac >= _REL_LATE, frac >= _REL_FINAL
        else:
            late, final = elapsed >= _LATE_S, elapsed >= _FINAL_S
        if final:
            parts.append(_PACING_FINAL)
        elif late:
            parts.append(_PACING_LATE)
        return "\n\n".join(parts)

    def _augment(self, request):
        # Mirror deepagents' own append_to_system_message: read the (fresh, per-call) system
        # message as content blocks and append one text block. No accumulation across turns --
        # each call gets a fresh request whose base system message is the original.
        sm = request.system_message
        blocks = list(sm.content_blocks) if sm is not None else []
        text = self._note()
        if blocks:
            text = f"\n\n{text}"
        blocks.append({"type": "text", "text": text})
        return request.override(system_message=SystemMessage(content_blocks=blocks))

    def wrap_model_call(self, request, handler):
        return handler(self._augment(request))

    async def awrap_model_call(self, request, handler):
        return await handler(self._augment(request))
# [...]
    """Thin wrapper that injects the default rubric into invocations.

    RubricMiddleware is dormant unless `rubric` is in the invocation state. This wrapper
    preserves the seed's standard call contract by adding the rubric when the caller
    omitted it, while still allowing a caller (or the evolutionary loop) to override it.
    """
# [...]
# The grader re-reads the transcript and reasons each time the agent would finish, so it
# is the seed's main cost driver: with max_iterations=3 a hard task can pay up to 3 extra
# full-model reasoning passes right at the end -- exactly when wall-clock is scarcest and a
# timeout auto-zeros the trial. We cap it at 2 (one re-grade after a fix). If a lighter
# grader model is available, the evolutionary loop should pass GRADER_MODEL to cut cost
# further; by default the grader inherits the injected solver model.
# The grader re-reads the transcript and reasons each time the agent would finish, so it is the
# seed's main cost driver; capped at 2 (one re-grade after a fix). Inherits the injected solver
# model by default (GRADER_MODEL=None).
# [...]
    """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
```

```
    can't be copied / has no thinking fields.

    Why the grader must not think: the grader is a structured-output classifier (it must emit
    a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
    with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
    the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
    Converse API rejects ("The final block in an assistant message cannot be 'thinking'"),
    making the grader fail and silently disabling the rubric gate (the seed's whole point).
    Extended thinking buys nothing for a short verdict classification, so we strip it on the
    grader only. The SOLVER keeps full thinking. Implemented harness-side via model_copy so we
    touch neither deepagents nor the shared model builder; the original model is unmutated.
    """
    """Return a copy of `model` with extended thinking disabled, or `model` unchanged if it
    can't be copied / has no thinking fields.

    Why the grader must not think: the grader is a structured-output classifier (it must emit
    a GraderResponse tool call). On Bedrock, forcing structured output binds the schema tool
    with tool_choice="any"; for a *thinking* Claude model that gets downgraded to "auto", and
    the grader then sometimes ends a turn on a `thinking` block with no tool call -- which the
    Converse API rejects ("The final block in an assistant message cannot be 'thinking'"),
    making the grader fail and silently disabling the rubric gate. Extended thinking buys nothing
    for a short verdict classification, so we strip it on the grader only. The SOLVER keeps full
    thinking. Implemented harness-side via model_copy; the original model is unmutated.
    """
# [...]
def build_agent(model, backend):
    grader_model = _thinking_free(GRADER_MODEL or model)
    agent = create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS)],
        # ONE change vs the seed: WallClockPacingMiddleware is prepended so the solver sees a live
        # clock + convergence guidance on every call. RubricMiddleware (the "done" gate) is kept
        # exactly as in the seed.
        middleware=[
            WallClockPacingMiddleware(),
            RubricMiddleware(model=grader_model, max_iterations=GRADER_MAX_ITERATIONS),
        ],
    )
    return _RubricAgent(agent, DEFAULT_RUBRIC)
```

D.2 Best Harness Evolved by MILO

Listing 14 is MILO's best harness on Terminal-Bench 2.1 with Opus 4.8 (RR@5 86.1), as a diff against its island's seed. It contains only general search and verification machinery (scaffold floor, deadline governor, independent-verifier gate, anti-stub checks) and no task-specific logic, which is why it transfers to Frontier-Bench (Tab. 3).

Listing 14 · MILO (ours): best harness, diff vs. its island's seed (B11, Lst. 4) RR@5 = 86.1

MILO's best harness (`best_harness_99bd6a17fdd6.py`; island 1, round 5; diff vs. its seed B11). The search deleted B11's one-shot plan gate, rewrote its prompt into a lead prompt with floor, clock, and verifier phases, and added three mechanisms that no baseline reached at once: a *scaffold floor* that produces a minimal valid version of every deliverable before deep work, so a timeout earns partial credit instead of zero; a *deadline governor* that reads the container clock and escalates to a triage directive near the deadline; and an *independent verifier gate* under which the lead cannot stop until a fresh adversarial `verifier` sub-agent signs off, relaxed near the deadline. Full source.

```
"""
deepagents_independent_verifier_monotonic.py -- philosophy: AN INDEPENDENT ADVERSARY
MUST SIGN OFF, NOBODY MISSES THE PLANE, AND THE GRADED STATE IS NEVER EMPTY.

LINEAGE & WHAT THIS BUILDS ON
-----------------------------
This builds on the lineage's BEST admitted node (df23e4454c65, pass=0.837) -- the
independent adversarial verifier (+VerifierGate, the top "what moved the needle" edge,
0.777->0.802) PLUS the deadline-aware governor (the next biggest edge, +0.035->0.837). Both
are kept verbatim because the search history shows they are the two proven wins on this
island. The assigned parent (274b9728fbe9, 0.802) lacks the governor; re-deriving it is
sanctioned ("do not undo a win"), but simply reproducing df23e4454c65 adds no search value,
so this node adds exactly ONE new coherent mechanism on top of that proven base.

THE RESIDUAL FAILURE THIS TARGETS (from this run's report.md + traces)
----------------------------------------------------------------------
After the verifier and the governor, the dominant remaining loss class is NOT semantic
rubber-stamping (the 3 traced fails are that, but fixing the VERIFIER was already tried --
997d4786bea0, prompt +2777 chars, scored -0.044 and was REJECTED -- so that direction is a
known dead end and is NOT retried here). The dominant loss class is TIMEOUT-ZEROS:

  >=14 of 21 failed tasks died at their exact agent_seconds cap with 0 tests passing:
    adaptive-rejection-sampler (900s, 0/9), chess-best-move (900s, 0/1),
    regex-chess (3600s, 0/4), train-fasttext (3600s, 0/2), write-compressor (900s, 0/3),
    gcode-to-text (900s, 0/2), make-doom-for-mips (900s, 0/3), ...

Crucially, MANY of these tasks grade with PARTIAL CREDIT for a correctly-SHAPED deliverable
that exists even before the hard logic is right. E.g. adaptive-rejection-sampler's first
graded checks only require a file named '/app/ars.R' to exist containing a function 'ars'
and a function 'test' -- a 60-second scaffold would have scored points. Instead the agent
explored the full algorithm for the entire budget, hit the cap, and left NOTHING on disk:
a flat 0 where a partial was free.

The governor already says "keep a working version saved" -- but that nudge fires MID-RUN
(past warn_frac). It is too late for a task where the agent spends the first 80% of the
budget exploring and has not yet created any artifact at all. The governor protects the END
of the run; nothing protects the START.

THE ONE NEW PRINCIPLE: MONOTONIC DELIVERABLE
--------------------------------------------
Keep a SCOREABLE artifact in the graded location from the first minutes through the
deadline -- the graded state is never empty. Concretely, a new ScaffoldFloorMiddleware
```

```
    injects, ONCE at the very first model call (so it is the first thing read after the task,
    regardless of prompt length), a FLOOR directive: before any deep work, create a minimal but
    VALID version of EVERY required deliverable (correct path, correct file type, content that
    at least parses / has the required symbols), confirm each is in place, and only THEN iterate
    to make it actually correct. This guarantees that a timeout converts a 0 into whatever
    partial credit the shape earns, instead of leaving nothing.

    This composes with -- and completes -- the deadline governor: ScaffoldFloor floors the graded
    state at t=0; the governor keeps it saved and lands the plane at the deadline. Together they
    are one coherent operating principle (the graded state is monotonically non-empty and never
    left broken), which is why this is ONE change, not a grab-bag.

    The injection reuses the EXACT proven mechanism the governor already uses
    (append_to_system_message inside awrap_model_call) -- a control hook, not just longer prose --
    so it inherits a 100%-tested path. It fires only on the first model call (one-shot) to avoid
    prompt noise on every subsequent turn. If the floor is genuinely impossible to scaffold for
    a task (e.g. the deliverable is a single computed scalar), the directive explicitly says to
    write a best-effort placeholder and move on, so it never wastes time fighting an unscaffold-
    able deliverable.

    Mutable surfaces: the floor wording, budget_sec, warn/triage fractions, the gate ceiling,
    the verifier/lead prompts.
    """

    import os
    # [...]
    from deepagents.middleware._utils import append_to_system_message
    # [...]
    class PlanGateMiddleware(AgentMiddleware):
        """Nudge a plan up front, and force one plan-completion review before stopping.

        Two cheap interventions, both bounded so a finished run is never looped forever:
          - before the first model call, inject a one-time directive to write the todo plan
            BEFORE touching the environment (the Plan-and-Solve "devise a plan first" step);
          - when the model tries to end, inject a one-time directive to walk the plan and
            confirm each step's end state with a concrete check (catch the missing step),
            then jump back to the model.
        """
    class Clock:
        """Shared container-wall-clock tracker (one per build -> concurrency-safe).

        Samples `date +%s` inside the sandbox so it tracks the same clock the harness timeout is
        measured against. `frac()` returns elapsed/budget in [0, inf) from the last sample; it is
        refreshed by the governor before every model call, so the gate (which runs right after a
        model call) reads a fresh-enough value. Before the first sample, frac() is 0.0 (i.e. we
        behave as if there is plenty of time -- fail-open).
        """
    # [...]
        def __init__(self, plan_nudge: bool = True, max_gates: int = 1) -> None:
        def __init__(self, backend, budget_sec: int) -> None:
            self._backend = backend
            self.budget = max(1, int(budget_sec))
            self._start: int | None = None
            self._elapsed: int = 0
            self._readable: bool = True

        async def refresh(self) -> None:
            if not self._readable:
                return
            try:
                res = await self._backend.aexecute("date +%s")
                now = int((res.output or "").strip().split("\n")[0])
            except Exception:
                self._readable = False  # can't read the clock -> stop trying; fail open.
                return
            if self._start is None:
                self._start = now
            self._elapsed = max(0, now - self._start)

        def frac(self) -> float:
            if self._start is None:
                return 0.0
            return self._elapsed / self.budget

        def remaining(self) -> int:
            return max(0, self.budget - self._elapsed)


    class ScaffoldFloorMiddleware(AgentMiddleware):
        """Floor the graded state at t=0: inject a one-shot directive (on the FIRST model call)
        to create a minimal VALID version of every deliverable BEFORE deep work begins.

        Why a middleware and not just prose in the lead prompt: this directive must be SALIENT at
        the very first decision the model makes, before it dives into exploration -- and it must be
        read regardless of how long the system prompt has grown. Appending it to the system
        message on the first call (the same proven append path the governor uses) guarantees that.
        It fires exactly once, so it adds no prompt noise on later turns where the iterate/verify
        discipline takes over.
        """

        def __init__(self) -> None:
    # [...]
            self.plan_nudge = plan_nudge
            self._injected = False

        def _floor_note(self) -> str:
            return (
                "\n\n[floor] BEFORE any deep work or exploration, spend your FIRST few actions "
                "creating a MINIMAL but VALID version of EVERY required deliverable: the correct "
                "file at the correct path, of the correct type/format, containing at least the "
                "structure the task names (required function/symbol names, a parseable skeleton, "
                "an empty-but-well-formed output of the right shape). Confirm each deliverable is "
                "in place with a concrete command. Many tasks award PARTIAL CREDIT for a correctly-"
                "shaped deliverable even before the hard logic is right, and if you run out of "
                "time a shaped deliverable scores while an empty directory scores 0. THEN iterate "
                "to make each deliverable actually correct, keeping it valid at every step (never "
                "leave it un-parseable mid-edit). If a particular deliverable truly cannot be "
                "scaffolded ahead of the computation (e.g. it is a single computed value), write a "
                "best-effort placeholder of the right shape and move on -- do not spend long here; "
                "the floor is a safety net, not the task."
            )

        async def awrap_model_call(self, request, handler):
```

```
        if not self._injected:
            self._injected = True
            request = request.override(
                system_message=append_to_system_message(request.system_message, self._floor_note())
            )
        return await handler(request)

    # Sync path: Harbor runs async; just pass through (mirrors the governor's sync no-op).
    def wrap_model_call(self, request, handler):
        return handler(request)


class DeadlineGovernorMiddleware(AgentMiddleware):
    """Make the wall-clock budget visible and escalate to a triage directive near the cliff."""

    def __init__(self, clock: Clock, warn_frac: float = 0.6, triage_frac: float = 0.82) -> None:
        super().__init__()
        self.clock = clock
        self.warn_frac = warn_frac
        self.triage_frac = triage_frac

    def _status(self) -> str | None:
        frac = self.clock.frac()
        if frac <= 0.0:
            return None  # no reading yet -> stay silent.
        remaining = self.clock.remaining()
        if frac >= self.triage_frac:
            return (
                f"\n\n[clock] ~{remaining}s left of your ~{self.clock.budget}s budget -- you are "
                "NEAR the deadline. LAND THE PLANE: stop exploring and make sure a CORRECT, "
                "VERIFIED result is saved to the graded location RIGHT NOW. One quick "
                "confirmation that the deliverable is in place and correct is enough -- do NOT "
                "start another long re-verification pass. If time remains after the result is "
                "safely saved you may keep improving, but never leave the graded state broken. "
                "A verified partial solution scores; running out of time scores 0."
            )
        if frac >= self.warn_frac:
            return (
                f"\n\n[clock] ~{remaining // 60} min left of your ~{self.clock.budget // 60} min "
                "budget. Prioritize the changes that most affect the graded result and keep a "
                "working version saved; defer nice-to-haves."
            )
        return None  # plenty of time -> silent to avoid prompt noise.

    async def awrap_model_call(self, request, handler):
        await self.clock.refresh()
        note = self._status()
        if note:
            request = request.override(
                system_message=append_to_system_message(request.system_message, note)
            )
        return await handler(request)

    # Sync path can't await the backend clock; Harbor runs async, so just pass through.
    def wrap_model_call(self, request, handler):
        return handler(request)


class VerifierGateMiddleware(AgentMiddleware):
    """Refuse to let the lead stop until an INDEPENDENT verifier has run since the last gate --
    UNLESS the deadline is near, in which case landing a working result wins.

    Evidence-based, not a hollow nudge: when the lead tries to end (an AIMessage with no tool
    calls), the gate scans the transcript for a `task`-tool dispatch naming the verifier that
    occurred AFTER the last gate fired. If found, the stop is allowed. If not, it re-injects a
    directive and jumps back to the model -- up to `max_gates`.

    Deadline awareness: once the clock is past `triage_frac`, the gate STOPS forcing fresh
    verification. On the timed-out tasks the verifier was being dispatched 4-6x; each full
    adversarial re-derivation is expensive, and forcing one more in the final minutes is
    exactly what converts a partial/near-pass into a timeout 0. Near the cliff we trust the
    governor's "land the plane" directive and let the agent finish with what it has.
    """

    def __init__(self, clock: Clock, max_gates: int = 3, verifier_name: str = "verifier",
                 triage_frac: float = 0.82) -> None:
        super().__init__()
        self.clock = clock
# [...]
        self._nudged = False
        self.verifier_name = verifier_name
        self.triage_frac = triage_frac
# [...]

    def before_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._maybe_plan(state)

    async def abefore_model(self, state: dict[str, Any], runtime) -> dict[str, Any] | None:
        return self._maybe_plan(state)

    def _maybe_plan(self, state) -> dict[str, Any] | None:
        if not self.plan_nudge or self._nudged:
            return None
        msgs = state.get("messages") or []
        # Only fire at the very start (before the agent has taken any action).
        if any(isinstance(m, AIMessage) for m in msgs):
            self._nudged = True
            return None
        self._nudged = True
        directive = (
            "Before running anything: understand the task, extract the concrete "
            "requirements (exact files, inputs, constraints, and the verifiable end "
            "state), and use `write_todos` to lay down a complete step-by-step plan. "
            "Then carry out the plan one step at a time, checking each step's result "
            "before moving on."
        )
        return {"messages": [{"role": "user", "content": directive}]}
        self._gate_at = 0  # message index at which we last gated (scopes "since")
# [...]
    def _verifier_dispatched_since(self, msgs, start: int) -> bool:
        """True if the `task` tool was called naming the verifier at/after index `start`."""
        for m in msgs[start:]:
            for tc in (getattr(m, "tool_calls", None) or []):
```

```
                args = tc.get("args", {}) if isinstance(tc, dict) else {}
                blob = (str(tc.get("name", "")) + " " + str(args)).lower()
                if self.verifier_name in blob:
                    return True
        return False

    # [...]
        # Only intervene when the lead is actually trying to STOP (no pending tool calls).
    # [...]
        # Deadline relaxation: near the cliff, do not force another expensive verification
        # round -- let the agent land a working result rather than re-verify into a timeout.
        if self.clock.frac() >= self.triage_frac:
            return None
        # Satisfied: an independent verification happened since we last gated -> allow stop.
        if self._verifier_dispatched_since(msgs, self._gate_at):
            return None
    # [...]
            return None
            return None  # ceiling reached; do not loop forever.
    # [...]
        self._gate_at = len(msgs)  # subsequent verification must be NEWER than this stop.
    # [...]
            "Before you finish: go back through your plan (the todos) and, for EACH step, "
            "confirm with a concrete command that it is actually done and correct -- not "
            "that you intended to do it. Pay special attention to any step you might have "
            "skipped. If anything is incomplete or wrong, fix it. Only stop once every "
            "step is verified against the real environment."
            "STOP -- you have not had this independently verified since your last change. "
            "Before finishing you MUST dispatch the `verifier` subagent via the `task` "
            "tool. Give it the full task requirements and tell it exactly where your "
            "deliverables are; it will re-derive the expected result from scratch and try "
            "to disprove your solution. If it reports NOT SATISFIED, fix the specific "
            "discrepancy it found and dispatch the verifier AGAIN. Only finish once a "
            "fresh verifier run reports SATISFIED with concrete evidence."
    # [...]
SYSTEM_PROMPT = """\
You are an autonomous agent solving a task in a sandboxed Linux environment (work in /app
unless told otherwise). The task is judged by the final state of the environment.

Work in two phases, the Plan-and-Solve way:

1. UNDERSTAND & PLAN. First read the task carefully and extract the concrete
   requirements: which exact files must exist or change, what inputs/parameters and
   constraints apply, and what the verifiable end state is. Inspect the working directory
   once. Then write a complete, ordered plan with `write_todos` -- one todo per required
   step. A missing step is the most common way these tasks fail, so make the plan
   exhaustive.

2. EXECUTE THE PLAN. Carry out the plan one step at a time. After each step, check the
   intermediate result with a concrete command before moving to the next; keep the todo
   list updated. If reality contradicts the plan, revise the plan rather than pushing on.

Before finishing, walk the whole plan again and verify each step's end state for real
(run the test, read the file back, hit the service). Do not rely on your assumption that
a step worked.
"""
LEAD_PROMPT = """\
You are the lead agent solving a task in a sandboxed Linux environment (work in /app unless
told otherwise). The task is judged by the FINAL STATE of the environment against a hidden
test suite you cannot see. Your own opinion that the work is done does not count -- an
independent verifier must confirm it.

Two cross-cutting disciplines govern everything below:
  * MONOTONIC DELIVERABLE -- the graded state must NEVER be empty. From your first few
    actions through the deadline, keep a SCOREABLE artifact in the graded location. Floor it
    early (a minimal valid version of every deliverable), then only ever improve it, never
    leaving it broken mid-edit. Many tasks award partial credit for a correctly-shaped
    deliverable; a timeout with nothing on disk scores 0.
  * WALL-CLOCK BUDGET -- you'll periodically see "[clock]" notes about time remaining; HEED
    them. Spend early time understanding the task and flooring the deliverables; as the
    budget runs low, stop exploring and make sure a correct, VERIFIED result is saved rather
    than risking finishing with nothing.

Work in phases:

0. FLOOR (first). Read the task and inspect the working directory once. Before any deep work,
   create a minimal but VALID version of EVERY required deliverable -- correct path, correct
   file type/format, containing at least the structure the task names (required function/
   symbol names, a parseable skeleton, an empty-but-well-formed output of the right shape).
   Confirm each is in place with a concrete command. This is your safety net; keep it short.

1. UNDERSTAND & PLAN. Extract the concrete, verifiable end state: which exact files must
   exist or change, the exact output format/values/paths required, and any numeric tolerances
   or reference behaviour implied. Then write a complete, ordered plan with `write_todos` --
   one todo per required step. A silently-skipped step is a common failure, so make the plan
   exhaustive.

2. EXECUTE. Carry out the plan one step at a time, checking each step's intermediate result
   with a concrete command before moving on. If reality contradicts the plan, revise the plan
   rather than pushing on. Improve the floored deliverable incrementally and keep it valid at
   every step -- so that if the clock runs out you still have a scoring result in place.

3. INDEPENDENT VERIFICATION (mandatory while time allows). Do NOT trust your own checks -- you
   built the solution with a particular mental model, and that is exactly the model that can
   be wrong. Before finishing, dispatch the `verifier` subagent via the `task` tool. Hand it
   the FULL task requirements (quote them) and tell it precisely where your deliverables live.
   The verifier runs in a fresh context and will independently re-derive the expected result
   and try to FALSIFY your solution. Trust its verdict over your own:
     - If it reports NOT SATISFIED, read the specific discrepancy it found, fix THAT, and
       dispatch the verifier again.
     - Only stop once a fresh verifier run reports SATISFIED with concrete evidence.
   IMPORTANT: verification costs time too. Keep the verifier focused, and once you are NEAR
   the deadline (the "[clock]" triage note fires), do not start another long re-verification
   pass -- confirm the deliverable is in place and finish, rather than timing out at 0.

Beware the "looks-right" trap: a file existing, JSON being well-formed, or your own
reconstruction agreeing with itself does NOT mean the values are correct. Correctness is
about matching the task's true semantics, which the verifier checks independently.
"""


VERIFIER_PROMPT = """\
You are an INDEPENDENT, ADVERSARIAL verification specialist. You did not build this
```

```
solution and you must not trust the lead agent's claims, notes, or self-checks -- they are
frequently confidently wrong in a way that passes their own tests but fails the real one.
Your default stance is GUILTY UNTIL PROVEN INNOCENT: assume the solution is wrong and try
hard to prove it, in your own fresh context.

Method:
1. Read the task requirements you were given as the source of truth. Identify the EXACT
   end state the hidden test would check: precise output values, formats, file paths,
   numeric tolerances, reference behaviour, edge cases.
2. RE-DERIVE the expected result yourself, from the original inputs/data in the
   environment -- NOT from the lead's intermediate artifacts or summaries. Where the task
   implies a reference computation (a known algorithm, a fit, a transform, a decode,
   a model's output on given inputs), reproduce it independently with your own command or
   script and compare to the lead's deliverable value-by-value.
3. Actively look for the ONE discrepancy that would fail the test: wrong values within
   tolerance vs. outside it, off-by-one, wrong units/normalization, wrong subset/coverage,
   wrong field names, extra/missing lines, formatting that won't parse, non-determinism,
   a path or filename that doesn't match, an executable that isn't actually runnable, a
   service that doesn't actually respond.
4. Run real commands to gather evidence. Inspect produced files by reading them back; run
   the deliverable on the actual inputs and check its real output; recompute references.
   Be efficient -- go straight for the highest-risk discrepancy first; you are on the same
   wall-clock budget as the lead, so do not burn it on low-value checks.

Report a single clear verdict:
  - "NOT SATISFIED" -- followed by the SPECIFIC discrepancies you found, each with the
    concrete evidence (the command you ran and what it showed, expected vs. actual). Be
    precise enough that the lead can fix exactly that.
  - "SATISFIED" -- ONLY if you independently reproduced/checked every requirement and found
    no discrepancy, with the evidence shown. If you could not fully verify something, say
    so explicitly rather than passing it.
Do not redesign the solution or implement fixes -- your job is to judge and to expose what
is wrong, not to repair it.
"""


# Budget in seconds. Defaults to the MODAL per-task cap on the timed-out tasks (900s) so the
# triage fires before the most common hard kill; env-overridable so the runner can match a
# different cap. The governor/gate-relaxation no-op gracefully if the clock can't be read.
BUDGET_SEC = int(os.environ.get("HARNESS_WALLCLOCK_BUDGET_SEC", "900"))
TRIAGE_FRAC = 0.82


def _subagents():
    # No `tools` key => the verifier inherits the lead's built-in shell/file tools, so it
    # can genuinely re-derive and inspect the real environment (not just reason about it).
    return [
        {
            "name": "verifier",
            "description": (
                "Independent adversarial verifier: re-derives the expected result from "
                "scratch and tries to disprove the lead's solution. Returns SATISFIED or "
                "NOT SATISFIED with concrete evidence. Dispatch before finishing."
            ),
            "system_prompt": VERIFIER_PROMPT,
        },
    ]
# [...]
def build_agent(model, backend):
    clock = Clock(backend, budget_sec=BUDGET_SEC)
    return create_deep_agent(
        model=model,
        backend=backend,
        system_prompt=SYSTEM_PROMPT,
        middleware=[PlanGateMiddleware(plan_nudge=True, max_gates=1)],
        system_prompt=LEAD_PROMPT,
        subagents=_subagents(),
        middleware=[
            # Floor the graded state at t=0 (one-shot, first model call).
            ScaffoldFloorMiddleware(),
            # Keep it saved and land the plane near the deadline.
            DeadlineGovernorMiddleware(clock, triage_frac=TRIAGE_FRAC),
            # Independent adversarial sign-off before stopping (relaxed near the cliff).
            VerifierGateMiddleware(clock, max_gates=3, triage_frac=TRIAGE_FRAC),
        ],
    )
```

E Instance-Level Scientific Discovery on Open Mathematical Problems

E.1 Records against the Best Known Bounds

Following Sec. 4.3, MILO evolves the harness h of the agent A_h that edits optimizer programs, with Opus 5 as the backbone. Each task τ pairs a starting construction (a frozen rank-2 to rank-6 leaderboard entry; rank 1 is held out) with a parent optimizer, a seed, and a 60 s run budget. A_h writes a revised optimizer π; the verifier V_τ runs π from the starting construction and scores the normalized gain over it, and harness fitness is the mean gain over the 30-task bank, rescored by trusted subprocesses. Three problems yielded a construction below the prior best (Tab. 8; all three are minimization problems, so lower is better). Each was rescored with the arena's own verifier, fetched from its API on 2026-09-09 and again on 2026-09-22, which reproduces every leaderboard score to the last digit; two independent implementations agree to within 2×10^{-16}.

Tab. 4 places these records beside other systems on the arena scale: the arena's baseline entries for AlphaEvolve (Novikov et al., 2025) and TTT-Discover (Yuksekgonul et al., 2026) (TTT-Discover:

Table 8: **Verified EinsteinArena records.** Prior best (the arena leader, unchanged between 2026-09-09 and 2026-09-22) vs. ours, both scored with the arena's verifier. All three are minimization problems (lower is better), and every margin exceeds the arena's minimum improvement for a new #1.

Problem (arena id)	Objective (minimization; lower is better)	AlphaEvolve (arena entry)	Prior best (Sept. 2026)	MILO	Margin	Min. impr.
Erdős minimum overlap (1)	minimize $\max_k \int h(x)(1-h(x+k))\,dx$	0.3809230351	0.3808585749 CodexProLong, 2026-08-15	**0.3808567744**	1.8×10^{-6}	10^{-7}
First autocorrelation ineq. (2)	minimize $\max(f\star f)/(\int f)^2$, $f\geq 0$	1.5052939684	1.5027436492 CodexProLong, 2026-08-14	**1.5027435984**	5.1×10^{-8}	10^{-8}
Third autocorrelation ineq. (4)	minimize $\lvert\max(f\star f)\rvert/(\int f)^2$, f signed	1.4556427954	1.4508066395 Poolish, 2026-08-23	**1.4488860143**	1.9×10^{-3}	10^{-5}

Erdős 0.3808753, first autocorrelation 1.5028629) and EvoX's published third-autocorrelation value of 1.4558 (Liu et al., 2026), whose discretized objective matches the arena's.

E.2 Best Optimizers Discovered by MILO

Each listing excerpts the best optimizer MILO discovered for the record task; line numbers are those of the original program and elided ranges are marked. AlphaEvolve released its final program for the first autocorrelation inequality (Georgiev et al., 2025), shown beside ours. For the other two problems no optimizer code is public: AlphaEvolve and TTT-Discover publish constructions only, EvoX a one-sentence description, and the arena leaders are anonymous, so the comparison there is at the construction level (Tab. 8).

Listing 15 · Erdős minimum overlap — **prior best 0.3808586 → ours 0.3808568**

Best optimizer discovered by MILO (excerpt of the 2,478-line program; construction on 512 grid points): the exact-verifier ratchet that gates every hand-off (`Ratchet.offer`) and the hot μ-continuation loop with a softmax-weighted FFT gradient and projected momentum step (`fast_link`). The hot-to-cold FISTA ladder (`fista_ladder`) and the equioscillation Newton stage (`newton_solve`) that complete the chain are not shown.

```
    def offer(self, x):
        """Project x back to the feasible set, score it exactly, keep if strictly better."""
        z = self.proj(x)
        if self.ker is not None:
            if float(self.ker.cvals(z)[0].max()) > self.braw + self.tol:
                return False
        elif score_raw(z) >= self.braw:
            return False
        z = exactify(z, self.S)
        if z is None:
            return False
        c = np.correlate(z, 1.0 - z, "full")
        m = float(np.max(c))
        if not (m < self.braw):
            return False
        if np.any(z < 0.0) or np.any(z > 1.0) or float(np.sum(z)) != self.S:
            sc = faithful(z)              # paranoia path: literal replica
        else:
            sc = m / z.size * 2.0         # identical arithmetic to the verifier on a feasible z
        if sc < self.bs:
            self.install(z, c, sc)
            return True
        return False
# ... lines 430-445 elided ...
def fast_link(x0, ker, rat, T, deadline, cfg, mu0, mu1, sig0, efw_on=True):
# ... lines 447-485 elided ...
    while it < T:
        now = time.monotonic()
        if now >= deadline:
            break
        frac = it / T
# ... lines 491-498 elided ...
        mun = mu0 * math.exp(lmr * frac)          # mu / n
        mu = mun * n
        step = lip * mun                          # == lip*mu/n, n-independent
        c, Fh, Fb = ker.cv(y)
        cmx = c.max()
# ... lines 504-509 elided ...
        np.subtract(c, cmx, out=wv)
        np.divide(wv, mu, out=wv)
        np.maximum(wv, EXPFLOOR, out=wv)
        np.exp(wv, out=wv)
        wv /= wv.sum()
        sig = sig0 * (1.0 - frac / sigfrac) if (sig0 > 0.0 and frac < sigfrac) else 0.0
        g = ker.wgrad(Fh, Fb, wv, sig)
        g -= float(g.sum()) / n                   # exact tangency to {sum x = S}
        np.multiply(g, step, out=tv)
        np.subtract(y, tv, out=tv)
        proj(tv, out=hn)
        thn = 0.5 * (1.0 + math.sqrt(1.0 + 4.0 * th * th))
        np.subtract(hn, hp, out=y)
        y *= (th - 1.0) / thn
        y += hn
        np.maximum(y, 0.0, out=y)                 # == np.clip(y, 0.0, 1.0, out=y)
        np.minimum(y, 1.0, out=y)
        hn, hp = hp, hn                           # hp is the point just computed
        th = thn
        it += 1
        if it % chk == 0:
            rat.offer(hp)
# ... lines 532-533 elided ...
    rat.offer(hp)
    return hp.copy(), it
```

Listing 16 · First autocorrelation inequality — **prior best 1.50274365 → ours 1.50274360**

AlphaEvolve's released final program (Georgiev et al., 2025) (excerpt of the 208-line program): the time-boxed main loop and its LP-based descent direction. It scores 1.5053; the 1.5032 construction came from a chain of 24 such heuristics.

```python
def search_for_best_sequence() -> float:
  """Function to search for the best coefficient sequence."""
  n = 300  # Increased n to 300
  best_sequence = [1] * n
  curr_sequence = best_sequence.copy()
  best_score = evaluate_sequence(curr_sequence)
  start_time = time.time()
  iteration_count = 0
  while time.time() - start_time < 1000:
    h_function = get_good_direction_to_move_into(curr_sequence)
    if h_function is not None:
      curr_sequence = h_function

    # Evaluate the current sequence and update the best score if necessary
    curr_score = evaluate_sequence(curr_sequence)
    if curr_score < best_score:
      best_score = curr_score
      best_sequence = curr_sequence
      print(f"New best score: {best_score}, elapsed time: {time.time() -
        start_time}")

    iteration_count += 1
    if iteration_count % 10 == 0:  # Increased frequency of perturbation
      time_left = max(0, 1000 - (time.time() - start_time))
      temperature = time_left / 1000.0
      curr_sequence = adaptive_perturb(curr_sequence, iteration_count,
        temperature, best_score, best_sequence)
  return best_sequence

# ... lines 38-122 elided ...
def get_good_direction_to_move_into(sequence: list[float]) -> float:
  """Returns the direction to move into the sequence."""
  n = len(sequence)
  sum_sequence = np.sum(sequence)
  normalized_sequence = [x / sum_sequence for x in sequence]
  rhs = np.max(np.convolve(normalized_sequence, normalized_sequence))
  g_fun = solve_convolution_lp(normalized_sequence, rhs)
  if g_fun is None:
    return None
  sum_g_fun = np.sum(g_fun)

  normalized_g_fun = [x / sum_g_fun for x in g_fun]

  t = 1
  initial_score = evaluate_sequence(sequence)
  momentum = [0.0] * n
  alpha = 0.5  # parameter for armijo condition

  while t > 1e-7:  # Reduced the minimum threshold for t
    # Incorporate momentum into the direction.
    new_direction = [(1 - 0.8) * x + 0.8 * y for x, y in
        zip(normalized_g_fun, momentum)] # Increased momentum weight
    # Normalize the new direction and incorporate it into the new sequence
    norm_dir = np.linalg.norm(new_direction)
    if norm_dir > 0:
      new_direction = [x / norm_dir for x in new_direction]
    temp_sequence =  [(1 - t) * x + t * y for x, y in zip(sequence,
        new_direction)]


    current_score = evaluate_sequence(temp_sequence)

    armijo_condition = current_score <= initial_score + alpha * t * np.sum(
        [x * y for x, y in zip(new_direction, [x - y for x, y in
        zip(sequence, temp_sequence)])]
    )
    if (
        armijo_condition or current_score < initial_score
    ):  # If the armijon condition is satisfied or a lower score is found
      momentum = [
          y - x for x, y in zip(sequence, temp_sequence)
      ]  # Update the momentum with the current direction
      return temp_sequence
    t = cubic_backtracking_line_search(sequence, new_direction,
        initial_score, alpha, t)
  return None
```

Best optimizer discovered by MILO (excerpt of the 251-line program; construction on 65,536 grid points): the annealed log-sum-exp Gauss–Newton step (`optimize`), with its temperature schedule, FFT gradient and Hessian-vector products, truncated conjugate gradient, and a backtracking acceptance that anneals faster when a step is rejected.

```python
def optimize(values: np.ndarray, seed: int, deadline: float) -> np.ndarray:
# ... lines 63-104 elided ...
    while True:
        now = time.monotonic()
        if now >= deadline:
            break
        frac = min(1.0, max(0.0, (now - t_begin) / span))
        temp = t_hi * (t_lo / t_hi) ** frac

        z = (conv_cur - cmax) * (1.0 / temp)
        w = np.exp(z)
        sw = float(w.sum())
        if not np.isfinite(sw) or sw <= 0.0:
            break
        phi = temp * np.log(sw) + cmax
        w *= 1.0 / sw

        spec2 = 2.0 * spec
        spec2c = np.conj(spec2)
        grad = proj(np.fft.irfft(spec2c * np.fft.rfft(w, nfft), nfft)[:n])
        gnorm = float(grad @ grad)
        if not np.isfinite(gnorm) or gnorm <= 0.0:
            break

        inv_t = 1.0 / temp

        def hess(d: np.ndarray) -> np.ndarray:
            dp = proj(d)
            y = np.fft.irfft(spec2 * np.fft.rfft(dp, nfft), nfft)[:length]
            y *= w
            out = np.fft.irfft(spec2c * np.fft.rfft(y, nfft), nfft)[:n]
            return proj(out) * inv_t

        # truncated conjugate gradient on  H d = -grad
        x = np.zeros(n)
        res = -grad
        p = res.copy()
        rs = gnorm
        for _ in range(cg_iters):
            hp = hess(p)
            php = float(p @ hp)
            if not np.isfinite(php) or php <= 0.0:
                break
            alpha = rs / php
            x += alpha * p
            res -= alpha * hp
            rs_new = float(res @ res)
            if not np.isfinite(rs_new) or rs_new <= 1e-8 * gnorm:
                break
            p = res + (rs_new / rs) * p
            rs = rs_new
            if time.monotonic() >= deadline:
                break
        if not np.all(np.isfinite(x)):
            break
        xn = float(np.abs(x).max())
        if xn <= 0.0:
            # no curvature information: fall back to a scaled gradient move
            x = -grad * (u.max() / max(float(np.abs(grad).max()), 1e-300))
        * 1e-3
            xn = float(np.abs(x).max())
            if xn <= 0.0:
                break

        accepted = False
        trial = min(1.0, step * 4.0)
        for _ in range(24):
            if time.monotonic() >= deadline:
                break
            cand = u + trial * x
            np.maximum(cand, 0.0, out=cand)
            s = float(cand.sum())
            if s > 0.0 and np.isfinite(s):
                cand *= 1.0 / s
                sp_c, cc = conv(cand)
                cm = float(cc.max())
                if np.isfinite(cm):
                    phi_c = temp * np.log(np.exp((cc - cm) * inv_t).sum())
        + cm
                    if np.isfinite(phi_c) and phi_c < phi:
                        u, spec, conv_cur, cmax = cand, sp_c, cc, cm
                        if cm < best_max:
                            best_max, best_u = cm, cand.copy()
                        step = trial
                        accepted = True
                        break
            trial *= 0.5
        if not accepted:
            step = max(step * 0.25, 1e-12)
            # nudge the anneal along so a stalled temperature is not
        repeated
            t_hi *= 0.7

    return best_u * total
```

Listing 18 · Third autocorrelation inequality — **prior best 1.4508066 → ours 1.4488860**

Best optimizer discovered by MILO (excerpt of the 574-line program; construction on 25,600 grid points): the log-sum-exp objective and its FFT gradient (`val_grad`), the fixed-iteration melt ladder (`run_path`), and the chain of warm-floor hops that re-melts the best point with jitter and publishes every improvement (`optimize`).

```
    def val_grad(self, f, T):
        dx, N = self.dx, self.N
        s = float(f.sum()) * dx
        if not np.isfinite(s) or s * s < 1e-9:
            return float("inf"), None, float("inf"), None
        sp = np.fft.rfft(f, N)
        u = np.fft.irfft(sp * sp, N)[: self.L]
        c = u * (dx / (s * s))
        cmax = float(c.max())
        if not np.isfinite(cmax):
            return float("inf"), None, float("inf"), sp
        with np.errstate(under="ignore", over="ignore", invalid="ignore"):
            e = np.exp((c - cmax) * (1.0 / T))
            z = float(e.sum())
            if not np.isfinite(z) or z <= 0.0:
                return float("inf"), None, cmax, sp
            val = cmax + T * float(np.log(z))
            w = e * (1.0 / z)
        wsp = np.fft.rfft(w, N)
        g = np.fft.irfft(wsp * np.conj(sp), N)[: self.n] * (2.0 * dx / (s * s))
        g -= 2.0 * dx * float(w @ c) / s
        if not np.all(np.isfinite(g)):
            return float("inf"), None, cmax, sp
        return val, g, cmax, sp
# ... lines 164-255 elided ...
def run_path(prob, f_start, t_hot, t_cold, stages, iters, eps, ratchet, deadline):
    """Geometric cooling t_hot -> t_cold with a FIXED iteration count per stage."""
    f = f_start.copy()
    stages = max(2, int(stages))
    if not (t_hot > t_cold > 0.0):
        return f
    per_stage = max(3, int(round(iters / stages)))
    ratio = (t_cold / t_hot) ** (1.0 / (stages - 1))
    T = t_hot
    for _j in range(stages):
        if time.monotonic() > deadline:
            break
        st = prob.val_grad(f, T)
        if st[1] is None:
            break
        f = lbfgs_stage(prob, f, T, eps, per_stage, ratchet, deadline, st)
        s = float(f.sum()) * prob.dx
        if s == 0.0 or not np.isfinite(s):
            break
        f *= 1.0 / s
        T *= ratio
    return f
# ... lines 278-388 elided ...
    rat = Ratchet(gm, sc_m)
    nfail = 0
    i = 0
    while True:
        rem = chain_deadline - time.monotonic()
        need = (0.55 * hop_iters + fin_iters) * per_iter
        if rem < need:
            break
        alpha, eps = arms[i % len(arms)]
        if ALPHA_RAND:
            alpha = float(np.exp(rng.uniform(np.log(ALPHA_LO), np.log(ALPHA_HI))))
            eps = float(10.0 ** rng.uniform(-6.0, -4.0))
        alpha *= float(np.exp(rng.uniform(-0.09, 0.09)))
        if base - gate.best_score < SAFE_GAIN and nfail >= DURESS_AFTER:
            # Nothing has landed in many hops.  The dead region on this family is
            # the MILD end, so push the melt hotter rather than colder.
            alpha = min(1.05, alpha * 1.15)
            eps = 1.0e-5
        t_hot = min(alpha, 1.05) * sc_m
        if t_hot <= 4.0 * t_warm:
            t_hot = 4.0 * t_warm
        start = rat.best
        s = float(start.sum()) * prob.dx
        if not np.isfinite(s) or s == 0.0:
            break
        start = start * (1.0 / s)
        if HOP_NOISE > 0.0:
            # A hop is DETERMINISTIC given (start, alpha, eps).  Without this
            # jitter a chain that always re-melts its own best point re-draws the
            # same landing and stalls -- measured here as 50 consecutive dead hops.
            # The amplitude follows the melt strength, so mild hops stay mild.
            sig = HOP_NOISE * min(1.0, alpha / 0.7)
            cand = start * np.exp(sig * rng.standard_normal(start.size))
            sc2 = float(cand.sum()) * prob.dx
            if np.isfinite(sc2) and sc2 != 0.0 and np.all(np.isfinite(cand)):
                start = cand * (1.0 / sc2)
        prev = rat.best_score
        hop_stop = min(chain_deadline - fin_iters * per_iter,
                       time.monotonic() + 2.2 * hop_iters * per_iter)
        run_path(prob, start, t_hot, t_warm, hop_stages, hop_iters,
                 eps, rat, hop_stop)
        if rat.best_score < prev:
            nfail = 0
            publish(rat.best)
```