HarnessOpt-Bench: Evaluating LLMs at Harness Optimization

Varun Ursekar[1], **Apaar Shanker**[1], **Yash Maurya**[1], **Shehab Yasser**[1], **Vijay S. Kalmath**[1], **Veronica Chatrath**[1], **Yuan (Emily) Xue**[1]

[1]Scale AI

varun.ursekar@scale.com | https://scale.com/research

Abstract

As LLMs are increasingly deployed within *agentic systems*, their capabilities depend not only on the model weights but also on the *harness*: the prompts, tools, control flow, memory, and orchestration code surrounding them. This makes automated *harness optimization* – the iterative and evaluation-guided improvement of a harness by an AI system – both an important route to improving AI systems and a demanding capability for AI systems themselves. Yet the community lacks a common protocol for measuring how well frontier LLMs perform at this task. We introduce HARNESSOPT-BENCH, a benchmark for end-to-end harness optimization under expensive and stochastic evaluation. An optimizer, an LLM paired with a coding harness, receives a target agent's seed harness, graded evaluation feedback, and a fixed target-evaluation budget. It edits the harness and nominates a final candidate, which is scored by its normalized gain over the seed on a held-out test partition that remains inaccessible throughout search. A trusted execution environment enforces the evaluation boundary, meters target-agent resource use, and preserves candidate versions for audit. We evaluate 5 frontier LLMs as optimizers both under a shared coding harness and under their native harnesses across 4 downstream tasks, over 111 scored runs. Experiment results show that optimizer models separate more than the coding harnesses they act through, native harnesses are not consistently superior, and gains vary substantially across tasks and seed regimes. These results establish harness optimization as a measurable and discriminative capability with large space for improvement.

1 Introduction

Language models are increasingly deployed within *harnesses*: the programs that specify their prompts, tools, control flow, context and memory, and the orchestration code invoking them [26, 25, 27]. The same model can exhibit substantially different capabilities under different harnesses [30, 38], which makes *harness optimization* – the iterative improvement of a harness on a measured outcome under a fixed budget – an increasingly important part of building capable AI systems.

Recent work asks whether AI systems can automate this process, extending a broader line on self-improving agents, agent meta-optimization, and automated research [20, 14, 28]. These approaches differ in the role they assign the model. ShinkaEvolve [11] and GEPA [1] use it as a mutation operator inside a larger evolutionary procedure,

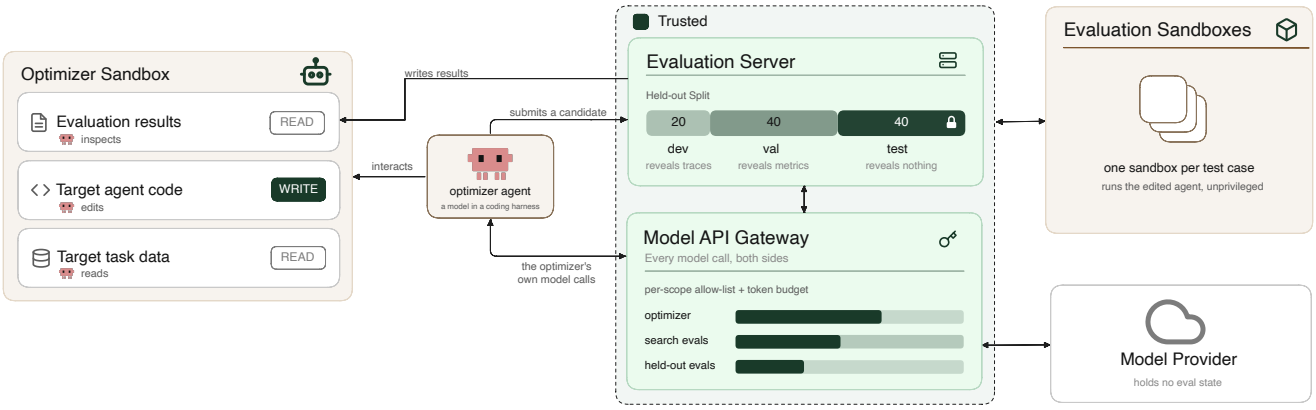

Figure 1. Trusted execution for held-out harness optimization. The optimizer can write only the target agent's harness. It can read the evaluation results and the target task data but not modify them. The optimizer receives per-case development traces and aggregate validation metrics, while test cases and scores remain inaccessible behind the trusted evaluation server. Each candidate is evaluated in isolated, unprivileged sandboxes, and every model call, both the optimizer's own and the evaluations', passes through a gateway that enforces model allow-lists and per-scope budgets. The test partition is evaluated only after the optimizer nominates a final candidate. Because held-out data, provider credentials, and budget enforcement are absent from the optimizer's sandbox, these restrictions are properties of the execution environment rather than instructions the optimizer is expected to follow.

whereas VERO [23] and MetaHarness [12] use coding agents as end-to-end optimizers that edit the harness as a codebase.

Which role a domain calls for depends on the cost of one reliable evaluation, and that cost is what makes harness optimization a test of more than coding ability. A test suite reports cheaply whether a code change is correct; the effect of a harness change must be estimated by running a stochastic agent over many cases, at substantial cost. An optimizer must therefore diagnose failures from incomplete evidence, implement system-level changes, spend a limited evaluation budget, separate real improvement from noise, and decide what to deploy. Where evaluation is cheap, selection over many candidates substitutes for reasoning about any one of them; where it is expensive and noisy, reasoning from prior evidence becomes the capability itself. As the systems being optimized grow more complex, evaluating them only gets more expensive, so the second regime is the one that increasingly matters. Our benchmark specifically targets tasks whose evaluation is itself costly and noisy (fixed-corpus research, terminal use) rather than ones that are cheap to score.

Independent of its place in that self-improvement loop, harness optimization is also long-horizon, plays out over a diverse and complex tool ecosystem rather than a narrow action space, and requires reasoning-driven interpretation of a stochastic system in pursuit of a measured metric rather than production of a single correct step. Each property is already the target of separate benchmarks; a task that combines all three is a demanding test of frontier capability in its own right.

Measuring the capability requires controlling how apparent improvement can arise. Methods are today evaluated with their own target agents, seeds, budgets, disclosure policies, and scoring protocols, so their results conflate the optimizer model, the coding harness it acts through, the target agent, and the protocol. Separating them requires three conditions: the target model, environment, and verifier held fixed; the final evaluation held out throughout search, so improvement reflects generalization rather than fit to a visible score; and a trusted execution boundary that enforces the budget, blocks access to held-out state, and preserves every candidate for audit. Under these conditions three questions become answerable: whether frontier models can be distinguished at this task (**RQ1**), where they fall short (**RQ2**), and how much the optimizer's own coding harness contributes relative to the model

(**RQ3**).

We make the following contributions:

- **A controlled benchmark for harness optimization.** We introduce HARNESSOPT-BENCH, in which an optimizer—an LLM paired with a coding harness—receives a target agent's seed harness, graded development and validation feedback, and a fixed target-evaluation budget. It edits the harness and nominates a final candidate, which is scored by its normalized gain over the seed on a held-out test partition that remains inaccessible throughout search.
- **A trusted, reproducible evaluation protocol.** The suite comprises 4 downstream tasks with pinned seeds, fixed and non-overlapping development, validation, and test splits, graded disclosure, and recorded baselines for both the seed and off-the-shelf harnesses. A trusted execution environment, building on VERO [23], enforces access and target-evaluation budgets, isolates held-out state, meters resource use, and versions every candidate for audit (Figure 1).
- **A controlled evaluation of optimizer models and coding harnesses.** We evaluate five frontier optimizer models under a shared coding harness and their respective native harnesses across 4 downstream tasks, yielding 111 scored optimizer runs, and evaluate two additional coding harnesses on one task. Under this paired design, differences among optimizer models using the shared harness are larger on average than the corresponding differences between shared and native harnesses. Native harnesses provide no consistent advantage, and achievable gains vary substantially across tasks and seed regimes (**RQ1**, **RQ3**).
- **Evidence of remaining capability gaps.** Release-level experiments show that HARNESSOPT-BENCH resolves variation among successive optimizer-model releases on OfficeQA (Section 5.2). Instrumented search trajectories show that broader intervention is associated with greater held-out gain, whereas detailed failure-trace inspection is rarely used and is not positively associated with gain (Section 5.3).

Although our experiments use coding agents as end-to-end optimizers, HARNESSOPT-BENCH is agnostic to optimizer design: any system that operates within the prescribed budget, edits the seed harness, and nominates a final candidate can enter. In this way, HARNESSOPT-BENCH turns harness engineering from an optimizer-specific demonstration into a reproducible evaluation target for measuring and developing systems that improve AI agents.

2 Related Work

Automated code and harness optimization. A long line of work, encompassing OPRO, FunSearch, AlphaEvolve, and ShinkaEvolve, has applied LLMs within larger search scaffolds to discover novel or optimized programs [29, 22, 17, 11]. Existing approaches to *harness optimization* differ along two axes: which parts of the harness they may modify, and the role assigned to the LLM during search. Prompt optimization methods such as DSPy [10] optimize prompts while holding the surrounding program fixed. Meta Context Engineering [31] searches over *skills* and context artifacts in a bi-level procedure. TextGrad [34], Trace [7], and LLM-AutoDiff [32] optimize arbitrary text components in chained workflows by back-propagating textual feedback using LLMs. GEPA [1] optimizes textual artifacts by using reflective feedback to guide an evolutionary process. A second line expands the search space to the agent program itself: STOP [35] recursively improves a scaffolding program. ADAS [8] searches over agent designs represented as code. AFlow [37] performs tree search over code-defined workflows. Darwin Gödel Machine [36], Gödel Agent [33], and SICA [21] study recursive *self-modification*.

Closest to the regime HARNESSOPT-BENCH measures are optimizers that act end-to-end: Ursekar et al. [23] and Lee et al. [12] let a coding agent read the target source, inspect scores and execution traces from prior candidates, and choose what evidence to gather, rather than orchestrating mutations in a fixed search algorithm.

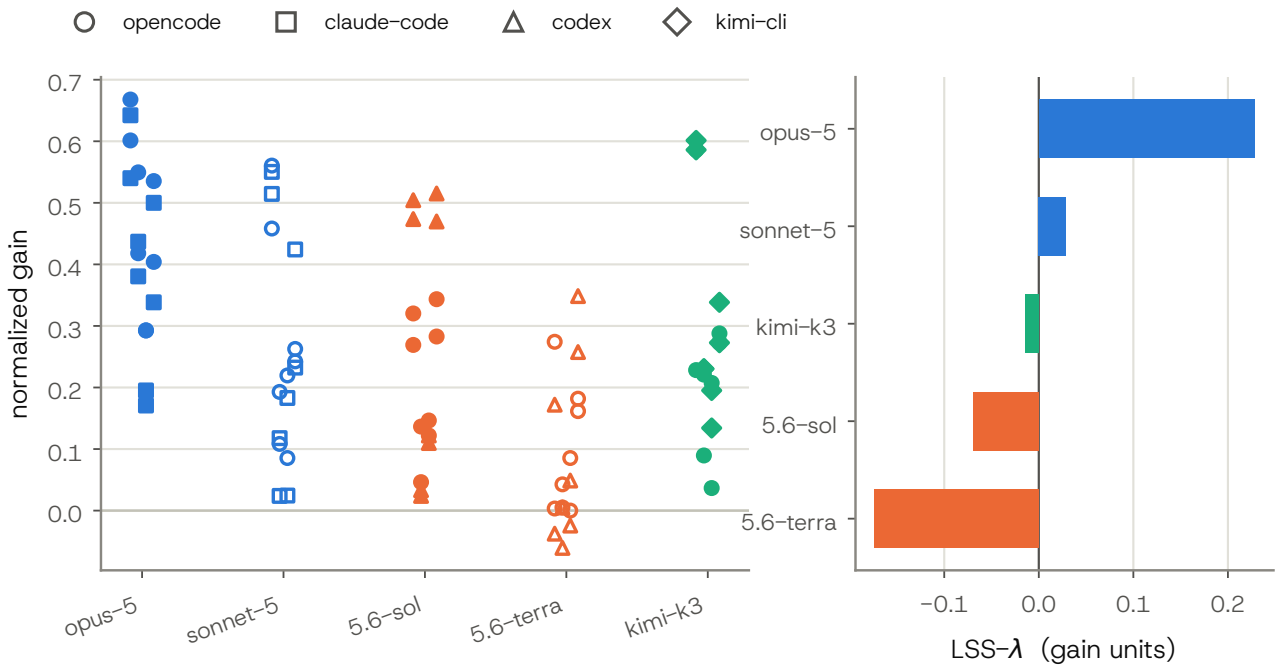

Figure 2. Optimizer models separate more than their coding harnesses. *Left:* normalized gain for every run in the controlled two-harness design. Marker shape denotes the harness; horizontal offsets prevent overlap and carry no task meaning. *Right:* LSS-λ, estimated from the balanced shared-harness grid on the three competent-seed tasks.

Other systems in this regime include Agentic Harness Engineering [13] and HarnessX [5]. Several of these works contribute optimization *methods*, evaluated on method-specific seeds, budgets, search spaces, and scoring protocols, which makes their reported outcomes difficult to compare. HARNESSOPT-BENCH is complementary: it fixes the optimization problem and the evaluation protocol so that optimizer models, harnesses, and search algorithms can be compared on common ground. HARNESSOPT-BENCH builds upon the infrastructure and protocol in VERO and contributes a suite of optimization tasks.

Coding agent benchmarks. Coding agent evaluation has grown from function-level synthesis [4, 2] to repository-scale tasks in real execution environments [9], and tasks requiring iterative optimization, such as machine learning engineering [3] and kernel optimization [19]. HARNESSOPT-BENCH similarly requires coding agents to navigate the target system codebase and iterate on environmental feedback; it differs in what the optimization target is (i.e. a harness), the noisiness of evaluation feedback, and the types of bounds imposed on its search.

3 HARNESSOPT-BENCH: Harness Optimization as a Task

Harness optimization is a constrained, stochastic program-optimization problem. Algorithm 1 summarizes the interaction protocol an optimizer must satisfy; this section defines each element in its general form, then fixes it for this work.

Algorithm 1 The HARNESSOPT-BENCH protocol

Require: H_0; $\theta = (\mathcal{M}, E, V)$; B; π
1: $\mathcal{C} \leftarrow \{H_0\}$
2: **while** $\sum_j c_j \leq B$ and no candidate is nominated **do**
3: **either**: commit $H' \in \mathcal{H}$ and add it to $\mathcal{C}$
4: **or**: choose $H \in \mathcal{C}$ and cases Q
5: $(\hat{s}, \varphi) \leftarrow F_\theta(H, Q)$
6: observe $\pi_{\mathcal{D}}(\hat{s}, \varphi)$
7: **end while**
8: nominate $H^+ \in \mathcal{C}$
9: server evaluates H^+ on $\mathcal{D}^{\text{test}}$
10: **report** g from Eq. 3

3.1 The optimization problem

Candidates and invariants. A candidate harness H is an executable codebase. No semantic partition is imposed between prompts, tool definitions, memory, and control flow. The optimizer may edit, add, or delete files subject to a fixed execution interface and a small set of immutable paths. We write $\mathcal{H}$ for the feasible set and $H_0 \in \mathcal{H}$ for the pinned seed. A task additionally fixes invariants $\theta = (\mathcal{M}, E, V)$: the models $\mathcal{M}$ available to a candidate, the environment $E(x)$ associated with each case x, and the verifier V mapping a completed trajectory to a score in $[0, 1]$. The optimizer may change H but not θ; changing θ defines a different task, not a different candidate. A harness' web access, for example, might be controlled by E.

Evaluation and disclosure. Cases are partitioned into disjoint development, validation, and test sets, $\mathcal{D}^{\text{dev}}$, $\mathcal{D}^{\text{val}}$, and $\mathcal{D}^{\text{test}}$. Executing harness H on case x produces a stochastic trajectory $\tau \sim \text{Rollout}(H, \theta, x)$, to which the verifier assigns score $V(\tau, x)$. We define the expected score on partition $\mathcal{D}$ as

$$\mathcal{E}_\theta(H; \mathcal{D}) = \mathbb{E}_{x \sim \mathcal{D}} \mathbb{E}_{\tau \sim \text{Rollout}(H, \theta, x)} \left[V(\tau, x) \right], \tag{1}$$

and abbreviate $\mathcal{E}_\theta(H; \mathcal{D}^{\text{test}})$ as $\mathcal{E}_\theta(H)$. During search, the optimizer may request an evaluation of H on a subset $Q \subseteq \mathcal{D}^{\text{dev}}$ or $Q \subseteq \mathcal{D}^{\text{val}}$:

$$F_\theta(H, Q) \longrightarrow (\hat{s}, \varphi), \tag{2}$$

where $\hat{s}$ estimates aggregate performance on Q (e.g. sample mean) and φ contains per-case outcomes and execution traces. A partition-specific disclosure policy $\pi_{\mathcal{D}}$ determines which of these outputs the optimizer observes. Development reveals case inputs, per-case outcomes, and traces to support diagnosis; validation reveals only an aggregate score to support selection. The test partition is inaccessible during search and is evaluated by the trusted server only after the optimizer nominates a candidate.

Budget. Each evaluation request j incurs a non-negative cost vector $c_j \in \mathbb{R}^d_{\geq 0}$, and search must satisfy $\sum_j c_j \leq B$ componentwise for a fixed budget vector B, so the optimizer chooses what to evaluate and at what fidelity. In HARNESSOPT-BENCH, the primary components of B are caps of 100 evaluation calls per partition and four full case passes on each of the development and validation partitions, plus a cap on total expendable target-model tokens. The optimizer's own inference is metered for observability but uncapped in this work, though the framework permits capping it.

Optimizer. An optimizer O is any program satisfying the interface of Algorithm 1 that receives H_0, interacts with F_θ under disclosure π and budget B, produces candidates $H_1, \ldots, H_T \in \mathcal{H}$, and nominates a final candidate H^+. In our work, the optimizer is an LLM operating through a coding harness.

Objective. The optimizer maximizes the expected improvement of its nominated candidate over the pinned seed on the held-out partition,

$$\max_{H \in \mathcal{H}} \quad [\mathcal{E}_\theta(H) - \mathcal{E}_\theta(H_0)] \quad \text{s.t.} \quad \sum_j c_j \leq B,$$

Since H_0 is pinned, $\mathcal{E}_\theta(H_0)$ is a constant within a task. Because test disclosure is empty, the optimizer never observes the quantity it maximizes. Raw improvement may not be comparable across tasks whose scoring scales differ, so wherever tasks are compared we use normalized gain

$$g = \frac{\mathcal{E}_\theta(H^+) - \mathcal{E}_\theta(H_0)}{1 - \mathcal{E}_\theta(H_0)}, \tag{3}$$

where negative values indicate a nominated candidate worse than the seed.

3.2 The HARNESSOPT-BENCH suite

Every pinned seed H_0 across HARNESSOPT-BENCH tasks is a small, deliberately untuned Python harness that leaves obvious headroom. Three of the four are competent but naive; the OfficeQA seed, for example, is a $\sim$130-line agent built on the OpenAI API, with three tools, a 24-turn loop, and a generic system prompt. GAIA's is a non-functional stub.

Table 3 in the appendix compares the pinned seeds H_0 to a number of open-source harnesses, showing that they perform comparably or worse than the weakest ones, providing ample headroom for improvement. The environment and the verifier are taken directly from each benchmark; environmental constraints such as limited network access and strict wall clock bounds are used without modification. Across all tasks, $|\mathcal{M}| = 1$, i.e. we use one pinned target model per task; these are listed in Table 3. Target models were chosen so that the seed would land in a measurable mid-range rather than at the floor or the ceiling.

Each task's $\mathcal{E}_\theta(H_0)$ is measured once, averaged over K=3 independent rounds, and pinned for reproducibility; a nominated candidate is likewise scored three times per test case and averaged. Because repeated scores can differ, we summarize this evaluation noise with a task-specific *resolution band* (Section 4); smaller differences are treated as unresolved.

Figure 1 details our execution infrastructure. Each optimizer is run in an isolated sandbox with evaluation results, task data, and the target harness in the filesystem. Similarly, each target agent rollout is performed in an ephemeral sandbox to control for noise introduced by environmental drift.

4 Experimental Setup

Task grid. We evaluate five optimizer models from three developers on all four optimization tasks in our suite: `claude-opus-5`, `claude-sonnet-5`, `gpt-5.6-sol`, `gpt-5.6-terra`, and `kimi-k3`, chosen as the current frontier release of each family. Each model is paired with two coding harnesses: a **shared harness** (`opencode`), held fixed across all models, and the model's **native harness**, namely `claude-code` for the Claude models, `codex` for the GPT models, and `kimi-cli` for Kimi. We refer to each model–harness pair as an *optimizer configuration*, giving 10 configurations in our core grid. Comparing models under `opencode` holds the coding harness fixed; comparing

Model	Optimizer harness *resolution band*	OfficeQA ±0.045	BrowseComp-Plus ±0.066	Terminal-Bench ±0.054	GAIA ±0.035
claude-opus-5	`claude-code`	0.59	0.41	0.18	0.42
claude-opus-5	`opencode`	**0.63**	**0.48**	**0.29**	0.47
claude-sonnet-5	`claude-code`	0.53	0.07	0.10	0.33
claude-sonnet-5	`opencode`	0.51	0.15	0.15	0.25
gpt-5.6-sol	`codex`	0.49	0.03	0.12	**0.49**
gpt-5.6-sol	`opencode`	0.29	0.09	0.13	0.31
gpt-5.6-terra	`codex`	0.07	−0.03	0.01	0.30
gpt-5.6-terra	`opencode`	0.14	0.02	0.04	0.17
kimi-k3	`kimi-cli`	0.59	0.23	0.16	0.31
kimi-k3	`opencode`	0.41	0.16	0.12	0.28

Table 1. Normalized gain for every optimizer on every task, one row per model-by-harness contestant so that a contestant reads across tasks rather than having to be found once per block of Table 2. Each entry is the mean over that contestant's rounds; the best in each column is bolded. Normalization expresses all tasks in common headroom units but does not remove systematic task differences, so emphasis remains within columns rather than across rows. Under each task name is its resolution band: a difference smaller than its own column's band is not a difference. GAIA is set apart because its seed is a non-functional stub with a measured-zero baseline: gain there is the raw held-out score, so it measures building a working agent rather than improving a competent one. `opencode` is the harness common to every model; each other harness is the native one for the model beside it. Generational-ladder rungs are excluded, as in Table 2, and so are the two additional coding harnesses run on GAIA alone. Table 2 gives the same gains with the range each mean is taken over.

a model across `opencode` and its native harness measures sensitivity to scaffold choice. We run two additional harnesses – `goose` and `mini-swe-agent`– across all models on GAIA to compare sensitivity across optimizer harnesses. Finally, for two of the five models – `claude-opus` and `gpt-5.x` – we run earlier releases of the same family using each family's native harness on OfficeQA, supporting the capability ladder we present in Section 5.2. The complete per-task results, including observed run ranges and properties of the generated harnesses, are reported in Appendix A (Table 2).

Scoring protocol. Scoring follows the protocol of Section 3.2: each held-out evaluation is the mean of three attempts per test case, matching the K=3 pooling behind each task's pinned baseline. Each optimizer configuration is run twice; we report ranges in Table 2. The optimizer model's inference is metered but left uncapped, so these results estimate what is achievable when the optimizer model's reasoning is not the scarce resource.

Analysis protocol. Three choices govern the analyses in Section 5.

- **Outcome.** We report the normalized gain g of Eq. 3: the fraction of the headroom above the pinned baseline that an optimizer captured.
- **Measurement resolution.** We estimate evaluation noise by scoring the same candidate twice on the same cases and carry the median discrepancy to the K=3 normalized-gain scale used for held-out scoring. The resulting task-specific *resolution band* is a descriptive threshold: differences smaller than the band are treated as unresolved, not as formal significance-test results. Table 1 reports each task's band.
- **Composite model score.** Normalized gain is defined within a task; normalization places tasks in common headroom units but does not remove systematic differences among them. To obtain a cross-task score for optimizer model m, let g_{mtr} denote its normalized gain on task t in qualifying replicate r under the shared harness,

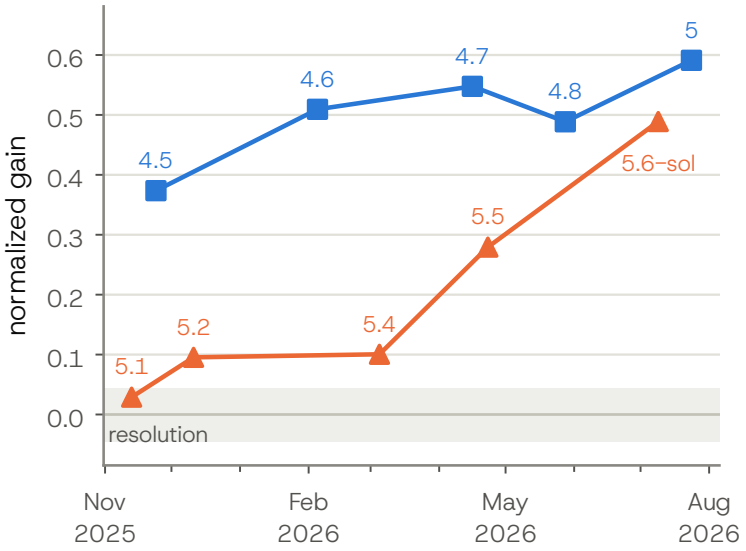

Figure 3. Gain across model releases. Successive Claude Opus (blue) and GPT (orange) releases on OfficeQA, with all factors fixed within each series except the optimizer model. Markers show release means and the shaded region is the OfficeQA resolution band, ± 0.045.

and first average replicates:

$$\bar{g}_{mt} = \frac{1}{|\mathcal{R}_{mt}|} \sum_{r \in \mathcal{R}_{mt}} g_{mtr}. \tag{4}$$

We then decompose these configuration-level gains as

$$\bar{g}_{mt} = \mu + \tau_t + \lambda_m + \varepsilon_{mt},$$
$$\sum_t \tau_t = 0, \quad \sum_m \lambda_m = 0,$$

where τ_t absorbs task-level differences and λ_m is the optimizer model effect. Because the shared-harness grid is balanced,

$$\widehat{\lambda}_m = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \bar{g}_{mt} - \frac{1}{|\mathcal{M}||\mathcal{T}|} \sum_{m' \in \mathcal{M}} \sum_{t \in \mathcal{T}} \bar{g}_{m't}. \tag{5}$$

We define $\mathrm{LSS}_\lambda(m) = \widehat{\lambda}_m$. Thus LSS-$\lambda$ is the model's task-adjusted mean performance relative to the evaluated models' grand mean, expressed in normalized-gain units; higher is better. The primary score uses the shared-harness runs on the tasks with competent seeds. Native-harness runs are excluded to hold the scaffold fixed.

5 Results

Table 1 compares the core experimental configurations across tasks on normalized gain. Appendix A reports the same results in Table 2, together with run ranges and properties of the generated harnesses by task.

5.1 Distinguishing frontier models

Model choice has a larger effect than coding-harness choice. Holding task and harness fixed, changing the optimizer model moves gain by 0.142 on average; holding task and model fixed, changing the harness moves it by 0.079. The

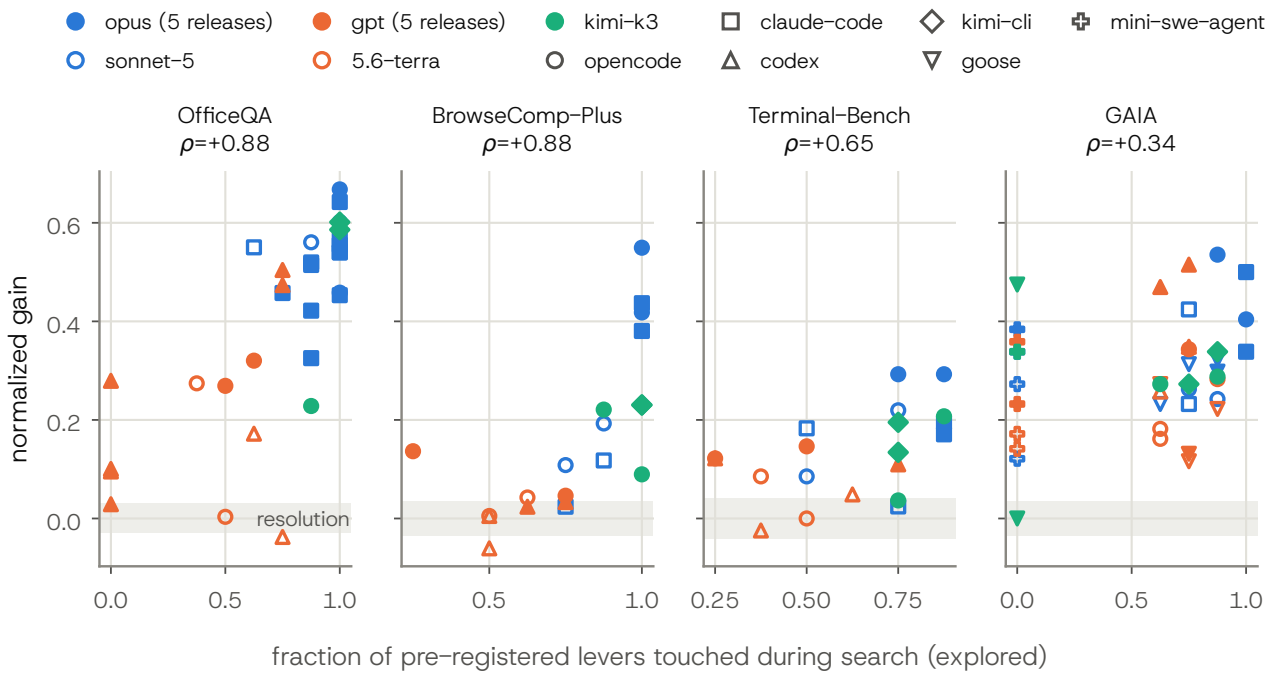

Figure 4. Explored breadth is associated with gain. Normalized gain against the fraction of eight pre-specified harness levers touched during search. Each point averages replicate runs for one optimizer configuration; Spearman ρ is computed separately by task. Lever coverage measures exploration, not changes retained in the submitted harness, and is correlated with overall modification volume.

model contrast is therefore about 1.8× larger. Both exceed the task resolution bands, although the harness contrast does so narrowly.

The extremes separate more clearly than the middle. The strongest configuration captures roughly two thirds of the available OfficeQA headroom and half of the BrowseComp-Plus headroom, while the weakest is unresolved from zero on BrowseComp-Plus and Terminal-Bench. Differences among intermediate configurations are often smaller than their round-to-round variation, supporting tiers rather than a complete ranking. Figure 2 shows the run-level spread and the corresponding task-adjusted model effects.

5.2 Tracking model progress

We test sensitivity to model progress using two OfficeQA release series, holding the target model, seed, budget, and coding harness fixed within each series. Across 5 GPT releases, gain rises monotonically from +0.03 to +0.49, with three of its four steps exceeding the task's resolution band. Across 5 Claude Opus releases, gain ranges from +0.37 to +0.59; gain is non-monotonic, but the first-to-last spread exceeds the task's resolution band.

5.3 Where current optimizers fall short

Trajectory-derived measures characterize how optimizers search; they are not independent measures of held-out capability.

Broader search is associated with greater gain. We identified eight harness levers on OfficeQA before examining the other tasks: prompt, context management, step cap, retry and timeout policy, tool schema, answer extraction,

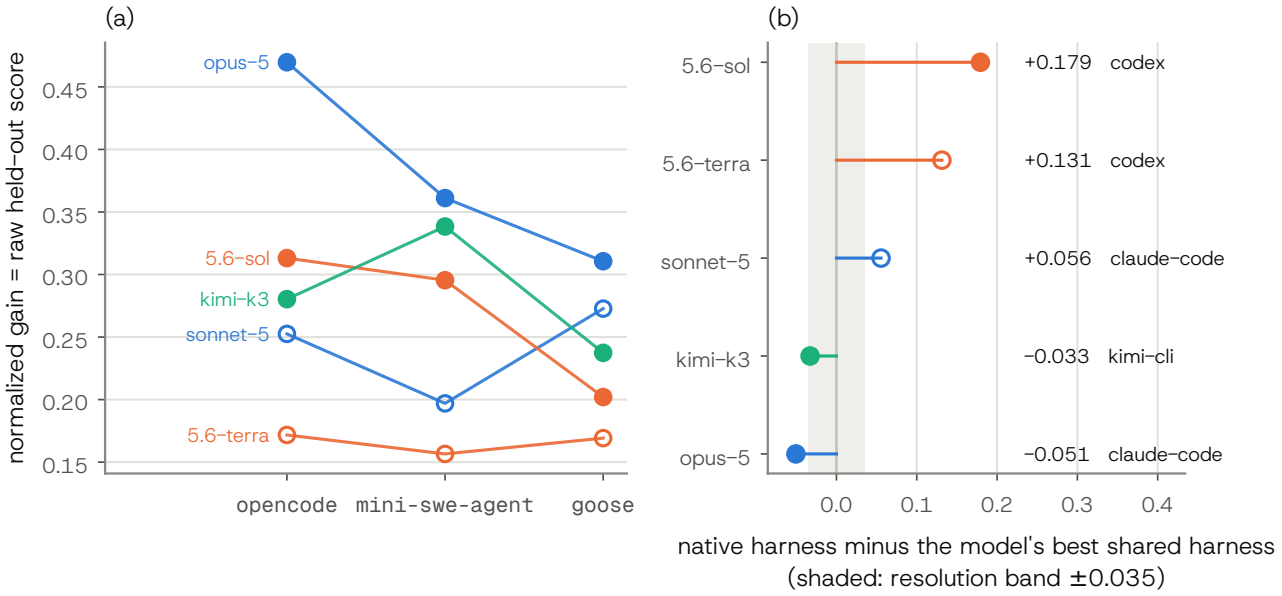

Figure 5. On the one task where the harness really varies, the best harness depends on the model. GAIA is the only task whose optimizer harness has more than two levels: every model ran under `opencode`, `goose` and `mini-swe-agent` as well as its own native harness. Its measured-zero baseline makes gain here the raw held-out score. **(a)** One line per model over a fixed harness order; the lines cross, so no harness holds its rank across models. **(b)** Each model's native harness against its own best shared one, against the task's resolution band. Both GPT models are far better under `codex` ($+0.179$ and $+0.131$); the two Claudes and Kimi sit within a band or two of zero either way.

retrieval policy, and reasoning effort. The fraction touched during search is positively associated with gain on every task, with ρ ranging from $+0.34$ to $+0.88$ (Figure 4). No other process measure we investigated had the same direction on all four tasks.

This is *exploration* rather than final candidate breadth. An optimizer may inspect or modify a lever without retaining the change: one configuration touched three quarters of the levers and made seven edits but shipped the original seed. Breadth is also correlated with total modification volume, so the data do not isolate breadth from search effort.

Trace reading is not associated with higher gain. The share of actions spent reading evaluation output is negatively associated with gain across the four tasks, from -0.31 to -0.64. Optimizers rely mainly on per-case score summaries: detailed trace spans were requested only 16 times by 7 of the 111 cells. This does not establish that diagnosis is unnecessary. For these tasks, per-case summaries may localize failures well enough that reading full traces does not justify its context cost.

Case passes, not evaluation calls, bind. The development and validation partitions each permit 200 evaluation invocations and four full case passes. The median optimizer uses 8 calls (4%) but 82% of its case allowance, and 55 of 100 cells exhaust at least one partition's case budget. Thus the case allowance constrains search, whereas the call cap does not.

Visible validation scores are optimistic. Most cells in Figure 13 fall below the identity line: the submitted candidate's test score is lower than the best validation score observed during search. The held-out partition is

therefore necessary to measure realized gain. The figure cannot distinguish selection-induced overfitting from a validation–test mismatch, so we claim only that the visible best score is optimistic.

5.4 The effect of the optimizer's own harness

Across the 20 model–task pairs evaluated under both conditions, the shared harness wins 11, the native harness wins 9, and 0 are tied. Native harnesses therefore have no consistent advantage. Restricting evaluation to a model's native tooling would not provide a reliable estimate of its harness-optimization ability.

In 11 pairs, the difference exceeds the task's resolution band, but the direction varies by model and task. The aggregate near-tie therefore reflects heterogeneous effects rather than uniformly small ones.

Figure 5 shows where the effect does live, on GAIA — the only task whose harness axis has more than two levels. The harnesses do not hold their rank across models, and the native-harness advantage is concentrated rather than absent: both GPT models are four to five resolution bands better under `codex`, while the two Claudes and Kimi sit within a band or two of zero. Reading only the shared harnesses would miss this, and would suggest that sensitivity to the harness scales with model capability; adding each model's native harness removes that pattern.

6 Conclusion

Harness engineering is becoming a model capability, not merely infrastructure around one. HARNESSOPT-BENCH makes that capability an empirical object: can a model diagnose, modify, and improve an agent as measured via a held-out reward? Current frontier models can, but unevenly. The strongest search broadly, yet their gains remain task-dependent and often too close to support a fine-grained ranking. By releasing HARNESSOPT-BENCH, we make this capability reproducible to measure and concrete to optimize. The next frontier is not merely better agents, but models that reliably make agents better.

Limitations

HARNESSOPT-BENCH is designed to be hack-resistant, not hackproof. The optimizer cannot access the test partition or alter the target model, environment, or verifier, but repeated development and validation feedback may still reward strategies specific to a fixed evaluation. Future versions should introduce per-run jitter in cases, tool behavior, and verifier implementation to distinguish general improvements from exploitation of stable evaluator artifacts.

The seed harness is itself a task-specific prior. Improving a mature agent tests diagnosis and refinement; starting from a stub tests construction. Our suite contains both regimes but does not vary seed complexity systematically. LSS-λ should therefore be interpreted relative to the present distribution of tasks and seeds. A controlled ladder of harness completeness and architectural complexity would reveal how optimizer performance changes with the strength of this prior.

Finally, candidates are restricted to Python and each task uses one pinned target model. Generalization to other languages, runtimes, agent architectures, and target models remains untested. Broader coverage, matched compute conditions, and additional replication are needed before treating HARNESSOPT-BENCH as a comprehensive measure of tool-integrated reasoning.

Ethics Statement

This work studies whether and how well LLMs can improve the code of other LLM agents. Automating agent improvement carries a dual-use tension, since the same capability that repairs and strengthens a benign agent could in principle be turned toward a harmful one. The benchmark's design limits this exposure. The optimization target is always a bounded, benign downstream task (document QA, deep research, multi-step reasoning, or terminal use), the score is the task's own verifier on a held-out split and nothing else, and every optimizer runs inside a trusted harness that meters spend, versions every change for audit, and confines the target to a fixed model behind an allow-list – a concrete precaution given evidence that comparable agent benchmarks can be gamed via evaluator hijacking or judge prompt-injection [24]. Nothing in the setup rewards or requires acquiring new capabilities, tools, or resources beyond editing a fixed agent's code.

Our benchmark reuses existing public datasets and benchmarks under their respective licenses and adds seed agents and split definitions that we release. The downstream corpora are public documents, and we introduce no personal or sensitive data. Because agent evaluation is compute-intensive, we report token usage as a first-class metric to make the cost of this line of work visible, and we keep the search budget denominated in evaluation calls and case-runs so that comparisons do not implicitly favor optimizers with larger compute budgets. Finally, we report the completed evaluation, including repeated runs, with explicit caveats (see the Limitations section) and avoid over-claiming fine-grained model rankings.

References

[1] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, C. Potts, K. Sen, A. Dimakis, I. Stoica, D. Klein, M. Zaharia, and O. Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

[3] J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, L. Weng, and A. Madry. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.07095.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

[5] T. Chen, S. Lu, K. Zhao, W. Meng, H. Teng, T. Li, C. Li, X. Liu, J. Liang, Z. Zhang, Y. Xie, H. Qu, K. Shao, and J. Luan. HarnessX: A composable, adaptive, and evolvable agent harness foundry, 2026. URL https://arxiv.org/abs/2606.14249.

[6] Z. Chen, X. Ma, S. Zhuang, P. Nie, K. Zou, S. Sharifymoghaddam, A. Liu, J. Green, K. Patel, R. Meng, M. Su, Y. Li, H. Hong, X. Shi, X. Liu, H. Oyarhoseini, N. Thakur, C. Zhang, L. Gao, W. Chen, and J. Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2026. URL https://openreview.net/forum?id=jjIKGiGqOo.

[7] C.-A. Cheng, A. Nie, and A. Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2406.16218.

[8] S. Hu, C. Lu, and J. Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

[9] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770.

[10] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.03714.

[11] R. T. Lange, Y. Imajuku, and E. Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349.

[12] Y. Lee, R. Nair, Q. Zhang, K. Lee, O. Khattab, and C. Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

[13] J. Lin, S. Liu, C. Pan, L. Lin, S. Dou, Z. Xi, X. Huang, H. Yan, Z. Han, T. Gui, and Y.-G. Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026. URL https://arxiv.org/abs/2604.25850.

[14] F. Meng, L. Du, Q. Chen, Z. Zhao, H. Lu, M. Hu, and M. Q. Shieh. Rsibench-data: Benchmarking data-centric research for recursive self-improvement, 2026. URL https://arxiv.org/abs/2607.25886.

[15] M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. M. Rytting, R. Marten, Y. Wang, A. Dimakis, A. Konwinski, and L. Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

[16] G. Mialon, C. Fourrier, C. Swift, T. Wolf, Y. LeCun, and T. Scialom. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2311.12983.

[17] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. URL https://arxiv.org/abs/2506.13131.

[18] K. Opsahl-Ong, A. Singhvi, J. Collins, I. Zhou, C. Wang, A. Baheti, O. Oertell, J. Portes, S. Havens, E. Elsen, M. Bendersky, M. Zaharia, and X. Chen. Officeqa pro: An enterprise benchmark for end-to-end grounded reasoning, 2026. URL https://arxiv.org/abs/2603.08655.

[19] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Re, and A. Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=yeoN1iQT1x.

[20] B. Rank, H. Bhatnagar, A. Prabhu, S. Eisenberg, K. Nguyen, M. Bethge, and M. Andriushchenko. Posttrain-bench: Can llm agents automate llm post-training?, 2026. URL https://arxiv.org/abs/2603.08640.

[21] M. Robeyns, M. Szummer, and L. Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025. URL https://arxiv.org/abs/2504.15228.

[22] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

[23] V. Ursekar, A. Shanker, V. Chatrath, Y. Xue, and S. M. Denton. VeRO: A harness for agents to optimize agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=zQzmwG2Nue.

[24] H. Wang, H. Li, Q. Mang, A. Cheung, K. Sen, and D. Song. Do androids dream of breaking the game? systematically auditing ai agent benchmarks with benchjack, 2026. URL https://arxiv.org/abs/2605.12673.

[25] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024. doi: 10.1007/s11704-024-40231-1.

[26] L. Weng. LLM powered autonomous agents. https://lilianweng.github.io/posts/2023-06-23-agent/, 2023.

[27] L. Weng. Harness engineering for self-improvement. *lilianweng.github.io*, July 2026. URL https://lilianweng.github.io/posts/2026-07-04-harness/.

[28] H. Wijk, T. Lin, J. Becker, S. Jawhar, N. Parikh, T. Broadley, L. Chan, M. Chen, J. Clymer, J. Dhyani, E. Ericheva, K. Garcia, B. Goodrich, N. Jurkovic, H. Karnofsky, M. Kinniment, A. Lajko, S. Nix, L. Sato, W. Saunders, M. Taran, B. West, and E. Barnes. Re-bench: Evaluating frontier ai r&d capabilities of language model agents against human experts, 2025. URL https://arxiv.org/abs/2411.15114.

[29] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2309.03409.

[30] Y. Yao, X. Tan, C.-H. Liu, Y. Li, Z. Wang, W. Yu, Z. Tan, Y. Tian, G. Zhao, L. Sun, X. Zhang, and T. Yang. Harness-bench: Measuring harness effects across models in realistic agent workflows, 2026. URL https://arxiv.org/abs/2605.27922.

[31] H. Ye, X. He, V. Arak, H. Dong, and G. Song. Meta context engineering via agentic skill evolution, 2026. URL https://arxiv.org/abs/2601.21557.

[32] L. Yin and Z. Wang. LLM-AutoDiff: Auto-differentiate any LLM workflow. *arXiv preprint arXiv:2501.16673*, 2025. URL https://arxiv.org/abs/2501.16673.

[33] X. Yin, X. Wang, L. Pan, L. Lin, X. Wan, and W. Y. Wang. Gödel agent: A self-referential agent framework for recursive self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. URL https://arxiv.org/abs/2410.04444.

[34] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

[35] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=46Zgqo4QIU.

[36] J. Zhang, S. Hu, C. Lu, R. Lange, and J. Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025. URL https://arxiv.org/abs/2505.22954.

[37] J. Zhang, J. Xiang, Z. Yu, F. Teng, X.-H. Chen, J. Chen, M. Zhuge, X. Cheng, S. Hong, J. Wang, et al. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.10762.

[38] Y. Zhang, J. Wang, Y. Ge, W. Xu, J. Hamm, and C. K. Reddy. Stop comparing LLM agents without disclosing the harness, 2026. URL https://arxiv.org/abs/2605.23950.

A Full optimizer results

Table 2 expands the compact gain comparison in Table 1. It reports the observed range across replicate runs and two properties of the resulting harnesses for every core contestant; the two additional harnesses evaluated on GAIA are included here as well. The generational-ladder runs remain in Figure 3, where release order is the comparison of interest.

Model	Harness	Gain	Levers	Tgt tokens (M)
OfficeQA	resolution band ±0.045 (normalized gain), baseline 0.341			
claude-opus-5	claude-code	0.59 (0.54–0.64)	1.00 (1.00–1.00)	9.97 (3.05–16.89)
claude-opus-5	opencode	0.63 (0.60–0.67)	1.00 (1.00–1.00)	3.77 (3.36–4.18)
claude-sonnet-5	claude-code	0.53 (0.51–0.55)	0.75 (0.62–0.88)	5.61 (4.92–6.29)
claude-sonnet-5	opencode	0.51 (0.46–0.56)	0.94 (0.88–1.00)	2.55 (2.31–2.79)
gpt-5.6-sol	codex	0.49 (0.47–0.50)	0.75 (0.75–0.75)	2.00 (1.59–2.41)
gpt-5.6-sol	opencode	0.29 (0.27–0.32)	0.56 (0.50–0.62)	1.17 (1.03–1.32)
gpt-5.6-terra	codex	0.07 (−0.04–0.17)	0.69 (0.62–0.75)	1.45 (1.31–1.58)
gpt-5.6-terra	opencode	0.14 (0.00–0.27)	0.44 (0.38–0.50)	1.26 (1.15–1.37)
kimi-k3	kimi-cli	0.59 (0.59–0.60)	1.00 (1.00–1.00)	2.77 (2.35–3.18)
kimi-k3	opencode	0.41 (0.23–0.59)	0.94 (0.88–1.00)	3.00 (2.21–3.79)
BrowseComp-Plus	resolution band ±0.066 (normalized gain), baseline 0.462			
claude-opus-5	claude-code	0.41 (0.38–0.44)	1.00 (1.00–1.00)	11.78 (9.95–13.61)
claude-opus-5	opencode	0.48 (0.42–0.55)	1.00 (1.00–1.00)	12.50 (11.13–13.88)
claude-sonnet-5	claude-code	0.07 (0.02–0.12)	0.81 (0.75–0.88)	14.47 (10.58–18.36)
claude-sonnet-5	opencode	0.15 (0.11–0.19)	0.81 (0.75–0.88)	16.10 (13.44–18.77)
gpt-5.6-sol	codex	0.03 (0.02–0.03)	0.69 (0.62–0.75)	4.04 (3.33–4.76)
gpt-5.6-sol	opencode	0.09 (0.05–0.14)	0.50 (0.25–0.75)	7.59 (5.18–10.00)
gpt-5.6-terra	codex	−0.03 (−0.06–0.01)	0.50 (0.50–0.50)	7.77 (7.59–7.96)
gpt-5.6-terra	opencode	0.02 (0.01–0.04)	0.56 (0.50–0.62)	8.47 (8.45–8.50)
kimi-k3	kimi-cli	0.23 (0.23–0.23)	1.00 (1.00–1.00)	15.65 (12.38–18.93)
kimi-k3	opencode	0.16 (0.09–0.22)	0.94 (0.88–1.00)	13.60 (10.71–16.49)
Terminal-Bench	resolution band ±0.054 (normalized gain), baseline 0.241			
claude-opus-5	claude-code	0.18 (0.17–0.20)	0.88 (0.88–0.88)	4.03 (2.71–5.36)
claude-opus-5	opencode	0.29 (0.29–0.29)	0.81 (0.75–0.88)	4.46 (3.86–5.07)
claude-sonnet-5	claude-code	0.10 (0.02–0.18)	0.62 (0.50–0.75)	3.19 (1.75–4.62)
claude-sonnet-5	opencode	0.15 (0.09–0.22)	0.62 (0.50–0.75)	4.81 (2.66–6.95)
gpt-5.6-sol	codex	0.12 (0.11–0.12)	0.50 (0.25–0.75)	1.79 (1.44–2.14)
gpt-5.6-sol	opencode	0.13 (0.12–0.15)	0.38 (0.25–0.50)	1.46 (1.24–1.67)
gpt-5.6-terra	codex	0.01 (−0.02–0.05)	0.50 (0.38–0.62)	1.12 (1.09–1.14)
gpt-5.6-terra	opencode	0.04 (0.00–0.09)	0.44 (0.38–0.50)	1.72 (1.13–2.32)
kimi-k3	kimi-cli	0.16 (0.13–0.20)	0.75 (0.75–0.75)	3.00 (2.47–3.54)
kimi-k3	opencode	0.12 (0.04–0.21)	0.81 (0.75–0.88)	2.59 (0.87–4.32)
GAIA	resolution band ±0.035 (normalized gain), baseline 0.000			
claude-opus-5	claude-code	0.42 (0.34–0.50)	1.00 (1.00–1.00)	1.79 (1.44–2.13)
claude-opus-5	goose	0.31 (0.30–0.32)	0.88 (0.88–0.88)	0.42 (0.30–0.54)
claude-opus-5	mini-swe-agent	0.36 (0.34–0.38)	0.00 (0.00–0.00)	1.42 (0.92–1.91)
claude-opus-5	opencode	0.47 (0.40–0.54)	0.94 (0.88–1.00)	1.68 (0.55–2.80)
claude-sonnet-5	claude-code	0.33 (0.23–0.42)	0.75 (0.75–0.75)	0.58 (0.29–0.88)
claude-sonnet-5	goose	0.27 (0.23–0.31)	0.69 (0.62–0.75)	0.29 (0.23–0.36)
claude-sonnet-5	mini-swe-agent	0.20 (0.12–0.27)	0.00 (0.00–0.00)	0.19 (0.04–0.33)
claude-sonnet-5	opencode	0.25 (0.24–0.26)	0.81 (0.75–0.88)	0.35 (0.22–0.47)
gpt-5.6-sol	codex	0.49 (0.47–0.52)	0.69 (0.62–0.75)	0.55 (0.20–0.91)
gpt-5.6-sol	goose	0.20 (0.13–0.27)	0.69 (0.62–0.75)	0.24 (0.21–0.27)
gpt-5.6-sol	mini-swe-agent	0.30 (0.23–0.36)	0.00 (0.00–0.00)	0.24 (0.13–0.35)
gpt-5.6-sol	opencode	0.31 (0.28–0.34)	0.81 (0.75–0.88)	0.39 (0.19–0.58)
gpt-5.6-terra	codex	0.30 (0.26–0.35)	0.69 (0.62–0.75)	0.42 (0.17–0.67)
gpt-5.6-terra	goose	0.17 (0.12–0.22)	0.81 (0.75–0.88)	0.21 (0.15–0.27)
gpt-5.6-terra	mini-swe-agent	0.16 (0.14–0.17)	0.00 (0.00–0.00)	0.11 (0.08–0.14)
gpt-5.6-terra	opencode	0.17 (0.16–0.18)	0.62 (0.62–0.62)	0.15 (0.15–0.15)
kimi-k3	goose	0.24 (0.00–0.47)	0.00 (0.00–0.00)	0.47 (1 run)
kimi-k3	kimi-cli	0.31 (0.27–0.34)	0.81 (0.75–0.88)	0.37 (0.35–0.39)
kimi-k3	mini-swe-agent	0.34 (1 run)	0.00 (0.00–0.00)	0.86 (1 run)
kimi-k3	opencode	0.28 (0.27–0.29)	0.75 (0.62–0.88)	0.61 (0.59–0.62)

Table 2. Optimizer performance by task, as *mean (observed range)* over a contestant's rounds. The Gain column repeats Table 1; generational-ladder rungs are excluded here and appear in Figure 3. Rows are ordered by model name, so a contestant sits at the same point in every block; blocks differ in length because two coding harnesses ran on one task only. A parenthetical is the range observed across that contestant's rounds, not a confidence interval — two rounds do not estimate dispersion. "–" is not a zero: it is a measure that was not computable for that cell. In 7 entries the contestant shipped the unmodified seed on at least one round, so that round re-measures the seed rather than a failed edit.

B The suite

Table 3 gives each task's design constants — its pinned target model, its exact split, and its seed agent's held-out baseline — together with what the same task scores under each off-the-shelf coding harness on the same held-out partition. Gain is measured against the seed baseline alone (Eq. 3); the off-the-shelf columns are context, not the reference — they say what the task's headroom is worth to an existing harness that nobody optimized.

Scores are means over $K=3$ rounds, with $\pm$ denoting the standard error across round means; timeouts count as zero. GAIA starts from a non-functional stub. Task data come from GAIA [16], OfficeQA Pro [18], BrowseComp-Plus [6], and Terminal-Bench 2.0 [15].

Task	Split (d/v/t)	Seed	Off-the-shelf harness				
			opencode	goose	openhands-sdk	mini-swe-agent	terminus-2
OfficeQA deepseek-v4-flash	49/98/99	0.341 ±0.023	0.727 ±0.010	0.505 ±0.129	0.713 ±0.011	**0.734** ±0.009	0.687 ±0.031
BrowseComp-Plus deepseek-v4-flash	33/66/66	0.462 ±0.020	0.434 ±0.018	0.615 ±0.013	0.657 ±0.035	**0.701** ±0.009	0.439 ±0.023
Terminal-Bench grok-build	17/36/36	0.241 ±0.009	**0.607** ±0.030	0.434 ±0.024	0.393 ±0.013	0.364 ±0.032	0.333 ±0.016
GAIA[§] gpt-5.4-mini	33/66/66	0.000	0.469 ±0.023	0.207 ±0.013	**0.508** ±0.029	0.172 ±0.022	0.202 ±0.005

Table 3. The HARNESSOPT-BENCH suite: each task's pinned target model (second line), its development/validation/test split, its seed agent's held-out score pooled over K=3 rounds, and what the same task scores under each off-the-shelf coding harness. An off-the-shelf column swaps a stock harness in for the seed program and changes nothing else — same dataset, same partition, same rounds, same target model — so the harness is the only variable; best per task in bold. They provide an indication of the seed harness' headroom. Every ± is the standard error of that cell's three round means ($\mathrm{sd}/\sqrt{K}$): re-run variation, not case-to-case spread. [§]this seed is a non-functional stub, so its baseline is a measured zero and gain there is the raw held-out score — the one task on which an optimizer's result and a stock harness's score are directly comparable. Timeouts are scored as zeros rather than dropped per each benchmarks specifications. Task data are drawn from GAIA [16], OfficeQA Pro [18], BrowseComp-Plus [6], and Terminal-Bench 2.0 [15].

C Model effects

Table 4 gives the model effect in gain units, the numeric form of the ordering Figure 2 plots. Two further estimators of the same quantity are computed and agree with it exactly on the ordering, so the ranking does not rest on the additive assumption; only the gain-unit estimator is reported, because it is the one the body quotes and the only one in units a reader can read directly as a fraction of headroom.

	LSS-λ	
Model	gain units	tier
resolution	± 0.058	
claude-opus-5	$+0.228$	1
claude-sonnet-5	$+0.029$	2
kimi-k3	-0.014	2
gpt-5.6-sol	-0.069	2
gpt-5.6-terra	-0.174	3

Table 4. The model effect on the balanced scope. LSS-λ is the model term of an additive fit over task and model, in normalized-gain units: what swapping the optimizer model is worth once the task is accounted for. This is the numeric form of the right panel of Figure 2, from the same call, so the two cannot disagree. *resolution* is the estimator's own measured split-round swing and the tier column cuts at it, so a shared tier means "closer together than re-running the grid moves them" and ranks within a tier are not resolved. Two further estimators of the same quantity — gain standardized within task, and mean within-group rank — are computed but not shown; all three produce the identical ordering (pairwise rank concordance ≥ 1.00), so the ranking here does not depend on the additive assumption. Scope: 15 contestant rows over 3 tasks on opencode alone, balanced. GAIA is excluded because its measured-zero baseline makes its gain a raw held-out score rather than a fraction of headroom. No harness effect is reported: this scope holds one harness, so the fit has no harness factor to estimate — the harness magnitude in Section 5.1 comes from the paired contrast instead.

D Supporting figures

All figures use the same run store and analysis set as the body. Where applicable, points average replicate rounds for one optimizer configuration on one task, and ρ is Spearman correlation within a task. As in the body, GAIA gain is its raw held-out score and is not pooled with the other tasks.

		OfficeQA	BrowseComp-Plus	Terminal-Bench	GAIA‡
Agent behaviour	Prompt	19/20	20/20	20/20	18/20
	Control loop	19/20	17/20	17/20	20/20
	Tool surface	7/20	15/20	10/20	18/20
	Tool implementation	7/20	16/20	10/20	18/20
	Retrieval	0/20	9/20	0/20	2/20
	Submission	8/20	7/20	2/20	18/20
Budgets	Turn budget	17/20	15/20	18/20	11/20
	Output cap	13/20	5/20	7/20	7/20
	Wall-clock budget	7/20	5/20	6/20	12/20
	Context management	7/20	7/20	10/20	7/20
Plumbing	Model client	14/20	11/20	8/20	11/20
	Initialization	4/20	6/20	5/20	17/20
	Environment setup	5/20	7/20	7/20	13/20
Not the agent	Tests	7/20	11/20	10/20	16/20
	Metadata	7/20	9/20	11/20	19/20
	Other	8/20	14/20	9/20	20/20

Figure 6. Edit prevalence by task. Share of balanced-grid runs that made at least one edit in each category; annotations give run counts. ‡GAIA starts from a non-functional stub.

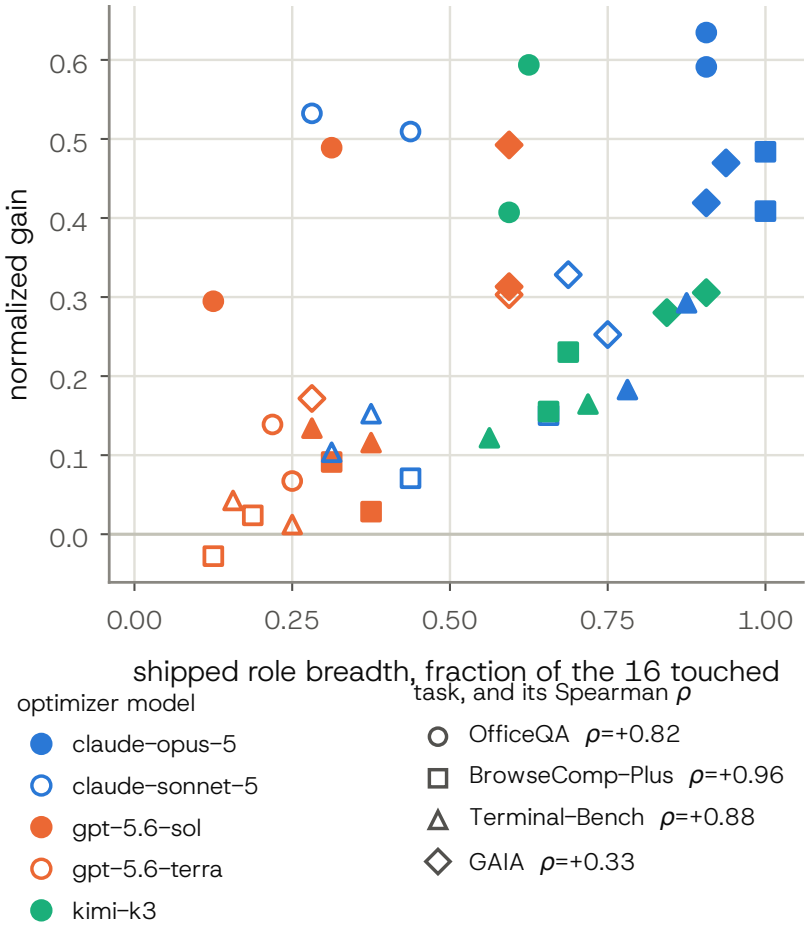

Figure 7. Shipped edit breadth and gain. Normalized gain against the fraction of edit categories present in the submitted harness. Figure 4 reports breadth explored during search.

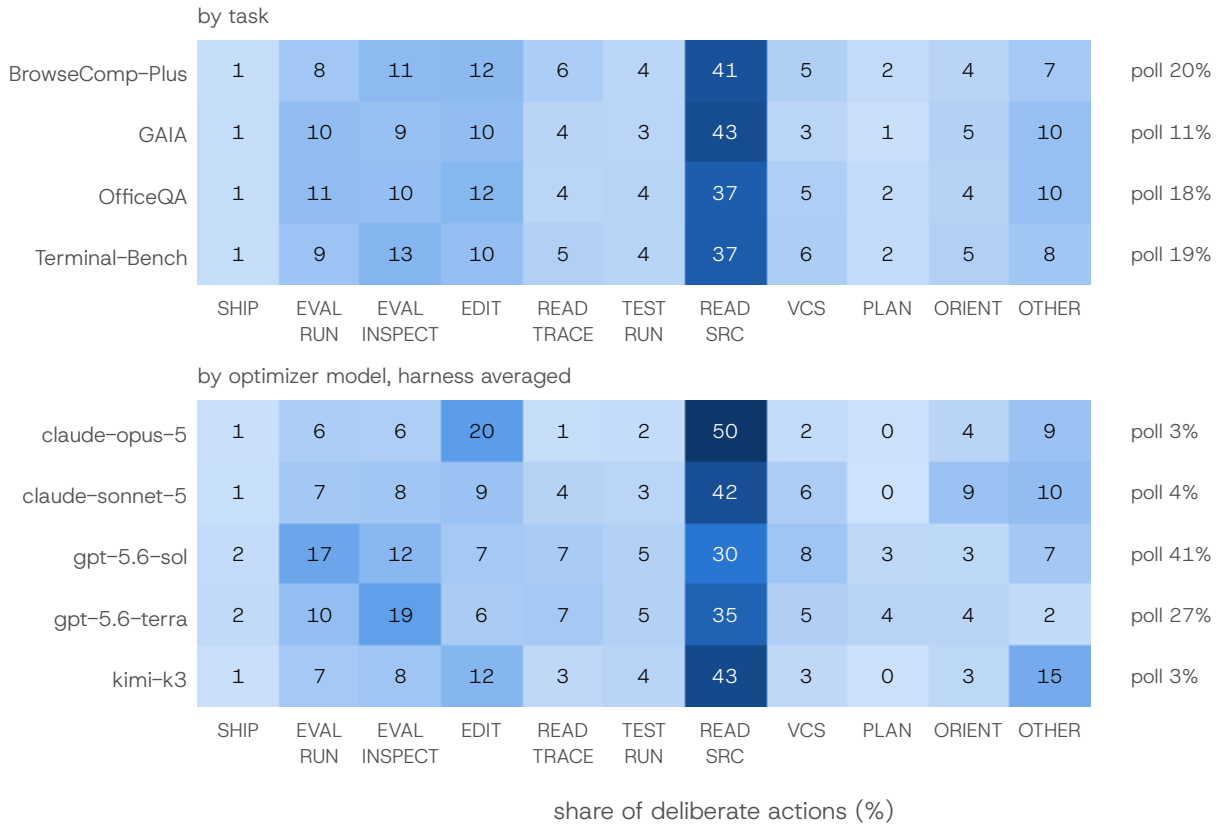

Figure 8. Optimizer action profiles. Share of classified, non-polling actions by task and optimizer model; polling is shown separately.

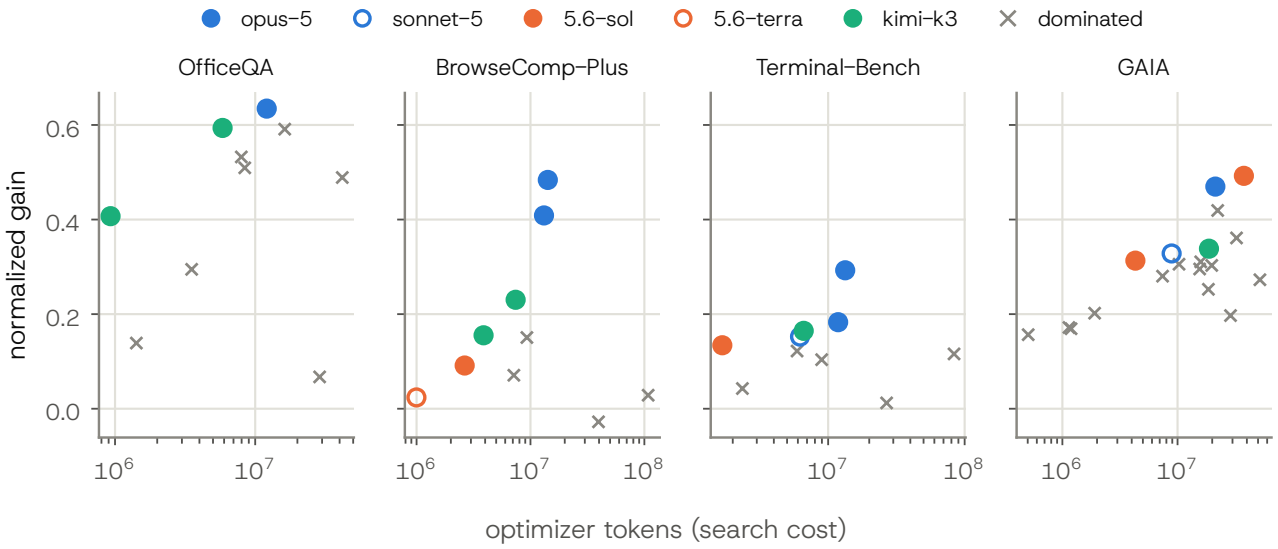

Figure 9. Search cost and gain. Optimizer token use versus normalized gain; filled points form the non-dominated frontier within each task.

Figure 10. Shipped-agent cost and gain. Inference tokens per held-out case versus normalized gain; filled points form the non-dominated frontier within each task.

Figure 11. Modification volume and gain. Lines changed in the target agent versus normalized gain; task-level rank correlations appear above each panel.

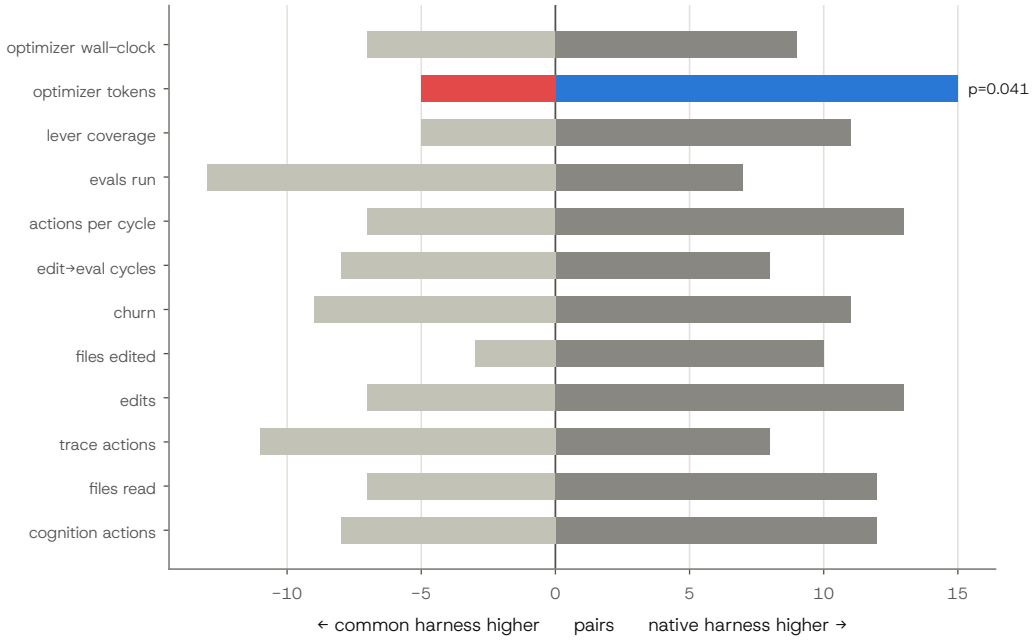

Figure 12. Native versus common optimizer harnesses. Bars count model–task pairs favoring the native harness or opencode; color marks a significant two-sided sign test.

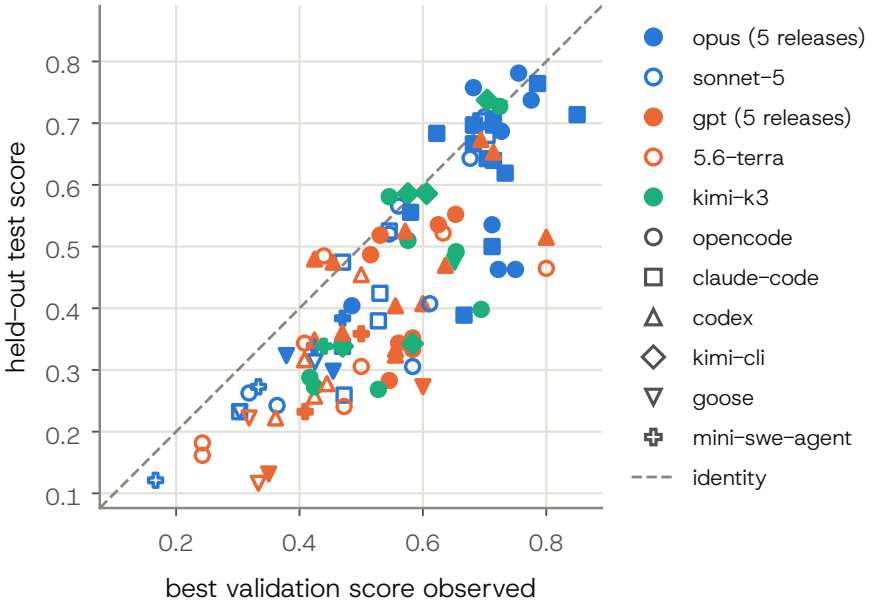

Figure 13. Validation versus held-out performance. Best observed validation score versus the submitted candidate's test score; the diagonal is equality. Cells without a validation measurement are omitted.

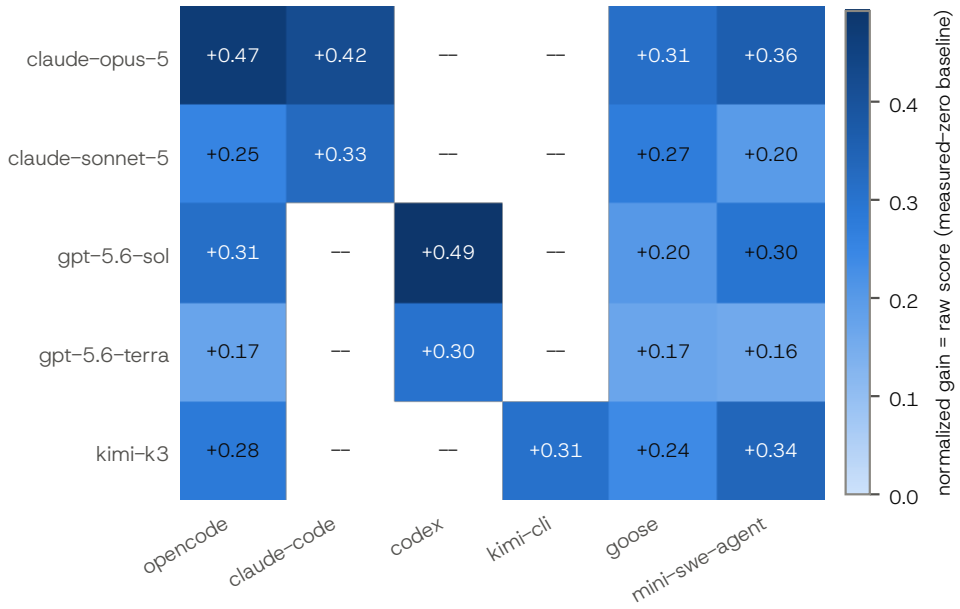

Figure 14. GAIA harness sweep. Mean normalized gain for each evaluated optimizer-model–harness pairing; dashes denote combinations not run.

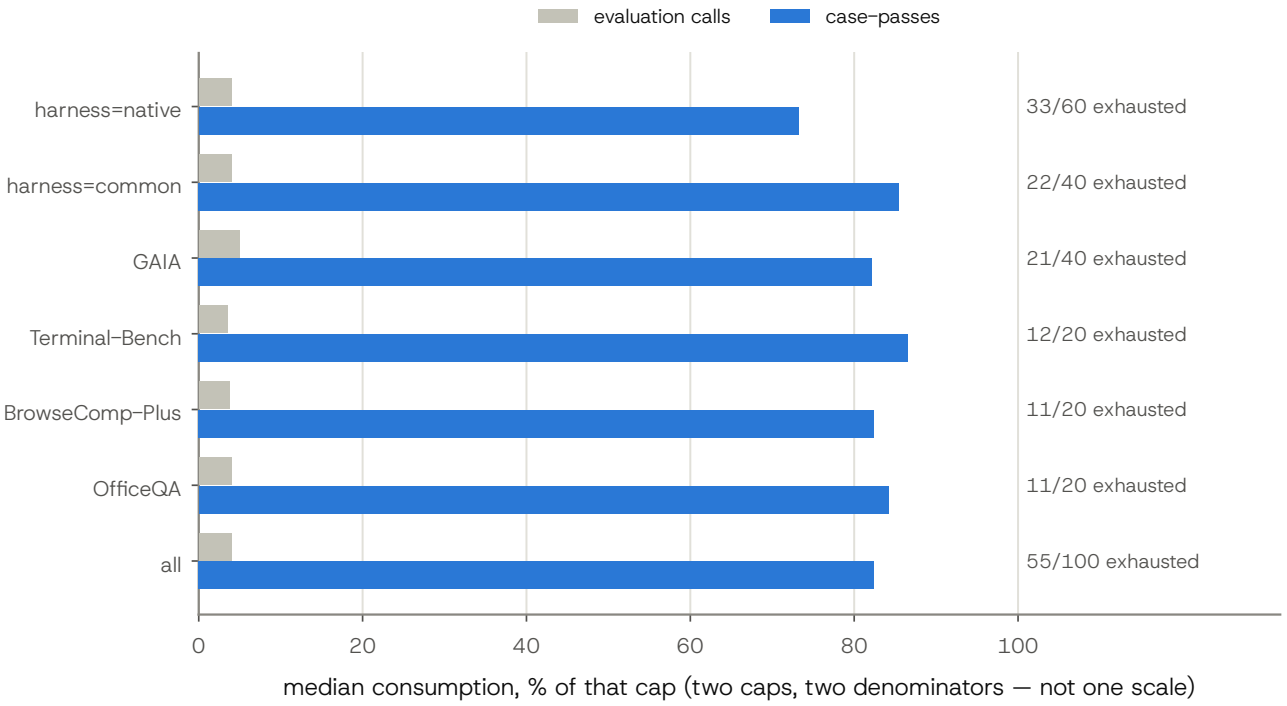

Figure 15. Case-pass budgets, not evaluation calls, bind. Median share of each cap consumed by task and harness scope; annotations count cells that exhausted a case-pass budget.

E Reproducibility

Each task pins its dataset by immutable reference and its split by a committed manifest. Splits are exact $20/40/40$ with no overlap, and the split generator verifies the committed tree byte-for-byte. Baselines are the seed agent's held-out score pooled over $K{=}3$ rounds and are re-pinned whenever a seed commit moves, using a script that reuses the original evaluation path. Every candidate an optimizer produces is an immutable Git commit, dependencies are pinned by lockfile, and each evaluation runs in an isolated sandbox with per-case timeouts taken from the dataset's own declared agent clock. The trusted gateway meters each evaluation's token usage as an input, cached, output, and total split, and stamps request-log records so that per-trial token attribution is reproducible from the run artifacts. Every quantity we report about an optimizer's process is recomputed from its own execution trace and from the evaluator's record of which evaluations it ran, so the analysis is reproducible from the released artifacts without re-running any search. The seed agents, split manifests, build configurations, and evaluation harness are released with the benchmark.