Meta-Harness: End-to-End Optimization of Model Harnesses

Yoonho Lee
Stanford

Roshen Nair
Stanford

Qizheng Zhang
Stanford

Kangwook Lee
KRAFTON

Omar Khattab
MIT

Chelsea Finn
Stanford

Project page w/ interactive demo: `https://yoonholee.com/meta-harness/`
Optimized harness: `https://github.com/stanford-iris-lab/meta-harness-tbench2-artifact`

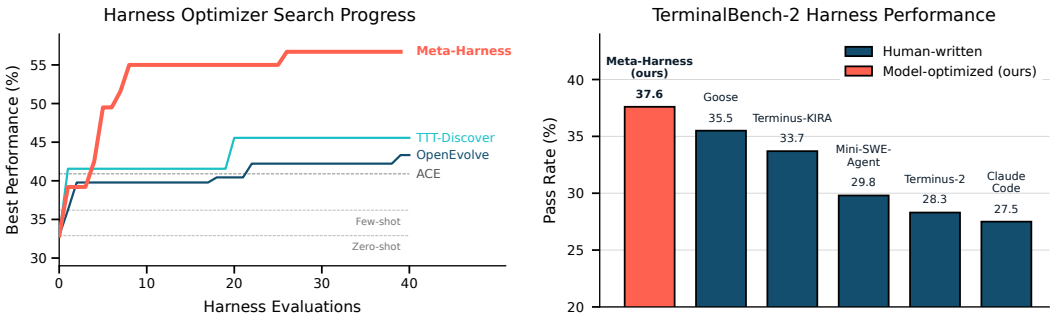

Figure 1: **(Left)** On text classification, Meta-Harness outperforms the best prior hand-designed harnesses (ACE) and existing text optimizers (TTT-Discover, OpenEvolve), matching the next-best method's final accuracy after just 4 evaluations. **(Right)** On TerminalBench-2, Meta-Harness outperforms all reported `Claude Haiku 4.5` harnesses.

Abstract

The performance of large language model (LLM) systems depends not only on model weights, but also on their *harness*: the code that determines what information to store, retrieve, and present to the model. Yet harnesses are still designed largely by hand, and existing text optimizers are poorly matched to this setting because they compress feedback too aggressively: they are memoryless, condition only on scalar scores, or restrict feedback to short templates or summaries. We introduce **Meta-Harness**, an outer-loop system that searches over harness code for LLM applications. It uses an agentic proposer that accesses the source code, scores, and execution traces of all prior candidates through a filesystem. On online text classification, Meta-Harness improves over a state-of-the-art context management system by 7.7 points while using 4× fewer context tokens. On retrieval-augmented math reasoning, a single discovered harness improves accuracy on 200 IMO-level problems by 4.7 points on average across five held-out models. On agentic coding, discovered harnesses surpass the best hand-engineered baselines on TerminalBench-2. Together, these results show that richer access to prior experience can enable automated harness engineering.

1 Introduction

Changing the harness around a fixed large language model (LLM) can produce a 6× performance gap on the same benchmark [46]. The *harness*—the code that determines what to store, retrieve, and show to the model—often matters as much as the model itself. This sensitivity has led to growing interest in **harness engineering**, the practice of refining the code around an LLM to improve the overall system's performance [35; 21; 10; 9]. But despite its importance, harness engineering remains largely manual: practitioners inspect failures,

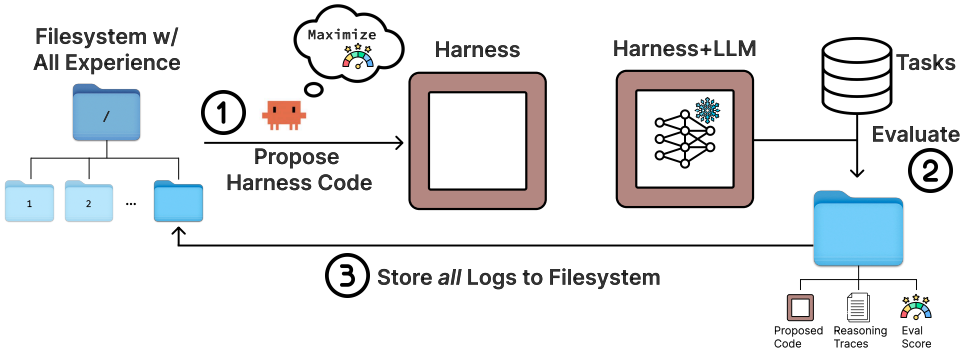

Figure 2: **Meta-Harness search loop. (1)** An agent reads a filesystem containing all prior candidates' source code, execution traces, and scores, and proposes a new harness. **(2)** We evaluate the proposed harness on evaluation tasks. **(3)** All logs (proposed code, reasoning traces, evaluation scores) are stored in the filesystem in a new directory, and the loop repeats.

Method	History	Log content	MTok/iter
OPRO [50]	Window	past (solution, score) pairs	0.002
TextGrad [52]	Last	textual feedback on current artifact	0.015
AlphaEvolve [34]	Window	program database + eval. scores	0.022
GEPA [1]	Summary	reflective feedback from rollout traces	0.008
Feedback Descent [25]	Summary	comparison + textual feedback	0.012
TTT-Discover [53]	Window	prev. solution fragment	0.026
`Meta-Harness`	**Full**	***all*** logs and scores	10.0

Table 1: **Comparison of text optimization methods and their settings.** Each row represents a method collapsed across tasks. `Mtok/iter` is our best estimate of the full context generated from one evaluation of a text artifact in the *largest setting considered in each paper*. This paper considers settings that yield orders-of-magnitude more context per artifact evaluation.

adjust heuristics, and iterate on a small number of designs. In this paper, we ask whether this process itself can be automated.

A natural starting point is recent work on text optimization, since harness engineering also involves iteratively improving text and code artifacts using feedback from prior attempts [37; 38; 34; 25; 1]. However, these methods are poorly matched to harness engineering because they typically operate with short-horizon or heavily compressed feedback: some condition only on the current candidate [30; 50; 52], others rely primarily on scalar scores [34; 12], and others restrict feedback to short templates or LLM-generated summaries [1; 25]. This is a pragmatic scalability choice, not evidence that longer-range dependencies are uninformative. Harnesses act over long horizons: a single choice about what to store, when to retrieve it, or how to present it can affect behavior many reasoning steps later. Compressed feedback often removes the information needed to trace downstream failures to earlier harness decisions. Across the tasks studied by several representative text optimizers, the available context per optimization step ranges from only 100 to 30,000 tokens (Table 1), far below the diagnostic footprint of harness search. More broadly, work on retrieval and memory-augmented language models suggests that useful context should often be accessed adaptively rather than monolithically packed into a single prompt [27; 47; 36; 55].

We address this limitation with **Meta-Harness**, an agentic harness for optimizing harnesses via end-to-end search (Figure 2). Its proposer is a coding agent, i.e., a language-model-based system that can invoke developer tools and modify code. The choice of coding agent (rather than raw LLM) matters because the amount of experience quickly exceeds context limits, so the proposer must decide *what* to inspect and validate edits through direct interaction with the codebase. Its key design choice is to expose **full history** through a *filesystem*, enabling selective diagnosis of raw prior code and execution traces rather than optimization from compressed per-candidate summaries. For every previous candidate harness, the filesystem stores the source code, evaluation scores, and execution traces, which the proposer retrieves via standard operations such as `grep` and `cat` rather than ingesting them as a single prompt. In practice, the proposer reads a median of **82 files per iteration** in our most demanding setting, referencing over 20 prior candidates per step (Appendix A). In the settings we

study, a single evaluation can produce up to 10,000,000 tokens of diagnostic information, roughly three orders of magnitude beyond the largest feedback budgets used in prior text optimization settings (Table 1).

We evaluate Meta-Harness on online text classification, mathematical reasoning, and agentic coding. On online text classification, harnesses discovered by Meta-Harness improve over Agentic Context Engineering (ACE, Zhang et al. [58]) by **7.7 points** while using $4\times$ fewer context tokens, and match the next-best text optimizer's final performance after 60 proposals with only four (Figure 1). On retrieval-augmented math reasoning, a single discovered harness improves accuracy on 200 IMO-level problems by **4.7 points** on average across five held-out models. On TerminalBench-2, the discovered harness **surpasses Terminus-KIRA and ranks #1 among all Haiku 4.5 agents**.

2 Related Work

At a high level, Meta-Harness brings ideas from the broader literature on credit assignment and meta-learning [39; 45; 3; 17; 43; 2] in a new regime enabled by recent advances in coding agents. Rather than updating model weights, the system assigns credit at the harness level: it uses experience from past rollouts to deliberately reason about which steps and components are responsible for failures, then rewrites the external code that governs future behavior. More specifically, the method lies at the intersection of several recent research threads; it is most directly related to work on adaptive access to external context, executable code search, and text optimization.

External memory and adaptive access. Several prior works note the benefits of treating large knowledge sources or long inputs as external resources that a language model accesses adaptively, rather than consuming them in a single pass. Specifically, retrieval-augmented generation [27], interleaved retrieval and reasoning [47], memory-based agents [36], or recursive language models [55] are mechanisms for adaptive access to external context. Meta-Harness uses a similar access pattern, but in the more demanding setting of harness engineering, where the proposer selectively inspects a large external history of code, scores, and execution traces to improve context-management procedures themselves.

Executable code search. Recent methods search over executable code for functions, workflows, or agent designs. Early work proposes using large models as mutation and crossover operators in evolutionary program search [26]. Later methods evolve designated functions within fixed program scaffolds [38], use meta-agents to program new agents from prior discoveries [20], or search over workflow graphs for agentic systems [57]. Another line of work searches over memory designs for continual-learning agents, where memory persists across task streams [56; 49]. In contrast, Meta-Harness searches over domain-specific harnesses, including prompt construction, retrieval, and state update strategies that reset between tasks. Its outer loop is deliberately minimal: instead of relying on a fixed scaffold, an archive of prior discoveries, or a persistent memory mechanism, it gives the proposer unrestricted filesystem access to prior experience. This lets the agent decide what information to inspect and enables search over full harness implementations rather than a predefined space of context-management procedures.

Text optimization methods. Meta-Harness is also closely related to methods such as ProTeGi, TextGrad, OPRO, GEPA, AlphaEvolve/OpenEvolve, and Feedback Descent, which iteratively improve prompts or other text artifacts using feedback from prior attempts [37; 30; 52; 50; 1; 34; 42; 25]. However, these methods are less well suited to harness engineering, where optimization targets a complete executable procedure, and the relevant environmental feedback is distributed across code, scores, and execution traces in a way that is hard to summarize up front. Rather than reacting only to aggregate scores or summaries, the proposer in Meta-Harness can reason over failed examples and their execution traces to propose targeted edits. See Table 1 for a comparison of problem scale considered in those papers and ours, and Figures 1 and 4 for a direct comparison with OpenEvolve, GEPA, and TTT-Discover in our problem setting.

3 Meta-Harness: A Harness for Optimizing Harnesses

This section describes Meta-Harness, our outer-loop procedure for searching over task-specific harnesses. Meta-Harness is built on the idea that harness optimization benefits from allowing a proposer to selectively inspect prior code and execution traces via filesystem access, rather than optimizing from lossy summaries or an additional hand-designed search structure. At a high level, it repeatedly proposes, evaluates, and logs new harnesses.

Meta-Harness is itself a harness in the broad sense (hence the name), since it determines what information the proposer model sees during search. Unless otherwise noted, we use *harness* to refer to the task-specific programs being optimized.

Objective. A harness is a stateful program that wraps a language model and determines what context the model sees at each step. The goal is simple: find the harness that makes the underlying model perform best on the target task distribution. Formally, let M denote a fixed language model and $\mathcal{X}$ a task distribution. For a harness H and task instance $x \sim \mathcal{X}$, we execute a rollout trajectory $\tau \sim p_M(H, x)$. The harness constructs prompts for M, the model responds, and the harness updates its state after each interaction. A task-specific reward function $r(\tau, x)$ scores the trajectory. The objective of harness optimization is to **find the harness that maximizes the expected final reward**:

$$H^* = \arg\max_H \mathbb{E}_{x\sim\mathcal{X},\tau\sim p_M(H,x)}\, r(\tau, x),$$

When multiple objectives are relevant (e.g., accuracy and context cost), we evaluate candidates under Pareto dominance and report the resulting frontier. In practice, this search has traditionally been carried out by human engineers and researchers, who iteratively refine prompts, context-management rules, and tool-use logic by hand.

Meta-Harness search loop. Meta-Harness uses a single coding-agent proposer with access to a growing filesystem $\mathcal{D}$ that serves as its feedback channel[1]. Here, a *coding agent* is a language-model-based system that can invoke developer tools and modify code. Unlike prior systems that externalize the improvement logic in a hand-designed search loop, Meta-Harness delegates diagnosis and proposal to the coding agent itself: it decides which prior artifacts to inspect, which failure modes to address, and whether to make a local edit or a more substantial rewrite. Equivalently, the proposer is not a raw next-token model operating on a fixed prompt assembled by the outer loop; it is an agent that retrieves information, navigates prior artifacts, and edits code as part of the search itself. Each evaluated harness contributes a directory containing its source code, scores, and execution traces (such as prompts, tool calls, model outputs, and state updates). The filesystem is typically far larger than the proposer's context window, so the proposer queries it through terminal tools such as `grep` and `cat` rather than ingesting it as a single prompt. At each iteration, the proposer first inspects prior code, scores, and execution traces, then reasons about likely failure modes before generating a new harness.

Meta-Harness maintains a population $\mathcal{H}$ and a Pareto frontier over evaluated harnesses, but imposes no parent-selection rule: the proposer is free to inspect *any* prior harness and its execution trace when proposing new ones. We run evolution for a fixed number of iterations and perform a final test-set evaluation on the Pareto frontier. This simplicity is deliberate: by leaving diagnosis and edit decisions to the proposer rather than hard-coding search heuristics, Meta-Harness can improve automatically as coding agents become more capable. The proposer never sees test-set results; its only feedback comes from the **search set**, the subset of task instances used to evaluate candidate harnesses during search and generate the feedback signal for improvement, and from execution traces logged during those search runs.

Advantages of code-space search. Harness optimization occurs in code space, where small changes to retrieval, memory, or prompt-construction logic can affect behavior many steps later, making local search heuristics poorly matched to the problem. By inspecting execution

[1] Based on earlier exploration, we think this workflow only became practical recently, following major improvements in coding-agent capabilities around early 2026.

Algorithm 1 Meta-Harness outer loop over harnesses

Input: tasks $\mathcal{X}$, LLM M, proposer P, iterations N
Initialize: population $\mathcal{H}$ ▷ Initial set of valid harnesses
Initialize: filesystem $\mathcal{D} \leftarrow \emptyset$ ▷ stores code, scores, traces
for $H \in \mathcal{H}$ **do**
    $E_H \leftarrow \text{Evaluate}(H, M, \mathcal{X})$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(H, E_H)\}$
for $t = 1 \dots N$ **do**
    Proposer P queries filesystem $\mathcal{D}$ ▷ inspects prior harnesses and scores
    Proposer P proposes k new harnesses $\{H_1, \dots, H_k\}$
    for H in $\{H_1, \dots, H_k\}$ **do**
        if H passes interface validation **then**
            $\mathcal{D} \leftarrow \mathcal{D} \cup \{(H, \textsc{Evaluate}(H, M, \mathcal{X}))\}$
return Pareto frontier of harnesses stored in $\mathcal{D}$

traces, the proposer can often infer *why* a harness failed and which earlier design choices likely contributed to the failure, not just *that* it failed, as illustrated by the search trajectories in Appendices A and A.2. There, we see that the proposer reads broadly across prior code and logs, then uses those traces to identify confounded edits, isolate likely causal changes, and shift toward safer modifications after repeated regressions. The proposer can therefore modify the harness at the level of algorithmic structure, ranging from changes to retrieval, memory, or prompt-construction logic to full program rewrites, rather than filling in templates or applying predefined mutation operators. In practice, it often starts from a strong prior harness, but this is an emergent strategy rather than a hard-coded rule. Although the search space is large, representing harnesses as programs provides a natural regularization bias: coding models tend to propose coherent algorithms rather than brittle, hard-coded solutions, which biases the search toward reusable context-management procedures. This action space is closely aligned with the read–write–execute workflows on which frontier coding assistants are trained.

Practical implementation. In our experiments, each harness is a single-file Python program that modifies task-specific prompting, retrieval, memory, and orchestration logic. In our experiments, the proposer P is Claude Code [4] with `Opus-4.6`. The proposer is guided by a minimal domain-specific skill that describes where to write new harnesses, how to inspect previous harnesses and their execution traces, and what files it can and cannot modify. The base model M varies by domain and is always frozen; see Section 4 for details. In our experiments, a typical run evaluates roughly 60 harnesses over 20 iterations. We provide additional tips for implementing Meta-Harness in a new domain in Appendix D.

4 Experiments

We evaluate Meta-Harness on three task domains: online text classification, math reasoning, and agentic coding. In each domain, we compare harnesses discovered by our search against domain-appropriate baselines using the standard evaluation metric. Please refer to each subsection for the precise experimental setup.

We compare against two main classes of methods. **(1) Human-designed strategies**: these are hand-crafted harnesses for each domain, representing the current state of the art in context construction. We describe these baselines in the corresponding subsections. **(2) Program-search methods:** these methods search over candidate harnesses using feedback and reward signals, but are designed for smaller-scale settings than harness engineering.

4.1 Online Text Classification

We follow the online text classification setup of Zhang et al. [58]; Ye et al. [51]: an LLM receives labeled examples one at a time, updates its memory, and is evaluated on a held-out test set. We use `GPT-OSS-120B` as the LLM text classifier, and consider the problem of

Harness	Datasets			Avg.	
	USPTO	S2D	Law	Acc	Ctx ↓
Zero-Shot	12.0	63.2	7.0	27.4	0
Few-Shot (8)	14.0	67.9	21.0	34.3	2.0
Few-Shot (32)	13.0	72.2	21.0	35.4	7.9
Few-Shot (all)	15.0	78.3	29.0	40.8	12.3
MCE [51]†	14.0	83.0	23.0	40.0	28.5
ACE [58]†	**16.0**	77.8	29.0	40.9	50.8
`Meta-Harness`	14.0	**86.8**	**45.0**	**48.6**	11.4

Table 2: Test-set metrics for all harnesses on the three datasets. Ctx denotes additional input tokens in context (thousands). †: implementation from Ye et al. [51]. ↓: lower is better. **Meta-Harness improves online text classification accuracy while using a smaller input context.**

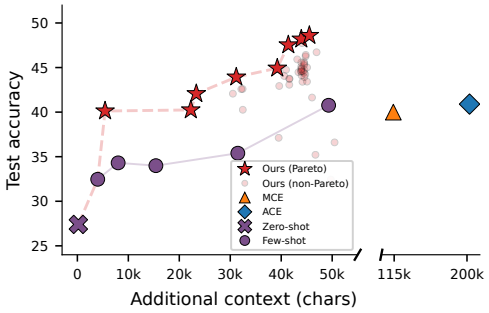

Figure 3: Pareto frontier of accuracy vs. context tokens on online text classification. **Meta-Harness achieves a stronger accuracy-context Pareto frontier than all comparison methods.**

designing a harness for text classification. We use three datasets, chosen for difficulty and domain diversity: **LawBench** (Law) [16] predicts criminal charges from case descriptions (215 classes); **Symptom2Disease** (S2D) [19] predicts diseases from symptom descriptions (22 classes); and **USPTO-50k** [40] predicts precursor reactants from product molecules (180 classes). We initialize the search population $\mathcal{H}$ from the main baseline harnesses in this setting: zero-shot, few-shot, ACE, and MCE. We ran 20 evolution iterations with two candidates per iteration, producing 40 candidate harnesses.

Comparison vs text optimizers. We compare Meta-Harness against representative methods for optimizing text. For a fair comparison, we use the same proposer configuration (`Opus-4.6` with max reasoning), select candidates solely based on search-set performance, and hold out the test sets until the final evaluation. Since evaluation is the main computational bottleneck, we give each method the same budget of proposal harness evaluations. We consider the following points of comparison:

- **Best-of-N**: independent samples from the seed with no search structure; a compute-matched control for whether search matters at all.
- **OpenEvolve** [42]: evolutionary search over programs with LLM mutation.
- **TTT-Discover** [54]: we use only the text-optimization component of their method, i.e., proposal selection via the PUCT reuse rule.

In this setting, Meta-Harness matches the best prior text optimizers (OpenEvolve, TTT-Discover) in $0.1\times$ the evaluations, and its final accuracy surpasses theirs by more than 10 points (Figure 1 and Table 4). We attribute this speedup to the intentional design choices that impose minimum necessary structure on the outer loop (Section 3). In particular, Meta-Harness preserves *full experience history using a filesystem* and allows the proposer to inspect anything necessary, whereas both OpenEvolve and TTT-Discover operate with more structured and substantially more limited proposer inputs than full filesystem access. We note that online text classification is the smallest-context setting we study (Table 1), so if structure-heavy text optimizers already lag here, their limitations may only grow in harder regimes.

Meta-Harness is 10× Faster and Converges to a Better Harness

In this setting, Meta-Harness matches the best prior text optimizers (OpenEvolve, TTT-Discover) with 10× fewer full evaluations, and its final accuracy surpasses theirs by more than 10 points.

To isolate which parts of the proposer interface matter most, we compare three conditions in online text classification: a scores-only condition, a scores-plus-summary condition in which the proposer receives LLM-generated summaries but no raw traces, and the full `Meta-Harness` interface with access to execution traces (Table 3). The results show a large gap in favor of the full interface: scores-only reaches 34.6 median and 41.3 best accuracy, while scores-plus-summary reaches 34.9 median and 38.7 best. By contrast, `Meta-Harness`

Method	Scores	Code	Summ.	Traces	Median ↑	Best Acc ↑	> ZS
Scores Only	✓	✓	×	×	34.6	41.3	26
Scores + Summary	✓	✓	✓	×	34.9	38.7	23
`Meta-Harness` (full)	✓	✓	-	✓	**50.0**	**56.7**	39

Table 3: Ablation of the information available to the proposer in online text classification. > ZS: number of runs whose accuracy exceeded the zero-shot baseline. The full Meta-Harness interface substantially outperforms scores-only and scores-plus-summary ablations. **Access to raw execution traces is the key ingredient for enabling harness search.**

reaches 50.0 median and 56.7 best accuracy, and even its median candidate outperforms the best candidate found under either ablation. We interpret this as evidence that full access to execution traces is the most important component of the interface: summaries do not recover the missing signal, and may even hurt by compressing away diagnostically useful details.

Comparison vs state-of-the-art harnesses. Our primary points of comparison are hand-designed harnesses for this problem setting: Agentic Context Engineering (ACE, Zhang et al. [58]), which uses reflective memory curation to build context over time, and Meta Context Engineering (MCE, Ye et al. [51]), which maintains and evolves a library of natural-language skills for context construction. As additional baselines, we evaluate zero-shot prompting and few-shot prompting with $N \in \{4, 8, 16, 32, \text{all}\}$ examples. Results in Table 2 show that Meta-Harness improves substantially over prior hand-designed harnesses. The selected `Meta-Harness`[2] reaches 48.6% accuracy, outperforming ACE by 7.7 points and MCE by 8.6 points. These gains do not come from using more context: `Meta-Harness` uses only 11.4K context tokens, versus 50.8K for ACE and 28.5K for MCE.

Method	Median	Best
GEPA [1]	32.6	40.2
Best-of-N	34.0	44.2
OpenEvolve [42]	39.1	43.3
TTT-Discover [54]	34.1	45.6
`Meta-Harness`	**50.0**	**56.7**

Table 4: Text classification accuracies of the harnesses proposed by different text optimizers (search set). **Meta-Harness is substantially more effective at harness optimization.**

Accuracy–Context Tradeoffs. Because Meta-Harness performs free-form optimization over harness code, we can express a joint preference for both accuracy and context cost rather than committing to a single scalar objective in advance. Given only the current metrics and the desired trade-off, the proposer is able to discover harnesses across a broad range of the frontier, yielding a smooth accuracy–context Pareto curve in Figure 3. This allows us to trade additional context for higher test accuracy in a controlled way, rather than committing to a single hand-designed operating point.

Out-of-distribution (OOD) task evaluation. We evaluate whether the discovered harness generalizes to entirely new datasets unseen during search. We consider nine diverse datasets, and describe them in detail in Appendix C.1. The selected `Meta-Harness` system achieves the best average accuracy (73.1%), outperforming ACE (70.2%) and all few-shot baselines (Table 5). Notably, we observe that naively adding more few-shot examples beyond 32 hurts performance in 7/9 tasks. Meta-Harness shows the highest performance on 6/9 datasets, suggesting that the discovered harness captures generally effective strategies for text classification rather than overfitting to the specific datasets used during search.

4.2 Harnesses for Retrieval-Augmented Reasoning

We study a somewhat non-standard setup for olympiad math solving: augmenting the model with the ability to retrieve examples from a large corpus. There is a good reason to expect retrieval to help mathematical reasoning in principle, because solutions often share reusable proof patterns, so previous reasoning traces contain information that a model may

[2]We slightly overload terminology for brevity: in the tables, `Meta-Harness` denotes the best discovered harness, whereas elsewhere it refers to the entire harness search procedure.

Harness	SciC	FiNER	Amz5	FPB	GoEmo	Bank77	News	SciT	TwHate	Avg Acc	Ctx ↓
Zero-shot	32.7	56.0	52.7	90.0	42.0	80.7	84.7	89.3	75.3	67.0	-
Few-shot (8)	34.0	63.0	54.0	90.0	44.0	82.7	84.7	**91.3**	76.7	68.9	2.2
Few-shot (32)	38.7	62.0	53.3	90.7	43.3	**86.0**	85.3	90.7	76.7	69.6	5.2
Few-shot (all)	35.3	61.0	50.0	93.3	42.7	80.7	84.0	90.0	76.7	68.2	7.4
ACE [58]	40.7	**74.0**	48.0	**96.7**	44.0	83.3	86.0	90.7	68.7	70.2	11.7
`Meta-Harness`	**53.3**	67.0	**60.0**	94.0	**46.0**	82.7	**86.7**	**91.3**	**77.3**	**73.1**	7.3

Table 5: OOD text classification dataset evaluation. We report test accuracy for each dataset and the average additional context tokens across all nine datasets. **`Meta-Harness` outperforms the next best method by 2.9 points on these 9 previously unseen tasks.**

Method	`GPT-5.4n`	`GPT-5.4m`	`Gem-3.1FL`	`Gem-3F`	`GPT-20B`	Avg.
No Retriever	23.0	28.8	28.6	42.6	47.6	34.1
Dense Retrieval (k=1)	27.1 (+4.1)	24.5 (-4.3)	31.3 (+2.7)	42.3 (-0.3)	46.9 (-0.7)	34.4 (+0.3)
Dense Retrieval (k=5)	31.1 (+8.1)	28.3 (-0.5)	37.1 (+8.5)	47.2 (+4.6)	46.7 (-0.9)	38.1 (+4.0)
Random Few-shot	23.1 (+0.1)	24.5 (-4.3)	31.0 (+2.4)	40.4 (-2.2)	41.8 (-5.8)	32.2 (-1.9)
BM25 Retrieval	30.2 (+7.2)	29.2 (+0.4)	32.8 (+4.2)	46.6 (+4.0)	48.9 (+1.3)	37.5 (+3.4)
`Meta-Harness`	31.7 (+8.7)	30.4 (+1.6)	34.9 (+6.3)	46.3 (+3.7)	50.6 (+3.0)	**38.8** (+4.7)

Table 6: Retrieval-augmented math problem solving on 200 IMO-level math problems. We show pass@1 averaged over three samples per problem, with absolute improvement over the baseline in parentheses. **The discovered Meta-Harness retrieval strategy improves reasoning on these IMO-level problems across all five held-out models, with a 4.7-point average gain over no retriever.**

be able to exploit at inference time. Yet retrieval has not become a standard ingredient in this setting, and prior work suggests that it has been much less successful on reasoning-intensive math benchmarks than in more fact-grounded domains [41; 48; 6]. The difficulty is that naive retrieval rarely surfaces the right traces in the right form. This suggests that success depends less on adding retrieval per se than on discovering the right retrieval policy. Rather than hand-designing that policy, we give Meta-Harness a hard set of olympiad problems and allow the retrieval behavior itself to emerge from search.

The retrieval corpus contains $\geq$500,000 solved problems from eight open-source datasets. We carefully deduplicated and decontaminated it against both evaluation benchmarks and the search set, confirmed that held-out problems have no exact prefix matches under our string-based filter, and manually inspected top BM25 retrievals for held-out examples (appendix C.2). We use Meta-Harness to optimize a harness for 40 iterations over a 250-problem search set of Olympiad-difficulty math problems (OlympiadBench + Omni-MATH hard), producing 109 candidate retrieval harnesses. We initialize the search population $\mathcal{H}$ from the main baseline harnesses in this setting: zero-shot, few-shot, and ACE. We select a single harness based on search-set performance using `GPT-OSS-20B` (Appendix B.2). We evaluate this harness on 200 previously unseen IMO-level problems drawn from IMO-AnswerBench, IMO-ProofBench, and ArXivMath [29; 6]. In addition to `GPT-OSS-20B`, we evaluate the same retrieval harness on four models not seen during search: `GPT-5.4-nano`, `GPT-5.4-mini`, `Gemini-3.1-Flash-Lite`, and `Gemini-3-Flash`. We follow the standard evaluation protocol of prior work [29] and report accuracy averaged over three samples per problem.

Results. Table 6 compares the discovered harness against no retrieval, dense retrieval using the separate embedding model `text-embedding-3-small`, random few-shot prompting, and BM25 retrieval. In contrast, Meta-Harness operates entirely in code space on top of the same BM25-based lexical retrieval stack as the sparse baseline, rather than introducing an additional dense encoder. The discovered retrieval harness outperforms the no-retrieval baseline across all five held-out models, with an average gain of **4.7 points**. It also matches or exceeds the strongest fixed baselines on average, outperforming BM25 retrieval by 1.3 points overall, while avoiding the regressions observed with dense retrieval and random few-shot prompting across several models.

Meta-Harness Improves Reasoning on IMO-Level Math Problems

In retrieval-augmented math reasoning, a single discovered retrieval harness transfers across five held-out models, improving accuracy by 4.7 points on average over no retrieval and yielding the strongest overall average among the compared methods.

4.3 Evaluating Agentic Coding Harnesses on TerminalBench-2

TerminalBench-2 [32] evaluates LLM agents on 89 challenging tasks that require long-horizon, fully autonomous execution under complex dependencies, and substantial domain knowledge. Prior work has shown that the choice agent harness has a large effect on performance on this benchmark. We initialize search from two strong open baselines, Terminus 2 [32] and Terminus-KIRA [24]. For this experiment, we perform search and final evaluation on the same 89-task benchmark. We use this benchmark as a *discovery problem* [53] in which the goal is to discover a harness configuration that improves performance on a hard, publicly contested benchmark. This is standard practice: public writeups already describe repeated benchmark-specific harness iteration on TerminalBench itself [18; 33; 24], and the benchmark is small and expensive enough that introducing a separate split would materially weaken the search signal. We additionally check for overfitting by manual inspection and regex-based audits for task-specific string leakage into evolved harnesses. We note that although the resulting harness is specialized to the TerminalBench-2 regime, autonomous completion of difficult long-horizon tasks from a single instruction is a core capability, and the benchmark consists of many tasks that frontier models and heavily engineered harnesses struggle with.

Results. We report results on the full benchmark in Table 7, evaluated on two base models: `Claude Opus 4.6` and `Claude Haiku 4.5`. On `Opus 4.6`, Meta-Harness discovers a harness achieving 76.4% pass rate, surpassing the hand-engineered Terminus-KIRA (74.7%) and ranking #2 among all `Opus 4.6` agents on the TerminalBench-2 leaderboard. The only higher-scoring `Opus 4.6` agent is ForgeCode (81.8%); however, we were unable to reproduce their reported result from the publicly available code alone, suggesting their leaderboard scores depend on components beyond the published repository. On the weaker `Haiku 4.5` model, the improvement is larger: Meta-Harness achieves 37.6%, outperforming the next-best reported agent (Goose, 35.5%) by 2.1 points. TerminalBench-2 is an actively contested benchmark with multiple teams directly optimizing for it, so the fact that an automatic search method can achieve benefits at this frontier is encouraging for long-horizon text-optimization loops.

Harness	Auto	Pass (%)
`Claude Opus 4.6`		
Claude Code	×	58.0
Terminus 2	×	62.9
Mux	×	66.5
Droid	×	69.9
TongAgents	×	71.9
MAYA-V2	×	72.1
Terminus-KIRA	×	74.7
Capy	×	75.3
ForgeCode	×	81.8
`Meta-Harness`	✓	**76.4**
`Claude Haiku 4.5`		
OpenHands	×	13.9
Claude Code	×	27.5
Terminus 2	×	28.3
Mini-SWE-Agent	×	29.8
Terminus-KIRA	×	33.7
Goose	×	35.5
`Meta-Harness`	✓	**37.6**

Table 7: Pass rate on TerminalBench-2. Results or others are from the official leaderboard. **Meta-Harness ranks #2 among all `Opus-4.6` agents and #1 among all `Haiku-4.5` agents on this competitive task.**

Qualitative behavior of the proposer. The harness search trajectory helps explain *why* Meta-Harness achieves these gains; we provide a detailed summary in Appendix A. In early iterations, the proposer combined plausible structural fixes with prompt-template edits and observed that both candidates regressed. It then explicitly hypothesized that the regressions were confounded by the shared prompt intervention, isolated the structural changes from the prompt rewrite, and ultimately pivoted toward a safer additive modification that became the best candidate in the run. This provides qualitative evidence that **filesystem access enables the proposer to inspect prior experience in enough detail to form causal hypotheses and revise the harness accordingly.**

Meta-Harness Surpasses Hand-Engineered Agents on TerminalBench-2

On TerminalBench-2, Meta-Harness automatically discovers harnesses that surpass Terminus-KIRA on Opus 4.6 and rank #1 among all Haiku 4.5 agents.

5 Discussion

Beyond outperforming existing harnesses, Meta-Harness has several practical advantages. Discovered harnesses generalize to out-of-distribution classification datasets (Table 5) and to unseen base models in the math setting (Table 6). A search run completes in a few hours of wall-clock time, yet produces readable, transferable strategies that can be reused across models, including future, stronger ones. Overfitting in code space is also more inspectable: brittle if-chains or hard-coded class mappings are visible on inspection in a way that weight-space overfitting is not. More broadly, our results suggest that the main advantage of Meta-Harness is not just search over code, but search with *selective access to prior diagnostic experience*. The proposer is not limited to scalar rewards or fixed summaries; it can inspect raw code, execution traces, and prior failures, then use that information to form and test hypotheses about what to change. The qualitative search trajectories in Appendix A.2 illustrate this behavior directly.

Our findings reflect a recurring pattern in machine learning [44]: once a search space becomes accessible, stronger general-purpose agents can outperform hand-engineered solutions. A natural next step for future work is to co-evolve the harness and the model weights, letting the strategy shape what the model learns and vice versa. While we evaluate on three diverse domains, our experiments demonstrate that harness search can work with one particularly strong coding-agent proposer (Claude Code); a broader study of how the effect varies across proposer agents remains for future work.

Acknowledgements

We thank KRAFTON AI for providing API credit support. This work is supported by OpenAI, KFAS, and Schmidt Sciences AI2050. We thank Anikait Singh and Jubayer Ibn Hamid for their valuable feedback and suggestions, and Sienna J. Lee for patiently listening to YL's half-formed thoughts during the early stages of this work.

References

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

[2] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models, 2023. URL https://arxiv.org/abs/2211.15661.

[3] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

[4] Anthropic. Claude code: An agentic coding tool. https://www.anthropic.com/claude-code, 2025.

[5] Anthropic and community contributors. agentskills/agentskills. GitHub repository https://github.com/agentskills/agentskills. Specification and documentation for Agent Skills, accessed March 27, 2026.

[6] Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions, February 2025. URL https://matharena.ai/.

[7] Francesco Barbieri, Jose Camacho-Collados, Leonardo Neves, and Luis Espinosa-Anke. Tweeteval: Unified benchmark and comparative evaluation for tweet classification, 2020. URL https://arxiv.org/abs/2010.12421.

[8] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, June 2023. ISSN 2475-1421. doi: 10.1145/3591300. URL http://dx.doi.org/10.1145/3591300.

[9] Birgitta Böckeler. Harness engineering. https://martinfowler.com/articles/exploring-gen-ai/harness-engineering.html, March 2026. martinfowler.com.

[10] Can Bölük. I improved 15 LLMs at coding in one afternoon. only the harness changed. https://blog.can.ac/2026/02/12/the-harness-problem/, February 2026.

[11] Iñigo Casanueva, Tadas Temčinas, Daniela Gerz, Matthew Henderson, and Ivan Vulić. Efficient intent detection with dual sentence encoders, 2020. URL https://arxiv.org/abs/2003.04807.

[12] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

[13] Harrison Chase. Langchain, October 2022. URL https://github.com/langchain-ai/langchain. Software, released 2022-10-17.

[14] Arman Cohan, Waleed Ammar, Madeleine van Zuylen, and Field Cady. Structural scaffolds for citation intent classification in scientific publications, 2019. URL https://arxiv.org/abs/1904.01608.

[15] Dorottya Demszky, Dana Movshovitz-Attias, Jeongwoo Ko, Alan Cowen, Gaurav Nemade, and Sujith Ravi. Goemotions: A dataset of fine-grained emotions, 2020. URL https://arxiv.org/abs/2005.00547.

[16] Zhiwei Fei, Xiaoyu Shen, Dawei Zhu, Fengzhe Zhou, Zhuo Han, Alan Huang, Songyang Zhang, Kai Chen, Zhixin Yin, Zongwen Shen, et al. Lawbench: Benchmarking legal knowledge of large language models. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 7933–7962, 2024.

[17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, 2017.

[18] ForgeCode. Benchmarks don't matter, 2025. URL https://forgecode.dev/blog/benchmarks-dont-matter/.

[19] Gretel AI. Symptom to diagnosis dataset. https://huggingface.co/datasets/gretelai/symptom_to_diagnosis, 2023. Accessed: 2026-01-22.

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

[21] Anthropic Justin Young. Effective harnesses for long-running agents. https://anthropic.com/engineering/effective-harnesses-for-long-running-agents, November 2025. Anthropic Engineering Blog.

[22] Phillip Keung, Yichao Lu, György Szarvas, and Noah A. Smith. The multilingual amazon reviews corpus, 2020. URL https://arxiv.org/abs/2010.02573.

[23] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

[24] KRAFTON AI and Ludo Robotics. Terminus-kira: Boosting frontier model performance on terminal-bench with minimal harness, 2026. URL https://github.com/krafton-ai/kira.

[25] Yoonho Lee, Joseph Boen, and Chelsea Finn. Feedback descent: Open-ended text optimization via pairwise comparison. In *arXiv preprint arXiv:2511.07919*, 2025.

[26] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models, 2022. URL https://arxiv.org/abs/2206.08896.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[28] Lefteris Loukas, Manos Fergadiotis, Ilias Chalkidis, Eirini Spyropoulou, Prodromos Malakasiotis, Ion Androutsopoulos, and Georgios Paliouras. Finer: Financial numeric entity recognition for xbrl tagging. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4419–4431. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.acl-long.303. URL http://dx.doi.org/10.18653/v1/2022.acl-long.303.

[29] Thang Luong, Dawsen Hwang, Hoang H. Nguyen, Golnaz Ghiasi, Yuri Chervonyi, Insuk Seo, Junsu Kim, Garrett Bingham, Jonathan Lee, Swaroop Mishra, Alex Zhai, Clara Huiyi Hu, Henryk Michalewski, Jimin Kim, Jeonghyun Ahn, Junhwi Bae, Xingyou Song, Trieu H. Trinh, Quoc V. Le, and Junehyuk Jung. Towards robust mathematical reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025. URL https://aclanthology.org/2025.emnlp-main.1794/.

[30] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

[31] Pekka Malo, Ankur Sinha, Pyry Takala, Pekka Korhonen, and Jyrki Wallenius. Good debt or bad debt: Detecting semantic orientations in economic texts, 2013. URL https://arxiv.org/abs/1307.5336.

[32] Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

[33] Jack Nichols. How we scored #1 on terminal-bench (52%), Jun 2025. URL https://www.warp.dev/blog/terminal-bench.

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

[35] OpenAI. Harness engineering: leveraging Codex in an agent-first world. https://openai.com/index/harness-engineering/, February 2026. OpenAI Blog.

[36] Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: Towards llms as operating systems. 2023.

[37] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.

[38] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[39] Jurgen Schmidhuber. A neural network that embeds its own meta-levels. In *IEEE International Conference on Neural Networks*, 1993.

[40] Nadine Schneider, Nikolaus Stiefl, and Gregory A Landrum. What's what: The (nearly) definitive guide to reaction role assignment. *Journal of chemical information and modeling*, 56(12):2336–2346, 2016.

[41] Srijan Shakya, Anamaria-Roberta Hartl, Sepp Hochreiter, and Korbinian Pöppel. Adaptive retrieval helps reasoning in llms – but mostly if it's not used, 2026. URL https://arxiv.org/abs/2602.07213.

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

[43] Jake Snell, Kevin Swersky, and Richard S. Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, 2017.

[44] Rich Sutton. The bitter lesson, 2019. *URL http://www. incompleteideas. net/IncIdeas/Bitter-Lesson. html*, 2019.

[45] Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pp. 3–17. Springer, 1998.

[46] 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 *arXiv preprint*, 2026. URL https://api.semanticscholar.org/CorpusID:285462974.

[47] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions, 2023. URL https://arxiv.org/abs/2212.10509.

[48] Chenghao Xiao, G Thomas Hudson, and Noura Al Moubayed. Rar-b: Reasoning as retrieval benchmark, 2024. URL https://arxiv.org/abs/2404.06347.

[49] Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. In *OpenReview*, 2026. URL https://api.semanticscholar.org/CorpusID:285454009.

[50] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

[51] Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026.

[52] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

[53] 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, 2026. URL https://arxiv.org/abs/2601.16175.

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

[55] Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2026. URL https://arxiv.org/abs/2512.24601.

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

[57] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation, 2025. URL https://arxiv.org/abs/2410.10762.

[58] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, V. Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and K. Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *arXiv preprint arXiv:2510.04618*, 2025.

[59] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification, 2016. URL https://arxiv.org/abs/1509.01626.

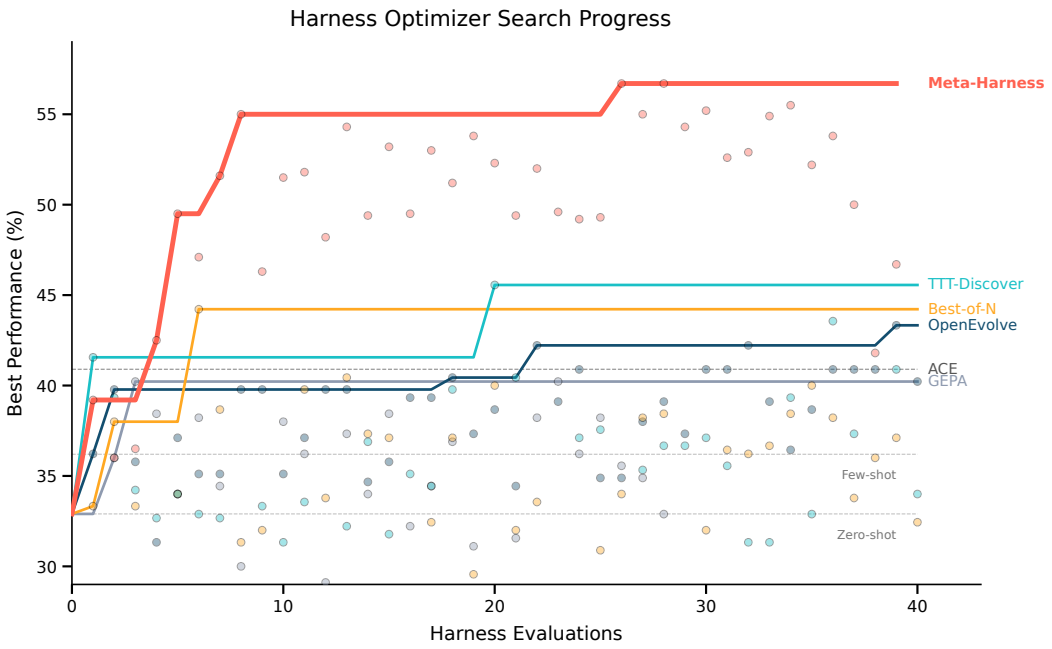

Figure 4: Search-set accuracy over evaluations for all compared text optimizers on online text classification. Each point is one candidate harness; lines track the best-so-far. Per-dataset curves are shown alongside the aggregate. **Meta-Harness reaches the final accuracy of OpenEvolve and TTT-Discover within the first 4 evaluations and continues improving, ending more than 10 points above all baselines.**

A Qualitative Proposer Behavior

This section examines how the proposer uses the filesystem during search, drawing on the TerminalBench-2 run (10 iterations, `Claude Opus 4.6`).

A.1 File Access Statistics

To verify that the proposer makes substantive use of the filesystem rather than defaulting to local edits, we recorded all file reads per iteration.

Table 8 summarizes the results. The proposer reads a median of 82 files per iteration (range 69–99), roughly evenly split between prior harness source code (41%) and execution traces (40%), with the remainder going to score summaries (6%) and other files (13%). This confirms that the proposer's access pattern is non-Markovian: it routinely inspects the majority of available history rather than conditioning only on the most recent parent.

Statistic	Value
Files read per iteration (median)	82
Files read per iteration (range)	69–99
File type breakdown	
Harness source code	41%
Execution traces	40%
Score/summary files	6%
Other	13%

Table 8: Proposer file access statistics from the TerminalBench-2 search run (10 iterations, `Claude Opus 4.6`). The proposer reads extensively from the filesystem, with roughly equal attention to prior source code and execution traces.

A.2 Qualitative Behavior: Causal Reasoning Over Prior Failures

The TerminalBench-2 search log reveals a clear narrative arc in which the proposer learns from its own regressions. Rather than wandering randomly through local edits, it forms an explicit diagnosis of why early candidates failed, then shifts toward a safer design pattern. All text inside the log boxes below is quoted verbatim from the proposer's recorded reasoning at each iteration (emphasis ours).

Iterations 1–2: promising bugfixes are confounded by prompt edits. The first two iterations both bundle plausible structural fixes with prompt-template modifications, and both regress sharply from the 64.4% Terminus-KIRA baseline. Iteration 1 targets observation corruption from leaked terminal markers and adds a loop breaker:

> Hypothesis: __CMDEND__ marker fragments leak into LLM observations on long-running tasks, causing the model to get confused and enter infinite no-tool-call loops. Stripping these markers + adding a loop breaker will recover wasted steps.

That candidate also introduced a new cleanup-oriented prompt template and a verification checklist. Iteration 2 proposes a different state-machine fix:

> Double-confirmation completion mechanism causes verification spirals. Observed in trajectories where the agent solves the task early but burns 15--40+ additional steps re-verifying because each verification command resets _pending_completion, requiring another task_complete → checklist → verify cycle.

This second candidate removes the pending-completion mechanism entirely, while also carrying over the marker stripping and the new prompt. It still regresses, which gives the proposer two failed candidates with different structural changes but one shared prompt intervention.

Iteration 3: the proposer identifies the confound. By iteration 3, the proposer explicitly infers that the regressions are not primarily due to the structural bugfixes themselves:

> Prior attempts: evo_marker_fix (58.9%, -5.6pp), evo_single_confirm (57.8%, -6.7pp) --- both regressed. **Root cause of regressions: Prompt template changes (cleanup directives) caused the agent to delete necessary state before task completion.** The structural bugfixes were confounded with harmful prompt changes. evo_strip_only isolates the two proven structural fixes.

This is the key causal step in the trajectory. The proposer notices that the common factor across the first two failures is not the particular bugfix, but the cleanup-heavy prompt rewrite. It therefore reverts to the original prompt and tests only the marker-stripping and loop-breaker. The resulting candidate still underperforms slightly (63.3%, -1.1pp), but it loses far less than the earlier versions, which supports the confound diagnosis.

Iterations 4–6: direct fixes to the diagnosed failure mode still regress. The next three iterations continue to probe the same part of the design space, but now with more explicit theories about why the completion logic is fragile. Iteration 4 attributes failures to a concrete state-machine bug in which verification commands reset the completion flag and trap the agent in repeated checklist cycles:

> Remove the two self._pending_completion = False lines that reset the completion flag when intermediate commands run. This fixes a state machine bug where: (1) Agent calls task_complete → sees QA checklist, _pending_completion = True (2) Agent runs verification commands → _pending_completion = False (bug!) (3) Agent calls task_complete again → sees checklist AGAIN → infinite loop.

The proposer even cites concrete trajectory evidence, noting that `configure-git-webserver` produced baseline failures with agents stuck in 30–60 step verification spirals after effectively solving the task. Iteration 5 tries to soften the cleanup language while preserving

confirmation, but still edits the prompt and regresses badly. Iteration 6 returns to the safer `evo_strip_only` base and proposes a systems-level optimization:

```
Empty-command turns waste full LLM round-trips when terminal output hasn't changed.
Smart-waiting (poll pane up to 3×5s) before the next LLM call saves 5--15 turns on
long-running tasks.
```

That change also regresses. By this point, the proposer has learned a specific empirical lesson: modifications to prompts and completion flow are high risk, even when the local hypothesis sounds reasonable.

Iteration 7: the winning candidate. After six consecutive regressions, the proposer shifts strategy from modifying the control loop to adding information before the loop begins:

```
All 6 prior iterations regressed from the 64.4% baseline because they modified the
completion flow, prompt template, or observation processing. evo_env_bootstrap takes
a different approach --- purely additive. It gathers an environment snapshot via a
single shell command before the first LLM call and appends it to the initial prompt.
No other methods are changed. This should eliminate 3--5 wasted exploration turns on
dependency-heavy tasks without risking regression on already-passing tasks.
```

This candidate is the best result so far. The important point is not just that iteration 7 wins, but that the proposer articulates *why* it should be safer: it avoids touching the previously fragile completion machinery and instead adds information that is useful mainly on hard tasks.

Iteration 8: composition. Having found one additive improvement, the proposer next attempts to compose it with an earlier structural fix:

```
Combining two orthogonal fixes --- env snapshot (saves early exploration turns) +
marker stripping with no-tool-call loop breaker --- will yield +1--3pp because they
address independent failure modes without touching prompts or confirmation flows
(which caused regressions in 5 of 7 prior iterations).
```

Iteration 10: cross-run transfer. The proposer references results from a separate earlier search run:

```
The evolution history showed ''don't cleanup service artifacts'' was worth +18pp. Iter
9 (evo_no_cleanup_directive) targeted the same idea but crashed before evaluation.
```

Summary. The search trajectory demonstrates that the proposer does more than random mutation. Across the first seven iterations, it identifies a confound, tests the confound-isolating hypothesis directly, observes that control-flow and prompt edits remain fragile, and then deliberately pivots to a purely additive modification that becomes the best candidate in the run. It subsequently tries to compose that winning idea with earlier fixes and even transfers lessons across runs. This kind of causal reasoning over prior failures is precisely what full-history filesystem access enables and what compressed-feedback optimizers cannot support.

B Discovered Harnesses

Meta-Harness discovers executable inference-time procedures specific to the problem setup at hand. These harnesses are structured, domain-specific policies, often with nontrivial control flow such as routing, filtering, and conditional context construction, selected solely by whether they improve search-set performance. This section presents compact, method-style abstractions of representative harnesses that summarize the main behaviors and control-flow decisions that drive inference-time behavior. For reference, the full implementation for each discovered harness is on the order of 100–1000 lines of code.

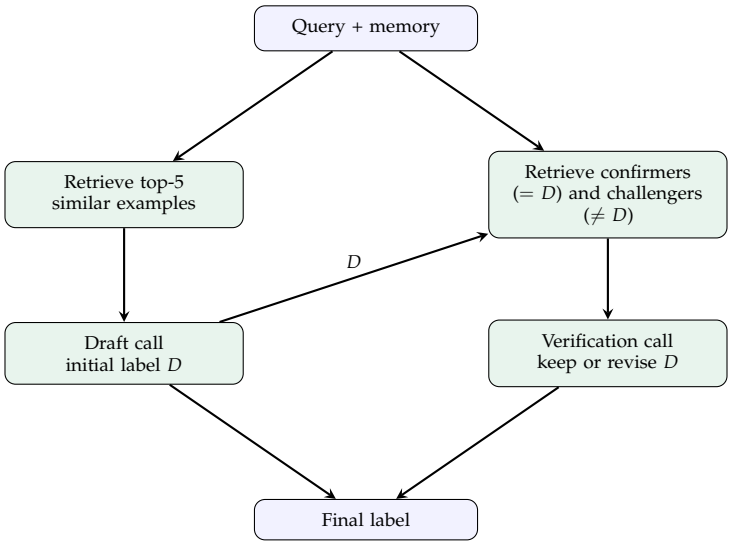

Figure 5: **Draft-verification classification harness.** The first call produces a draft label from a short retrieved context. The second call retrieves evidence for and against that draft and returns the final prediction.

B.1 Text Classification Harness

In online text classification, Meta-Harness discovers a family of memory-based harnesses rather than a single canonical policy. Table 9 reports the Pareto frontier of non-dominated variants from the main search, all selected solely by search-set performance. We highlight two representative endpoints here: `Meta-Harness (Draft Verification)`, the lowest-context frontier point, and `Meta-Harness (Label-Primed Query)`, the highest-accuracy frontier point used in the main text.

Overview. Both harnesses maintain a growing memory of past labeled examples and build prompts from that memory at inference time. What differs is the control flow used to interrogate the memory. `Meta-Harness (Draft Verification)` uses two short calls and explicitly tests the model's first guess against retrieved counterexamples, while `Meta-Harness (Label-Primed Query)` spends a larger single-call budget on making the label space and local decision boundaries explicit. Figures 5 and 6 summarize these two programs.

Meta-Harness (Draft Verification). The corresponding discovered file is `draft_verification.py`. This lightweight variant turns prediction into a two-call procedure. It first retrieves the 5 most similar labeled examples and makes a draft prediction. It then re-queries the same memory conditioned on that draft label, retrieving 5 *confirmers* with the same label and 5 *challengers* with different labels, and asks the model whether to maintain or revise its initial answer. The key discovered behavior is that the second retrieval depends on both the query and the draft prediction, so the harness can surface counterexamples targeted at the model's current guess rather than only generic near neighbors. If too few labeled examples have been accumulated, the program falls back to a standard single-call few-shot prompt.

- **Stage 1: Draft.** Retrieve the 5 nearest labeled examples and ask for an initial prediction.
- **Stage 2: Verification.** Condition retrieval on the draft label, then show both supporting and challenging examples before making the final prediction.
- **Cold start.** If fewer than 5 labeled examples are available, skip the two-stage procedure and use a standard single-call few-shot prompt.
- **Why it is cheap.** Both calls use short retrieved contexts, so the overall context cost stays near the low end of the frontier even with two model invocations.

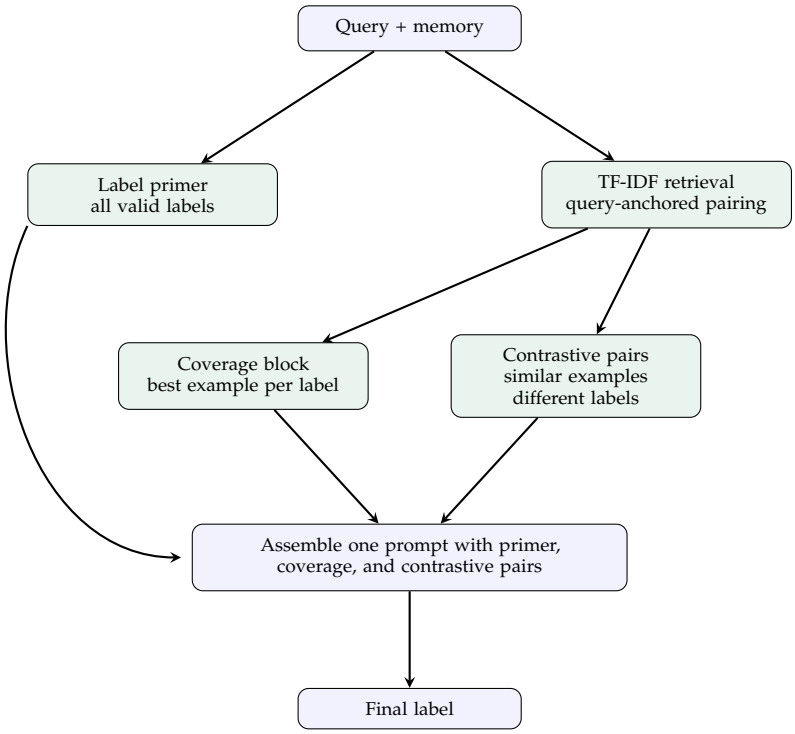

Figure 6: **Label-primed query-anchored classification harness.** The program builds a single prompt that exposes the label space, then populates it with query-relevant coverage examples and local contrastive pairs.

Meta-Harness (Label-Primed Query). The corresponding discovered file is `label_primed_query_anchored.py`. This strongest variant uses a single larger call built from three parts. It begins with a *label primer* listing the valid output labels, then constructs a *coverage* section with one query-relevant example per label, and finally adds *query-anchored contrastive pairs* that place highly similar examples with different labels side by side. The coverage block exposes the full label space, while the contrastive block sharpens local decision boundaries around the current query. In code, the harness implements this with TF-IDF retrieval over past labeled examples and a query-anchored pairing rule that chooses contrasting examples from the same local neighborhood.

- **Label primer.** List the valid output labels before showing any examples, so the model sees the full answer space up front.
- **Coverage block.** For each known label, retrieve the most query-relevant labeled example and include one representative example per class.
- **Contrastive block.** Build pairs of highly similar examples with different labels, so the prompt exposes local decision boundaries around the current query.
- **Retrieval rule.** Use TF-IDF similarity and query-anchored partner selection rather than label-agnostic nearest neighbors.

B.2 Math Retrieval Harness

This subsection describes the retrieval harness discovered by Meta-Harness for mathematical reasoning (Section 4.2). The final harness is a compact four-route BM25 program whose structure emerged through search rather than being manually specified after the fact. All design choices below—the routing predicates, reranking terms, deduplication thresholds, and per-route example counts—were selected by the outer loop across 40 iterations of evolution.

Variant	Datasets			Avg metrics	
	USPTO ↑	Symptom ↑	LawBench ↑	Avg ↑	Ctx ↓
`Meta-Harness (Draft Verification)`	**18.0**	85.4	17.0	40.1	5.4
`Meta-Harness (Error-Annotated)`	9.0	87.7	24.0	40.2	22.3
`Meta-Harness (CoT Replay)`	13.0	88.2	25.0	42.1	23.3
`Meta-Harness (Cluster Coverage)`	12.0	86.8	33.0	43.9	31.2
`Meta-Harness (Cascade Retrieval)`	12.0	86.8	36.0	44.9	39.2
`Meta-Harness (RRF + Contrastive)`	**18.0**	89.6	35.0	47.5	41.4
`Meta-Harness (Relevance + Contrastive)`	**18.0**	**90.6**	36.0	48.2	43.9
`Meta-Harness (Label-Primed Query)`	14.0	86.8	**45.0**	**48.6**	45.5

Table 9: Pareto-optimal discovered variants from the main text-classification search, trading off average accuracy against context cost. The selected system in the main text is `Meta-Harness (Label-Primed Query)`. Ctx denotes average additional characters in input context (thousands).

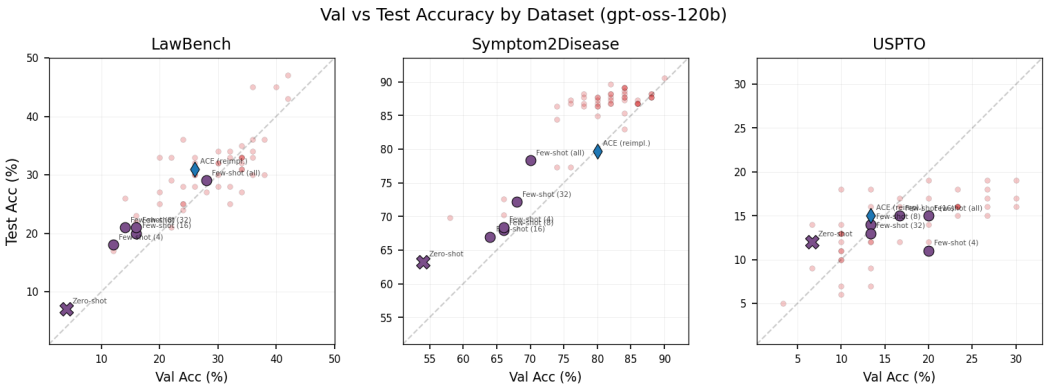

Figure 7: Search-set vs. test accuracy per dataset for discovered text-classification strategies. Each pink dot is a discovered strategy; baselines are labeled. The dashed diagonal is $y{=}x$.

Overview. At inference time, the harness assigns each problem to exactly one of four routes: combinatorics, geometry, number theory, or a default route for algebra and other problems. The gates are implemented as lightweight lexical predicates over the problem statement, including keyword sets and a small number of regex features for geometry notation. The harness does not aggregate outputs across routes: once a route is selected, only that route retrieves examples for the final prompt. All routes use BM25 as the underlying retrieval mechanism over the filtered corpus described above. The BM25 index uses a math-aware tokenizer that preserves LaTeX tokens (e.g., `\frac`, `^{2}`) as atomic units. The selected harness is a merge of two successful search lineages, autonomously combined by the proposer during search: one contributed a stronger geometry route based on raw BM25, while another contributed a stronger combinatorics route based on deduplication and difficulty reranking. Figure 8 gives a compact flowchart view of the final program.

- **Combinatorics:** fetch 20 BM25 candidates, deduplicate to 8, rerank by lexical score and difficulty, then return the top 3. This is the main route where the harness explicitly trades off diversity against hard-problem matching.
- **Geometry:** return 1 hard NuminaMath reference together with 2 raw BM25 neighbors. Search consistently prefers raw structural matches here over difficulty reranking.
- **Number theory:** fetch 12 BM25 candidates and rerank using lexical score, difficulty, and a small bonus for solutions that state a technique early. This favors examples whose proof strategy is explicit.
- **Default:** fetch 10 BM25 candidates, rerank by lexical score and difficulty, and choose an adaptive number of examples based on how concentrated the top retrieval scores are.

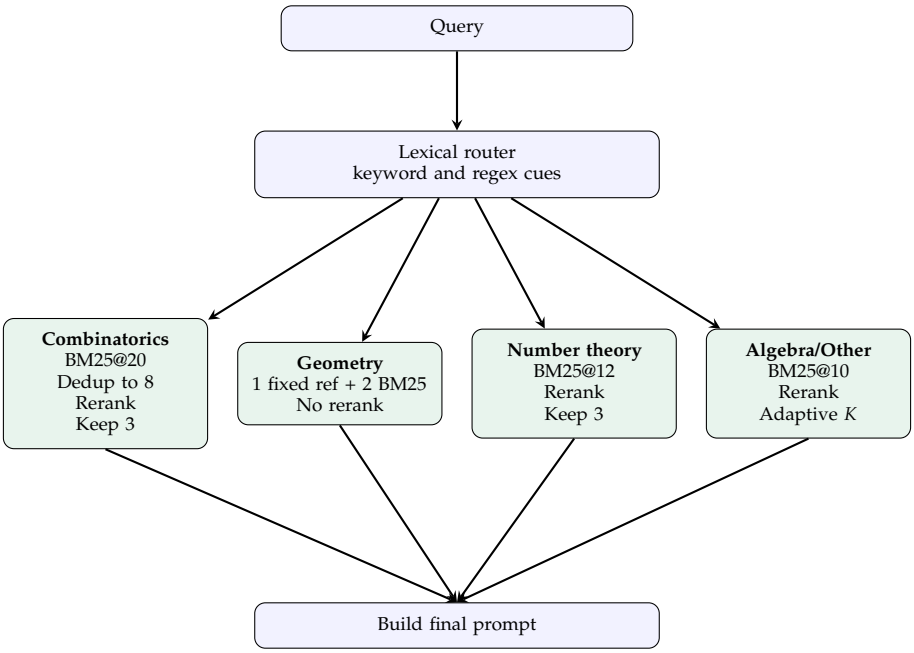

Figure 8: **Discovered math retrieval harness.** A lexical router assigns each query to one of four subject-specific retrieval policies. The selected policy retrieves examples, which are inserted into the final prompt.

B.3 TerminalBench-2 Harness

The discovered TerminalBench-2 harness builds on Terminus-KIRA [24], inheriting its native tool calling (replacing Terminus 2's ICL-based JSON parsing), 30KB output cap, and multi-perspective completion checklist. The main modification discovered by Meta-Harness is **environment bootstrapping**: before the agent loop begins, the harness runs a compound shell command to gather a snapshot of the sandbox environment and injects it into the initial prompt. The proposer's hypothesis, recorded verbatim from the search log, was:

```
Hypothesis: ‘‘Injecting an environment snapshot (OS, installed languages, package
managers, /app contents) before the first LLM turn will reduce wasted exploration
episodes by 3--5 turns on dependency-heavy tasks’’

Changes: ‘‘Added _gather_env_snapshot() that runs a single compound shell command to
collect working directory, /app listing, available languages (python, gcc, node, java,
rustc, go), package managers (pip, apt) [. . . ] and injects as [Environment Snapshot]
block’’
```

The snapshot includes: the working directory, a listing of `/app` (truncated to 20 entries for large directories), available programming languages and their versions (Python, GCC, G++, Node, Java, Rust, Go), installed package managers (pip, apt-get), and available memory. This eliminates the 2–4 exploratory turns that agents typically spend discovering what tools and files are available, allowing the model to begin productive work immediately. The bootstrapping command is guarded by a 15-second timeout and fails silently, so it does not break the agent in unusual environments. The full implementation adds roughly 80 lines on top of Terminus-KIRA. Figure 9 summarizes the harness structure.

Per-task analysis. Compared to Terminus-KIRA, the discovered harness gains on 7 of 89 tasks, with the largest improvements on `protein-assembly` and `path-tracing`. The gaining tasks share a common property: they require domain-specific tooling whose availability cannot be assumed in advance (bioinformatics libraries, rendering pipelines, chess engines, cryptographic utilities, CoreWars simulators). Without the bootstrap, the agent spends its

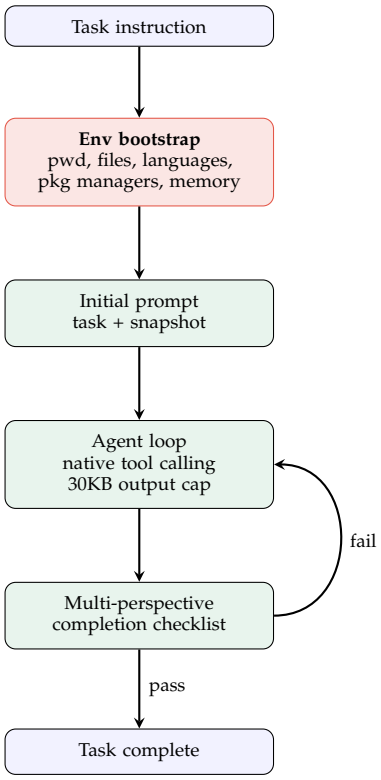

Figure 9: **Discovered TerminalBench-2 harness.** The harness inherits Terminus-KIRA's native tool calling, output cap, and completion checklist (green). The environment bootstrap (red) is the component discovered by Meta-Harness: it gathers a sandbox snapshot before the agent loop begins, eliminating early exploratory turns.

first 2–4 turns probing the environment; on tasks with tight turn budgets or where early wrong assumptions cascade, those wasted turns can be the difference between pass and fail. This suggests that the bootstrap's value is largest when the environment is non-obvious, and the task requires the agent to match its strategy to what is actually installed.

C Dataset Details

C.1 OOD Text Classification Datasets

- **SciCite** is a 3-way citation-intent classification benchmark introduced by Cohan et al. [14]. Each example consists of a citation context from a scientific paper, labeled by the citation's rhetorical role, such as background, method, or result. The task tests whether a model can infer why one paper cites another from the local scientific context.
- **FiNER-139** is a financial numeric entity recognition benchmark introduced by Loukas et al. [28]. It consists of word-level annotations from financial filings with 139 fine-grained XBRL entity types, making it substantially more fine-grained than standard sentence-level classification tasks. The benchmark tests whether a model can identify and classify numeric financial entities from context.
- **Amazon Reviews** is the English portion of the Multilingual Amazon Reviews Corpus introduced by Keung et al. [22]. In our setting, it is used as a 5-way review rating prediction task, where the label corresponds to the review's star rating. This benchmark evaluates general-domain sentiment and rating prediction from product review text.
- **Financial PhraseBank** is a 3-way financial sentiment benchmark introduced by Malo et al. [31]. It consists of sentences from financial news and related economic text labeled

as positive, neutral, or negative with respect to market sentiment. The task evaluates domain-specific sentiment classification in finance.

- **GoEmotions** is a fine-grained emotion classification benchmark introduced by Demszky et al. [15]. It contains English Reddit comments annotated with 27 emotion categories plus a neutral category, and is commonly treated as a 28-way classification task. The benchmark tests nuanced affect recognition beyond coarse positive-negative sentiment.
- **Banking77** is a fine-grained intent classification benchmark introduced by Casanueva et al. [11]. It contains online banking user utterances labeled with 77 intents, covering a wide range of customer service requests. The task evaluates single-domain intent detection with a large label space.
- **AG News** is a 4-way news topic classification benchmark commonly associated with the text classification setup of Zhang et al. [59]. Examples are labeled with broad news categories such as world, sports, business, and science/technology. It is a standard general-domain benchmark for topic classification.
- **SciTail** is a science-domain textual entailment benchmark in which the task is to predict whether a hypothesis is entailed by a premise sentence in a science-focused inference setting [**?**].
- **TweetEval (Hate)** is the hate-speech subset of the TweetEval benchmark introduced by Barbieri et al. [7]. It is a binary tweet classification task for detecting hateful versus non-hateful content within a unified social-media evaluation suite. This benchmark tests robust classification in noisy, short-form social media text.

C.2 Math Retrieval Corpus

Table 10 lists the datasets composing the retrieval corpus used in Section 4.2. The raw sources contain more problems than the final corpus; several filtering steps were applied before merging. NuminaMath-1.5 was filtered to competition-math subsets (AMC/AIME, olympiad references, number theory, inequalities, and related sources), discarding lower-quality web-scraped entries. OpenMathReasoning was deduplicated to one solution per problem (retaining the solution with the highest pass rate on an independent verifier), and problems whose source matched any evaluation benchmark family (IMO, AIME, HMMT, SMT, USAMO, Putnam) were removed before deduplication. The entire corpus was then decontaminated against all evaluation benchmarks and the search set used during harness search, using exact prefix matching followed by fuzzy Jaccard similarity (threshold 0.8); any corpus problem matching an eval problem under either criterion was discarded. Solutions from OpenMathReasoning and DeepMath are truncated to 5,000 characters to limit retrieval context length. At runtime, the selected harness further restricts retrieval to entries with non-empty solutions shorter than 4,000 characters. Retrieved solutions are truncated again to 3,000 characters when inserted into the prompt. For the geometry route, the harness also constructs a separate hard-reference index from NuminaMath problems with difficulty greater than 6.

C.3 Math IMO-level Test Set

The main text aggregates results over 200 IMO-level problems drawn from IMO-AnswerBench, IMO-ProofBench, ArXivMath December 2025, and ArXivMath January 2026. The 200-problem evaluation set consists of a stratified 100-problem subset of IMO-AnswerBench, together with all problems from the other three benchmarks. This per-benchmark breakdown is useful because the four datasets mix answer-style, proof, and research-style problems, which are aggregated together in the main paper for brevity. When included, the table in this section should report each benchmark separately for both `Base` and `Meta-Harness` across the five held-out models.

Dataset	Problems	Sol. Len	Proof
OpenMathReasoning	281,743	5,000†	34%
DeepMath-103K	103,021	5,000†	0%
NuminaMath-1.5	129,520	1,376	13%
PolyMath	11,083	363	0%
Omni-MATH	4,289	829	0%
FineProofs-SFT	4,275	3,977	100%
AIME 1983–2024	933	—	0%
Putnam-AXIOM	492	888	100%
Total	**535,356**	5,000†	22%

† Truncated at 5,000 characters; actual solutions are longer.

Table 10: Datasets in the math retrieval corpus (535K problems total). Sol. Len is the median solution length in characters. Proof indicates whether the dataset contains proof-type problems (by answer or problem type field).

Dataset	Problems
IMO-AnswerBench	100
IMO-ProofBench	60
ArXivMath Dec. 2025	17
ArXivMath Jan. 2026	23
Total	**200**

Table 11: Breakdown of the 200-problem IMO-level evaluation set.

D Practical Implementation Tips

Meta-Harness is largely domain-agnostic: we expect it to apply in any setting where a language model is wrapped by a task-specific harness. Applying it in a new domain, however, requires operating in a relatively new regime of LLM-assisted coding, where the proposer conditions on long-horizon histories of prior runs and writes programs whose effects may only become visible many steps later. In getting this workflow to work reliably, we found a small set of practical choices that mattered consistently across the three domains studied in this paper. The guidelines below are not themselves scientific claims about the method; they are engineering lessons from building and running the system, which we hope will make it easier for future work to apply Meta-Harness in other domains.

- **Write a good skill.** The skill text is the primary interface for steering the search, and its quality is the strongest lever on whether the loop works. The proposer receives a natural-language skill [5] that defines its role, the directory layout, CLI commands, and output format. In practice, the skill should constrain outputs and safety-relevant behavior, not the proposer's diagnosis procedure: it should specify what is forbidden, what artifacts to produce, and what objectives to optimize, while leaving the model free to inspect scores, traces, and prior code as needed. Our intuition from inspecting logs from Meta-Harness runs is that after enough iterations, the accumulated traces often shape the proposer's behavior more than the skill itself. In our experience, iterating on the skill text had a larger effect on search quality than changing iteration count or population size. Expect to run a few short evolution runs (3–5 iterations each) specifically to debug and refine the skill before committing to a full run.
- **Start with a baseline harness and a search set that is hard for it.** Write a simple baseline (e.g., few-shot prompting), then construct the search set by either filtering for examples that the baseline gets wrong or selecting a diverse subset of difficult instances. The search has little to optimize if the baseline already saturates the evaluation. Keep the search set small enough for roughly 50 full evaluations per run (50–100 examples in our

classification experiments, 88 problems for math retrieval); a fast, discriminative eval is more valuable than a large one.

- **Log everything in a format that is easy to navigate.** Evaluation code should write code, scores, and execution traces in a form that the proposer can query reliably. In practice, this means using machine-readable formats such as JSON, organizing artifacts hierarchically, choosing reasonable and consistent file names, and adopting naming schemes that make simple tools such as regex search work well.
- **Make logs queryable through a small CLI (optional, but helpful).** Each harness gets a directory containing source code, scores, and execution traces, but as the history grows, raw filesystem access alone becomes cumbersome. A short CLI that lists the Pareto frontier, shows top-k harnesses, and diffs code and results between pairs of runs can make the experience store much easier to use, and querying such CLIs is closely aligned with the workflows on which coding agents are trained. If relevant offline experience exists (rollouts from other models, solved problem corpora, relevant papers), converting it into the same directory structure can also help warm-start exploration and ground new ideas. This layer helps the proposer save tokens it may have wasted on navigation.
- **Lightweight validation before expensive benchmarks.** Write a small validation test that imports the module, instantiates the class, and calls both methods on a tiny set of examples. Harnesses proposed during the search should pass this test before being fully evaluated. A simple test script can catch most malformed or nonfunctional candidates in seconds and keep the cost of failures near zero.
- **Automate evaluation outside the proposer.** Running evals is simple enough that it is not worth making the proposer do it. A separate harness should score candidates and write results to the filesystem.

E Extended Related Work

This appendix expands the brief discussion in Section 2 and situates Meta-Harness relative to several neighboring lines of work that we could not cover in detail in the main text. A recurring distinction is that Meta- Harness optimizes executable harness implementations and provides the proposer with selective access to prior code, scores, and execution traces via the filesystem.

AlphaEvolve / OpenEvolve. AlphaEvolve [34] and OpenEvolve [42] evolve code via LLM-guided mutations with structured feedback: the proposer receives a program database with scalar scores (4–22K tokens per step; Table 1) and applies fixed mutation strategies to tournament-selected parents. These methods are designed for algorithm discovery and optimization (mathematical conjectures, scheduling heuristics, hardware kernels), where the search target is a single stateless function with a clean scalar objective, and mutations are local. Harness engineering is a different regime: harnesses are stateful programs that accumulate experience across many examples, and a single design choice (e.g., what to store in memory) can cascade through an entire evaluation sequence. Meta-Harness addresses this by giving an unstructured coding agent full filesystem access, letting it selectively read any prior candidate's source code, execution traces, and scores.

GEPA. GEPA [1] is the closest text optimizer in terms of feedback richness, providing rollout traces per candidate. It is designed for prompt optimization on tasks with short feedback loops (math problems, instruction-following, code optimization), where each rollout is a single LLM call or a short pipeline. In this regime, per-candidate reflection works well: one prompt, one answer, one score. Harness engineering requires reasoning across many examples and many candidates simultaneously: understanding why a retrieval strategy works for one class of problems but degrades on another requires comparing execution traces across the full population. GEPA operates on one candidate at a time (2–8K tokens per step; Table 1), with a fixed critique format that must anticipate what information is relevant. Meta-Harness gives the proposer access to *all* prior candidates simultaneously and lets the agent decide what to examine.

Prompt orchestration frameworks. Several systems provide structured abstractions for composing multi-stage LLM programs. LMQL [8], LangChain [13], and DSPy [23] make prompt engineering more systematic by providing higher-level interfaces for prompt templates, control flow, and modular LLM pipelines. These frameworks help developers specify and organize LLM programs, but they still typically require manual design of retrieval policies, memory updates, and orchestration logic. Meta-Harness operates at a different level: it searches over the *implementation* of these policies in executable code, treating the harness itself as the optimization target.