MEMOHARNESS: AGENT HARNESSES THAT LEARN FROM EXPERIENCE

Anonymous authors
Paper under double-blind review

https://anonymous.4open.science/r/MemoHarness

ABSTRACT

An agent harness is the external control layer that turns a base LLM into an executable agent by managing context, tools, orchestration, memory, decoding, and output handling. While harness design strongly affects agent behavior, most automatic improvement methods optimize narrower artifacts such as prompts, pipelines, or workflows, and deployed agents usually reuse a single global harness for all cases. We introduce MEMOHARNESS, an adaptive harness optimization framework that learns from its own executions. MEMOHARNESS decomposes the harness into six editable control dimensions, stores per-case diagnoses and distilled global patterns in a dual-layer experience bank, and adapts the learned harness to each test case using retrieved experience without test-time labels, feedback, or additional search. In our evaluation across shell-agent, code-generation, and analytical-reasoning benchmarks, MEMOHARNESS improves over the fixed harnesses we compare against and shows selective transfer to unseen suites and base models. Its additional context can also remain cost-competitive when much of the retrieved experience is cacheable. These results provide evidence that execution experience is a practical substrate for building agent harnesses that are more adaptive than a single static configuration, while leaving broader claims about statistical robustness and component attribution to future work.

1 INTRODUCTION

The performance of an LLM agent depends not only on the underlying language model but on the entire *agent harness* that surrounds it. By *agent harness*, we mean the external control layer that turns a base LLM into an executable agent by specifying how context is constructed, which tools and retrieval channels are exposed, how inference is orchestrated across turns, what memory is retained, and how outputs are validated and returned. In practice, this includes the prompt construction policy, tool and retrieval interfaces, decoding parameters, orchestration topology, memory management, and output handling. Practitioner experience consistently shows that harness design alone can swing end-to-end task success by tens of percentage points with the same base model and the same tools (Lopopolo, 2026; LangChain, 2025).

Despite this, automated optimization of the *harness itself* remains rare. Most automatic improvement methods optimize a narrower artifact: prompts or instructions (Fernando et al., 2023), declarative LM pipelines (Khattab et al., 2023), or agent workflows (Zhang et al., 2024). These methods are valuable, but they do not jointly edit the full control layer that determines context assembly, tool access, generation policy, orchestration, memory, and output handling. Recent work on Meta-Harness (Lee et al., 2026) moves closer by treating harness code as the object of search, but the result is still a training-time artifact intended to be reused at deployment. The second missing piece is therefore *test-time adaptation*: a deployed agent should be able to specialize its harness to the particular case in front of it without test-time labels, feedback, or a new search run.

This setting creates three challenges. First, the harness is high-dimensional and coupled: changing retrieval can alter the useful prompt format, decoding budget, workflow topology, memory policy, and validation behavior. Second, benchmark scores alone are weak supervision for search; they reveal whether a run succeeded but not which harness dimension caused the failure or what should be

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

transferred to future cases. Third, test-time adaptation must avoid leakage: it may use only the visible input and experience accumulated during training, yet it must handle cases that differ in domain, ambiguity, reasoning depth, retrieval needs, and output format.

We address this gap with MEMOHARNESS, an agent harness that learns from its own past executions. MEMOHARNESS makes three design choices. **(1) Six-dimensional harness space.** We decompose the harness along the temporal flow of inference into six editable control surfaces (context, tool, generation, orchestration, memory, output), turning harness search into structured editing over separable dimensions rather than over a single opaque prompt (§2.3). **(2) Dual-layer experience bank.** During search, MEMOHARNESS accumulates both per-case execution entries and distilled global patterns about what works, what fails, and how dimensions interact, so that each iteration becomes diagnostic rather than only score-driven (§2.4). **(3) Test-time case adaptation.** At test time, the search-derived global harness W^* is adapted to a case-specific harness $W(x_j)$ by retrieving similar past cases and relevant patterns from the bank, without any test-time feedback, gradient updates, or extra search rounds (§2.6). A correctness-first ranking rule, with execution cost only as a tiebreaker, keeps the search from drifting onto cheap-but-wrong configurations. Figure 1 summarizes the full pipeline.

We evaluate MEMOHARNESS across benchmarks spanning shell tooling, code generation, and analytical reasoning. On the primary shell-agent benchmark under our evaluation protocol, MEMOHARNESS improves over the strongest fixed-harness baseline we run (0.806 vs. 0.722) while reducing per-task dollar cost relative to the strongest commercial baselines in that comparison. The learned harness also shows selective, not uniform, transfer without retraining: it improves some unseen evaluation suites and raises average task success across the held-out base models we evaluate by $+0.098$. These results point to a practical role for adaptive harnesses: execution experience can be reused to specialize the control layer case by case, although larger-scale studies and finer component ablations are needed to establish the robustness and source of the gains.

Contributions. The main contributions of this work are as follows:

- A *structured harness optimization framework* that decomposes a monolithic agent harness into six editable control surfaces and pairs them with a dual-layer experience bank, so that search accumulates reusable diagnostic knowledge rather than only scalar scores (§2.3 and §2.4).
- A *test-time adaptation mechanism* that adapts the search-derived global harness to each new case using retrieved successes, failures, and global patterns, without test-time feedback (§2.6).
- Empirical results across in-domain, cross-dataset, and cross-model evaluations showing improved task success under our protocol, selective positive transfer across unseen evaluation suites, gains on the evaluated base-model families, and favorable cost behavior when retrieved context is reusable through caching (§3).

2 METHOD

Overview. We study *adaptive harness optimization* for a fixed base language model. A *harness* is the surrounding system that determines what information is retrieved, how it is assembled, how model calls are orchestrated, and how raw outputs are post-processed. Rather than treating optimization as only a training-time search for one reusable global artifact, MEMOHARNESS learns a *global base harness* during search and applies a *case-specific adaptation rule* at test time. A *controller* uses retrieved execution experience to decide how to update the harness during search and adapt it to each test case. The method combines a structured six-dimensional harness space with a dual-layer experience bank and follows a **correctness-first** principle: the primary reward determines ranking, while lower execution cost serves only as a secondary tiebreaker. Figure 1 summarizes the full pipeline, which decomposes into a training-time search phase and a test-time case-adaptation phase.

2.1 MOTIVATION

LLM harnesses are usually deployed as a single benchmark-level configuration: one prompt template, decoding policy, tool interface, and orchestration pattern is selected and then reused for all cases. This is brittle when cases differ in domain, required reasoning depth, retrieval needs, or output format. A harness that is strong on average can still be systematically mismatched to particular case types.

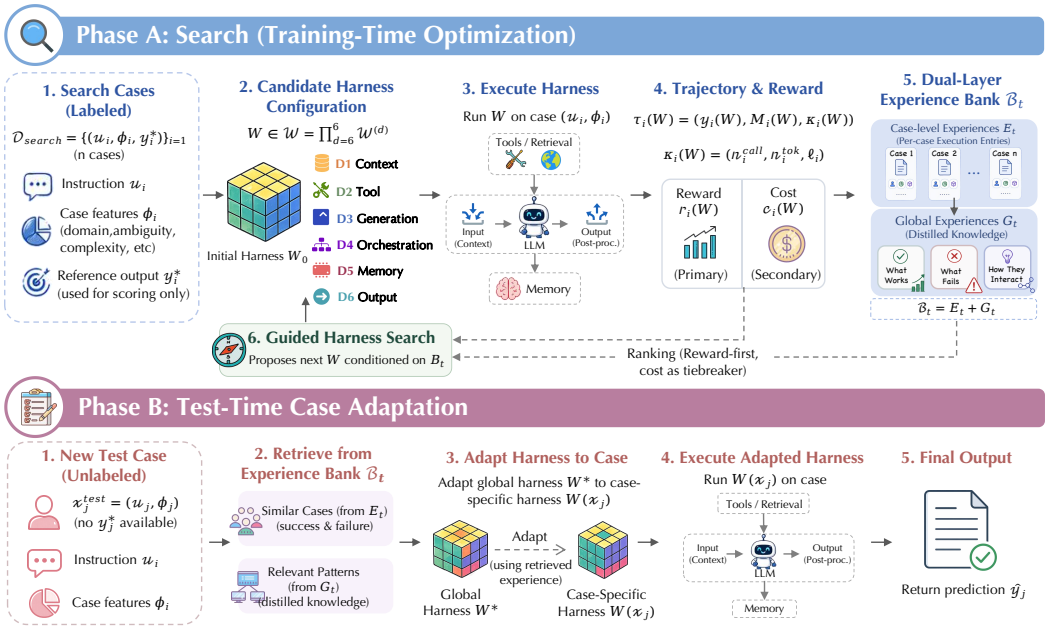

Figure 1: Overview of MEMOHARNESS. **Phase A (Search, training-time optimization)** runs a guided search over the six-dimensional harness space $\mathcal{W} = \prod_{d=1}^{6} \mathcal{W}^{(d)}$: each candidate harness W is executed on labeled search cases, scored by a correctness-first reward $r_i(W)$ with token usage $c_i(W) = n_i^{\text{tok}}$ as tiebreaker, and the resulting trajectories are stored in a dual-layer experience bank $\mathcal{B}_t = (E_t, G_t)$ of per-case entries and distilled global patterns; the search policy π then proposes the next W conditioned on $\mathcal{B}_t$. **Phase B (Test-Time Case Adaptation)** takes an unlabeled test case x_j^{test}, retrieves similar cases and relevant patterns from $\mathcal{B}_t$, and maps the global harness W^* to a case-specific harness $W(x_j)$, which is then executed to produce the final prediction $\hat{y}_j$.

MEMOHARNESS addresses this mismatch by making harness search diagnostic and case-adaptive. It decomposes the harness into six editable control dimensions, records which dimensions explain successes and failures, and uses that experience to adapt the learned global harness once per test case.

2.2 PROBLEM FORMULATION

Definition 2.1 (Benchmark Cases). *We distinguish between labeled search cases and unlabeled evaluation cases. The search set is*

$$\mathcal{D}_{\text{search}} = \left\{ x_i^{\text{s}} = (u_i,\ \phi_i,\ y_i^{\star}) \right\}_{i=1}^{n}, \tag{1}$$

where u_i is the input instruction, ϕ_i denotes case features such as domain, ambiguity, complexity, and whether external knowledge is required, and $y_i^{\star}$ is the reference output used only for search-time scoring. The evaluation set is

$$\mathcal{D}_{\text{test}} = \left\{ x_j^{\text{test}} = (u_j,\ \phi_j) \right\}_{j=1}^{m}. \tag{2}$$

At test time, the adaptation rule may condition on (u_j, ϕ_j) and the experience bank, but not on the unknown reference output.

Definition 2.2 (Harness Configuration). *For each dimension $d \in \{1, \ldots, 6\}$, let $\mathcal{W}^{(d)}$ denote the set of admissible choices for that functional control surface. A harness configuration is an element of the product space*

$$W \in \mathcal{W} = \mathcal{W}^{(1)} \times \mathcal{W}^{(2)} \times \cdots \times \mathcal{W}^{(6)}, \qquad W = \left(W^{(1)}, \ldots, W^{(6)}\right), \tag{3}$$

where each component $W^{(d)}$ controls one functional dimension of the inference pipeline (see Section 2.3).

Table 1: Six dimensions of the MEMOHARNESS harness space.

Dimension	Stage	Definition and example operation
D1 Context assembly	Pre-call input construction	*Builds the model input from instructions, constraints, retrieved material, and examples.* `structure prompt; add demos; compress context`
D2 Tool interaction	External tool and retrieval use	*Controls when and how the harness calls external tools or retrievers.* `enable retrieval; set top-k; rerank evidence`
D3 Generation control	Decoding configuration	*Sets the sampling and budget parameters for model generation.* `raise max tokens; lower temperature; sample candidates`
D4 Orchestration	Workflow topology	*Chooses the sequence of model calls and intermediate reasoning steps.* `single call -> plan/execute/refine`
D5 Memory management	Cross-call state persistence	*Determines what state persists across calls and what stale context is removed.* `keep state; summarize trace; drop stale context`
D6 Output processing	Post-call output handling	*Transforms raw model output into the final answer returned by the harness.* `extract answer; validate schema; choose fallback`

Executing the visible part (u_i, ϕ_i) of a case under configuration W produces a prediction $y_i(W)$. We denote the corresponding execution trajectory by $\tau_i(W)$; it records the prediction, intermediate model-tool trace (including invoked tools and intermediate outputs), and runtime statistics: model-call count, total token usage, and latency. We use total token usage $n_i^{\text{tok}}(W)$ as the search-time secondary cost, $c_i(W) = n_i^{\text{tok}}(W)$, with lower values preferred. Call count and latency are logged for diagnostic and reporting purposes but do not enter the harness-selection rule. Monetary cost in dollars is computed offline from token counts at the public list prices of the base model (RQ6) and is therefore separate from search-time selection. Given a user-specified primary metric R, the search-time task reward is $r_i(W) = R\big(y_i(W),\ y_i^\star\big)$, with larger values preferred.

The search problem is to identify a strong global harness $W^\star$ from candidate executions on $\mathcal{D}_{\text{search}}$. The evaluation-time problem is to construct, for each unlabeled case (u_j, ϕ_j), a specialized harness $W(x_j)$ using only test-visible information and retrieved experience.

2.3 SIX-DIMENSIONAL HARNESS SPACE

> **Design rationale.** Rather than treating the harness as a monolithic prompt or agent template, we decompose it according to the *temporal information flow* of inference. Each dimension corresponds to a distinct functional stage, enabling targeted diagnosis and repair while making cross-dimensional interactions explicit.

This decomposition (Table 1) turns harness search into structured editing over *separable control surfaces* instead of treating the entire harness as a single opaque object.

2.4 DUAL-LAYER EXPERIENCE BANK

MEMOHARNESS maintains an experience bank as a typed pair

$$\mathcal{B}_t = \big(\mathcal{E}_t,\ \mathcal{G}_t\big), \tag{4}$$

where $\mathcal{E}_t$ is the set of per-case execution entries accumulated up to search iteration t, and $\mathcal{G}_t$ is the set of distilled global patterns. We use a pair rather than a set union because entries and patterns have different schemas.

Per-case experience. For each search case and iteration, we store an execution entry

$$\xi_i^{(t)} = \Big(i,\, t,\, \phi_i,\, W_t,\, \Delta_i^{(t)},\, \tau_i(W_t),\, r_i(W_t),\, c_i(W_t),\, z_i^{(t)}\Big), \tag{5}$$

where $\Delta_i^{(t)}$ records the changes in W_t relative to the most recent harness previously applied to case i (or W_0 if no such execution exists). The diagnosis is produced by a diagnostic operator

$$z_i^{(t)} = g\big(x_i^{\mathrm{s}},\, W_t,\, \tau_i(W_t),\, r_i(W_t)\big), \tag{6}$$

whose output contains a success indicator $s_i^{(t)} \in \{0,1\}$, a primary failure dimension $d_{i,\mathrm{prim}}^{(t)} \in \{1,\dots,6\} \cup \{\emptyset\}$ (with $\emptyset$ when no failure dimension is assigned), secondary contributing dimensions $\mathcal{D}_{i,\mathrm{sec}}^{(t)}$, and a natural-language analysis $a_i^{(t)}$. We also maintain lightweight per-case statistics, including consecutive failures, recent average reward, reward trend, and dimension-wise failure counts.

Global patterns and retrieval. Every N search iterations, a distillation operator extracts persistent cross-case regularities from failure clusters,

$$\mathcal{G}_t \leftarrow \mathcal{G}_{t-1} \cup \mathrm{Distill}\big(\mathcal{E}_{\le t}\big), \tag{7}$$

where each global pattern summarizes a recurring phenomenon, its supporting evidence, and the expected effect of a targeted harness change. The controller does not read the entire bank directly; instead, it issues a structured query $q \in \mathcal{Q}$ over case features, failure statistics, iteration ranges, and dimension-specific diagnoses, yielding a retrieved slice $\mathcal{S}_t(q) = \mathrm{Retrieve}\big(\mathcal{B}_t,\, q\big)$. A retrieved slice may contain global patterns, filtered per-case entries, and aggregated statistics, which keeps the controller context bounded even as the bank grows over time.

2.5 Training-Time Harness Optimization

Training proceeds over the labeled search set $\mathcal{D}_{\mathrm{search}}$ for T iterations. We begin from a *minimal harness* W_0: no demonstrations, no structured instruction scaffolding, no external tools, deterministic single-call decoding, no cross-call memory, and raw output passthrough.

> **Starting from scratch.** Initializing from a minimal harness ensures that every component of the final configuration is *justified by empirical evidence* accumulated during search, rather than inherited from potentially suboptimal defaults.

At search iteration t, the controller forms a query q_t from the current harness and the accumulated bank, retrieves a bounded evidence slice $\mathcal{S}_{t-1}(q_t)$, and proposes the next configuration:

$$q_t = Q\big(W_{t-1},\, \mathcal{B}_{t-1}\big), \qquad \mathcal{S}_{t-1}(q_t) = \mathrm{Retrieve}\big(\mathcal{B}_{t-1},\, q_t\big), \qquad W_t = \Pi_{\mathrm{train}}\big(W_{t-1},\, \mathcal{S}_{t-1}(q_t)\big). \tag{8}$$

Here Q is a query constructor and Π_{train} is the retrieval-conditioned controller. The proposed harness is then executed on every search case $x_i^{\mathrm{s}} \in \mathcal{D}_{\mathrm{search}}$ to collect trajectories $\tau_i(W_t)$, compute rewards $r_i(W_t)$ and costs $c_i(W_t)$, and produce diagnosis signals $z_i^{(t)}$. All resulting entries are written to $\mathcal{E}_t$, and every N rounds the system distills persistent failure clusters into new entries in $\mathcal{G}_t$.

Correctness-first selection. For each candidate W_t, let $\bar{r}_t$ and $\bar{c}_t$ denote the arithmetic means of $r_i(W_t)$ and $c_i(W_t)$, respectively, over the n cases in $\mathcal{D}_{\mathrm{search}}$. Let $\mathcal{C}_T = \{W_0, \dots, W_T\}$ denote the explored candidates. If a cost budget or resource constraint is imposed, let $\mathcal{C}_{\mathrm{feas}} \subseteq \mathcal{C}_T$ denote the feasible candidates; otherwise $\mathcal{C}_{\mathrm{feas}} = \mathcal{C}_T$. The final global harness $W^\star$ is selected by *lexicographic ordering*:

$$W^\star \in \underset{\mathrm{lex},\, W_t \in \mathcal{C}_{\mathrm{feas}}}{\arg\max} \big(\bar{r}_t,\, -\bar{c}_t\big). \tag{9}$$

This makes the optimization objective explicit: primary reward is optimized first, while lower token usage ($\bar{c}_t$) serves only as a secondary tiebreaker among candidates with equal primary reward.

2.6 Test-Time Case Adaptation Without Feedback

After training, MemoHarness enters the evaluation phase on a test set $\mathcal{D}_{\mathrm{test}}$. Unlike training, test-time adaptation operates *without any iterative feedback loop*. For each unseen case $x = (u, \phi)$,

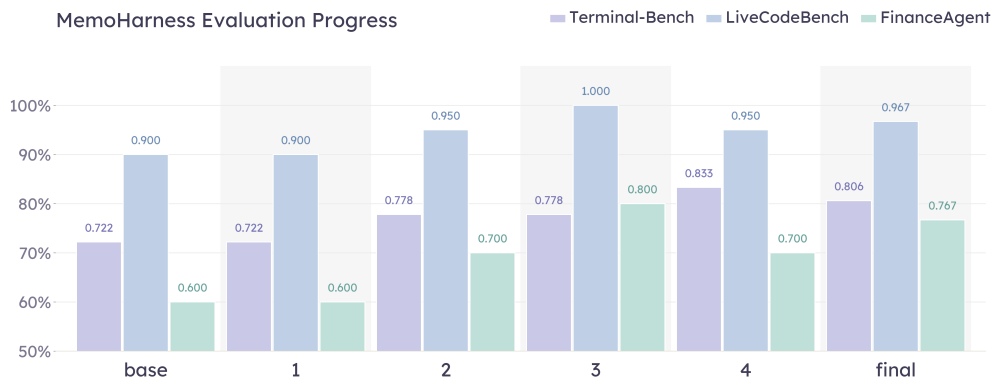

Figure 2: Harness quality at six checkpoints (`base`, four intermediate iterations, and the final selected harness) across three benchmarks. The final score is higher than `base` on every benchmark, but it is not always equal to the in-training peak: `LiveCodeBench` touches 1.000 at iteration 3, and `Terminal-Bench` reaches 0.833 at iteration 4 because the shipped harness is selected by validation rather than by peeking at the held-out evaluation split.

the controller adapts the learned global harness $W^\star$ using only test-visible information and the frozen experience bank $\mathcal{B}_T$.

Test-time case adaptation. For the current case, the controller retrieves global patterns, feature-matched historical slices, and the most similar successful and failed training examples. Let $\psi(u)$ denote an instruction representation, and let u_ξ denote the instruction associated with historical entry ξ. We score the similarity between a test case $x = (u, \phi)$ and an entry ξ by

$$\rho_\psi(x, \xi) = \cos\big(\psi(u),\ \psi(u_\xi)\big). \tag{10}$$

Let $s(\xi)$ denote the success indicator stored in entry ξ, and let $\mathcal{E}_T^+ = \{\xi \in \mathcal{E}_T : s(\xi) = 1\}$ and $\mathcal{E}_T^- = \{\xi \in \mathcal{E}_T : s(\xi) = 0\}$ denote successful and failed historical entries. The retrieved neighborhoods are

$$\mathcal{N}_K^+(x) = \underset{\xi \in \mathcal{E}_T^+}{\mathrm{TopK}}\big[\rho_\psi(x, \xi)\big], \qquad \mathcal{N}_K^-(x) = \underset{\xi \in \mathcal{E}_T^-}{\mathrm{TopK}}\big[\rho_\psi(x, \xi)\big], \tag{11}$$

where TopK returns the entries with the largest scores. Together with a feature-conditioned retrieval slice, these neighborhoods form the test-time evidence

$$\mathcal{S}_{\text{test}}(x) = \Big(\mathcal{N}_K^+(x),\ \mathcal{N}_K^-(x),\ \mathrm{Retrieve}\big(\mathcal{B}_T,\ Q_{\text{test}}(x)\big),\ \mathcal{G}_T\Big). \tag{12}$$

Conditioned on this evidence, the controller emits a case-specific harness

$$W(x) = \Pi_{\text{test}}\big(W^\star,\ x,\ \mathcal{S}_{\text{test}}(x)\big), \tag{13}$$

which is then executed for the current evaluation case. If evaluation labels or metrics later become available, the resulting trajectory may optionally be appended to the experience bank for future reuse, but no learning, re-selection, or distillation is triggered during the current evaluation.

Relation to global harness search. The central distinction of MEMOHARNESS is the separation of *global optimization* from *instance-level adaptation*. Training identifies a robust global harness under **correctness-first** selection, while evaluation uses the experience bank to specialize that harness to the requirements of each individual case. This allows simple cases to remain lightweight, while retrieval-heavy, multi-step, or format-sensitive cases can invoke richer orchestration only when the retrieved evidence warrants it.

3 EXPERIMENTS

We organize the evaluation around six research questions. RQ1 asks whether adaptive harness optimization improves absolute task success against existing agent harnesses. RQ2 examines whether search produces stable improvements over iterations and across task families. RQ3 tests the contribution of case adaptation and experience reuse, including sensitivity to experience-bank size. RQ4

tests whether learned harness changes transfer to unseen evaluation suites. RQ5 asks whether the same harness transfers across base models. RQ6 measures whether the added retrieval and adaptation machinery is cost-effective at inference time.

3.1 SETUP

We evaluate MEMOHARNESS on three task families: long-horizon shell agency (`Terminal-Bench`), single-shot code generation (`LiveCodeBench`), and multi-step financial reasoning (`FinanceAgent`). On `Terminal-Bench`, we compare against four released or benchmark-provided harness baselines under a shared evaluation protocol; when a baseline is not a model-agnostic harness, we use the closest reproducible configuration available in that system rather than claiming a pure scaffold-only transplant. For transfer, we also evaluate the learned harness on eight external suites and six additional base models. For the main evaluations, we report mean task success rate averaged over repeated runs, using the validation-selected harness rather than the in-training peak. Appendix C gives benchmark, baseline, model, and cost-accounting details.

3.2 RESULTS

RQ1 Does adaptive harness optimization improve task success?

Figure 3 compares MEMOHARNESS with two types of baselines on `Terminal-Bench`: four released or benchmark-provided harnesses (`Terminus-2`, `Claude Code`, `OpenCode`, and `Codex`), and two harness-optimization methods (Meta-Harness (Lee et al., 2026) and AHE (Lin et al., 2026)). All methods are evaluated on the same 18 held-out tasks. MEMOHARNESS achieves the highest success rate of 0.806, improving over `Codex`, the strongest released-harness baseline, by 0.084, and over Meta-Harness and AHE by 0.139 and 0.028, respectively.

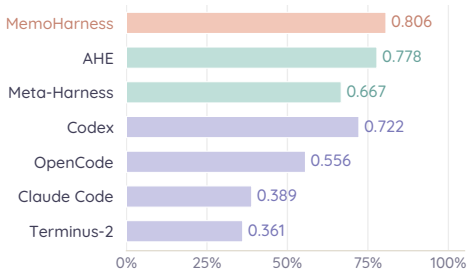

Figure 3: Baseline comparison on `Terminal-Bench`. Higher is better; MEMOHARNESS reaches the highest.

To evaluate performance on a larger held-out set, we additionally evaluate MEMOHARNESS under an alternative `Terminal-Bench` allocation with 33 search tasks and 56 held-out tasks. The full system achieves a success rate of 81.1% on this allocation, complementing the main 18-task evaluation.

`Terminal-Bench` tasks involve multi-step tool use and stateful execution, where planning, context management, and intermediate verification affect task completion. MEMOHARNESS uses experience from previous executions to refine these control decisions and adapt the learned harness to individual tasks.

RQ2 How does harness quality evolve during search?

To understand how the harness improves over the course of optimization, we examine its quality at intermediate checkpoints across all three benchmarks (Figure 2) and the per-iteration trajectory on the two non-`Terminal-Bench` benchmarks (Figure 4).

Figure 2 shows the final selected harness improves over `base` on all three benchmarks: $0.722 \rightarrow 0.806$ on `Terminal-Bench`, $0.900 \rightarrow 0.967$ on `LiveCodeBench`, and $0.600 \rightarrow 0.767$ on `FinanceAgent`. The largest gain appears on `FinanceAgent`, where the base harness leaves the most room; `LiveCodeBench` is near saturation and improves less. The final checkpoint is sometimes below the in-training peak because selection is validation-based not peak-test-score based.

Figure 4 shows that `FinanceAgent` continues to improve across search rounds, reaching 65.0% near iterations 8 and 9, while `LiveCodeBench` remains in a narrow 91.2%–95.0% band. Thus harness search helps most on longer-horizon agentic workloads, while still giving smaller gains in near-saturated single-shot code generation.

The different trajectories are useful diagnostically. On `FinanceAgent`, later iterations still discover beneficial edits, suggesting that multi-step analytical tasks expose diverse failure modes that can be repaired over time. On `LiveCodeBench`, by contrast, the base model already solves most

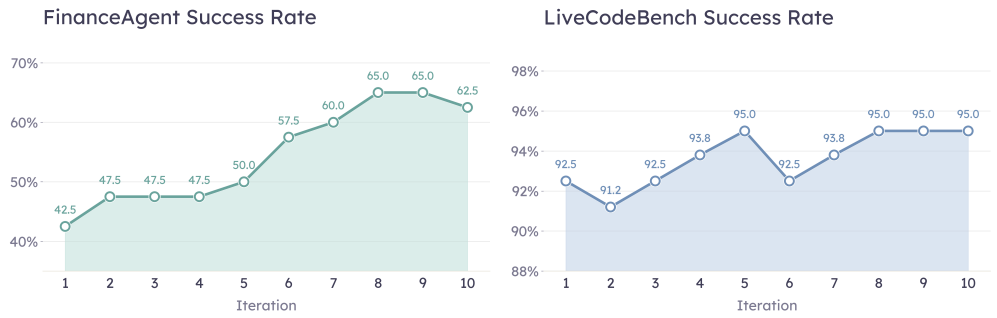

Figure 4: Per-iteration success rate on `FinanceAgent` (left) and `LiveCodeBench` (right) over 10 search rounds. `FinanceAgent` continues to benefit from additional iterations (rising from 42.5% to a 65.0% peak around iterations 8 and 9), whereas `LiveCodeBench` saturates almost immediately near the base-model ceiling and oscillates within a ∼4pt band.

Table 2: Component ablations and experience-bank size sensitivity on `Terminal-Bench`. Scores are mean ± run-level SD (%).

(a) Component ablations

Condition	Mean ± SD
$W^\star$ uniform	75.9 ± 5.2
$W(x)$ without global patterns	75.9 ± 6.9
$W(x)$ without retrieval	72.2 ± 4.5
Full $W(x)$	$\mathbf{80.6 \pm 5.3}$

(b) Bank-size sensitivity

Retained	Mean ± SD
25%	75.9 ± 2.6
50%	75.9 ± 10.5
75%	77.8 ± 0.0
100%	$\mathbf{80.6 \pm 5.3}$

cases, so search mostly operates in a low-headroom regime where small validation differences matter. This supports the use of validation-based selection and motivates the operation-level breakdown in Appendix G, which asks which atomic edits are associated with reward improvements.

RQ3 How do case adaptation and experience reuse contribute to performance?

To distinguish case adaptation from a stronger static harness, we compare full $W(x)$ with uniform $W^\star$ and remove global patterns or per-case retrieval. All conditions use the same test set and evaluation setup. We also vary the retained experience-bank fraction to test sensitivity to experience coverage.

Table 2 shows that full case adaptation raises mean success from 75.9% to 80.6%, a 4.7-point gain over uniform $W^\star$. Removing global patterns reduces the mean to 75.9%; removing retrieval causes the largest drop, to 72.2%. The full bank also achieves the highest mean, compared with 75.9–77.8% for smaller retained fractions, although variability does not decrease monotonically with bank size. Overall, these results suggest that both case adaptation and experience reuse contribute to the observed gains.

RQ4 Do learned harness changes transfer to unseen suites?

A learned harness is more convincing if it transfers not only across search checkpoints and base models, but also across evaluation suites that were never used during search. To test this, we take the final harness learned on each source benchmark under `GPT-5.3-Codex` and evaluate it on the eight external suites described in Appendix C. We select these suites to span diverse domains and interaction requirements, testing whether learned harness changes transfer beyond the task families used for search. For each source benchmark, we compare against the same untuned `Codex` base harness; all reported metrics are the suites' native scores.

Table 4 shows positive but selective transfer. The `Terminal-Bench` harness gives the broadest lift, improving `MMMLU` by +0.030, `SWE-Bench Pro` by +0.059, and `GAIA` by +0.052. The `LiveCodeBench` harness improves `MMMLU`, while the `FinanceAgent` harness mainly improves `LawBench` and `MedAgentBench` (+0.052). Saturated suites such as `HumanEvalFix` and `Reasoning-Gym-Easy` show no movement.

These results argue against two extreme. The learned harness is not a universally dominant prompt template: some source benchmarks produce conservative or domain-specific edits, and `LawBench`

Table 4: Cross-dataset generalization of search-derived harnesses under the shared base model GPT-5.3-Codex. The first row is the shared Codex baseline; each MemoHarness row is learned from the listed search source. HEFix, StrongR, RG-Easy, Law, SWE-Pro, and MedAgent abbreviate HumanEvalFix, StrongReject, Reasoning-Gym-Easy, LawBench, SWE-Bench Pro, and MedAgentBench, respectively. Higher is better.

Search source	Harness	MMMLU	HEFix	StrongR	RG-Easy	Law	SWE-Pro	GAIA	MedAgent
Shared baseline	Codex	0.818	1.000	0.879	0.947	0.675	0.706	0.895	0.474
Terminal-Bench	**MemoHarness**	**0.848**	1.000	**0.909**	0.947	**0.676**	**0.765**	**0.947**	**0.491**
FinanceAgent	**MemoHarness**	0.818	1.000	**0.909**	0.947	**0.682**	0.706	**0.912**	**0.526**
LiveCodeBench	**MemoHarness**	**0.879**	1.000	**0.909**	0.947	0.669	0.706	**0.912**	**0.491**

remains mixed. Yet the gains are not confined to the source benchmark. The strongest transfer appears when the source search task is long-horizon and tool-centric, suggesting that some learned control decisions, such as more robust instruction following or software-task structure, may survive a change in evaluation suite. Detailed behavioral analysis and failure cases are provided in Appendix H.

RQ5 Does the learned harness transfer across base models?

A learned harness is most useful if it transfers across base models without retraining. We take the harness produced by searching with GPT-5.3-Codex and apply it directly to six additional models spanning four families.

Table 3: Cross-model transfer on Terminal-Bench. Higher is better.

Model	Base	MemoHarness
GPT-5.3-Codex	0.722	**0.806**
Claude-Sonnet-4.6	0.530	**0.583**
Gemini-3.1-Pro	0.611	**0.694**
Qwen3.5-397B-A17B	0.444	**0.528**
GLM-5	0.500	**0.733**
GPT-4.1	0.500	**0.538**
DeepSeek-V3.2	0.333	**0.444**

Table 3 shows that every evaluated model improves over its base setting in our transfer study. The mean gain is $+0.098$, ranging from $+0.038$ on GPT-4.1 to $+0.233$ on GLM-5.

The transfer pattern is important because the harness was not re-searched for each model. Improvements on both proprietary and open-weight models suggest, but do not prove, that the learned changes are not purely model-specific prompt quirks. They are consistent with a more portable execution policy for how an LLM should gather context, invoke tools, manage intermediate state, and finalize outputs on terminal tasks. The smaller gain on GPT-4.1 suggests that stronger or already well-calibrated base settings may leave less room for harness intervention, but this pattern should be confirmed on larger model and task sets.

RQ6 Is adaptive harnessing cost-effective?

Table 9 reports token usage and dollar cost on the Terminal-Bench 18-task held-out evaluation split under GPT-5.3-Codex. Costs are computed offline from public list prices and are reported for comparison only; harness selection uses raw token usage as the secondary tiebreaker (Eq. 9). MEMOHARNESS uses more raw input tokens because it retrieves from the experience bank, but most of that context is cached in our evaluation (13.32M of 14.18M input tokens). Its total reported cost is therefore \$6.89, below Codex and Claude Code while achieving higher task success under this cost-accounting protocol. A detailed analysis of cost–accuracy trade-offs and cache-retention sensitivity is provided in Appendix I.

4 CONCLUSION

We presented MEMOHARNESS, an agent harness that turns past executions into reusable experience for future cases. By combining a structured six-dimensional harness space, a dual-layer experience bank, and test-time case adaptation, MEMOHARNESS makes harness search both diagnostic and reusable. The results suggest that improving the control layer around an LLM can be a practical complement to model scaling and manual harness engineering. Future work should study fully unsupervised search, larger-scale validation, more detailed component attribution, and online experience accumulation across deployments.

AI USE STATEMENT

In this work, we used generative AI tools to assist with code implementation and manuscript preparation. We have reviewed all AI-assisted work, LLM-generated code and the manuscript have been reviewed and checked by authors. The authors take responsibility for the final content of this work, including the manuscript, implementation, and reported results.

REFERENCES

Anthropic. Building effective agents. Anthropic Engineering Blog, December 2024. URL https://www.anthropic.com/engineering/building-effective-agents. Published December 19, 2024.

Anthropic. Claude Code: Agentic coding in the terminal. https://github.com/anthropics/claude-code, 2025a. GitHub repository, accessed 2026-04.

Anthropic. Writing effective tools for agents with agents. Anthropic Engineering Blog, September 2025b. URL https://www.anthropic.com/engineering/writing-tools-for-agents. Published September 11, 2025.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Y. He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?, 2026. URL https://openreview.net/forum?id=9R2iUHhVfr.

Zhiwei Fei, Xiaoyu Shen, Dawei Zhu, Fengzhe Zhou, Zhuo Han, Alan Huang, Songyang Zhang, Kai Chen, Zhixin Yin, Zongwen Shen, Jidong Ge, and Vincent Ng. LawBench: Benchmarking legal knowledge of large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7933–7962, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.452. URL https://aclanthology.org/2024.emnlp-main.452/.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023. URL https://arxiv.org/abs/2309.16797.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Yixing Jiang, Kameron C. Black, Gloria Geng, Danny Park, James Zou, Andrew Y. Ng, and Jonathan H. Chen. Medagentbench: A realistic virtual ehr environment to benchmark medical llm agents, 2025. URL https://arxiv.org/abs/2501.14654.

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.

LangChain. Context engineering. LangChain Blog, July 2025. URL https://blog.langchain.com/context-engineering-for-agents/. Published July 2, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. Autoflow: Automated workflow generation for large language model agents, 2024. URL https://arxiv.org/abs/2407.12821.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026. URL https://arxiv.org/abs/2604.25850.

Ryan Lopopolo. Harness engineering: Leveraging codex in an agent-first world. OpenAI Engineering Blog, February 2026. URL https://openai.com/index/harness-engineering/. Published February 11, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

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, 2026. URL https://arxiv.org/abs/2601.11868.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. OctoPack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mw1PWNSWZP.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. Codex CLI: Lightweight coding agent for the terminal. https://github.com/openai/codex, 2025. GitHub repository, accessed 2026-04.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs, 2024. URL https://arxiv.org/abs/2406.11695.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses, 2026. URL https://arxiv.org/abs/2603.25723.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search, 2023. URL https://arxiv.org/abs/2305.03495.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Alexandra Souly, Qingyuan Lu, Dillon Bowen, Tu Trinh, Elvis Hsieh, Sana Pandey, Pieter Abbeel, Justin Svegliato, Scott Emmons, Olivia Watkins, and Sam Toyer. A StrongREJECT for empty jailbreaks. In *The Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=KZLE5BaaOH.

SST. OpenCode: An open-source ai coding agent for the terminal. https://github.com/sst/opencode, 2024. GitHub repository, accessed 2026-04.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=GqYSunGmp7.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for ai software developers as generalist agents, 2024. URL https://arxiv.org/abs/2407.16741.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2023. URL https://arxiv.org/abs/2309.03409.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023a. URL https://arxiv.org/abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, 2023b. URL https://arxiv.org/abs/2210.03629.

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.

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, 2024. URL https://arxiv.org/abs/2410.10762.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023. URL https://arxiv.org/abs/2310.04406.

Appendix Contents

A	Limitations	p. 12
B	Implementation details	p. 13
C	Experimental details	p. 14
D	Models used	p. 14
E	Comparison scope	p. 15
F	Related work	p. 15
G	Operation-level diagnostic	p. 16
H	Behavioral analysis and failure cases	p. 16
I	Detailed cost analysis	p. 17

A Limitations

This study has several limitations. The main controlled Terminal-Bench experiments use an 18-task held-out split, complemented by a larger 33-search/56-test full-system evaluation; broader

controlled evaluation remains future work. The search phase currently assumes evaluable task feedback, such as reference outputs or executable verifiers, which may limit direct application to open-ended tasks with ambiguous or delayed rewards; test-time adaptation itself remains label- and feedback-free. Our current implementation optimizes a predefined six-dimensional harness space, leaving broader or automatically discovered control spaces unexplored. Cross-model transfer is evaluated using harnesses learned from a single source model, and the frozen experience bank leaves long-term experience accumulation, updating, and retirement under changing tasks and tools to future work.

B Implementation Details

This appendix describes how the abstract components of Section 2, namely the harness space, the diagnostic operator, and the cost proxy, are realized in our reference implementation. None of these choices alters the algorithmic structure of MemoHarness; they specify how the abstract objects are concretely instantiated in the experiments reported in Section 3.

Harness representation. A harness configuration is defined in the main text as a tuple $W = (W^{(1)}, \ldots, W^{(6)})$ over the six functional dimensions (Section 2.3). In the implementation, W is materialized as a *harness bundle*: a structured policy file that records the current D1 to D6 configuration, paired with lightweight textual scaffolding specifying the agent's operating rules, a persistent playbook, and the distilled memory currently in scope. An edit to a harness therefore corresponds to editing both the structured dimension summary and the accompanying instructions that instantiate those choices at execution time, keeping the textual scaffolding in sync with the typed dimension state.

Initialization. All experiments reported in this paper initialize from the minimal harness W_0 in which demonstrations, retrieval, structured scaffolding, cross-call memory, and output validators are all disabled. The implementation also supports resuming from a previously archived bundle when one is present, but this deployment-time convenience is not used for the reported benchmark results. In both modes, the search procedure optimizes over the same six-dimensional space $\mathcal{W} = \prod_{d=1}^{6} \mathcal{W}^{(d)}$.

Diagnostic instantiation. The diagnostic operator g from Section 2.4 is instantiated with verifier-grounded signals from the execution environment. Rather than requiring a full causal explanation for each failure, the implementation maps observable evidence, including verifier outcomes, exception types, timeouts, missing artifacts, command failures, and execution traces, to one of the six dimensions $d \in \{1, \ldots, 6\}$, yielding a per-case diagnosis $z_i^{(t)}$ that is intentionally coarse and serves as a stable optimization signal. More persistent structure is recovered at the global layer: when repeated failures accumulate, they are distilled into cross-case entries in $\mathcal{G}_t$ together with supporting evidence and a dimension-level repair suggestion.

Cost proxy. The main text instantiates the search-time cost as total token usage, $c_i(W) = n_i^{\mathrm{tok}}(W)$, while call count and latency are used only for diagnostic reporting (Section 2.2). The implementation follows this exactly: token usage is the only cost quantity that enters the lexicographic selection rule (Eq. 9), and dollar cost is computed offline at report time from token counts and the public list prices of the base model. Token usage was chosen because it is available consistently across runs and correlates directly with inference cost; call count and latency are logged for debugging and for the breakdowns in RQ6, but never affect selection. This preserves the correctness-first ordering described in the main text while avoiding dependence on less stable runtime measurements.

Global pattern distillation. The experience bank $\mathcal{B}_t = (\mathcal{E}_t, \mathcal{G}_t)$ pairs per-case execution entries with distilled global patterns (Section 2.4). In the implementation, distillation runs both on a fixed schedule (every N search rounds, as in the main text) and opportunistically when repeated failures accumulate on the same case. The distiller consumes recent failure histories, reward trends, dimension assignments, and compact configuration snapshots, and emits a bounded number of new global patterns into $\mathcal{G}_t$. Capping the number of newly emitted patterns per round keeps the controller's prompt budget bounded across iterations while preserving recurring lessons from prior executions.

C EXPERIMENTAL DETAILS

Benchmarks. We use three benchmarks chosen to span distinct task families. `Terminal-Bench` is a long-horizon shell-agent benchmark involving multi-step tool use, file editing, and process management. We use an 80/20 (8:2) train/evaluation split of the full 89-task suite for the main experiments, yielding an 18-task held-out evaluation split for absolute performance and cost reporting. A supplementary allocation uses 33 search tasks and 56 held-out tasks to evaluate the full system; its result is reported separately in RQ1. `LiveCodeBench` consists of recent competitive-programming problems and stresses single-shot code generation rather than multi-step agency. `FinanceAgent` requires multi-step analytical reasoning over financial documents and tool calls, exercising the longer-horizon analytical end of the workload spectrum. Together the three benchmarks cover shell tooling, code generation, and analytical reasoning, three regimes in which harness design plays qualitatively different roles.

For cross-dataset evaluation, we test the learned harnesses on eight external suites that were not used during search: `MMMLU` (Hendrycks et al., 2021), `HumanEvalFix` from OctoPack (Muennighoff et al., 2024), `StrongReject` (Souly et al., 2024), `Reasoning-Gym-Easy` (Stojanovski et al., 2026), `LawBench` (Fei et al., 2024), `SWE-Bench Pro` (Deng et al., 2026), `GAIA` (Mialon et al., 2023), and `MedAgentBench` (Jiang et al., 2025). Together, these suites span knowledge-intensive question answering, code repair, safety refusal behavior, verifiable reasoning, legal reasoning, long-horizon software engineering, general assistant tasks, and clinical workflows, testing transfer across diverse domains and interaction requirements beyond the task families used for search.

Baselines. On `Terminal-Bench` (Merrill et al., 2026) we compare against four agent-harness frameworks: OpenAI's `Codex` (OpenAI, 2025), Anthropic's `Claude Code` (Anthropic, 2025a), the open-source `OpenCode` (SST, 2024), and `Terminus`, the neutral test-bed agent that ships with the benchmark. We run each baseline using its released default configuration or the closest reproducible configuration available under the shared evaluation protocol. Where the framework permits a model-agnostic generator swap, we use `GPT-5.3-Codex`; where it does not, we treat the result as a comparison to the released system rather than as a pure scaffold-only ablation. Of the four, `Codex` is the strongest baseline in our setup and serves as the most demanding reference point for MEMOHARNESS.

Models and protocol. The harness search uses `GPT-5.3-Codex` as the base model. For cross-model transfer, we evaluate the search-derived harness, without any further training or retraining, on six other base models from four families: `Claude-Sonnet-4.6`, `Gemini-3.1-Pro`, `Qwen3.5-397B-A17B`, `GLM-5`, `GPT-4.1`, and `DeepSeek-V3.2`. For the main evaluations, we report mean task success rate per benchmark, averaged over repeated runs. Because reported values are run averages, they are not constrained to integer multiples of the 18 held-out `Terminal-Bench` tasks. The harness used at evaluation time is the one selected by validation at the end of search, not the in-training peak shown in Figures 2 and 4. For cost reporting, token counts are averaged over repeated runs on the `Terminal-Bench` evaluation split; dollar costs use the public list prices of `GPT-5.3-Codex` for cached input, non-cached input, and output tokens.

Hyperparameters and splits. The Controller–Bank loop runs for $T = 10$ outer iterations. Global pattern distillation uses a dual trigger: it runs after $M = 5$ new bank entries or after $N = 3$ consecutive failures on the same case, whichever occurs first. At each round, the controller receives a compact bank summary containing $K_{\text{succ}} = 10$ recent successes and $K_{\text{fail}} = 10$ recent failures. We do not use semantic retrieval for these summaries ($D2.\texttt{top_k} = 0$). Dataset splits in the main experiments use a fixed 80/20 (8:2) random partition with seed 42, so repeated runs and the held-out evaluation pass use identical task IDs. Generation is deterministic in all reported runs: temperature 0.0, top-$p = 1.0$, candidate count 1, and maximum generation budget 8192 tokens.

D MODELS USED

Table 5 lists the seven base models evaluated in this paper. The harness search itself uses `GPT-5.3-Codex` as the source model (Section 3.1); the remaining six models participate only in the cross-model transfer study in RQ5, where they execute the search-derived harness with no further

Table 5: Base models used in this paper. "Access" indicates whether the model's weights are publicly released or served only via a proprietary API. `GPT-5.3-Codex` is the source model used during harness search; the remaining six are evaluated only at test time, as described in RQ5.

Model	Provider	Access
`GPT-5.3-Codex`	OpenAI	Proprietary (API)
`Claude-Sonnet-4.6`	Anthropic	Proprietary (API)
`Gemini-3.1-Pro`	Google DeepMind	Proprietary (API)
`GPT-4.1`	OpenAI	Proprietary (API)
`Qwen3.5-397B-A17B`	Alibaba	Open weights
`GLM-5`	Zhipu AI	Open weights
`DeepSeek-V3.2`	DeepSeek	Open weights

training. "Access" distinguishes proprietary models served only behind an API from models whose weights have been publicly released.

E Comparison Scope

Meta-Harness (Lee et al., 2026) is the closest prior system to our training-time search setting because it also treats the harness as the object of optimization. We compare MemoHarness with Meta-Harness and AHE (Lin et al., 2026) on the same `Terminal-Bench` test set. These comparisons evaluate complete systems; model and tool configurations are not fully matched. RQ3 reports controlled ablations of case adaptation and experience reuse.

F Related Work

F.1 Optimization for Agents

Recent work studies how to automatically improve prompts, inference policies, and agentic workflows rather than treating them as fixed hand-written artifacts. Early agentic prompting and tool-use methods such as ReAct (Yao et al., 2023b) and Toolformer (Schick et al., 2023) established that performance improves substantially when inference is augmented with actions, tools, and external feedback. Planning-oriented inference methods such as Tree of Thoughts (Yao et al., 2023a) and Language Agent Tree Search (Zhou et al., 2023) further demonstrated that search over intermediate reasoning states can improve deliberative decision making.

Prompt and instruction optimization subsequently became an explicit objective. OPRO treats the language model itself as an optimizer over instruction space (Yang et al., 2023); ProTeGi performs prompt optimization using textual gradients and beam search (Pryzant et al., 2023); and Promptbreeder evolves both task prompts and mutation prompts (Fernando et al., 2023). A complementary line of work improves generation behavior through iterative self-feedback: Self-Refine repeatedly generates feedback and revisions (Madaan et al., 2023), while Reflexion stores verbal reflections in memory to improve subsequent trials (Shinn et al., 2023). However, these methods optimize *within a single prompt turn* and do not consider the broader system-level configuration that surrounds the model.

System-level optimization methods move beyond a single prompt. DSPy formulates language-model pipelines as declarative programs that can be compiled against a task metric (Khattab et al., 2023); MIPRO extends this line by optimizing instructions and demonstrations for multi-stage language-model programs (Opsahl-Ong et al., 2024); and TextGrad casts optimization of compound AI systems as textual backpropagation (Yuksekgonul et al., 2024). For agent workflows, AutoFlow automatically generates natural-language workflow programs (Li et al., 2024), and AFlow optimizes code-represented agent workflows with search and execution feedback (Zhang et al., 2024). Closest to our setting, Meta-Harness searches directly over harness code (Lee et al., 2026), and AlphaEvolve demonstrates the power of evolutionary code-level improvement under evaluator feedback (Novikov et al., 2025). *Across this line of work, optimization is primarily a pre-deployment procedure over prompts, programs, workflows, or harness code. Less explored is harness-level adaptation to the*

specific requirements of each individual test case at deployment time. MEMOHARNESS addresses this gap through its dual-layer experience bank and test-time case adaptation.

F.2 HARNESS ENGINEERING

Recent practitioner work increasingly argues that agent performance depends as much on the surrounding harness as on the underlying model. Anthropic's *Building Effective Agents* emphasizes simple compositional workflows, careful context management, and deliberate tool design in production systems (Anthropic, 2024). Their subsequent engineering note on tool design argues that tool naming, descriptions, response shaping, and evaluation directly affect both task success and efficiency (Anthropic, 2025b). LangChain's writing on *context engineering* similarly frames reliability as a problem of deciding what should be written, selected, compressed, and isolated across long agent trajectories (LangChain, 2025). OpenAI's *Harness engineering* post extends this view to software agents, stressing repository legibility, structured in-repo knowledge, and feedback loops as first-class engineering artifacts rather than incidental implementation details (Lopopolo, 2026).

Academic work has only recently begun to formalize these intuitions. Meta-Harness makes the harness itself the object of search (Lee et al., 2026), while Natural-Language Agent Harnesses externalize harness behavior as editable natural-language artifacts executed by a shared runtime (Pan et al., 2026). AHE (Lin et al., 2026) refines coding-agent harnesses through an observability-driven loop that uses structured execution evidence and task-level feedback to guide edits to harness components. `Terminal-Bench` provides a realistic command-line evaluation substrate on which different software-agent harnesses can be compared (Merrill et al., 2026). In parallel, SWE-agent and OpenHands demonstrate that interface design, environment access, and execution scaffolding are central determinants of agent performance rather than incidental implementation details (Yang et al., 2024; Wang et al., 2024). *While this emerging literature establishes harness engineering as a first-class concern, we are not aware of prior work that accumulates reusable diagnostic experience across search iterations and leverages it for case-level adaptation at test time*. This is the contribution that MEMOHARNESS makes.

G OPERATION-LEVEL DIAGNOSTIC ON ADJACENT-ITERATION TRANSITIONS

A scalar score curve only tells us *whether* the harness improved at each iteration; it does not tell us *which* edits were associated with the improvement. To get at this, we examine every adjacent-iteration transition (*previous*→*next*) on a fixed task and ask: when a particular atomic shell operation appears in the harness output for the first time (absent in the previous iteration's harness output, present in the next), is reward more likely to go up than the all-transition baseline? Table 6 reports, for each frequently-newly-added operation, the number of newly-added occurrences (n_{add}), the number of those followed by a reward increase (n_{pos}), the resulting positive-transition rate, the lift of that rate over the all-transition baseline (in percentage points), and the same lift expressed as a relative change. Operations such as `cat`, `sed`, `which`, and `test` are strongly associated with reward improvement (lift ranging from $+17$ to $+60$ pp), suggesting that they often appear in transitions that repair inspection or condition-checking gaps. By contrast, `curl`, `echo`, and `grep` are weakly or negatively associated in this analysis, indicating that their appearance is less predictive of reward-improving transitions. This kind of operation-level decomposition is only possible because the dual-layer experience bank stores per-case execution traces alongside the scalar score, and it provides a direct signal for which directions the search policy should be biased toward in subsequent iterations.

H BEHAVIORAL ANALYSIS AND FAILURE CASES

MMMLU behavior across search sources. The three source harnesses are evaluated on the same 33 `MMMLU` task IDs, with matching outcomes on 30 tasks. Table 7 reports success counts and tool and token usage. The `LiveCodeBench` source solves the most tasks despite using fewer web searches and lower average token usage than the other two sources. Trajectory analysis suggests that the three differing outcomes involve retrieval triggering, evidence integration, confirmation bias, and translation preservation, rather than tool-call volume alone.

Table 6: Operation-level lift analysis on adjacent-iteration transitions. The unit of observation is a single *previous→next* iteration pair on a fixed task in which a given atomic operation was *newly added* (absent before, present after). n_{add}: newly-added occurrences; n_{pos}: occurrences followed by a reward increase; *Pos. rate*: $n_{\text{pos}}/n_{\text{add}}$; *Lift (pp)*: *Pos. rate* minus the all-transition baseline rate ($\sim$13.2%), in percentage points; *Rel. lift*: the same lift relative to the baseline. Green: above-baseline lift; red: below-baseline.

Operation	n_{add}	n_{pos}	Pos. rate	Lift (pp)	Rel. lift
`cat`	11	8	72.7%	+59.5	+451.9%
`sed`	11	4	36.4%	+23.2	+175.9%
`which`	15	5	33.3%	+20.2	+152.9%
`test`	46	14	30.4%	+17.3	+130.9%
`pip`	10	3	30.0%	+16.8	+127.6%
`python3`	19	5	26.3%	+13.1	+99.7%
`strings`	4	1	25.0%	+11.8	+89.7%
`pdftotext`	15	3	20.0%	+6.8	+51.8%
`head`	5	1	20.0%	+6.8	+51.8%
`grep`	9	1	11.1%	−2.1	−15.7%
`echo`	28	3	10.7%	−2.5	−18.7%
`curl`	19	1	5.3%	−7.9	−60.1%
`file`	3	0	0.0%	−13.2	−100.0%
`jq`	3	0	0.0%	−13.2	−100.0%
`rg`	3	0	0.0%	−13.2	−100.0%
`apt-get`	2	0	0.0%	−13.2	−100.0%
`wc`	2	0	0.0%	−13.2	−100.0%
`pip3`	2	0	0.0%	−13.2	−100.0%

Table 7: `MMMLU` behavior on the same 33 task IDs. Search and execution counts are aggregate calls; the last column is average total tokens per task.

Source	Solved	Web searches	Exec calls	Tasks with tools	Average tokens
`FinanceAgent`	27/33	18	0	4	12,458
`LiveCodeBench`	29/33	15	0	5	12,421
`Terminal-Bench`	28/33	18	3	3	15,985

Terminal-Bench boundary cases. On `qemu-alpine-ssh`, MEMOHARNESS succeeds in three of six runs. The reported Codex failure ends with `ssh: connect to host localhost port 2222: Connection refused`. Successful MEMOHARNESS trajectories repeatedly check QEMU process state, serial/monitor sockets, DHCP, `sshd`, and port 2222 until the verifier passes. In contrast, `filter-js-from-html` fails in all six runs, while `polyglot-c-py` and `protein-assembly` succeed in only two of six runs each. Failure analysis identifies non-interference, final-file hygiene, and strict sequence invariants as remaining constraints across these cases. These qualitative observations characterize execution behavior and its limits without isolating the effect of case-specific adaptation.

I DETAILED COST ANALYSIS

Cost–accuracy trade-offs. Table 8 reports observed cost–accuracy results for released-system configurations on the `Terminal-Bench` 18-task held-out split, with `GPT-5.3-Codex` as the base model for MEMOHARNESS. Dollar costs use API token counts and public list prices. This cost summary is reported separately from the aggregate token-cost comparison in Table 9.

With 94.15% of its input tokens cached, MEMOHARNESS costs only \$0.0027 more per task than `Codex`, with a 2.8-percentage-point higher mean success rate and a slightly lower cost per success.

Table 8: Cost analysis on `Terminal-Bench` (18-task held-out evaluation split). Costs are in USD. Cost per success is average cost per task divided by mean success rate; cache share is the fraction of input tokens served from cache. MEMOHARNESS success is averaged over six runs.

Framework	Success (%)	Cost / task ($)	Cost / success ($)	Cache share (%)
Codex	77.8	0.3555	0.4571	94.64
Terminus-2	27.8	0.3664	1.3190	22.92
Claude Code	38.9	0.6100	1.5686	39.01
OpenCode	55.6	0.1738	0.3129	93.65
MemoHarness	**80.6**	**0.3582**	**0.4444**	**94.15**

Table 9: Cost analysis on `Terminal-Bench` (18-task held-out evaluation split) with `GPT-5.3-Codex` as the shared base model. Token counts are reported in millions; lower dollar cost is better.

Framework	Input (M)	Cached (M)	Non-cached (M)	Output (M)	Cost ($)
Codex	8.23	4.33	3.90	0.19	10.28
Terminus	3.96	0.94	3.03	0.09	6.68
Claude Code	7.32	3.11	4.21	0.11	9.51
OpenCode	5.48	5.07	0.41	0.05	2.34
MemoHarness	**14.18**	**13.32**	**0.86**	**0.22**	**6.89**

It is both cheaper and more accurate than `Claude Code` and `Terminus-2`, while `OpenCode` remains cheaper per task and per success at lower accuracy. These comparisons reflect observed usage rather than a matched-budget evaluation.

Sensitivity to cache retention. Cached input is measured via the API field `prompt_tokens_details.cached_tokens`; fresh input is total input minus cached input. For cached input n_c, fresh input n_f, output n_o, and their respective per-token prices p_c, p_f, p_o, we reprice the same execution records while retaining a fraction α of the observed cached input:

$$C(\alpha) = \alpha n_c p_c + \big[n_f + (1-\alpha)n_c\big]p_f + n_o p_o. \tag{14}$$

Here $\alpha = 1$ reproduces observed cache usage, while $\alpha = 0$ charges all input at the non-cached rate; execution records and success rates are held fixed.

Table 10: Sensitivity of cost per task (USD) to retention of observed cached input tokens. Full retention preserves the measured cache share of each method; it does not assume that all input tokens are cached.

Framework	Cache retention (α)				
	0%	25%	50%	75%	100%
Codex	1.4309	1.1620	0.8932	0.6244	0.3555
Terminus-2	0.4439	0.4245	0.4051	0.3858	0.3664
Claude Code	0.8809	0.8131	0.7454	0.6777	0.6100
OpenCode	0.6806	0.5539	0.4272	0.3005	0.1738
MemoHarness	**1.4440**	**1.1726**	**0.9011**	**0.6297**	**0.3582**

Table 10 shows that MEMOHARNESS cost rises from $0.3582 per task at full retention to $1.4440 without cache reuse. It remains close to `Codex` throughout, while `OpenCode` remains cheaper. At zero retention, MEMOHARNESS also costs more than `Claude Code` and `Terminus-2`, so its cost advantage over these two systems depends on cache reuse.