MESH-HARNESS: SELF-IMPROVING AGENT HARNESSES VIA BANDIT-GUIDED COMPOSITIONAL EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

An *agent harness* is the code that organizes context, maintains state, and coordinates tool calls for a language model. We study how to improve the harness under a limited evaluation budget while keeping model weights fixed. Our method, **MESH-Harness**, organizes each harness into functional modules with explicit role-specific interfaces, allowing alternative implementations of each module to be substituted and recombined. It uses shared module representations and full-covariance LinUCB to score candidate combinations based on predicted performance and exploration value. Mixed-start coordinate ascent selects complete configurations for evaluation without enumerating the combinatorial space. Validation traces then guide local code edits, and the resulting candidates are incorporated into fixed-capacity role-specific pools for subsequent recombination. On text tasks, retrieval-augmented mathematical reasoning, code generation, and interactive scientific tasks, MESH-Harness outperforms Meta-Harness by 5.70, 7.01, 2.00, and 5.00 points, respectively, under matched candidate-evaluation budgets. Iterative harness optimization improves MESH-Harness by 5.63–7.79 points over its first-round configurations. For the reported configurations, aggregate test-time cost is 44.2% lower than that of Meta-Harness, while total cost including search is 14.6% lower. These results show that combining module-level design reuse with feedback-driven compositional search can systematically improve agent harnesses while keeping overall optimization cost under control.

1 INTRODUCTION

Large language models (LLMs) can act as agents by retrieving information, calling tools, and interacting with environments (Yao et al., 2023). Their behavior also depends on the external program that controls memory, context construction, tool use, and failure recovery. This program is referred to as the *agent harness*. Unlike a single prompt, a harness specifies how successive model calls work together to solve a task. For example, harnesses for long-running agents support this coordination through explicit state handoffs, progress records, and execution checks (Young, 2025). Because the harness governs how model calls work together, optimizing it offers a way to improve an agent without updating model weights. AutoHarness synthesizes action validators and policy code through iterative refinement using environment feedback (Lou et al., 2026). Meta-Harness searches over task-specific Python programs, allowing a proposer agent to inspect previous candidates' code, scores, and execution traces (Lee et al., 2026). Both approaches make the surrounding program an object of optimization rather than a fixed implementation.

The difficulty is that evaluating a harness can be much more expensive than proposing it. Even a small edit may require an end-to-end validation run involving many model calls and environment interactions. Meta-Harness likewise identifies candidate evaluation as its main computational bottleneck (Lee et al., 2026). Under a limited budget, the search must therefore decide not only what to generate, but what to execute. We study how to use past evaluations to make these decisions and obtain larger improvements within a fixed number of candidate evaluations.

We introduce **MESH-Harness**, a bandit-guided framework for compositional harness evolution. Contextual bandits use shared features to generalize from observed rewards and balance predicted performance against uncertainty (Li et al., 2010; Chu et al., 2011). To apply this idea to harnesses, we organize each harness into functional slots with fixed interfaces, maintain a candidate pool

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

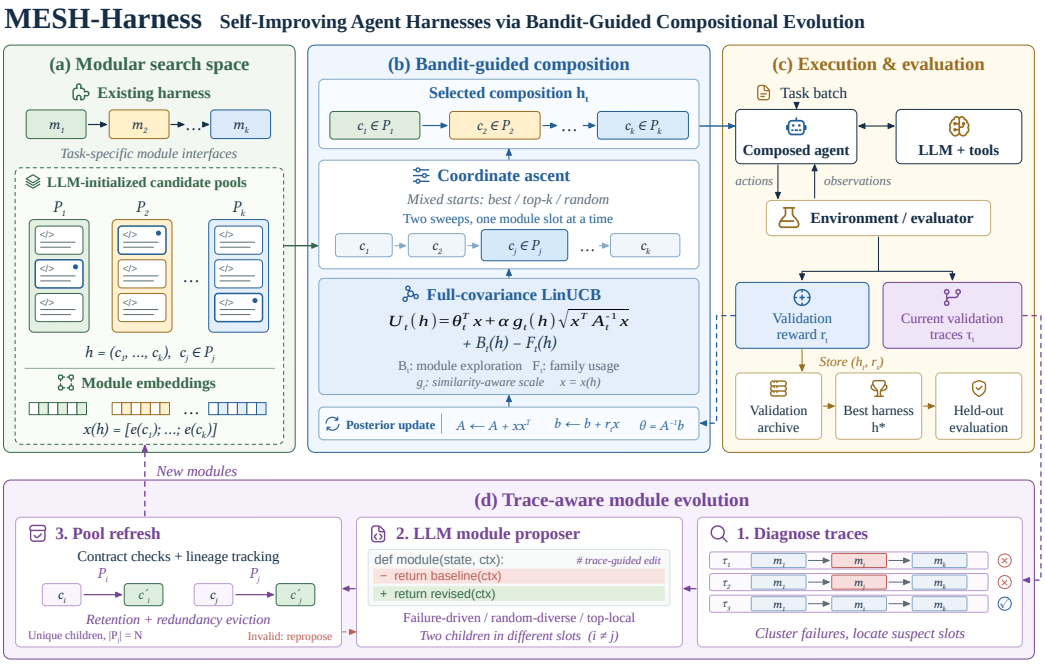


Figure 1: **Overview of MESH-Harness.** (a) Each functional slot has a fixed interface and a pool of LLM-initialized implementations with cached embeddings. (b) Full-covariance LinUCB scores configurations with one implementation per slot, and mixed-start coordinate ascent selects a configuration for evaluation. (c) The selected harness is executed to obtain validation rewards and traces. (d) Trace diagnosis guides edits to implementations and fixed-capacity pool updates. Rewards update the bandit model; new implementations join their respective pools for subsequent composition.

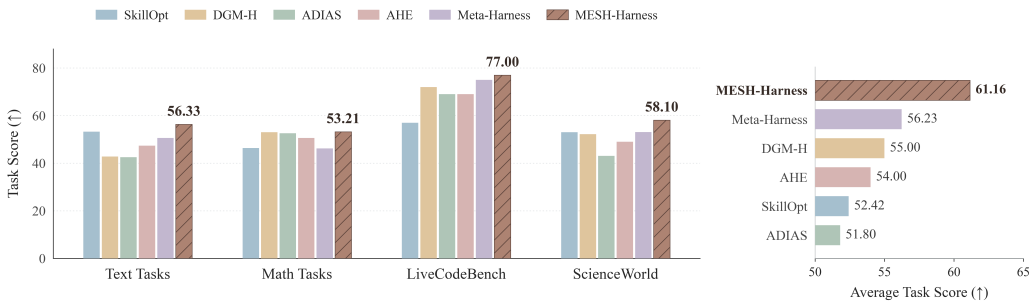
per slot, and select one implementation per slot. Each implementation is encoded separately; concatenating the selected embeddings in slot order represents a complete harness. A shared surrogate learns from observed harness scores to predict the performance of untested combinations of module implementations.

Full-covariance LinUCB combines these predictions with an uncertainty bonus to score module combinations. We approximately maximize this criterion through mixed-start coordinate ascent, drawing one start per round from the incumbent, top-K archived configurations, or a random combination and substituting implementations within one slot at a time rather than enumerating the Cartesian product of candidate pools. Selection uses only cached representations and surrogate scores; the chosen complete harness is then executed on the validation tasks. This separates inexpensive candidate comparison from costly task evaluation.

Selection alone cannot discover implementations absent from the candidate pools. MASPOB selects prompt combinations using bandit-guided coordinate ascent but does not update its candidate pools (Hong et al., 2026). MESH-Harness instead updates the available implementations in each slot during search. Validation traces provide evidence of local failures, such as missing information or repeated unsuccessful actions, and guide an LLM in editing implementations while preserving their slot interfaces. New implementations enter their slot's fixed-capacity pool for combination with candidates in other slots. Each execution thus supplies two kinds of feedback: scores update the selection model, while traces guide module evolution. A new implementation need not trigger an immediate end-to-end evaluation.

We evaluate MESH-Harness on text tasks, retrieval-augmented mathematical reasoning, LiveCodeBench (Jain et al., 2025), and ScienceWorld (Wang et al., 2022). Under matched candidate-evaluation budgets, it improves final test scores over Meta-Harness by 5.70, 7.01, 2.00, and 5.00 points, respectively. MESH-Harness improves by 5.63–7.79 points over its first-round configurations. On ScienceWorld, it starts at 51.90, below the ReAct reference score of 55.50, and reaches 58.10. Across all four tasks, the reported configurations reduce aggregate test-time cost by 44.2% relative to Meta-Harness. Total cost, including search and testing the reported configurations, is 14.6% lower.

Our contributions are:



(a) Performance comparison across tasks.

(b) Overall performance comparison.

Figure 2: **Test performance across four task domains.** (a) Text macro-average, overall Math pass@1, LiveCodeBench solve rate under the fixed cap-8 protocol, and ScienceWorld mean progress score. (b) Unweighted mean of the four task scores. All scores are on a 0–100 scale, with higher values indicating better performance. MESH-Harness achieves the highest score on all four aggregate metrics.

- A bandit-guided harness optimizer that allocates a limited evaluation budget using shared performance predictions and uncertainty estimates.
- A module-level representation and mixed-start coordinate ascent procedure that reuse evaluation feedback across configurations without enumerating the full combinatorial space.
- A trace-driven module evolution mechanism, evaluated on four tasks in terms of final performance, search gains, and search and test costs.

2 Related Work

Improving agents through external components. Recent work improves agents by optimizing the external artifacts that govern their behavior, ranging from natural-language skill documents to executable harnesses. ACE (Zhang et al., 2026c) incrementally curates execution experience into context playbooks, while SkillOpt (Yang et al., 2026) refines reusable skill documents through trajectory-driven edits and validation-gated updates. MCE (Ye et al., 2026) co-evolves context artifacts and the context-engineering skills that produce them. At the code level, Meta-Harness (Lee et al., 2026) gives an agentic proposer access to historical programs, scores, and traces to guide harness revision. ADIAS (Jiang et al., 2026) maintains persistent issue states to direct full-code modifications, and DGM-H (Zhang et al., 2026a) makes both the task agent and its self-improvement procedure editable. AEvo (Zhang et al., 2026b) uses accumulated evolution evidence to guide a meta-agent in editing the search procedure or agent context that governs subsequent evolution. Closely related to our work, AHE (Lin et al., 2026) exposes harness components as editable files and uses structured trajectory evidence and edit-outcome checks to guide revision and rollback. Whereas AHE organizes optimization around successive revisions to a working harness, MESH-Harness maintains alternative implementations in per-slot pools and learns which complete compositions to evaluate. Implementations discovered in different iterations can thus be combined by selecting one per slot, without requiring another LLM rewrite.

Bandit-guided search under limited evaluation budgets. Feature-based bandits (Li et al., 2010; Chu et al., 2011) provide a way to allocate evaluations by sharing reward information across candidates and balancing exploitation with uncertainty-driven exploration. TRIPLE (Shi et al., 2024) casts prompt selection as fixed-budget best-arm identification. For multi-agent systems, MASPOB (Hong et al., 2026) combines a topology-aware GNN surrogate with UCB-guided coordinate search to select prompt combinations from given candidate domains. MESH-Harness applies uncertainty-guided selection to configurations with one executable implementation per functional slot. A shared reward surrogate over component representations allows past evaluations to inform the ranking of untested compositions, while coordinate search avoids enumerating the full combination space. Unlike search over an unchanged candidate domain, this selection process is interleaved with trace-driven component evolution, so new implementations can enter the pools as optimization proceeds.

Together, these choices separate implementation generation from evaluation allocation. MESH-Harness reuses existing implementations without additional LLM rewriting, shares feedback across candidate compositions, and introduces new implementations for existing slots from execution

evidence. This allows search to exploit prior discoveries without remaining confined to the initial candidate pools.

3 PRELIMINARIES

With a fixed execution model M and tool set, a *harness* h is a program that organizes model calls, maintains state, and controls interaction with the environment. On an instance $z \sim \mathcal{D}$, it produces a trajectory $\tau \sim \mathrm{Run}(M, h, z)$ with task score $R(z, \tau)$. Harness optimization seeks to improve expected performance without changing model weights:

$$J(h) = \mathbb{E}_{z\sim\mathcal{D},\ \tau\sim\mathrm{Run}(M,h,z)}\left[R(z,\tau)\right]. \tag{1}$$

We organize the harness into k functional slots with fixed interfaces $\mathcal{I} = \{I_1, \ldots, I_k\}$. A slot specifies a fixed functional responsibility, such as retrieval or failure recovery, rather than a single model call; execution may include branches and loops. Before round t, pool $P_{j,t}$ contains candidate implementations satisfying I_j. Substitution occurs within a slot; each configuration selects one implementation per slot:

$$h = (c_1, \ldots, c_k), \quad c_j \in P_{j,t}, \qquad \mathcal{H}_t = \prod_{j=1}^{k} P_{j,t}. \tag{2}$$

The executable harness is $\mathrm{Compose}_{\mathcal{I}}(h)$; N candidates per slot yield N^k configurations.

Each round evaluates a complete configuration h_t, yielding a validation score r_t, traces $\mathcal{T}_t$, and resource-use records κ_t, all stored in an archive $\mathcal{A}_t$. The score is aggregated under the task's validation protocol and is an observed estimate rather than the unknown $J(h_t)$. A bandit pull selects a complete harness: no separate module rewards are observed. We seek a high-performing configuration within T evaluations and select the final harness from the validation archive. Test data are not used for generation, selection, or stopping.

4 METHOD

MESH-Harness combines bandit-guided configuration selection with trace-driven module evolution. Validation scores update a shared performance model, while execution traces guide changes to the candidate pools. Algorithm 1 summarizes the procedure; Appendix B specifies its state transitions and interfaces.

4.1 MODULE POOLS AND SHARED REPRESENTATIONS

An LLM initializes N implementations per slot from the task description, interfaces, and permitted training information. Candidates for a retrieval slot, for example, may rank evidence differently. They must pass their slot's interface checks and basic execution checks; duplicates are rejected, and invalid implementations are revised using error feedback.

A fixed encoder maps each candidate's role description, interface, and implementation summary to an embedding, followed by a fixed projection and normalization to obtain $e(c) \in \mathbb{R}^d$. The

Algorithm 1 MESH-Harness

Require: $M, \mathcal{I}, N, T, S, L$; validation evaluator
  Initialize checked per-slot pools $\mathcal{P} = \{P_j\}$ and embeddings
  $A \leftarrow \lambda I_{kd}$, $b \leftarrow 0$, $\mathcal{A} \leftarrow \varnothing$, $h^\star \leftarrow \perp$
  Initialize use counts and lineage records
  for $t = 1, \ldots, T$ **do**
    Form U_t using Eq. (6)
    $h \leftarrow \textsc{MixedStart}(\mathcal{P}, \mathcal{A}, h^\star)$
    $h \leftarrow \textsc{CoordinateSearch}(h, U_t, \mathcal{P}, S)$
    $h_t \leftarrow \textsc{NoveltyFallback}(h, \mathcal{P}, \mathcal{A})$
    $(r_t, \mathcal{T}_t, \kappa_t) \leftarrow \textsc{Evaluate}(M, h_t)$
    Update (A, b) using Eq. (9)
    Archive $(h_t, r_t, \mathcal{T}_t, \kappa_t)$; update usage
    $h^\star \leftarrow \textsc{SelectVal}(\mathcal{A})$
    if $t < T$ **and** $t \bmod L = 0$ **then**
      $\mathcal{Z} \leftarrow \textsc{Diagnose}(\mathcal{T}_t, \mathcal{A})$
      $\mathcal{C} \leftarrow \textsc{ProposeChecked}(\mathcal{P}, \mathcal{I}, \mathcal{Z})$
      for all accepted $(j, c'_j) \in \mathcal{C}$ **do**
        $c_j^- \leftarrow \textsc{Evict}(P_j, \mathcal{A}, h^\star)$
        $P_j \leftarrow (P_j \cup \{c'_j\}) \setminus \{c_j^-\}$
        Cache $e(c'_j)$ and lineage; $n(c'_j) \leftarrow 0$
      end for
    end if
  end for
  return $h^\star$

configuration representation concatenates embeddings in slot order:

$$x(h) = [e(c_1); \ldots; e(c_k)] \in \mathbb{R}^{kd}. \tag{3}$$

The shared surrogate (i.e., $\hat{\mu}_t(h)$ in Section 4.2) predicts harness performance from $x(h)$, whose fixed slot order aligns module implementations by functional role across configurations. An observation from one configuration can therefore affect the ranking of other combinations, including those that have not been executed. Embeddings are cached; newly proposed implementations are encoded in the same space without resetting the model.

4.2 Bandit-Guided Compositional Search

Following linear contextual bandits (Li et al., 2010; Chu et al., 2011), we maintain shared reward statistics over configuration features. Let $p = kd$ and $x_s = x(h_s)$. Before round t,

$$A_{t-1} = \lambda I_p + \sum_{s<t} x_s x_s^\top, \qquad b_{t-1} = \sum_{s<t} r_s x_s, \tag{4}$$

where $\lambda > 0$, $A_0 = \lambda I_p$, and $b_0 = 0$. With $\hat{\theta}_{t-1} = A_{t-1}^{-1} b_{t-1}$, the predicted performance and exploration signal are

$$\hat{\mu}_t(h) = x(h)^\top \hat{\theta}_{t-1}, \qquad \sigma_t(h) = \sqrt{x(h)^\top A_{t-1}^{-1} x(h)}. \tag{5}$$

We retain the full information matrix, including cross-slot feature co-occurrences, for joint parameter estimation and configuration-level uncertainty. The predicted mean remains additive over slots: $\hat{\mu}_t(h) = \sum_{j=1}^{k} e(c_j)^\top \hat{\theta}_{j,t-1}$, where $\hat{\theta}_{j,t-1}$ is the parameter block for slot j. **The shared reward model and uncertainty estimate guide the selection of complete configurations**; actual performance is determined by end-to-end execution. Appendices A.1–A.2 derive observation transfer and cross-slot uncertainty.

To account for module coverage and lineage reuse as the pools evolve, we use

$$U_t(h) = \hat{\mu}_t(h) + \alpha g_t(h) \sigma_t(h) + B_t(h) - F_t(h), \tag{6}$$

with

$$B_t(h) = \frac{\eta}{k} \sum_{j=1}^{k} \frac{1}{\sqrt{1 + n_{t-1}(c_j)}}, \qquad F_t(h) = \frac{\gamma}{k} \sum_{j=1}^{k} \log(1 + u_{j,t-1}(f(c_j))). \tag{7}$$

Here, $n_{t-1}(c)$ counts prior evaluations containing c, $f(c)$ identifies its lineage family, and $u_{j,t-1}(f)$ counts that family's uses in slot j over the latest W evaluations. Thus, B_t *favors underused modules and* F_t *discourages repeated lineage selection*.

For similarity-aware exploration, set $g_t(h) = \max\{g_{\min}, 1 - \beta \bar{s}_t(h)\}$. Across slots, $\bar{s}_t(h)$ averages each candidate's maximum nonnegative cosine similarity to previously evaluated modules in the same slot and its parent when available; missing references contribute zero. This reduces the exploration premium for close variants. The coefficients $\alpha, \eta, \gamma, \beta \geq 0$ and $g_{\min} \in (0, 1]$ control these terms, which affect selection only, not the reward update.

We approximately maximize U_t by mixed-start coordinate ascent, building on the coordinate search used in MASPOB (Hong et al., 2026). Each round draws one start from a mixture of the incumbent, the top-K archived configurations restorable by the task adapter, and a random combination; an empty archive gives a random start. The incumbent supports local refinement, while archived and random starts provide alternative search points. We then sweep through the slots:

$$c_j \leftarrow \arg\max_{c \in P_{j,t}} U_t(h_{-j}, c), \qquad j = 1, \ldots, k, \tag{8}$$

where (h_{-j}, c) substitutes candidate c for the implementation in slot j and retains the latest choices elsewhere. Each update scores a complete configuration. We use two sweeps by default, favor unevaluated configurations on ties, and use random fallback sampling to seek a new configuration when the result has already been evaluated. The S sweeps require $O(S \sum_j |P_{j,t}|)$ surrogate evaluations (Appendix A.3). Throughout these sweeps, U_t remains fixed and no harness is executed. Only the selected h_t receives a task evaluation, after which

$$A_t = A_{t-1} + x_t x_t^\top, \qquad b_t = b_{t-1} + r_t x_t. \tag{9}$$

We also update the archive, module-use counts, and lineage statistics.

4.3 TRACE-DRIVEN MODULE EVOLUTION

After every L evaluations, while budget remains, MESH-Harness groups validation traces by failure type and identifies slots whose implementations warrant revision. Diagnosis uses current failures, recurring errors, and previous edits in the relevant lineages. The proposer mixes failure-guided repair, exploration of less-covered designs, and local refinement of strong configurations. Each refresh targets two distinct slots and proposes one child implementation per slot from its interface, parent code, and execution evidence. Proposals undergo the checks in Section 4.1, with a bounded number of retries for invalid implementations. A child implements its parent's slot and can be combined with candidates for other slots from other rounds.

Each accepted child replaces one candidate in its slot's pool, keeping the pool size fixed:

$$P_{j,t+1} = (P_{j,t} \cup \{c'_j\}) \setminus \{c^-_j\}. \tag{10}$$

Among eligible candidates, we evict the one with the lowest retention value: the mean observed reward of configurations containing it. Unevaluated candidates have value zero and do not inherit parent scores. Ties favor removing the candidate most similar to others in the pool. The incumbent's selected implementations and the newly inserted child are protected during the update; historical implementations and lineage records remain archived. Pools without an accepted child are unchanged.

Refreshes preserve A_t and b_t. A new implementation receives no immediate task evaluation: subsequent bandit decisions determine when it is selected and which implementations fill the other slots. Search therefore learns both which combinations to evaluate and which implementations to add, without assigning an evaluation to every proposal. The final harness is chosen by the prespecified validation criterion and its implementations are frozen for testing. Generation and embedding costs count toward search expenditure; deployment executes only the selected harness, without further search. Algorithm 1 omits time subscripts on mutable state. Appendix B.3 details pool updates, and Appendix C.5 specifies the final selection criterion.

5 EXPERIMENTS

We evaluate final performance under a limited candidate-evaluation budget, examine the contributions of selection and module evolution, and assess the cost and reuse of the resulting harnesses.

5.1 EXPERIMENTAL SETUP

Tasks and evaluation. We study text tasks on LawBench (Fei et al., 2024), USPTO (Schneider et al., 2016), and Symptom-to-Diagnosis dataset (Gretel AI, 2023); retrieval-augmented mathematics constructed from OlympiadBench (He et al., 2024) and MathArena (Dekoninck et al., 2026); code generation on LiveCodeBench (Jain et al., 2025); and interaction in ScienceWorld (Wang et al., 2022). We report the macro-average over the three text datasets, overall Math pass@1 on the combined test set, LiveCodeBench solve rate under the fixed cap-8 protocol, and mean ScienceWorld progress. Math pass@1 divides the total number of correct answers across both datasets by their total number of test instances, weighting each instance equally (Appendix C.4). All methods use the same splits (Appendix C.1). Search and final harness selection use validation data; the selected harnesses are frozen before held-out testing. All reported test scores are means over three test runs of the same frozen harness.

Comparisons and budget. We compare with SkillOpt (Yang et al., 2026), DGM-H (Zhang et al., 2026a), ADIAS (Jiang et al., 2026), AHE (Lin et al., 2026), and Meta-Harness (Lee et al., 2026). All optimization methods use the same execution model and a budget of 40 complete candidate evaluations per task. Meta-Harness evaluates 20×2 candidates, and MESH-Harness evaluates 40×1 compositions. Meta-Harness starts from fixed harnesses, including few-shot (Brown et al., 2020) and BM25 (Robertson & Zaragoza, 2009), in Tables 2–3, or handcrafted ReAct (Yao et al., 2023) harnesses for LiveCodeBench and ScienceWorld; MESH-Harness initializes per-slot pools with LLM-generated implementations.

Implementation. Both optimizers use Claude Code (Anthropic, 2026a) with Claude Opus 5 (Anthropic, 2026b) at `medium` effort. Unless stated otherwise, task execution uses DeepSeek V4 Flash (Xu et al., 2026) with thinking disabled. MESH-Harness uses Qwen3-Embedding-8B (Zhang et al., 2025), 10 candidates per slot, $\alpha = 0.8$, and two coordinate-search sweeps per evaluation. After every five evaluations, while budget remains, it proposes one child implementation in each of two distinct slots. Appendix C.3 gives the full settings and execution limits.

Table 1: **Held-out test performance.** All methods use 40 candidate evaluations; scores average three test runs. Text: dataset macro-average; Math: pooled pass@1; LiveCodeBench: cap-8 solve rate; ScienceWorld: mean progress. Scores use a 0–100 scale. Bold marks the column maximum; Δ compares MESH-Harness with Meta-Harness.

Method	Text	Math	LiveCodeBench	ScienceWorld
SkillOpt	53.27	46.40	57.00	53.00
DGM-H	42.78	53.00	72.00	52.20
ADIAS	42.50	52.60	69.00	43.10
AHE	47.39	50.60	69.00	49.00
Meta-Harness	50.63	46.20	75.00	53.10
MESH-Harness	**56.33**	**53.21**	**77.00**	**58.10**
Δ vs. Meta-Harness	+5.70	+7.01	+2.00	+5.00

Table 2: **Text reference harnesses.** Fixed Meta-Harness seeds appear above the divider; scores are macro-averages.

Harness	Score
No memory	26.07
Few-shot (8)	41.00
Few-shot (32)	46.87
Few-shot (all)	50.00
ACE memory	50.00
MCE memory	48.93
Meta-Harness	50.63
MESH-Harness	**56.33**

Table 3: **Math reference harnesses.** Fixed Meta-Harness seeds appear above the divider; scores are overall pass@1.

Harness	Score
No retrieval	23.20
Random few-shot	41.42
BM25	42.18
ACE	42.38
Meta-Harness	46.20
MESH-Harness	**53.21**

5.2 Performance under a Fixed Evaluation Cap

Main comparison. MESH-Harness achieves the highest reported score on each of the four aggregate metrics (Table 1 and Figure 2). Relative to Meta-Harness, the gains are 5.70 points on Text, 7.01 on Math, 2.00 on LiveCodeBench, and 5.00 on ScienceWorld. The advantage therefore spans context construction, retrieval, execution feedback, and sequential interaction without changing the execution model. MESH-Harness is particularly strong on Text and ScienceWorld, while reaching the performance level of the strongest Math competitor, DGM-H (53.21 versus 53.00).

Where the gains arise. MESH-Harness exceeds the strongest fixed references by 6.33 points on Text and 10.83 on Math (Tables 2–3). The text improvement over Meta-Harness is driven by LawBench (+8.00) and USPTO (+11.00), offsetting a decrease on Symptom-to-Diagnosis (84.90 to 83.00). Both math subsets improve: OlympiadBench reaches 63.00 (+4.30), and MathArena reaches 43.10 (+9.80). MESH-Harness also exceeds the handcrafted ReAct baselines used to initialize Meta-Harness: 77.00 versus 71.00 on LiveCodeBench and 58.10 versus 55.50 on ScienceWorld (Figure 3). Appendix D shows how the selected harnesses organize task-specific evidence: Text highlights how each retrieved example matches the query and differs from the other examples; Math retains the full problem, planning notes, and exact-answer requirements while omitting retrieved passages; and LiveCodeBench compares public-test outcomes across two drafts to guide repair, drawing on a passing alternative when only one draft fails a test.

Improvement beyond initialization. Figure 3 shows that MESH-Harness improves by 5.63–7.79 points over its first evaluated compositions. On ScienceWorld, it rises from 51.90 to 58.10, despite beginning below the handcrafted reference at 55.50. These results establish that the final performance is not simply inherited from the first composition. The reference scores describe within-method changes; the matched-cap comparison is the final performance in Table 1.

5.3 Contributions of Selection and Evolution

The following ablations use the three text tasks, the default executor, and the candidate-evaluation cap from Section 5.1.

Selecting which compositions to evaluate matters. Randomly selecting one implementation per slot scores 48.80, 7.53 points below the full method (Table 4). Removing only the UCB uncertainty

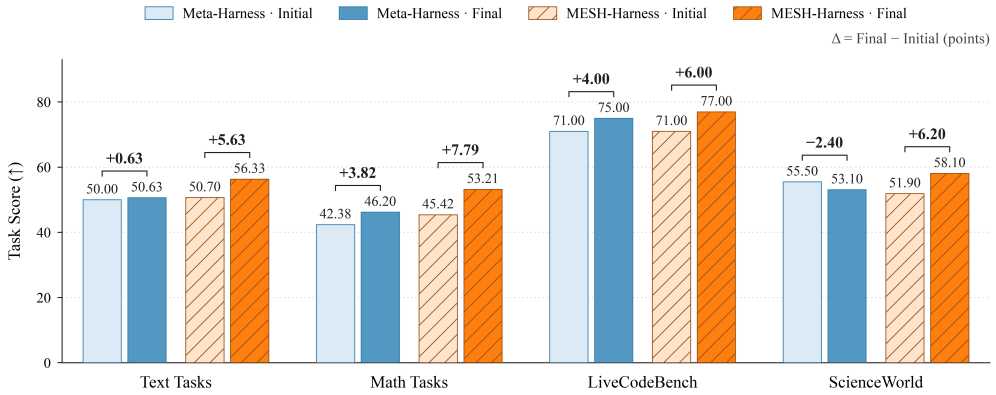


Figure 3: **Final performance relative to method-specific references.** "Initial" denotes a fixed reference seed for Meta-Harness and the first evaluated composition for MESH-Harness, not a shared initialization. Meta-Harness references are 50.00 (Text), 42.38 (Math, ACE), 71.00 (LiveCodeBench), and 55.50 (ScienceWorld). Brackets show final minus reference.

Figure 4: **Exploration strength.** Text macro-average versus α; the star marks the default. Dashed lines show Meta-Harness and random selection.

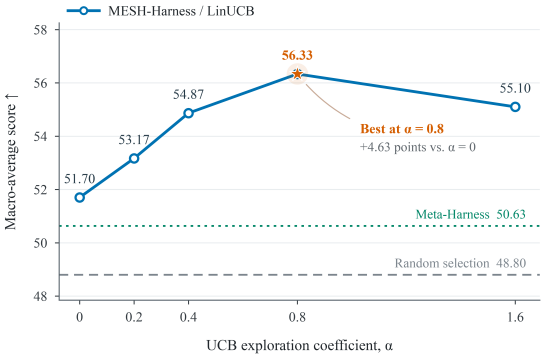


Table 4: **Configuration selection.** Text macro-average; Δ is relative to full MESH-Harness. Rows follow the α sweep at left.

Selection	Score	Δ
No UCB bonus ($\alpha = 0$)	51.70	−4.63
$\alpha = 0.2$	53.17	−3.16
$\alpha = 0.4$	54.87	−1.46
Full ($\alpha = 0.8$)	**56.33**	0.00
$\alpha = 1.6$	55.10	−1.23
Random composition	48.80	−7.53
Meta-Harness	50.63	−5.70

term ($\alpha = 0$), while retaining predicted reward, coverage, and lineage terms, yields 51.70. Thus, guided selection without the uncertainty term improves on random selection, but still trails the full acquisition rule by 4.63 points. Performance increases from $\alpha = 0$ to 0.8 and then falls at 1.6 (Figure 4), favoring a moderate exploration bonus among the tested values.

Evolving the pools improves performance. Keeping the initial pools for all slots fixed yields 51.20, a 5.13-point drop (Table 5). Refreshing them without historical traces, while retaining pool refresh and bandit statistics, recovers 2.60 points; restoring trace access adds a further 2.53. Together with the selection ablation, these results support the two parts of the design: use reward feedback to allocate evaluations among compositions, and execution evidence to improve the implementations available for future composition.

Table 5: **Module evolution.** Text macro-average over three test runs and change from the full method.

Variant	Score	Δ
MESH-Harness	**56.33**	0.00
No pool refresh	51.20	−5.13
Refresh without historical traces	53.80	−2.53
Meta-Harness	50.63	−5.70

5.4 Cost and Reuse of Optimized Harnesses

Search and execution costs. Table 6 separates optimization-inclusive costs from execution costs. Across the four reported workloads, MESH-Harness reduces search-plus-test expenditure from \$195.0537 to \$166.6078 (14.6%) and test-only expenditure from \$3.5528 to \$1.9835 (44.2%). Both costs are lower on every task; the largest test-time savings occur on LiveCodeBench (27.0%) and ScienceWorld (59.5%). The final score improvements therefore do not rely on more expensive

Table 6: **Search and execution costs (USD).** Search + test includes optimization and testing the selected configurations; test only excludes optimization. Savings are relative to Meta-Harness. Totals sum the four reported task workloads.

	Search + test			Test only		
Task	Meta	MESH	Saving	Meta	MESH	Saving
Text	63.1843	51.1643	19.0%	0.1709	0.1662	2.8%
Math	58.8652	47.2431	19.7%	0.5475	0.4987	8.9%
LiveCodeBench	37.1387	34.7826	6.3%	0.5273	0.3847	27.0%
ScienceWorld	35.8655	33.4178	6.8%	2.3071	0.9339	59.5%
Total	**195.0537**	**166.6078**	**14.6%**	**3.5528**	**1.9835**	**44.2%**

Figure 5: **Score–cost tradeoff.** Circles: MESH-Harness; squares: Meta-Harness. The dashed line connects nondominated evaluated configurations. Cost uses a log scale.

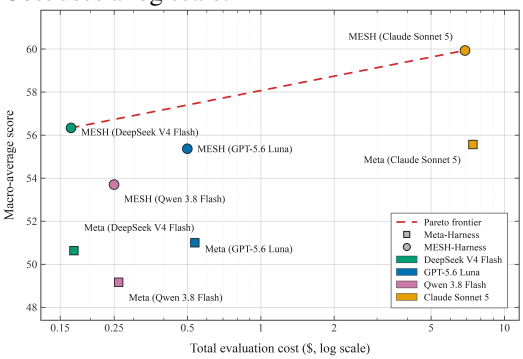


Table 7: **Frozen-harness transfer.** Text macro-average and execution cost (USD), without further search. Bold marks the better result within each model.

	Score ↑		Cost ↓	
Execution model	Meta	MESH	Meta	MESH
DeepSeek V4 Flash	50.63	**56.33**	0.1709	**0.1662**
GPT-5.6 Luna	51.00	**55.37**	0.5359	**0.4987**
Qwen 3.8 Flash	49.17	**53.70**	0.2608	**0.2496**
Claude Sonnet 5	55.57	**59.93**	7.4215	**6.8874**

execution of the selected harnesses. These savings are consistent with module reuse reducing LLM rewriting during search, and with compact contexts and targeted repair reducing execution-time token use (Appendix D).

Reuse across execution models. We freeze the text harnesses and replace only the executor with GPT-5.6 Luna (OpenAI, 2026), Qwen 3.8 Flash (Qwen Team, 2026), or Claude Sonnet 5 (Anthropic, 2026c). Including the default executor, MESH-Harness retains a 4.36–5.70-point advantage and a 2.8–7.2% cost reduction across all four models (Table 7). Among the measured configurations, its DeepSeek and Sonnet variants are the two nondominated points (Figure 5), offering the lowest cost and highest score, respectively.

Reuse of an existing optimized harness. Starting from a Meta-Harness solution, 20 additional MESH-Harness evaluations improve the text macro-average from 50.63 to 54.37, with gains on all three datasets (Table 8). Compositional refinement can therefore build on an existing optimizer's output, not only on MESH-Harness's own initialization. This experiment studies further refinement, separately from the matched-cap comparison.

Table 8: **Refining a Meta-Harness solution.** Held-out text scores before and after 20 additional MESH-Harness candidate evaluations.

Dataset	Meta-Harness solution	+ MESH refinement
LawBench	51.00	**55.00**
Symptom-to-Diagnosis	84.90	**90.10**
USPTO	16.00	**18.00**
Macro-average	50.63	**54.37**

6 Conclusion

We introduced MESH-Harness, which combines shared module representations, bandit-guided compositional search, and trace-driven module evolution to improve agent harnesses without changing model weights. Across four task domains, it achieves the highest aggregate scores among the evaluated methods and outperforms Meta-Harness under matched candidate-evaluation budgets. Text ablations support guided selection and module evolution; experiments show transfer across execution models and gains from refining a Meta-Harness solution. Aggregate test-time and combined search-and-test costs are respectively 44.2% and 14.6% lower than Meta-Harness. Reusing and refining implementations within fixed functional slots can thus improve task performance while reducing execution costs.

AI USE STATEMENT

We used generative AI tools to improve the clarity, grammar, and presentation of the manuscript and to assist with implementing, debugging, and refining research code. Generative models also serve as the proposer and executor in MESH-Harness, as described in the method and experimental setup. The authors reviewed and revised AI-assisted text and inspected and tested AI-assisted code. The authors take full responsibility for the final manuscript, implementation, and experimental results.

ETHICS STATEMENT

This work studies the optimization of agent harnesses with existing language models and benchmark datasets spanning text tasks, mathematical reasoning, code generation, and simulated scientific interaction. The experiments evaluate model outputs and executable programs within the specified benchmark protocols; they do not involve recruiting human participants or deploying systems in real-world decision-making. Diagnosis, legal, and chemical outputs are used only as benchmark predictions. We are not aware of specific ethical concerns arising from the experiments reported in this study.

REPRODUCIBILITY STATEMENT

We provide the code for MESH-Harness in the supplementary material. Algorithm 1 and Section 4 specify the MESH-Harness procedure, while Appendix A provides the theoretical analysis and Appendix B expands the optimization loop, composition selection, and module refresh procedures. Section 5.1 identifies the models, benchmarks, baselines, and evaluation protocol. Appendix C details the data splits, retrieval resources, memory boundaries, search hyperparameters, execution limits, scoring rules, and cost accounting. Appendix D describes the selected harnesses and their task-specific control flow. All comparisons use the shared task splits and scoring protocols described in the paper. Final harnesses are selected on validation data and frozen before testing; reported test scores average three executions of each frozen harness. The models, benchmarks, and retrieval sources are identified and cited in the manuscript.

REFERENCES

Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. *Advances in neural information processing systems*, 24, 2011.

AIMO-Corpus. PolyMath. Hugging Face dataset, 2026. URL `https://huggingface.co/datasets/AIMO-Corpus/PolyMath`.

Anthropic. Claude Code overview. Claude Code documentation, 2026a. URL `https://code.claude.com/docs/en/overview`. Accessed September 25, 2026.

Anthropic. Introducing Claude Opus 5. Anthropic, July 2026b. URL `https://www.anthropic.com/news/claude-opus-5`.

Anthropic. Introducing Claude Sonnet 5. Anthropic, June 2026c. URL `https://www.anthropic.com/news/claude-sonnet-5`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandits with linear payoff functions. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 208–214. JMLR Workshop and Conference Proceedings, 2011.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with llms. *arXiv preprint arXiv:2605.00674*, 2026.

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.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Zhengyang Tang, et al. Omni-math: A universal olympiad level mathematic benchmark for large language models. In *International Conference on Learning Representations*, volume 2025, pp. 100540–100569, 2025.

Gretel AI. Symptom to Diagnosis Dataset. Hugging Face dataset, 2023. URL https://huggingface.co/datasets/gretelai/symptom_to_diagnosis. Adapted from the Symptom2Disease dataset.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, et al. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 138306–138322, 2026.

Zhi Hong, Qian Zhang, Jiahang Sun, Zhiwei Shang, Mingze Kong, Xiangyi Wang, Yao Shu, and Zhongxiang Dai. MASPOB: Bandit-based prompt optimization for multi-agent systems with graph neural networks. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=A5venxSvpw.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Lekang Jiang, Bohan Tang, Stephan Goetz, and Yiwen Guo. Adias: Automated design of interactive agentic systems. *arXiv preprint arXiv:2608.06410*, 2026.

Yoonho Lee, Roshen Sanjay Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=tmbOUyFx3R.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. NuminaMath. Hugging Face dataset, 2024. URL https://huggingface.co/datasets/AI-MO/NuminaMath-1.5.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pp. 661–670, 2010.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset. *arXiv preprint arXiv:2504.16891*, 2025.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. OpenAI, July 2026. URL https://openai.com/index/gpt-5-6/.

Qwen Team. Latest model: Qwen3.8-Flash. QwenCloud documentation, 2026. URL https://docs.qwencloud.com/developer-guides/getting-started/latest-model. Accessed September 25, 2026.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and trends® in information retrieval*, 4(1-2):1–174, 2009.

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.

Jack Sherman and Winifred J Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.

Chengshuai Shi, Kun Yang, Zihan Chen, Jundong Li, Jing Yang, and Cong Shen. Efficient prompt optimization through the lens of best arm identification. *Advances in Neural Information Processing Systems*, 37:99646–99685, 2024.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=P1jHroBS5E.

Justin Young. Effective harnesses for long-running agents. Anthropic Engineering, November 2025. URL https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

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

Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, et al. Harnessing agentic evolution. *arXiv preprint arXiv:2605.13821*, 2026b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026c.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

A THEORETICAL SUPPLEMENTS

This section analyzes the mechanisms that make compositional search useful under a limited evaluation budget: information transfer through a shared surrogate, uncertainty over module combinations, and inexpensive coordinate-wise selection. The first two results follow from the ridge state and the search rule themselves. The final result explicitly states the additional confidence-envelope assumption needed to relate acquisition-based selection to the task objective.

A.1 HOW ONE EVALUATION INFORMS OTHER COMPOSITIONS

Let $p = kd$, and fix the embedding map throughout a search run. At a given round, write

$$A = \lambda I + \sum_{s<t} x_s x_s^\top, \qquad b = \sum_{s<t} r_s x_s, \qquad \widehat{\theta} = A^{-1} b, \tag{11}$$

where $\lambda > 0$. Since each slot embedding has unit norm, $\|x(h)\|_2^2 = k$. In particular, A is positive definite and $x(h)^\top A^{-1} x(h) \le k/\lambda$ for every composition, including a composition that has not been evaluated.

Proposition A.1 (Transfer through the shared ridge state)**.** *Suppose the next evaluated harness has feature vector x and reward r. Let $A^+ = A + xx^\top$ and $b^+ = b + rx$. For any fixed comparison vector v, the posterior mean and squared uncertainty satisfy*

$$v^\top \widehat{\theta}^+ - v^\top \widehat{\theta} = \frac{v^\top A^{-1} x}{1 + x^\top A^{-1} x}\big(r - x^\top \widehat{\theta}\big), \tag{12}$$

$$v^\top (A^+)^{-1} v = v^\top A^{-1} v - \frac{(v^\top A^{-1} x)^2}{1 + x^\top A^{-1} x}. \tag{13}$$

Proof. The Sherman–Morrison identity (Sherman & Morrison, 1950) gives

$$(A + xx^\top)^{-1} = A^{-1} - \frac{A^{-1} x x^\top A^{-1}}{1 + x^\top A^{-1} x}. \tag{14}$$

Multiplying by $b + rx$ and using $A^{-1} b = \widehat{\theta}$ yields $\widehat{\theta}^+ = \widehat{\theta} + A^{-1} x (r - x^\top \widehat{\theta})/(1 + x^\top A^{-1} x)$. Left multiplication by $v^\top$ proves Eq. (12); taking the quadratic form of Eq. (14) proves Eq. (13). □

These identities make the reuse mechanism explicit. An evaluation changes another composition's prediction through its overlap with the evaluated direction in the A^{-1} geometry, not merely through exact module identity. The update also depends on the prediction residual: a surprising result has a different effect from one already explained by the surrogate. Uncertainty cannot increase at a fixed vector after an observation, and its reduction is largest along directions coupled to the evaluated composition. This is surrogate-level information transfer; it is not an assignment of the observed reward to every selected module as an independent contribution.

The same identities apply after a pool refresh. A new implementation is encoded in the existing feature space, so it can be scored immediately with the accumulated (A, b). Its creation does not add an observation, and it receives neither its parent's reward nor a synthetic reward. Retiring an implementation changes the feasible set of compositions but does not invalidate the stored observations. This separation between the feasible set and the statistical state is what allows the pools to evolve without restarting the surrogate.

A.2 THE ROLE OF FULL COVARIANCE IN COMPOSITIONAL UNCERTAINTY

Partition $\widehat{\theta}$ and A by functional slot. The mean prediction remains additive:

$$\widehat{\mu}(h) = \sum_{j=1}^{k} \widehat{\theta}_j^\top e(c_j). \tag{15}$$

Full covariance does not introduce explicit pairwise reward features. Rather, its off-diagonal blocks retain which slot representations have been observed together:

$$A_{ij} = \lambda \, \mathbf{1}\{i = j\} I_d + \sum_{s<t} e(c_{i,s}) e(c_{j,s})^\top. \tag{16}$$

These blocks affect both the fitted coefficients and the uncertainty of a proposed combination.

A two-slot example isolates the distinction. Let $d = 1$, and suppose all n observations used $x_+ = (1, 1)^\top$. Then

$$A = \lambda I + n x_+ x_+^\top, \qquad \sigma^2(x_+) = \frac{2}{\lambda + 2n}, \qquad \sigma^2(x_-) = \frac{2}{\lambda}, \quad x_- = (1, -1)^\top. \tag{17}$$

Repeatedly observing the same pairing resolves the common direction but provides no information about the orthogonal contrast. Replacing A by its block diagonal would assign both directions uncertainty $2/(\lambda + n)$ and lose that distinction. In a compositional search, this matters when modules have repeatedly appeared with the same partners: frequent exposure of each module need not imply that a different pairing is well understood.

The distinction is useful for evaluation allocation: exposure to individual modules and exposure to their joint directions are different forms of information. The embedding-based mean is a surrogate for program performance, while end-to-end execution supplies the observed reward. The selection ablations in Table 4 complement this algebraic analysis by measuring the effect of the acquisition rule under the fixed evaluation cap.

A.3 Monotonicity and computational cost of coordinate search

During one selection step, the pools, posterior, usage counts, and lineage statistics are held fixed. Thus the acquisition $U_t(h)$ is a fixed function on the current Cartesian product. A coordinate update enumerates the active candidates for one slot while retaining the other slots.

Proposition A.2 (Monotonic coordinate updates)**.** *If each coordinate update includes the currently selected module among its choices and selects an acquisition maximizer, the acquisition is nondecreasing over the coordinate updates. If the procedure is continued using only strict improving moves, it terminates at a coordinate-wise maximizer on a finite product space.*

Proof. The current composition is feasible in every coordinate subproblem, so the selected replacement has acquisition at least as large. With strict improving moves, a composition cannot be revisited. Finiteness therefore implies termination; at termination no single-coordinate replacement improves the acquisition. □

The implementation uses two sweeps as a bounded local-ascent procedure. The proposition characterizes each feasible coordinate update; convergence to a coordinate-wise maximizer would additionally require continuing strict improvement to termination. Novelty-based tie breaking preserves nondecrease for equal acquisition values. Duplicate avoidance is a subsequent selection step, detailed in Section B.2.

For slot pool sizes $N_1, \ldots, N_k$ and S sweeps, the coordinate comparisons require $S \sum_j N_j$ acquisition evaluations, plus final scoring and any start-selection or duplicate-handling overhead. Exhaustive comparison would require $\prod_j N_j$ evaluations. With $N_j = 10$ and $S = 2$, this is 140 coordinate comparisons instead of 10^7 for the seven-slot tasks, and 100 instead of 10^5 for the five-slot ScienceWorld controller. These counts concern inexpensive surrogate comparisons, not benchmark executions. Only the selected complete harness consumes a candidate-evaluation slot.

The dense statistical state occupies $O(p^2)$ space. With a cached inverse, a rank-one update and a quadratic-form score each require $O(p^2)$ arithmetic; the agent implementation uses this update. The text and math implementation instead stores (A, b) and evaluates the corresponding dense linear solves. These implementations realize the same statistical update with different linear-algebra costs. The comparison above concerns acquisition-query complexity; embedding newly proposed modules is accounted for as initialization or refresh work.

A.4 Decomposing uncertainty and approximate selection

Coverage, lineage, and similarity adjustments modify evaluation priorities while leaving the observed rewards unchanged. The following decomposition separates these adjustments from uncertainty

and approximate maximization using the confidence-envelope formulation of linear bandit analysis (Abbasi-Yadkori et al., 2011). Consider a round on which, for some $q_t \geq 0$, the surrogate satisfies the uniform envelope

$$|J(h) - \widehat{\mu}_t(h)| \leq q_t \sigma_t(h) \quad \text{for all } h \in \mathcal{H}_t. \tag{18}$$

Write $V_t = \widehat{\mu}_t + q_t \sigma_t$ and $D_t = U_t - V_t$. Let h_t^* maximize J over the current pools, and define the acquisition-search error $\varepsilon_t = \max_{h \in \mathcal{H}_t} U_t(h) - U_t(h_t) \geq 0$.

Proposition A.3 (A conditional one-step decomposition)**.** *Under Eq.* (18)*,*

$$J(h_t^*) - J(h_t) \leq 2q_t \sigma_t(h_t) + \mathrm{osc}_{\mathcal{H}_t}(D_t) + \varepsilon_t, \tag{19}$$

where $\mathrm{osc}(D) = \max D - \min D$.

Proof. The envelope gives $J(h_t^*) \leq V_t(h_t^*)$ and $J(h_t) \geq V_t(h_t) - 2q_t\sigma_t(h_t)$. Moreover, $V_t(h_t^*) - V_t(h_t) = U_t(h_t^*) - U_t(h_t) + D_t(h_t) - D_t(h_t^*) \leq \varepsilon_t + \mathrm{osc}(D_t)$. Combining the inequalities proves the claim. □

For MESH-Harness, $D_t = (\alpha g_t - q_t)\sigma_t + B_t - F_t$. Equation (19) separates three sources of selection error: uncertainty in the chosen composition, the range of the acquisition adjustment, and the gap left by approximate search. It is a conditional statement under Eq. (18); the empirical α is a search coefficient rather than an asserted calibrated confidence radius. Coverage and lineage regularize exploration of an evolving candidate space, while coordinate search bounds the cost of comparing those candidates.

Pool refresh changes the feasible set without resetting the statistical state. Protecting the incumbent preserves an already evaluated configuration, while retaining historical embeddings allows subsequent selection to reuse all completed observations. Section B makes these state transitions explicit, including the different roles of the composition reward, the pool-retention statistic, and the validation criterion used for final selection.

B ALGORITHMIC DETAILS

This section expands Algorithm 1 into the procedures used for initialization, composition selection, and pool refresh. The central design is to separate proposing an implementation from deciding to evaluate it. A checked module enters a reusable pool; only a complete composition selected by the bandit receives an end-to-end validation run. The resulting score updates the shared surrogate, while the associated traces inform subsequent code changes.

B.1 SEARCH STATE AND COMPLETE PROCEDURE

The state consists of the active pools $\mathcal{P} = \{P_j\}_{j=1}^{k}$, a catalog $\mathcal{V}$ retaining all accepted implementations and their cached embeddings, the ridge statistics (A, b), the validation archive $\mathcal{A}$, and the set $\mathcal{K}$ of evaluated composition keys. A key is the ordered tuple of component identifiers. Usage records $\mathcal{U}$ store each module's evaluation count, accumulated reward, parent, lineage family, and selected rounds. An initial module starts its own family; a child retains its parent's family. The encoder and task interfaces remain fixed throughout the run.

Algorithm 2 uses t to count completed candidate evaluations, starting from one. The task adapter EVALUATEVAL implements the dataset-specific execution and scoring rules in Section C. SELECTVAL applies the prespecified validation criterion in Section C.5; neither the acquisition score nor a test score is substituted for that criterion.

Algorithm 2 MESH-Harness: complete optimization procedure

Require: Fixed executor M, proposer G, encoder E, ordered interfaces $\mathcal{I}$
Require: Permitted initialization data D_{init}, training data D_{tr}, validation data D_{val}
Require: Fixed task resources $\mathcal{C}$; pool size N, evaluation budget T, refresh interval L

$\mathcal{P}, \mathcal{V}, \mathcal{A}, \mathcal{K}, \mathcal{U} \leftarrow \varnothing$; $h^\star \leftarrow \perp$
for $j = 1, \ldots, k$ **do**
  $P_j \leftarrow \varnothing$
  for $i = 1, \ldots, N$ **do**
    $c \leftarrow$ CHECKEDPROPOSAL$(G, I_j, D_{\text{init}}, \mathcal{V}_j, \text{initial})$ ▷ Algorithm 5
    Cache $e(c) \leftarrow$ ENCODENORMALIZE(E, I_j, c); initialize its family and zero usage
    $P_j \leftarrow P_j \cup \{c\}$; $\mathcal{V}_j \leftarrow \mathcal{V}_j \cup \{c\}$
  end for
end for
$A \leftarrow \lambda I_{kd}$; $b \leftarrow 0$; initialize the seeded random generator
for $t = 1, \ldots, T$ **do**
  Hold $(A, b, \mathcal{P}, \mathcal{U})$ fixed during the following selection step
  $h_t \leftarrow$ SELECTCOMPOSITION$(\mathcal{P}, \mathcal{V}, A, b, \mathcal{A}, \mathcal{U}, \mathcal{K}, h^\star)$ ▷ Algorithm 3
  $H_t \leftarrow \text{Compose}_{\mathcal{I}}(h_t)$; $x_t \leftarrow [e(c_{1,t}); \ldots; e(c_{k,t})]$
  $(r_t, \mathcal{T}_t, \kappa_t) \leftarrow$ EVALUATEVAL$(M, H_t, D_{\text{tr}}, D_{\text{val}}, \mathcal{C})$
  $A \leftarrow A + x_t x_t^\top$; $b \leftarrow b + r_t x_t$
  $\mathcal{A} \leftarrow \mathcal{A} \cup \{(t, h_t, r_t, \mathcal{T}_t, \kappa_t)\}$; $\mathcal{K} \leftarrow \mathcal{K} \cup \{\text{key}(h_t)\}$
  for all $c \in h_t$ **do**
    $n(c) \leftarrow n(c) + 1$; $s(c) \leftarrow s(c) + r_t$; append t to its usage history
  end for
  Update recent family uses and committed validation-trace history in $\mathcal{U}$
  $h^\star \leftarrow$ SELECTVAL$(\mathcal{A})$
  if $t < T$ **and** $t \bmod L = 0$ **then**
    $(\mathcal{P}, \mathcal{V}, \mathcal{U}) \leftarrow$ REFRESHPOOLS$(t, h_t, h^\star, \mathcal{P}, \mathcal{V}, \mathcal{A}, \mathcal{U})$
      ▷ Algorithm 4; no task evaluation and no change to $(A, b, \mathcal{K})$
  end if
  Persist the completed-round state and resource-use records
end for
$h^\star \leftarrow$ SELECTVAL$(\mathcal{A})$
return FREEZE$(h^\star, \mathcal{V}, \mathcal{I}, M, \mathcal{C})$ and the validation archive

B.2 MIXED-START COMPOSITION SELECTION

The acquisition in Eq. (6) is evaluated on complete compositions, even when only one slot is being changed. For a candidate c in slot j, let $\mathcal{R}_j(c)$ contain previously evaluated modules in that slot and its direct parent, when available. With normalized embeddings, the similarity term is

$$s_j(c) = \max\big(\{0\} \cup \{e(c)^\top e(v) : v \in \mathcal{R}_j(c)\}\big), \qquad \bar{s}(h) = \frac{1}{k}\sum_{j=1}^{k} s_j(c_j). \tag{20}$$

Thus $U(h)$ uses the ridge mean, full-covariance uncertainty scaled by $\max(g_{\min}, 1 - \beta\bar{s}(h))$, the module-coverage bonus, and the recent-family penalty. All these quantities are computed from cached representations and completed evaluations. Coordinate comparisons do not call the proposer, executor, or task evaluator.

Algorithm 3 Mixed-start coordinate search with novelty-aware tie breaking

Require: Search state from Algorithm 2; $S = 2$, $K = 5$, random-attempt limit $R = 200$

function SELECTCOMPOSITION($\mathcal{P}, \mathcal{V}, A, b, \mathcal{A}, \mathcal{U}, \mathcal{K}, h^\star$)
  Construct U from Eqs. (6)–(7) and (20)
  $\mathcal{B} \leftarrow$ top-K validation-reward entries restorable by the task's archive adapter
  Draw $u \sim \mathrm{Uniform}[0, 1)$
  if $u < 0.65$ **and** $h^\star \neq \perp$ **then**
    $h \leftarrow h^\star$
  else if $u < 0.85$ **and** $\mathcal{B} \neq \varnothing$ **then**
    $h \leftarrow$ a uniformly sampled composition from $\mathcal{B}$
  else
    $h \leftarrow$ RANDOMNEW($\mathcal{P}, \mathcal{K}, R$)
  end if
  for $s = 1, \ldots, S$ **do**
    for $j = 1, \ldots, k$ **do**
      For every $c \in P_j$, form $h^{(c)} \leftarrow (h_{-j}, c)$ and compute $U(h^{(c)})$
      $c^+ \leftarrow \arg\max_{c \in P_j} \big(U(h^{(c)}), \mathbf{1}\{\mathrm{key}(h^{(c)}) \notin \mathcal{K}\}, \mathrm{birth}(c), \mathrm{id}(c)\big)$
      $h \leftarrow (h_{-j}, c^+)$ ▷ Lexicographic maximization; retain latest choices elsewhere
    end for
  end for
  if $\mathrm{key}(h) \in \mathcal{K}$ **then**
    $h \leftarrow$ RANDOMNEW($\mathcal{P}, \mathcal{K}, R$)
  end if
  return h
end function

function RANDOMNEW($\mathcal{P}, \mathcal{K}, R$)
  for $a = 1, \ldots, R$ **do**
    Draw each c_j independently and uniformly from P_j; $h \leftarrow (c_1, \ldots, c_k)$
    if $\mathrm{key}(h) \notin \mathcal{K}$ **then**
      return h
    end if
  end for
  return the last sampled h
end function

The start mixture is sampled once per evaluation, rather than running three searches and evaluating their winners. The archived-start adapter reconstructs component identities from stored code. The agent runner restricts this bank to fully active compositions; the Text/Math runner can restore historical starts, whose slots are replaced from the active pools during the first sweep. The evaluated composition is in the active product space in either case. The monotonicity result in Proposition A.2 applies whenever the current coordinate is among the feasible replacements. Novelty is a tie breaker, not an additional reward, and bounded random sampling is the duplicate fallback rather than an exhaustive search for an unseen composition.

B.3 TRACE-GUIDED PROPOSALS AND FIXED-CAPACITY POOL UPDATES

Refresh ranks validation failure groups by recurring count and then diagnostic confidence. Failure-guided actions scan primary and secondary suspected slots, using the corresponding module in the latest composition. Coverage-oriented actions rank active parents by recent family use and evaluation count. Local-refinement actions rank incumbent modules by recent family use. Remaining ties use the runner's deterministic creation-time and identifier ordering. These alternatives supply different parent designs through the same slot interface.

The default plan first reserves an available failure-guided action, then fills the remaining position from the rotating order $(\text{failure}, \text{failure}, \text{coverage}, \text{local})$. A slot can appear at most once. At refresh q, the order is rotated by $(q-1) \bmod 4$; refreshes follow evaluations 5, 10, 15, 20, 25, 30, and 35. NEXTACTION below applies these rankings while excluding already selected slots. The planning loop is bounded by 12 attempts in Text/Math and $4\max(3,k)$ in the agent runner.

Algorithm 4 Trace-guided refresh without an immediate candidate evaluation

Require: Proposer G, encoder E, interfaces $\mathcal{I}$; two children per refresh in distinct slots

function REFRESHPOOLS($t, h_t, h^\star, \mathcal{P}, \mathcal{V}, \mathcal{A}, \mathcal{U}$)
  $\mathcal{Z} \leftarrow$ recurring failure groups and representative validation traces from $\mathcal{A}$
  $\Pi \leftarrow [\,]$; $J \leftarrow \varnothing$
  $a \leftarrow$ NEXTACTION(failure, $\mathcal{Z}, h_t, h^\star, \mathcal{P}, \mathcal{U}, J$)
  if $a \neq \perp$ **then**
    Append a to Π; add its slot to J
  end if
  $\omega \leftarrow$ the rotating default mode order for this refresh
  for $i = 1, \ldots, R_{\text{plan}}$ **do**
    if $|\Pi| = 2$ **then break**
    $a \leftarrow$ NEXTACTION($\omega_i, \mathcal{Z}, h_t, h^\star, \mathcal{P}, \mathcal{U}, J$)
    if $a \neq \perp$ **then**
      Append a to Π; add its slot to J
    end if
  end for
  Require $|\Pi| = 2$; otherwise report an incomplete refresh plan
  if some planned slot has no old module outside $h^\star$ that can be evicted **then**
    return $(\mathcal{P}, \mathcal{V}, \mathcal{U})$ ▷ Keep the fixed-capacity pools unchanged
  end if
  $\mathcal{C}_{\text{new}} \leftarrow \varnothing$
  for all actions $a = (j, c_{\text{parent}}, \text{mode}) \in \Pi$ **do**
    $Z_j \leftarrow$ slot contract, parent code, failure evidence, and bounded related history
    $c'_j \leftarrow$ CHECKEDPROPOSAL($G, I_j, Z_j, \mathcal{V}_j, \text{mode}$) ▷ Algorithm 5
    Cache $e(c'_j)$; set its parent and family; initialize $n(c'_j) = s(c'_j) = 0$
    $\mathcal{C}_{\text{new}} \leftarrow \mathcal{C}_{\text{new}} \cup \{(j, c'_j)\}$
  end for
  for all $(j, c'_j) \in \mathcal{C}_{\text{new}}$ **do**
    $P_j^+ \leftarrow P_j \cup \{c'_j\}$; $\mathcal{D}_j \leftarrow \{c \in P_j : c \notin h^\star\}$
    $c_j^- \leftarrow \arg\min_{c \in \mathcal{D}_j} \big(v(c), \delta(c; P_j^+), \text{id}(c)\big)$
    $P_j \leftarrow P_j^+ \setminus \{c_j^-\}$; retain both implementations in $\mathcal{V}_j$
    Mark c'_j active and c_j^- retired; record the proposal and replacement in $\mathcal{U}$
  end for
  Require $|P_j| = N$ for every slot; commit the updated pool and history records
  return $(\mathcal{P}, \mathcal{V}, \mathcal{U})$ ▷ A, b, and evaluated keys are unchanged
end function

The retention and redundancy quantities used in the lexicographic eviction rule are

$$v(c) = \begin{cases} s(c)/n(c), & n(c) > 0, \\ 0, & n(c) = 0, \end{cases} \qquad \delta(c; P) = \min_{v \in P \setminus \{c\}} \big(1 - e(c)^\top e(v)\big). \tag{21}$$

Lower observed mean reward is removed first; smaller nearest-neighbor distance breaks a tie. The incumbent and new child are protected during replacement. Retired code remains archived; later selection uses the ordinary acquisition, without forced child evaluations.

B.4 ADMISSIBILITY, HISTORY CONSTRUCTION, AND FROZEN EXECUTION

Initialization and refresh share the same admissibility principle: a candidate must implement its assigned responsibility and be executable through the declared interface. CHECK in Algorithm 5 combines syntax and signature validation, the permitted-capability checks, task-specific return-shape and state checks, and small execution fixtures. It also rejects duplicate identifiers and duplicate implementations within the slot's historical catalog. Rejections are returned to the proposer as concrete correction feedback. The default limit is 20 attempts per requested module for both initialization and refresh.

Algorithm 5 Checked module generation and isolated evaluation of the frozen harness

function CHECKEDPROPOSAL($G, I_j, Z, \mathcal{V}_j, \text{mode}$)
  $F \leftarrow [\,]$ ▷ Rejected code and admissibility feedback
  for $a = 1, \ldots, 20$ **do**
    $c \leftarrow \text{PROPOSE}(G, I_j, Z, \text{mode}, F)$
    $(\text{ok}, \text{reason}) \leftarrow \text{CHECK}(c, I_j, \mathcal{V}_j)$
    if ok **then**
      return c
    end if
    Append the rejected implementation and reason to F
  end for
  Report generation exhaustion; do not commit an unchecked module or an undersized pool
end function

function EVALUATEFROZEN($H^\star, D_{\text{test}}, D_{\text{tr}}, \mathcal{C}$)
  for $r = 1, 2, 3$ **do**
    Initialize an independent test repetition r
    Reset task memory and environment episodes; initialize permitted training memory
    $S^{(r)} \leftarrow \text{TRUSTEDTEST}(H^\star, D_{\text{test}}, D_{\text{tr}}, \mathcal{C})$
    Store predictions, trusted scores, and resource use; do not update the optimizer
  end for
  return $\frac{1}{3}\sum_{r=1}^{3} S^{(r)}$
end function

History is selective context, not a new retrieval corpus. Failure statistics are accumulated over the committed validation history. The proposer receives a bounded view: up to eight error groups, at most two representative examples per group, up to six related family edits, and up to three higher-scoring historical trajectories for currently failed problems. Contrasts are chosen preferentially from configurations differing in fewer slots. The additional history context is limited to 24,000 characters by dropping complete records. Each record retains its originating evaluation and component identities, so the proposer can distinguish an observed result from the stated intent of an untested edit. These contrasts guide a modification hypothesis; the next complete-harness evaluation supplies its observed reward.

Three state transitions have different roles. A completed evaluation changes (A, b), the archive, and usage statistics. A refresh changes which programs are available while retaining those statistics. Finalization freezes a validation-selected composition and removes the optimizer from the execution path. In particular, the empirical mean $v(c)$ is used for pool retention, whereas the complete-harness reward r_t is the observation for the bandit. A new module starts with zero observations and is represented immediately in the same feature space. This is the operational counterpart of Proposition A.1: reuse does not require assigning an inherited or synthetic reward to a new implementation.

Evaluation budget. The default initialization creates 70 modules for the seven-slot tasks or 50 for ScienceWorld. Seven refreshes can add at most 14 modules while preserving an active space of 10^7 or 10^5 compositions. The budget contains 40 complete-composition evaluations; generation, embedding, and checks are accounted for separately. Final testing follows Algorithm 5.

C Implementation Details

This section describes the datasets, runtime settings, and evaluation protocol used in Section 5.1. MESH-Harness and Meta-Harness share the task data, execution model, and scoring rules. Their search budgets count complete candidate evaluations; each candidate may make several model calls while solving the task.

C.1 Datasets and retrieval resources

Table 9 summarizes the task splits. Training data provide examples or initial memory, validation data guide harness optimization and selection, and held-out data measure the selected harness. The splits remain fixed across methods and repeated test runs.

Table 9: Task splits and prediction units. A dash indicates no separate training split. Math contains 200 validation problems from each source; its 500 test problems comprise 254 from OlympiadBench and 246 from MathArena. ScienceWorld counts are episodes.

Dataset	Train	Validation	Test	Prediction or interaction unit
USPTO	50	30	100	Precursor strings for a product molecule
Symptom-to-Diagnosis	200	50	212	A diagnosis from a symptom description
LawBench	200	50	100	A set of charges from a case description
Mathematical reasoning	–	400	500	A final mathematical answer
LiveCodeBench	100	100	100	One selected Python program per problem
ScienceWorld	66	22	102	One interactive episode

Text tasks. The three text datasets require different outputs. USPTO asks for reactant precursors given a product molecule (Schneider et al., 2016); Symptom-to-Diagnosis asks for a diagnosis from symptoms (Gretel AI, 2023); and LawBench asks for the applicable charges in a legal case (Fei et al., 2024). For symptom diagnosis, we use Gretel's 22-class adaptation of the original Symptom2Disease dataset, which filters diagnosis categories and rewrites symptom descriptions with an LLM. We use the same task-specific output and scoring rules for all harnesses.

Mathematical reasoning. Math combines answer-based problems from OlympiadBench (He et al., 2024) and MathArena (Dekoninck et al., 2026). The development set contains 200 problems from each source, totaling 400 problems, and is used for harness optimization. The held-out test set contains 254 OlympiadBench problems and 246 MathArena problems, totaling 500 problems. Final evaluation follows the common held-out protocol used for Meta-Harness: the two sources are pooled, and every test instance receives equal weight. Section C.4 defines this aggregation. The development and test questions are disjoint, and answers are evaluated by the corresponding source-specific programmatic graders.

All retrieval-capable math harnesses use a fixed corpus of 433,926 problem–solution pairs drawn from NuminaMath (Li et al., 2024), Omni-MATH (Gao et al., 2025), DeepMath (He et al., 2026), OpenMathReasoning (Moshkov et al., 2025), and PolyMath (AIMO-Corpus, 2026). Table 10 gives its composition. The corpus is filtered against the evaluation questions and deduplicated before search; the same corpus is used for validation and testing. Retrieval indexes the full corpus. Each solution is truncated to at most 4,000 characters during preprocessing and 3,000 characters when inserted into a prompt.

Code generation. We use fixed training, validation, and test splits from LiveCodeBench release v6 (Jain et al., 2025), with 100 problems in each split. The test set contains 23 easy, 22 medium, and 55 hard problems. A harness can use public examples to test and repair its program; private tests are reserved for final scoring. The cap-8 protocol is defined in Section C.4.

Interactive scientific tasks. ScienceWorld (Wang et al., 2022) uses the episode splits provided with ADIAS (Jiang et al., 2026), covering 22 task types. Each type contributes three training variations and one validation variation, with 102 held-out episodes in total. Episodes use the easy simplifications, a

Table 10: Math retrieval corpus after evaluation-overlap filtering and deduplication.

Retrieval source	Problem–solution pairs
NuminaMath	55,968
Omni-MATH	3,201
DeepMath	98,670
OpenMathReasoning	267,647
PolyMath	8,440
Total	433,926

limit of 30 agent turns, and an environment step limit of 200. During search, each iteration samples 15 training episodes without replacement and evaluates all 22 validation episodes.

C.2 Memory and information available to each harness

The optimizer's validation archive, a harness's runtime memory, and the fixed retrieval corpus serve different roles. Validation scores and traces guide code evolution. Runtime memory supports the current task, while retrieval supplies external examples. Held-out answers remain with the evaluator throughout execution.

Text. Each candidate builds a separate memory for each dataset through one pass over its training examples, receiving the gold label after each training prediction. Validation and test predictions use this training-derived memory without adding held-out labels. Memory is rebuilt for each candidate and each final repetition.

Math. The harness receives the problem, its answer-format instructions, and access to the fixed retrieval corpus. The evaluator retains the reference answer and grader. Retrieved solutions belong to the external corpus; development and test solutions are not added to it. A candidate's verifier can inspect and refine an answer using the available context, but cannot consult the reference answer.

LiveCodeBench and ScienceWorld. Code-generation harnesses receive problem statements, starter-code interfaces, and public examples. Only public-test results can be used for repair. ScienceWorld harnesses receive the task description and observations from their own actions, with memory reset at the start of every episode. Validation traces from these interactions can guide subsequent module proposals, as specified in Section B.4.

C.3 Search settings and execution limits

Text, Math, and LiveCodeBench have seven functional slots; ScienceWorld has five. Each slot starts with ten LLM-generated implementations. Table 11 lists the shared search parameters. The module encoder is Qwen3-Embedding-8B (Zhang et al., 2025); each slot is represented by a normalized 128-dimensional vector, cached throughout the run. Newly proposed modules use the same encoder and dimensionality reduction as the initial pool.

Models and token budgets. Both optimizers use Claude Code with Claude Opus 5 at medium effort for proposals and DeepSeek V4 Flash with thinking disabled for execution, as in Section 5.1. Maximum output lengths per executor call are 512 tokens for Text, 2,400 for Math, 8,000 for LiveCodeBench, and 512 for ScienceWorld. Text and ScienceWorld use temperature zero; the selected math harness uses temperature 0.1.

Task-specific computation. Math permits one primary solver call and at most one verifier call per problem. LiveCodeBench allocates separate draft and repair calls and skips repair when a draft passes every public example. ScienceWorld permits at most one solver call per agent turn. Retrieval, output parsing, and program tests are local operations. Executor identity and scoring remain fixed while the optimizer changes module implementations.

Table 11: Default search settings. The window W and refresh interval are measured in completed candidate evaluations.

Setting	Value
Candidate-evaluation budget	40
Implementations per slot	10
Feature dimensions	$d = 128$ per slot; $p = 896$ for seven slots, $p = 640$ for five
Ridge regularization; UCB coefficient	$\lambda = 1$; $\alpha = 0.8$
Coverage; family-reuse penalty	$\eta = 0.04$; $\gamma = 0.04$
Similarity attenuation; minimum scale	$\beta = 0.25$; $g_{\min} = 0.5$
Recent family-usage window	$W = 10$
Start probabilities	Incumbent 0.65, archived 0.20, random 0.15
Archived start candidates	Top $K = 5$ configurations
Coordinate sweeps	2 per selection
Novelty sampling; module proposal limit	200 random attempts; 20 proposals per requested module
Pool refresh	Every 5 evaluations, while budget remains
Children per refresh	At most 2, in distinct slots
Search seed	42

C.4 Metrics and aggregation

Scores are reported on a 0–100 scale. Each selected harness is frozen and evaluated three times on the same held-out data. For task score $S^{(r)}$ in repetition r, the reported score is $S = \frac{1}{3}\sum_{r=1}^{3} S^{(r)}$. The cross-task score in Figure 2 is the unweighted mean of the four task scores.

Text. Let $s_d^{(r)}$ be accuracy on dataset d in repetition r. Text gives equal weight to the three datasets:

$$S_{\text{Text}}^{(r)} = \frac{s_{\text{USPTO}}^{(r)} + s_{\text{Symptom}}^{(r)} + s_{\text{Law}}^{(r)}}{3}. \tag{22}$$

USPTO uses exact set agreement on normalized precursor strings; Symptom-to-Diagnosis uses exact diagnosis agreement after case, whitespace, and punctuation normalization; LawBench requires agreement on the full set of charges. The same dataset-macro convention defines the validation reward.

Math. We use the pooled pass@1 protocol shared with Meta-Harness. Let $n_{\text{O}} = 254$ and $n_{\text{A}} = 246$ be the numbers of test instances from OlympiadBench and MathArena, and let $C_{\text{O}}^{(r)}$ and $C_{\text{A}}^{(r)}$ be their correct-answer counts in repetition r. Then

$$S_{\text{Math}}^{(r)} = 100\,\frac{C_{\text{O}}^{(r)} + C_{\text{A}}^{(r)}}{n_{\text{O}} + n_{\text{A}}}. \tag{23}$$

Every instance has equal weight, so the two source-level percentages are weighted by their test-set sizes. Each source's programmatic grader evaluates the final extracted answer. Internal retrieval or verification calls are part of one harness execution; the resulting final answer is the prediction scored for that instance.

LiveCodeBench: cap-8 solve rate. For each problem, the evaluator concatenates the public and private test cases, in that order, and retains at most eight. The selected program solves the problem if it passes every retained case within the six-second timeout per case. The solve rate is the percentage of test problems solved. A harness may generate several drafts or use public feedback for repair, but only its final selected program is scored. The cap refers to test cases per problem.

ScienceWorld. For episode i, let $s_{i,t}^{(r)}$ be the native environment score at step t. The episode contributes its highest attained progress, clipped to the 0–100 range:

$$z_i^{(r)} = \min\Big(100, \max\Big(0, \max_t s_{i,t}^{(r)}\Big)\Big), \qquad S_{\text{SW}}^{(r)} = \frac{1}{102}\sum_{i=1}^{102} z_i^{(r)}. \tag{24}$$

Partial progress therefore contributes to the score, while full episode completion corresponds to 100.

C.5 FINAL SELECTION AND COST ACCOUNTING

Validation-based selection. The final harness is selected from the validation archive. Math ranks candidates by development pass@1, breaking ties by lower mean input-token use and then by candidate identifier. LiveCodeBench and ScienceWorld use validation score, then lower total token use and candidate identifier. Text uses validation accuracy under the strict component interface. The selected code, executor settings, and retrieval resources are fixed before testing. Each test repetition starts with fresh task memory and environment episodes.

Search budget. MESH-Harness evaluates 40 complete compositions; Meta-Harness evaluates two candidates in each of 20 proposal rounds (Lee et al., 2026). Pool initialization, embedding, interface checks, and coordinate comparisons are optimization work. Reference-harness evaluations are recorded separately from the 40 new-candidate evaluations. All initialization and reference-execution expenses are included in search cost.

Reported cost. The test-only totals in Table 6 measure execution of the frozen harnesses. Search-plus-test totals additionally include proposer, executor, and embedding usage during optimization, with provider-reported cache accounting. At deployment, only the selected harness and its task resources are required. Infrastructure interruptions are resumed as the same evaluation; an executed answer or program that fails the task receives the task's failure score.

D Analysis of Optimized Harnesses

We analyze the final harnesses selected by validation performance on the four tasks. Their structures show how MESH turns a pool of reusable implementations into task-specific programs: modules determine what information to retain, how to present it to the model, and how to act on the resulting response. We first examine these decisions within each task, then relate the selected components to their origins in the pool. The boxes reproduce prompt or code excerpts from the final implementations, with only line breaks or outer indentation adjusted for display.

D.1 Text: evidence composition with explicit answer contracts

The Text harness separates seven decisions: maintaining training examples, defining the answer space, retrieving evidence, composing context, allocating computation, checking the response, and extracting the final answer. A shared answer contract connects these decisions across single-label prediction, multi-label prediction, and structured-string generation. The budget policy identifies the requested answer type from the input and stored examples, allowing downstream components to preserve the semantics of each task. Structured strings, for example, remain open-ended outputs rather than entries in a finite label inventory.

The selected memory module removes exact input–target duplicates and, when its pool exceeds capacity, preferentially removes redundant examples from large signature groups. The retriever ranks examples using weighted feature overlap, input-length proximity, and script-dependent features, while suppressing near-duplicates and repeated answer labels. The composer displays at most four supplied examples and adds a short "Decisive difference" annotation to each: features shared with the query but absent from the other displayed cases are identified explicitly. Retrieval and composition thus serve complementary roles: one selects examples, while the other exposes the distinctions the model should consider.

Text: explicit label-set reasoning

```
Examine each actor and each act on its own and decide
which labels it supports.
Report every applicable label. A combination never seen
together in the cases below is allowed as long as each
label is a known atomic label.
Before answering, re-scan the input for a supported label
you left out, then drop any label with no direct support
in the input.
```

The answer contract continues through generation and parsing. A JSON field carries the final answer, with separate rules for a complete label, a combination of labels, or a structured string. For the latter, the prompt requests preservation of stereochemistry, charges, and ring closures. The optimized program therefore joins evidence selection, comparison, and answer formatting through explicit interfaces. These connected choices make complete-program evaluation relevant: an example is useful to the solver only through the context and decision rules with which it is presented.

D.2 Math: compact problem context and exact-answer discipline

The Math harness contains seven slots for query analysis, budget allocation, retrieval, context composition, solving, verification, and answer parsing. The selected combination centers on the requested mathematical quantity and its output form. Its analyzer distinguishes counting, finding all solutions, extremal questions, and exact-value questions, then extracts constraints and corresponding obligations. These include completeness for a solution set, or an attaining construction and a matching bound for an extremum.

The selected composer preserves the complete problem and adds compact planning notes and an exact-answer contract. It does not insert the retrieved passages, even though retrieval is available through the task interface. This is a concrete choice within the optimization space: a component may determine which available information enters the solver context. Here, the resulting prompt keeps the problem, its constraints, and the required form of the answer together.

Math: compact context and exact answers

```
Solve one olympiad math problem. No reference material
is supplied.

EXACT FORM: keep radicals, fractions, factorials and
symbolic constants unevaluated rather than decimal; use
set or interval notation for collections or ranges;
omit units unless the problem asks for them.

Run the check, then emit the complete JSON final_answer;
if later checking corrects it, emit a new complete JSON
object with the corrected value.
```

The solver adds a problem-dependent instruction: maintain a ledger when many named quantities occur, substitute explicitly given numerical constants when applicable, or proceed through the governing identity. For long derivations it requests forward progress without repeated restarts and a complete answer once a defensible value is available. The verifier checks specific completion defects, including a missing answer after truncation and an unresolved symbol where a definite result is requested. The parser first uses explicit answer fields, then applies answer-region and exact-form fallback rules. The resulting division of labor carries the answer specification from problem analysis through derivation and final extraction, while preserving a compact solver context.

D.3 LiveCodeBench: complementary drafts and failure-directed repair

The LiveCodeBench harness separates problem analysis, budget selection, prompt construction, draft generation, public-test feedback, repair selection, and code parsing. The analyzer extracts the entry-point contract, requested output, input-size bounds, and boundary conditions. The prompt distinguishes a function or class method from a standard-input program and reproduces public examples. The selected budget requests two drafts with repair enabled. Both drafts use temperature zero, with different instructions: one models the process directly; the other is asked to re-derive the requested output quantity independently, using requirement sentences extracted from the statement when available.

Each draft is evaluated on the public tests. The repair selector compares failures across drafts: a failing row is *local* if another draft passes it, *contested* if the drafts produce different wrong outputs, and *shared* otherwise. Syntax, timeout, and runtime defects take precedence when choosing the repair instruction.

LiveCodeBench: public-feedback arbitration

```
mine = _clip(r.get('actual'), 300)
tag = 'shared'
for o in others:
    c = _cell(o, i)
    if c is None:
        continue
    if c.get('passed') is True:
        tag = 'local'
        break
    if _clip(c.get('actual'), 300) != mine:
        tag = 'contested'
```

The feedback interface retains the evidence needed for this comparison, including outputs and failures for individual public-test rows. The repair module can therefore use a passing alternative to guide a local correction, or direct attention to the required output when multiple drafts disagree or fail together. Its prompt includes a small set of failing rows, requests preservation of passing behavior, and warns against special-casing public examples. This case illustrates how reusable components can

exchange structured execution evidence: draft generation supplies alternatives, feedback characterizes them, and repair turns those observations into a targeted instruction.

D.4 SCIENCEWORLD: PLANNING WITH OBSERVED PRECONDITIONS

The ScienceWorld harness uses five slots: evidence memory, goal-phase planning, action-candidate ranking, a focus guard, and recovery. Evidence memory records attempted actions, observed changes, score increments, locations, and focus history. The planner converts this record into a concrete next subgoal; ranking and guard modules constrain which actions are proposed and accepted. Recovery uses the same state and permission interfaces to replace an invalid or repeatedly ineffective action.

The selected planner also illustrates how a new implementation is incorporated into the pool. It was introduced at search iteration 24 by a trace-aware, failure-driven refresh targeting incomplete goals. The planner constructs an ordered chain of preconditions: perform a permitted focus action, navigate toward an absent target, inspect a visible object's exact identity, establish a missing circuit element, measure an unresolved property, or execute a report branch supported by an observation. The following excerpt distinguishes an unknown property from a satisfied report condition.

ScienceWorld: observation-dependent planning

```
try:
    val = observed_property(r['condition'], st, bd)
except Exception:
    val = None
if val is None:
    if pending is None:
        pending = r
elif val is True and ready is None:
    ready = r
```

When progress stalls, the planner can move to a less-attempted later precondition from its chain. The focus guard and recovery module then check proposed actions through their own interfaces. The refreshed planner was selected at iterations 27, 29, 32, 35, and 36, and the final iteration-36 combination joins it with four components from the initial pool. A new planning policy can therefore be evaluated and reused while the surrounding evidence, ranking, guard, and recovery implementations remain available for recombination.

D.5 REUSE, SELECTION, AND EVOLUTION IN THE FINAL PROGRAMS

The final Text, Math, and LiveCodeBench harnesses each combine seven implementations from the initial pool; ScienceWorld combines four initial implementations with the refreshed planner. Their provenance illustrates the two roles of the pool: preserving alternatives for combination selection and admitting new implementations during search. Retention and evolution operate at the level of individual slots, while evaluation measures the behavior of the assembled program.

Across tasks, the selected programs organize different forms of evidence: comparisons among examples in Text, mathematical constraints and answer forms in Math, public-test outcomes in LiveCodeBench, and observed preconditions in ScienceWorld. Stable interfaces allow these control policies to be stored and recombined without prescribing a single reasoning workflow. Together with the main-text ablations, these cases connect the search design to the concrete programs it produces.