

HarnessNet: Feedback-driven Hierarchical Joint Harness Evolution for Multi-agent Systems

Zhaoyuan Qiu Sachith Seneviratne Yu Xia Saman Halgamuge

University of Melbourne, Melbourne, Australia

`zhaoyuanqiu@student.unimelb.edu.au`

Abstract

AI agents can evolve by using feedback from task environments to revise their harnesses, the software that manages model inputs, tool use, memory, and execution, while keeping the underlying foundation models frozen. Recent studies focus on this process for single agents, leaving underexplored the harness evolution in Multi-agent System (MAS), which requires coordinating harness changes across multiple agents. However, using a single global optimizer, the straightforward extension from single agent harness evolution, faces a growing context and workload as the MAS scale increases. Access restrictions introduce further challenges: agents may be unable to share internal information with a common optimizer, and some agents may lack permission to access the environment feedback which is for the entire MAS. To address these challenges, we propose HarnessNet, a framework for hierarchical harness evolution in MAS. Its basic unit, Agentron, consists of an evolvable forward harness, a foundation model, and a local optimizer that can observe and modify only that forward harness. Multiple Agentrons are orchestrated into a layered graph, and feedback is propagated backward from the task environment through the system so that each optimizer only receives node-specific information relevant to its own Agentron from downstream layer. Joint harness evolution can therefore be divided into smaller, coordinated tasks with separate contexts, without requiring every optimizer process to access all Agentron trajectories or raw environment feedback. This reduces the context and workload handled by individual optimizers, and keeps optimizer access local. We conduct experiments on AgentBench with multiple foundation models to demonstrate the feasibility of training HarnessNet. A comparison study against single global optimizer approach shows that HarnessNet achieves comparable performance gains while respecting local privacy, with fewer tokens and less time required per optimizer call. A migration study further shows that the evolved harnesses remain effective after replacing the foundation models, suggesting the transferability.

Keywords: Multi-agent Systems, Harness Engineering, Hierarchical Evolution

1 Introduction

The behavior of Large Language Model (LLM) based agents depends on both the underlying models and the software infrastructure [1, 2]. Agent development therefore places growing emphasis on refining the software that organizes and directs model capabilities. This trend reflects an emerging paradigm shift in AI system optimization: from a primary focus on model weights toward model-external components that govern system behavior, release or extend the capabilities attainable with a given foundation model, exemplified with prompt engineering [3, 4], context engineering [5, 6], and, more recently, harness engineering [7, 8]. Expressed in natural language and code, these software components and their revisions are directly accessible to human inspection and audit, supporting efforts to improve system security and controllability. Viewing the harness itself as an optimization target motivates harness evolution, in which task feedback guides iterative harness refinement while the underlying foundation models remain frozen.

Recent work has explored closed-loop feedback-driven agent harness evolution as a form of agent system training, but has focused mainly on single-agent harness evolution [9–14], leaving joint harness evolution in multi-agent systems underexplored. In single-agent approaches that use a coding agent as the optimizer, this optimizer analyzes the agent’s execution trajectories and modifies its harness based on feedback. A direct extension to multi-agent systems would assign a single optimizer the responsibility of analyzing all agents’ trajectories and updating their harnesses jointly. However, such a global optimizer extension faces challenges: **(1)** as the system scales up, this one-for-all design has a growing context and workload burden, which can degrade its performance [15–17]; **(2)** in scenario with distinct privacy or access constraints, agents may not be able to expose their internal trajectories to a shared optimizer [18–20], or directly access the environment feedback of the whole system. These limitations motivate a solution to joint harness evolution that decomposes multi-agent system harness evolution into small, independently executable tasks while coordinating harness updates across agents, using local feedback. In each task, optimizer visibility and authority are restricted to the agent being updated and preserve local privacy and

access boundary.

To address these gaps, we propose HarnessNet, a framework of hierarchical, feedback-driven joint harness evolution for multi-agent systems. Its fundamental unit, Agentron, couples an evolvable agent harness with a local optimizer instance whose visibility and authority are confined to that unit. HarnessNet organizes these units into a layered supergraph and propagates unit-specific feedback hierarchically to coordinate local harness updates. We use an automated task-specific initialization procedure to reduce the need for manual harness design, while persistent textual records of harness revisions and optimizer reasoning make the evolution process auditable. Experiments on AgentBench [21] demonstrate significant improvements in task performance through iterative hierarchical harness evolution. Further migration studies show that evolved harnesses retain useful task-specific mechanisms across different foundation models. This work makes the following contributions.

1. We introduce Agentron, an AI unit that combines an evolvable forward harness with a local optimizer instance, functioning as a building block for cascade evolvable AI systems.
2. We propose HarnessNet, which coordinates local harness updates through hierarchical feedback propagation, achieving gains comparable to global optimization while reducing per-optimizer context and workload, preserving local privacy and access boundaries.
3. We develop an automated task-specific initialization procedure and evaluate HarnessNet on AgentBench tasks, demonstrating the feasibility of automated initialization and the effectiveness of subsequent harness evolution.
4. We assess the transferability of evolved HarnessNet systems through migration studies across foundation models. Results indicate that task-specific rules learned within agent harnesses remain jointly effective on different models.

2 Related Work

2.1 Agent Improvement Beyond Model Weights

Beyond the foundation model, an agent relies on semantic artifacts that shape its behavior and execution strategy. Refining these artifacts offers a way to improve agent performance without updating model weights. ProTeGi [22], OPRO [23], PromptAgent [24], and GEPA [25] optimize task instructions through iterative search and feedback, while MemAPO [26] accumulates reusable strategies and failure patterns to guide prompt refinement. Context-oriented approaches include Context Language

Models [27] and Agentic Context Engineering [28], complemented by ACON’s feedback-driven context compression [29] and HiAgent’s hierarchical working memory management [30]. Reflexion [31], A-MEM [32], and Dynamic Cheatsheet [33] improve the reuse of past experience through textual reflection, memory organization, and persistent strategy accumulation. AutoSkill [34], MUSE-Autoskill [35], CoEvoSkills [36], SkillGrad [37], and MetaSkill-Evolve [38] further explore the acquisition and refinement of reusable agent skills. Together, these methods establish prompts, context, memory, and skills as targets for agent improvement beyond model weights.

2.2 Agent Harness Evolution

Agent harness evolution extends individual artifact refinement to the agent harness, the broader agent software infrastructure, as the optimization target. SICA [39], Darwin Gödel Machine [40], and MOSS [41] show that coding agents can revise their own software to create agent substrates. Meta-Harness searches over complete harness implementations using source code, traces, and scores [9]. AHE structures harness components and execution evidence to support observable edits [11]. Self-Harness couples model-specific weakness discovery with proposal validation [42]. Life-Harness [10], Continual Harness [12], and Adaptive Auto-Harness [13] further extend harness evolution to runtime interventions and non-stationary task streams. MILO [43] uses multi-agent to optimize agent harness. However, these works focus on single agent harness evolution, leaving underexplored the scenario of refining agent harnesses in multi-agent systems with end-to-end feedback. HarnessNet closes this gap by introducing a joint harness evolution process through hierarchical feedback propagation and node-specific evolution.

2.3 Multi-agent System Optimization

Multi-agent system (MAS) optimization has explored agent selection, prompt refinement, and workflow search, leaving harness underexplored as an optimization scope. DyLAN [44] selects task-specific agent teams and adapts communication structures. GPT-Swarm [45] represents language agents as optimizable graphs and updates node prompts and graph connectivity, while Agentic Supernet [46] searches over query-dependent multi-agent architectures. ANN [47] combines layered execution with textual backpropagation to refine prompts and collaboration structures. EvolveNet [48] aggregates program adaptations from local deployments to improve a shared harness. MASS [49] coordinate the optimization of agent prompts and collaboration structures. MASPO [50] jointly refines agent prompts by evaluating their effects on downstream agents, whereas AgentGrad [51] uses agent-level interventions to identify failure sources and guide targeted prompt updates. EvoAgentX integrates prompt and workflow optimizers [52]. Beyond these approaches, TPGO [53] repre-

sents MAS textual configurations as a Textual Parameter Graph and optimizes its node content and structure. Autogenesis [54] provides versioned resource interfaces and a closed-loop protocol for managing and evolving heterogeneous agent components. In contrast, HarnessNet deliberately makes executable agent harnesses as the optimization object, coordinating local revisions to jointly optimize execution strategies, input-output handling, and memory management on different agent harness.

3 Methods

3.1 Agentron & HarnessNet

3.1.1 Agentron

We introduce Agentron, a harness-based evolvable AI unit whose evolution is driven by execution trajectories and feedback. It is defined as

$$A = (m, h, \mathcal{O}),$$

which contains three components: **(1)** a frozen Large Language Model (LLM) m , which serves as the foundation model and provides reasoning capability through text; **(2)** a modifiable Forward Harness (FH) h , which orchestrates interactions with the model and defines the unit’s local behaviour, including context processing, tool use, and input-output handling; and **(3)** a frozen optimizer $\mathcal{O}$, an independent coding agent that analyzes execution trajectories, modifies the harness at the code level, and, when applicable, generates feedback for upstream units based on the resulting harness changes and the upstream inputs recorded in the trajectories.

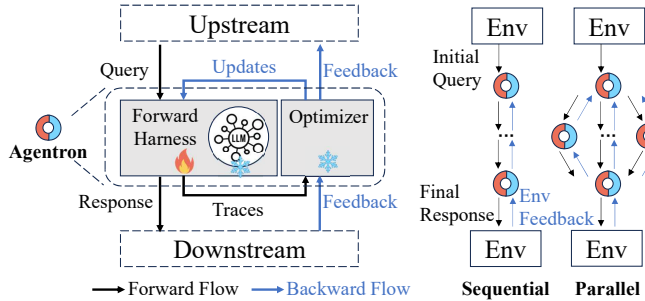


Figure 1: (Left) Agentron; (Right) Agentron Expansibility

As illustrated in Figure 1, Agentron operates in two phases. In the forward phase, given a query or a set of messages from upstream units, denoted by x , it produces an output y and an execution trace τ that records evidence for subsequent analysis:

$$(y, \tau) = A.Forward(m, h, x).$$

In the backward phase, given downstream feedback s and the forward trace τ , the frozen optimizer $\mathcal{O}$ propose an

updated harness H' and generate feedbacks f for upstream units when applicable:

$$(h', f) = A.Backward(\mathcal{O}, h, \tau, s).$$

The foundation model M remains unchanged in both phases. Only forward harness h is updated. An Agentron can receive messages from multiple upstream units and transmit outputs to multiple downstream units. Its harness determines how incoming information is selected, transformed, and aggregated, while its optimizer can generate feedback for different upstream units. This design supports building of cascade systems composed of multiple Agentron units through sequential stacking or parallel branching, with the task environment as both the starting point and the endpoint.

3.1.2 HarnessNet

We introduce HarnessNet, a layered multi-agent system composed of multiple Agentron units. Its backbone is represented by the non-cyclic directed graph

$$G = (V_1, \dots, V_L, E),$$

where

$$V_\ell = \{v_i^\ell \mid 1 \leq i \leq N^\ell\}$$

is the set of Agentron nodes in layer ℓ . Each node is an Agentron of the form

$$v_i^\ell = (m_i^\ell, h_i^\ell, \mathcal{O}_i^\ell),$$

and

$$E = \bigcup_{\ell=1}^{L-1} (V_\ell \times V_{\ell+1})$$

specifies full connectivity between each pair of adjacent layers.

As illustrated in Figure 2-Left, HarnessNet operates through a forward process and a backward process.

In the forward process (Algorithm 1), nodes are executed in topological order. Each node receives the outputs of the connected nodes in the preceding layer. Its forward harness selects, transforms, and aggregates these messages, then it goes through a multi-round agentic reasoning process before producing and broadcasting outputs for the next layer. The first layer receives the task input from the environment, while the last layer returns the system response. Nodes in intermediate layers process information according to their roles and the behaviour specified by their forward harnesses. By default, the first and last layers each contain a single node, although multiple nodes can be used to handle multiple environment inputs or outputs. Since nodes within the same layer do not depend on one another, their harness update processes can be executed in parallel.

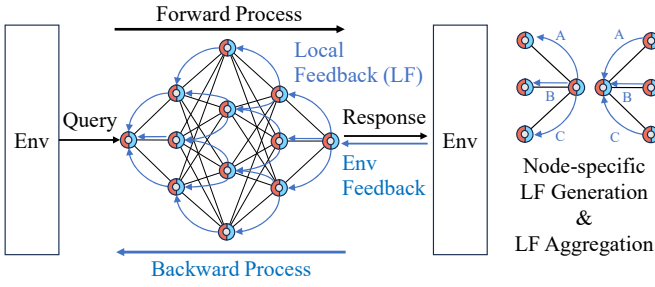


Figure 2: The dual-phase of HarnessNet (Left; some links are omitted for clarity) and the propagation of feedback (Right).

Algorithm 1: HarnessNet Forward Process

Input: HarnessNet backbone G , task query $\mathcal{Q}$
Output: Layer outputs y , forward trace $\mathcal{T}$

```

1   $y, \mathcal{T} \leftarrow \emptyset, \emptyset$ 
2  for  $\ell = 1, \dots, L$  do
3     $y^\ell \leftarrow \emptyset$ 
4     $x^\ell \leftarrow y^{\ell-1}$  if  $\ell > 1$  else  $\mathcal{Q}$ 
5    for each  $v_i^\ell \in V_\ell$  in parallel do
6       $y_i^\ell, \mathcal{T}_i^\ell \leftarrow v_i^\ell.\text{Forward}(m_i^\ell, h_i^\ell, x^\ell)$ 
7       $y^\ell, \mathcal{T} \leftarrow y^\ell \cup y_i^\ell, \mathcal{T} \cup \mathcal{T}_i^\ell$ 
8    end for
9     $y \leftarrow y \cup y^\ell$ 
10 end for
11 return  $(y, \mathcal{T})$     (Retrieve  $y^L$  for final responses.)
```

Algorithm 2: Hierarchical Backward Process

Input: HarnessNet backbone G , forward trace $\mathcal{T}$, environment feedback s
Intermediate: Upstream-layer feedback $\mathcal{F}$, downstream-layer feedback g
Output: Updated HarnessNet harnesses H'

```

1   $H' \leftarrow \emptyset$ 
2  for  $\ell = L, \dots, 1$  do
3     $\mathcal{F}^\ell \leftarrow \emptyset$ 
4    for each  $v_i^\ell \in V_\ell$  in parallel do
5       $(m_i^\ell, h_i^\ell, \mathcal{O}_i^\ell) \leftarrow v_i^\ell$ 
6       $g_i^\ell \leftarrow \{\mathcal{F}_{k \rightarrow i}^{\ell+1} \mid v_k^{\ell+1} \in V_{\ell+1}\}$  if  $\ell < L$  else  $s$ 
7      if  $\ell > 1$  then
8         $(h_i^\ell)', \{f_{i \rightarrow j} \mid v_j^{\ell-1} \in V_{\ell-1}\} \leftarrow$ 
9         $v_i^\ell.\text{Backward}(\mathcal{O}_i^\ell, h_i^\ell, \mathcal{T}_i^\ell, g_i^\ell)$ 
10        $\mathcal{F}^\ell \leftarrow \mathcal{F}^\ell \cup \{f_{i \rightarrow j} \mid v_j^{\ell-1} \in V_{\ell-1}\}$ 
11       else  $(h_i^\ell)' \leftarrow v_i^\ell.\text{Backward}(\mathcal{O}_i^\ell, h_i^\ell, \mathcal{T}_i^\ell, g_i^\ell)$ 
12       end if
13        $v_i^\ell \leftarrow (m_i^\ell, (h_i^\ell)', \mathcal{O}_i^\ell)$ 
14        $H' \leftarrow H' \cup (h_i^\ell)'$ 
15     end for
16   end for
17 return  $H'$ 
```

In the hierarchical backward process (Algorithm 2), environment feedback is received by the final layer. Nodes are then evolved layer by layer in reverse topological order, based on the execution traces and downstream feed-

back. Each local optimizer instance updates the forward harness and, where applicable, generates feedback based on the resulting harness revisions and the predecessor agents forward outputs. This process enables node-specific local feedback generation and aggregation (Figure 2-Right), while allowing nodes in the same layer to evolve in parallel.

3.2 Task-aware HarnessNet Initialization

Constructing a multi-agent system typically requires heavy human input about agent roles, functions, and workflow. To overcome this problem, HarnessNet uses an Initializer $\mathcal{I}$, implemented as a coding agent, to automatically plan and build initial forward harnesses for the Agentrons in the predefined HarnessNet backbone G .

As shown in Algorithm 3, the initializer receives task-environment knowledge K_e , information about the nodes' reasoning backends K_r , and a default harness protocol h specifying the repository structure, core modules, and interfaces. Using curated task-specific harness protocol h is possible for more customized initialization. Actual task cases are strictly excluded from K_e to prevent data leakage.

Algorithm 3: HarnessNet Initialization

Input: HarnessNet backbone G , initializer $\mathcal{I}$, harness protocol h
task env knowledge K_e , reasoning backend info K_r
Output: Initial HarnessNet harnesses H^0

```

1   $H^0 \leftarrow \emptyset$ 
2   $V_{\text{spec}} \leftarrow \mathcal{I}.\text{Plan}(G, K_e, K_r)$ 
3  for  $\ell = 1, \dots, L$  do
4    for each  $v_i \in V_\ell$  in parallel do
5       $H_i^l \leftarrow \mathcal{I}.\text{Implement}((V_{\text{spec}})_i^l, h)$ 
6      if  $\mathcal{I}.\text{Test}(H_i^l)$  is Fail do
7         $H_i^l \leftarrow \mathcal{I}.\text{Repair}(H_i^l)$ 
8       $H^0 \leftarrow H^0 \cup H_i^l$ 
9    end for
10 end for
11 return  $H^0$ 
```

As illustrated in Figure 3, the Initialization proceeds in three stages. **(1)** The initializer develops a nodes specification for the network, assigning node roles and specifying strategies for memory, execution, tool use, and input-output parsing. During this process, the Initializer is instructed to assign complementary functions across nodes and avoids homogeneity. **(2)** Each node's forward harness is implemented according to this specification and the protocol h . **(3)** Initializer performs a smoke test on each harness and attempts a repair if the test fails. The resulting initial harnesses are assigned to the nodes of G to form the HarnessNet instance. More details about the Initializer implementation are provided in Appendix A.

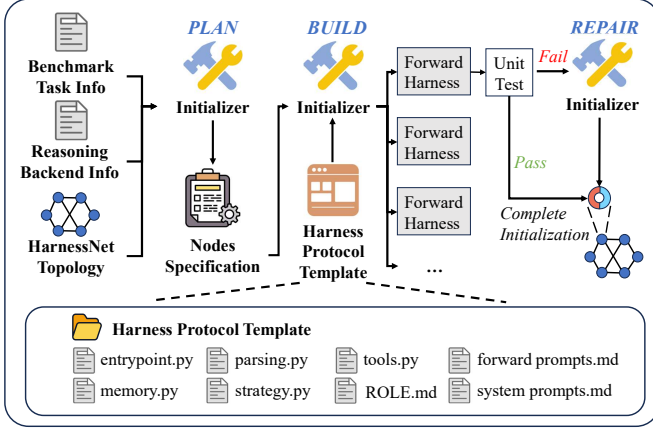


Figure 3: HarnessNet 3-Stage Task-aware Initialization Workflow, with a template provided as reference.

3.3 HarnessNet Training

Algorithm 4: HarnessNet Validation Process

Input: HarnessNet backbone G , validation set $\mathcal{E}^{val}$

Intermediates: Forward trace $\mathcal{T}$, layer outputs y , evolution step k , env feedback s , training results $\mathcal{U}$, validation results $\mathcal{V}$

Output: Validation results $\mathcal{V}$

```

1   $\mathcal{V} \leftarrow \emptyset$ 
2  for  $i = 1, \dots, \|\mathcal{E}^{val}\|$  do
3     $(y_i, \mathcal{T}_i) \leftarrow G.Forward(\mathcal{E}_i^{val}.Query)$  (Algorithm 1)
4     $s_i \leftarrow \mathcal{E}_i^{val}.Feedback(y_i^L)$ 
5     $\mathcal{V} \leftarrow \mathcal{V} \cup s_i$ 
6  return  $\mathcal{V}$ 

```

We train HarnessNet following a batch-training-and-validate schema inspired by neural network training (Algorithm 5). At each iteration, tasks from $\mathcal{E}^{train}$ are randomly shuffled and evaluated with a fixed all nodes harness snapshot H^k until a batch of b failed cases is collected. A backward pass updates the harnesses using this batch, after which the updated snapshot H^{k+1} is evaluated on $\mathcal{E}^{val}$. The final snapshot H^* is selected from all evaluated snapshots based on both training and validation performance. Given a fixed batch size, using only failed cases reduces the number of backward passes required to traverse the $\mathcal{E}^{train}$, but successful cases can also be included and might help with the evolution process according to some recent works [14, 29, 55]. Similar to traditional neural network training, we select the best training checkpoint by considering both the training performance $\mathcal{U}$ and validation performance $\mathcal{V}$.

Algorithm 5: HarnessNet Training Loop

Input: HarnessNet backbone G , failure batch size b , training set $\mathcal{E}^{train}$, validation set $\mathcal{E}^{val}$, max training rounds T

Intermediates: Forward trace $\mathcal{T}$, layer outputs y , evolution step k , env feedback s , training results $\mathcal{U}$, validation results $\mathcal{V}$, checkpoints $\mathcal{H}$

Output: Best HarnessNet harnesses checkpoint H^*

```

1   $k, \mathcal{T}, s, \mathcal{U}, \mathcal{V}, \mathcal{H} \leftarrow 0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$ 
2   $H^0 \leftarrow \{h_i^\ell | v_i^\ell \in V_\ell, 1 \leq \ell \leq L\}$  (FH0)
3   $\mathcal{H} \leftarrow \mathcal{H} \cup H^0$ 
4   $\mathcal{V}^0 \leftarrow Validate(G, \mathcal{E}^{val})$  (Algorithm 4)
5   $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}^0$ 
6  for  $t = 0, \dots, T - 1$  do
7     $\mathcal{E}^{train} \leftarrow \mathcal{E}^{train}.Shuffle()$ 
8     $H^k, \mathcal{U}^k \leftarrow \{h_i^\ell | v_i^\ell \in V_\ell, 1 \leq \ell \leq L\}, \emptyset$ 
9    for  $i = 1, \dots, \|\mathcal{E}^{train}\|$  do
10      $(y_i, \mathcal{T}_i) \leftarrow G.Forward(\mathcal{E}_i^{train}.Query)$  (Algorithm 1)
11      $s_i \leftarrow \mathcal{E}_i^{train}.Feedback(y_i^L)$ 
12      $\mathcal{U}^k \leftarrow \mathcal{U}^k \cup s_i$ 
13     if  $IsFailure(s_i)$  then
14        $\mathcal{T}, s \leftarrow \mathcal{T} \cup \mathcal{T}_i, s \cup s_i$ 
15       if  $\|s\| = b$  or  $i == \|\mathcal{E}^{train}\|$  then (Backward)
16          $H^{k+1} \leftarrow G.Backward(\mathcal{T}, s)$  (Algorithm 2)
17          $\mathcal{V}^{k+1} \leftarrow Validate(G, \mathcal{E}^{val})$  (Algorithm 4)
18          $\mathcal{U}, \mathcal{V}, \mathcal{H} \leftarrow \mathcal{U} \cup \mathcal{U}^k, \mathcal{V} \cup \mathcal{V}^{k+1}, \mathcal{H} \cup H^{k+1}$ 
19          $\mathcal{T}, s, \mathcal{U}^k, \mathcal{V}^{k+1} \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$ 
20          $k \leftarrow k + 1$ 
21       end if
22     end if
23   end for
24 end for
25  $H^* \leftarrow GetBest(\mathcal{H}, (\mathcal{U}, \mathcal{V}))$ 
26 return  $H^*$  (Trained HarnessNet Harnesses)

```

4 Experiments

4.1 Experiment Settings

4.1.1 Dataset

We use AgentBench [21], a light-weight agentic benchmark, to demonstrate the training feasibility of proposed Agentron and HarnessNet. We focus on ALFWorld and WebShop, whose environment-training splits reuse the original benchmarks' official training data, while other domains rely on synthetic training data, which may introduce train-test distribution shifts [56]. To save training efforts while preventing testset leakage, we created training and validation set using the randomly sampled cases from *alfworld-env_train* and *webshop-env_train*, and then test the selected harness checkpoints on *alfworld-std* and *webshop-std*, the official benchmark testsets. Case quantity details are in Table 1.

4.1.2 Models, Initializer and Optimizer

To simplify the setup and reduce computational costs, in the experiments, we equip Agentrons in the same Har-

Table 1: Experiment Data Splits

Benchmark	ALFWorld			WebShop		
Split	Train	Validation	Test	Train	Validation	Test
Case Num	168	42	109	320	80	200

nessNet with independent instances of the same foundation model and optimizer. Models include Qwen3.5-9B [57], Gemma4-8b-e4b [58], Ornith 1.5-9b [59] and Qwen3.8-27B [60], using ollama [61] as reasoning backend, GPT5.6-SOL High [62] combined with codex-cli 0.153.4 as the Initializer and Optimizer. We use zero temperature and fixed dataset sampling/shuffling seeds to improve reproducibility. Complete model parameter settings are provided in Appendix B.

4.2 Training Effectiveness

We assess training feasibility for Agentron and HarnessNet in five steps: (1) Evaluate foundation models on the test sets to establish the Model-only baseline. (2) Instantiate three minimal topologies and initialize their harnesses using Algorithm 3. (3) Measure pre-training test performance (FH0). (4) Train each system using Algorithm 5, assessing convergence and guiding stopping with training and validation curves. (5) Test the selected forward harness checkpoints (FH_starred). We compute *Harness Gain* relative to Model-only and *Training Gain* relative to FH0. Results show that feedback-driven iterative training can improve performance on unseen task instances. Training records are in Appendix C.

4.2.1 Agentron Training

We firstly verify the training feasibility of the fundamental unit, Agentron, which is also a single-node HarnessNet. As the first column in Figure 4 shows, the training process quickly improves checkpoint performance and converges even though the initialized FH0 was bad. Test results are in Table 2.

Table 2: Agentron Training Effects

Benchmark Domain	Test	Score (↑)	Harness Gain	Training Gain
ALF-World	Model-only	0.5688	-	-
	FH0	0.2202	-61.29%	-
	FH_starred	0.7798	+37.10%	+254.13%
WebShop	Model-only	0.5670	-	-
	FH0	0.5559	-1.96%	-
	FH_starred	0.6950	+22.57%	+25.02%

4.2.2 Training with Cross-layer Feedback

After verifying Agentron’s training feasibility, we build the minimal 2-node harnessnet to check the training feasibility when the cross-layer feedback propagation is enabled. As the second column in Figure 4 shows, the FH0 is well initialized with similar performance compared to Model-only baseline, and subsequent training further improves its performance. Test results are in Table 3.

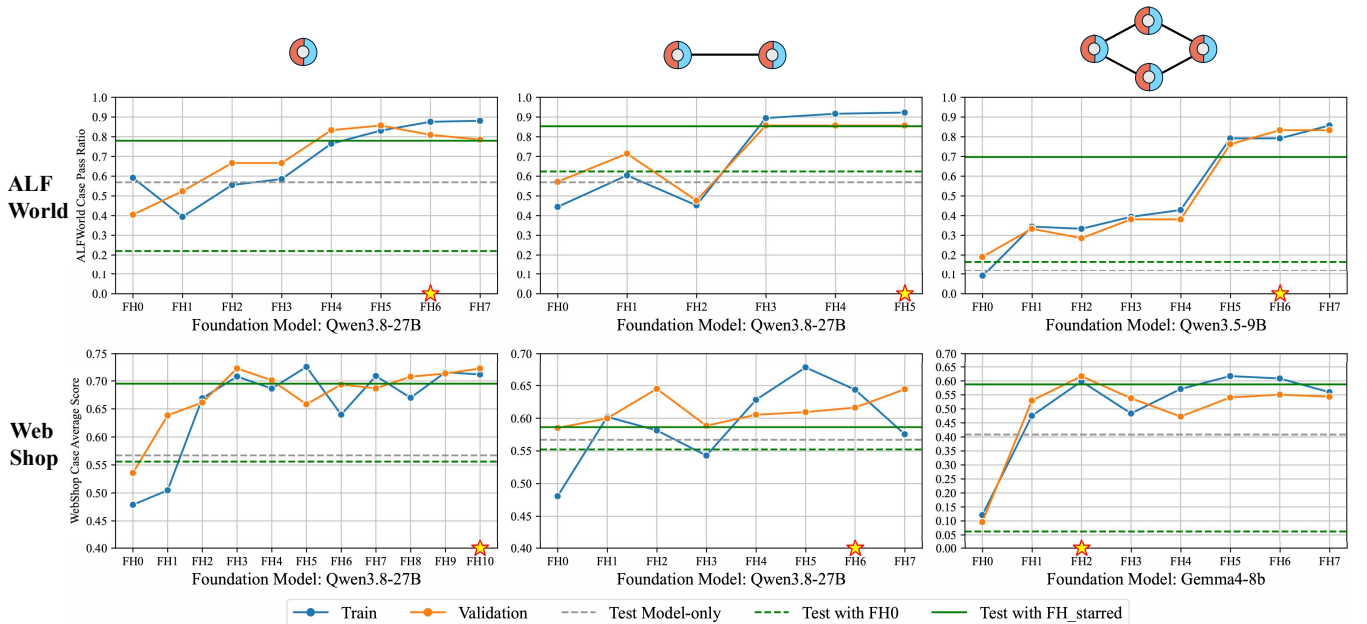


Figure 4: Training Curves and Test Results on AgentBench of Minimal HarnessNet Topologies.

Table 3: 2-Node HarnessNet Training Effects

Benchmark Domain	Test	Score (↑)	Harness Gain	Training Gain
ALF-World	Model-only	0.5688	-	-
	FH0	0.6239	+9.69%	-
	FH_starred	0.8532	+50.00%	+36.75%
WebShop	Model-only	0.5670	-	-
	FH0	0.5523	-2.59%	-
	FH_starred	0.5892	+3.92%	+6.68%

4.2.3 Training with Minimal Node-specific Feedback Generation & Aggregation

Finally, we evaluate a 4-node HarnessNet with an input node, an output node, and two parallel hidden nodes to assess training feasibility with cross-layer, node-specific feedback generation and aggregation mechanism. To reduce computational costs, we use the smaller Qwen3.5-9B [57] and Gemma4-8b-e4b [58] models in place of Qwen3.8-27B [60], which is used in earlier experiments. As shown in the third column of Figure 4, Qwen3.5-9B fails 88.07% ALFWorld test cases in the Model-only setting. However, the 4-node HarnessNet starts from a beneficial FH0 and, through training, achieves performance comparable to that obtained with Qwen3.8-27B. On WebShop, training improves the Gemma4-8b-e4b system from a poorly performing FH0 to a checkpoint that surpasses the Qwen3.8-27B Model-only baseline and matches 2-node HarnessNet training performance. Table 4 reports the test results.

Table 4: 4-Node HarnessNet Training Effects

Benchmark Domain	Test	Score (↑)	Harness Gain	Training Gain
ALF-World	Model-only	0.1193	-	-
	FH0	0.1651	+38.39%	-
	FH_starred	0.6972	+484.41%	+322.29%
WebShop	Model-only	0.4083	-	-
	FH0	0.0624	-84.72%	-
	FH_starred	0.5875	+43.89%	+841.51%

4.3 Ability Migration

We assess harness transferability by replacing Qwen3.5-9B [57] in the trained 4-node ALFWorld HarnessNet with either Ornith1.5-9B [59], an open-source model fine-tuned from Qwen3.5-9B, or Gemma4-8b-e4b [58], an open-source model from a different family. We evaluate the migrated harness checkpoints on the same validation set during training process. As shown in the upper panel of Figure 5, both replacement models benefit from the harnesses evolved with Qwen3.5-9B and exhibit similar performance improvement trends across checkpoints. We

then select FH_starred based on validation performance and evaluate it on the same test set. All three models fail most test case in the Model-only setting. With the evolved 4-node harnesses, however, Gemma4-8b-e4b and Ornith1.5-9B achieve success rates of 68.81% and 49.54%, which coheres with the results observed with Qwen3.5-9B. These results indicate that task-specific mechanisms learned within individual HarnessNet harnesses remain jointly effective across foundation models, including models from a different family. Details of the validation and test are provided in Appendix D.

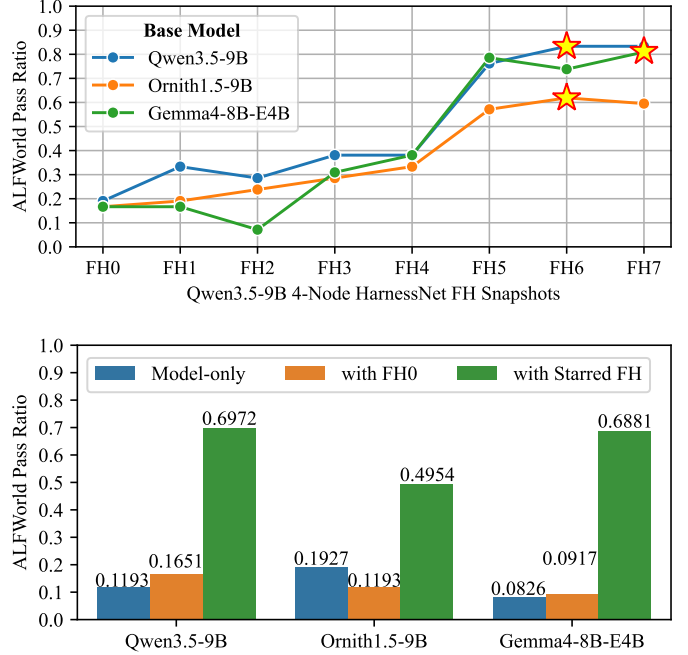


Figure 5: Validation (Up) and Test Scores (Bottom) on ALFWorld across Different Foundation Models and Identical 4-node HarnessNet FH Snapshots Trained with Qwen3.5-9B.

4.4 Hierarchical Approach vs. Global Optimizer

Algorithm 6: HarnessNet Global Optimizer Backward Process

Input: HarnessNet backbone G , forward trace $\mathcal{T}$, environment feedback s

Intermediate: Global harness set $\mathcal{H}$, update set $\mathcal{U}$

Output: Updated harnesses H'

```

1  $H \leftarrow \{h_i^\ell | v_i^\ell \in V_\ell, 1 \leq \ell \leq L\}$ 
2  $H' \leftarrow \mathcal{O}_{\text{global}}.\text{Backward}(G, H, \mathcal{T}, s)$ 
3 for each  $(h_i^\ell)' \in H'$  do
4    $(m_i^\ell, h_i^\ell, \mathcal{O}_i^\ell) \leftarrow v_i^\ell$ 
5    $v_i^\ell \leftarrow (m_i^\ell, (h_i^\ell)', \mathcal{O}_i^\ell)$ 
6 end for
7 return  $H'$ 
```

To evaluate whether hierarchical backward optimization reduces the burden on individual optimizers while maintaining effective HarnessNet evolution, we compare it with a global optimizer approach (Algorithm 6) on the 4-node HarnessNet topology. Both methods start from identical initialized FH0 and the first batch of failed trajectories, and undergo same backward steps on identical training and validation sets. Finally, We compare the checkpoint performances and optimizer costs. Result details are in Appendix E.

Figure 6-Left illustrates two approaches. In the hierarchical backward process, each Agentron uses its local optimizer to update its forward harness based on node-specific feedback from the downstream layer. By contrast, the global optimizer approach uses a global optimizer, implemented as an independent optimizer instance while disabling other Agentron local optimizers. It is granted

with access to all nodes forward harnesses and forward trajectories to jointly update the entire system based on environment feedback.

As Figure 6-Right shows, both approaches improve over initialization, with hierarchical optimization achieving comparable performance on WebShop and substantially higher performance on ALFWorld within the same number of backward steps.

Figure 7 compares the optimizer cost. Although the total cost may rise due to fixed optimizer prompt prefixes, local optimizer calls require fewer per-run input and output tokens and less execution time than global optimizer calls. These results indicate that hierarchical optimization reduces per-optimizer context and workload burdens while maintaining effective system-level improvement.

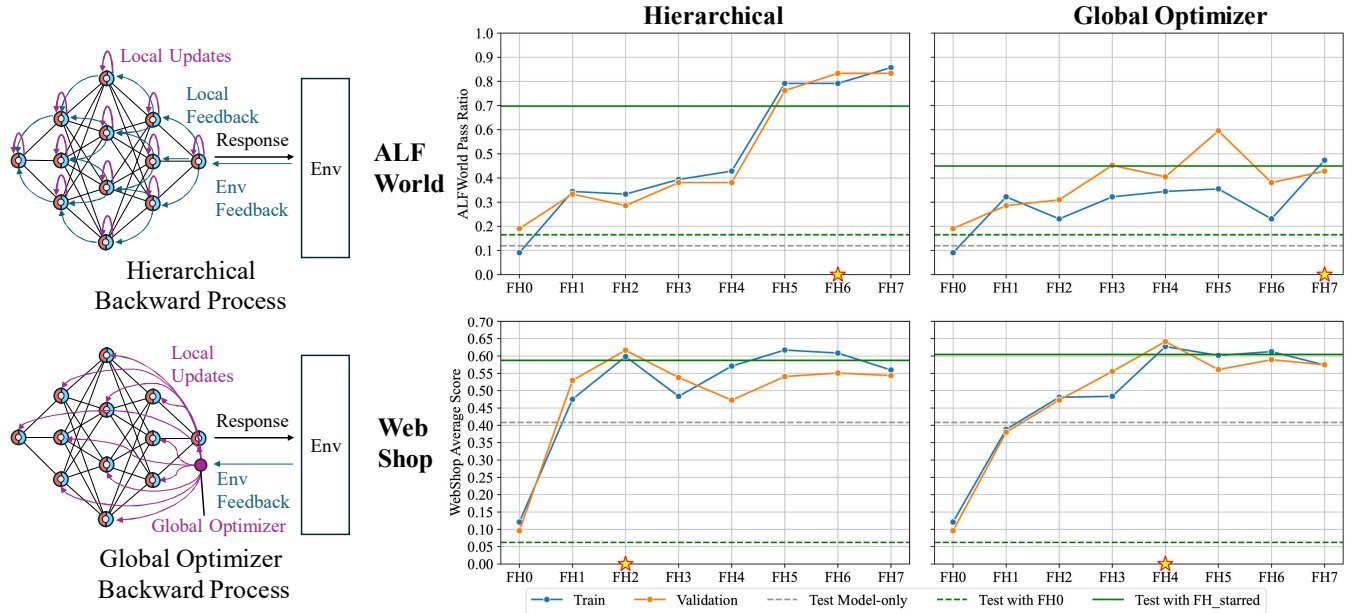


Figure 6: (Left) Illustration of Hierarchical Approach & Global Optimizer Approach; (Right) Training Performance.

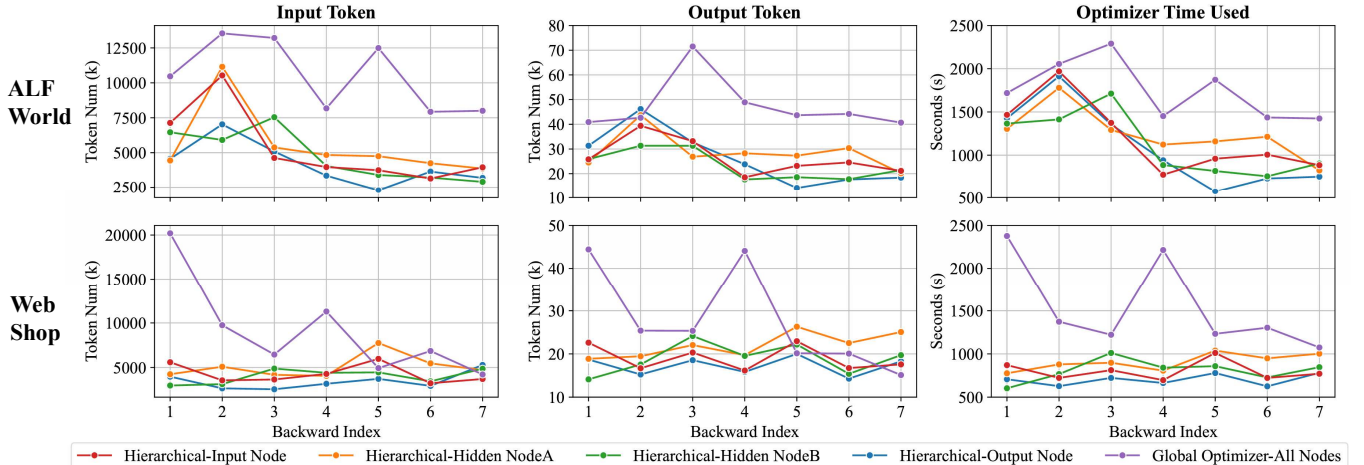


Figure 7: Per-call Optimizer Cost and Latency of Hierarchical Approach vs. Global Optimizer Approach.

4.5 Case Study - Learn Rules, Not Answers

We examine whether the performance gains reflect reusable task-specific mechanisms encoded in the evolved harnesses rather than memorized answers to training tasks. We analyze the 4-node ALFWorld experiment with Qwen3.5-9B [57], shown in the upper-right panel of Figure 4. The foundation models remain frozen, and all updates are confined to the forward harness repositories. This makes the evolution process directly inspectable. We divide the process into four stages based on the training and validation curves; stage headings report checkpoint validation pass rates.

FH0 Initialization. The initial HarnessNet uses a 4-node topology with the following role division:

- **Input Node:** Organizes the task, current observation, interaction history, and admissible actions for the downstream.
- **Hidden NodeA:** Independently analyzes which actions are currently feasible and check their preconditions.
- **Hidden NodeB:** Tracks goal progress and recommends the next useful transition toward task completion.
- **Output Node:** Reconciles the two hidden-node analyses and selects one action from the current admissible set.

At FH0, these nodes contain only general role prompts and forwarding logic. They do not have explicit mechanisms for loop prevention, object-state tracking, multi-object progress, or search planning.

FH0 (19.05%) → FH1 (33.33%). Training and validation performance improve sharply in this stage. Unreliable environment interaction is the main FH0 failure mode: 25 of the first 40 failed cases terminate with invalid actions. Updates in FH1 include:

- **Input Node:** Preserves the latest observation and records held objects and searched locations.
- **Hidden NodeA:** Add detects for repeated information-gathering actions to prevent step waste.
- **Hidden NodeB:** Add canonical actions recommending and starts tracking object count, transformation state, and add the one-object inventory constraint.
- **Output Node:** Converts malformed suggestions into canonical commands, rejects unavailable actions, and resolves conflicts using current environment evidence.

These changes establish more reliable multi-step interaction with the environment. Invalid-action failures decrease from 25 to 5 in the next failure batch, shifting the

main bottleneck from action validity to long-horizon task completion.

FH1 (33.33%) → FH4 (38.10%). Performance improves gradually and non-monotonically. FH1 often searches while already holding the target, confuses similar object names or action meanings, and allows old history to override the latest location or inventory. Updates through FH4 include:

- **Input Node:** Introduces a Goal→Now→Progress→Next→Later structure to organize state and action order.
- **Hidden NodeA:** Uses the latest location and inventory as state and checks immediate action preconditions.
- **Hidden NodeB:** Preserves completed deliveries, tracks held and distinct object instances, and requires observed evidence before marking transformations complete.
- **Output Node:** Uses current state and action history to avoid cycles and unnecessary search or actions.

By FH4, all 40 failures reach the step limit rather than terminating with invalid actions. Ten cases make partial pickup, transformation, or placement progress, while the other 30 perform valid but overly long searches. Thus, training reduces state and action errors but exposes search efficiency as the next bottleneck.

FH4 (38.10%) → FH5 (76.19%). This stage yields the largest training and validation gains. All 40 FH4 failures reach the step limit, and 16 cases repeatedly search locations of the same kind instead of trying another plausible source. Updates in FH5 include:

- **Input Node:** Learns source-family priors for common target objects and switches families after repeated misses.
- **Hidden NodeA:** Detects search stagnation and proposes an unvisited location from a different plausible family.
- **Hidden NodeB:** Refreshes its plan from the latest visible objects and visited locations, prioritizing visible targets and avoiding unsupported locations.
- **Output Node:** Remains unchanged.

In the next 35-case failure batch, no trajectory visits three or more distinct locations from one family consecutively before switching families. This greatly improves the training/validation performance.

FH5 (76.19%) → FH7 (83.33%). Performance improves from FH5 to FH6 and then saturates, with no further validation gain at FH7. Remaining failures include leaving plausible source locations too early, attempting

remote examination, repeating stale destinations, treating proposed actions as completed, and selecting the same object instance again. Updates through FH7 include:

- **Input Node:** Prioritizes unvisited sources and stops repeating ignored destinations.
- **Hidden NodeA:** Distinguishes proposed actions from executed actions, and produces feasible action analysis.
- **Hidden NodeB:** Tracks completed deliveries and individual object instances, avoids stale search locations, and includes required steps such as opening an appliance before using it.
- **Output Node:** Uses NodeA’s feasibility analysis to correct NodeB’s plan when a prerequisite is missing.

These changes refine search ordering and multi-step execution, producing the smaller improvement from FH5 to FH6, while FH7 maintains the same validation pass rate.

Overall, the evolution process preserves the FH0 node roles while adding complementary ALFWorld-specific mechanisms inside each node forward harness. The final harnesses contain neither case IDs nor stored case-specific action sequences. Moreover, the best evolved checkpoint improves independent test performance from 16.51% at FH0 to 69.72% at FH6 (Table 4). These observations support the interpretation that HarnessNet learns reusable interaction rules rather than memorizing training answers.

5 Conclusion

We propose HarnessNet and its fundamental unit, Agentron, for feedback-driven hierarchical joint harness evolution in multi-agent systems. Experiments and studies shows: **(1)** training them is feasible and improves the task performance; **(2)** evolved harnesses capabilities are transferrable across foundation models; and **(3)** hierarchical evolution reduces per-optimizer-call workload and context burdens while keeping comparable training effects.

References

- [1] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/36b7acf6f6010652b3f2a433774a66fe-Abstract-Conference.html.
- [2] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.
- [3] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):195:1–195:35, 2023. doi: 10.1145/3560815.
- [4] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, Hyojung Han, Sevin Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynter, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2024. URL <https://arxiv.org/abs/2406.06608>.
- [5] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models, 2025. URL <https://arxiv.org/abs/2507.13334>.
- [6] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield. Effective context engineering for AI agents. Anthropic Engineering Blog, September 2025. URL <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>. Published September 29, 2025. Accessed October 3, 2026.
- [7] Justin Young. Effective harnesses for long-running agents. Anthropic Engineering Blog, November 2025. URL <https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents>. Published November 26, 2025. Accessed October 3, 2026.
- [8] Ryan Lopopolo. Harness engineering: Leveraging Codex in an agent-first world. OpenAI Engineering Blog, February 2026. URL <https://openai.com/index/harness-engineering/>. Published February 11, 2026. Accessed October 3, 2026.
- [9] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.
- [10] Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents. *arXiv preprint arXiv:2605.22166*, 2026.
- [11] Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.
- [12] Seth Karten, Joel Zhang, Tersoo Upaa, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.
- [13] Zewen Liu, Zhan Shi, Yisi Sang, Bing He, Minhua Lin, Tianxin Wei, Dakuo Wang, Benoit Dumoulin, Wei Jin, and Hanqing Lu. Adaptive auto-harness: Sustained self-improvement for agentic system deployment on open-ended task streams. *arXiv preprint arXiv:2606.01770*, 2026.
- [14] Wen Jiang, Mingmin Chu, Yimeng Tian, Qianxin Zhang, Hao-fei Yang, Rui Yang, Yang Liu, Tao Lv, and Fangming Li. HarnessEvolve: Learning from reference trajectories for reliable agent self-evolution, 2026. URL <https://arxiv.org/abs/2609.00829>.
- [15] Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Babu Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. Context length alone hurts LLM performance despite perfect retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 23281–23298. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.1264. URL <https://aclanthology.org/2025.findings-emnlp.1264/>.
- [16] Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management, 2025. URL <https://arxiv.org/abs/2508.21433>.
- [17] Zhezhen Hao, Tianfu Wang, Huanshuo Dong, Ziyang Liu, Hong Wang, Xiankun Lin, Qiang Lin, Can

- Wang, Hande Dong, and Jiawei Chen. Evolve as a team: Collaborative self-evolution for llm-based multi-agent systems, 2026. URL <https://arxiv.org/abs/2605.29790>.
- [18] Eugene Bagdasarian and Marco Gruteser. Open and emergent problems in agentic privacy and security: A contextual angle. Google Research Blog, October 2026. URL <https://research.google/blog/open-and-emergent-problems-in-agentic-privacy-and-security-a-contextual-angle/>. Published October 5, 2026. Accessed October 7, 2026.
- [19] Gurusha Juneja, Jayanth Naga Sai Pasupulati, Alon Albalak, Wenyue Hua, and William Yang Wang. MAGPIE: A benchmark for Multi-AGent contextual PrIvacy evaluation, 2025. URL <https://arxiv.org/abs/2510.15186>.
- [20] Faouzi El Yagoubi, Godwin Badu-Marfo, and Ranwa Al Mallah. AgentLeak: A benchmark for internal-channel privacy leakage in multi-agent LLM systems, 2026. URL <https://arxiv.org/abs/2602.11510>.
- [21] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuan-yu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046, 2024.
- [22] Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with “gradient descent” and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7957–7968. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.494. URL <https://aclanthology.org/2023.emnlp-main.494/>.
- [23] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://arxiv.org/abs/2309.03409>.
- [24] Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. PromptAgent: Strategic planning with language models enables expert-level prompt optimization. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=22pyNMuIoa>.
- [25] Lakshya A. Agrawal, Shangyin Tan, Dilara Soyulu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.
- [26] Guanbao Liang, Yuanchen Bei, Sheng Zhou, Yuheng Qin, Huan Zhou, Bingxin Jia, Bin Li, and Jiajun Bu. Generalizable self-evolving memory for automatic prompt optimization, 2026. URL <https://arxiv.org/abs/2603.21520>.
- [27] Rulin Shao, Shannon Zejiang Shen, Junjie Oscar Yin, Yuetai Li, Minheng Wang, Hamish Ivison, Radha Poovendran, Nathan Lambert, Teng Xiao, Mike Lewis, Wen tau Yih, Luke Zettlemoyer, and Pang Wei Koh. Context language models, 2026. URL <https://arxiv.org/abs/2609.37725>.
- [28] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.
- [29] Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. ACON: Optimizing context compression for long-horizon LLM agents. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 55321–55375. PMLR, 2026. URL <https://proceedings.mlr.press/v306/kang26b.html>.
- [30] Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32779–32798. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1575. URL <https://aclanthology.org/2025.acl-long.1575/>.
- [31] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- [32] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural*

- Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.
- [33] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheat-sheet: Test-time learning with adaptive memory, 2025. URL <https://arxiv.org/abs/2504.07952>.
 - [34] Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. AutoSkill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026.
 - [35] Huawei Lin, Peng Li, Jie Song, Fuxin Jiang, and Tieying Zhang. MUSE-Autoskill: Self-evolving agents via skill creation, memory, management, and evaluation. *arXiv preprint arXiv:2605.27366*, 2026.
 - [36] Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. CoEvoSkills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026.
 - [37] Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. SkillGrad: Optimizing agent skills like gradient descent. *arXiv preprint arXiv:2605.27760*, 2026.
 - [38] Zefeng Wang, Minxi Yan, Jinhe Bi, Sikuan Yan, Volker Tresp, and Yunpu Ma. MetaSkill-Evolve: Recursive self-improvement of LLM agents via two-timescale meta-skill evolution. *arXiv preprint arXiv:2607.05297*, 2026.
 - [39] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.
 - [40] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.
 - [41] Qianshu Cai, Yonggang Zhang, Xianzhang Jia, Huajiang Zheng, Wei Xue, Jun Song, Xinmei Tian, and Yike Guo. MOSS: Self-evolution through source-level rewriting in autonomous agent systems. *arXiv preprint arXiv:2605.22794*, 2026.
 - [42] Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026.
 - [43] Prithwish Jana, Mononito Goswami, Hao Liu, Xinyu Li, Langlin Huang, Zhehui Huang, Zhishen Huang, Patrick Blöbaum, Anoop Deoras, Purak Jain, Nikos Kanakaris, and Sahika Genc. Milo: Automated harness discovery via orchestrated multi-agent evolution, 2026. URL <https://arxiv.org/abs/2609.38349>.
 - [44] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic LLM-powered agent network for task-oriented agent collaboration. *arXiv preprint arXiv:2310.02170*, 2023.
 - [45] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.
 - [46] Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025.
 - [47] Xiaowen Ma, Yunpu Ma, Chenyang Lin, Sikuan Yan, Jinhe Bi, Zixuan Cao, Yijun Tian, Volker Tresp, and Hinrich Schuetze. Self-evolving multi-agent systems via textual backpropagation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Findings of the Association for Computational Linguistics: ACL 2026*, pages 9918–9951, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.483. URL <https://aclanthology.org/2026.findings-acl.483/>.
 - [48] Jun Nie, Yonggang Zhang, Qianshu Cai, Yiuming Cheung, Xinmei Tian, and Bo Han. Evolvernet: Collaborative harness evolution for agent self-improvement, 2026. URL <https://arxiv.org/abs/2608.04968>.
 - [49] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arik. Multi-agent design: Optimizing agents with better prompts and topologies, 2026. URL <https://arxiv.org/abs/2502.02533>.
 - [50] Zhexuan Wang, Xuebo Liu, Li Wang, Zifei Shan, Yutong Wang, Zhenxi Song, and Min Zhang. MASPO: Joint prompt optimization for LLM-based multi-agent systems. In Tong Zhang, Miroslav Dudík, Martin Jaggi, Alekh Agarwal, Sharon Li, Dale Schuurmans, Jerry Zhu, Felix Berkenkamp, Hanze Dong, and Alberto Bietti, editors, *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 128503–128534. PMLR, 06–11 Jul 2026. URL <https://proceedings.mlr.press/v306/wang26ew.html>.

- [51] Jaewon Chu, Jinwoo Seo, Jaewon Cho, Jeehye Na, Yunyang Xiong, Youngdae Kim, and Hyunwoo J. Kim. Agentgrad: Intervention-guided prompt optimization for multi agent systems, 2026. URL <https://arxiv.org/abs/2609.08572>.
- [52] Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. EvoAgentX: An automated framework for evolving agentic workflows. *arXiv preprint arXiv:2507.03616*, 2025.
- [53] Shan He, Runze Wang, Zhuoyun Du, Huiyu Bai, Zouying Cao, Yu Cheng, and Bo Zheng. Learning to evolve: A self-improving framework for multi-agent systems via textual parameter graph optimization. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Findings of the Association for Computational Linguistics: ACL 2026*, pages 30719–30732, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1534. URL <https://aclanthology.org/2026.findings-acl.1534/>.
- [54] Wentao Zhang, Zhe Zhao, Haibin Wen, Yingcheng Wu, Cankun Guo, Ming Yin, and Bo An. Auto-genesis: A self-evolving agent protocol, 2026. URL <https://arxiv.org/abs/2604.15034>.
- [55] Kunfeng Chen, Qihuang Zhong, Juhua Liu, and Bo Du. SkillCAT: Contrastive, assessment-augmented and topology-aware skill self-evolution for LLM agents, 2026. URL <https://arxiv.org/abs/2606.13317>.
- [56] Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, Rui Lu, Hongning Wang, Jie Tang, and Yuxiao Dong. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework, 2025. URL <https://arxiv.org/abs/2510.04206>.
- [57] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL <https://qwen.ai/blog?id=qwen3.5>.
- [58] Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.
- [59] Ornith Team. Ornith-1.5: From self-scaffolding to self-improvement, 2026. URL https://ornith.ai/ornith_1_5.html.
- [60] Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL <https://qwen.ai/blog?id=qwen3.8>.
- [61] Ollama. Ollama. <https://github.com/ollama/ollama>. Software. Accessed: 2026-10-02.
- [62] OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. <https://openai.com/index/gpt-5-6/>, 2026. Accessed: 2026-09-30.

HarnessNet: Supplementary Material

Appendix Navigation

Appendix A Initializer Implementation Details

Appendix B Forward-Model Configurations

Appendix C Training Experiments

Appendix D Migration Study

Appendix E Hierarchical vs. Global Optimizer Comparison Study Details

Appendix A Initializer Implementation Details

The HarnessNet initializer constructs the trainable forward-harness baseline (FH0) before any benchmark case is used for optimization. It is implemented as an auditable controller protocol with three model-facing stages: PLAN, BUILD, and REPAIR. The controller, rather than the model, owns job publication, state transitions, validation, and final acceptance. The wording of the prompt templates reproduced below is copied verbatim from the implementation at repository revision `cc3cf57`; line wrapping and continuation indentation are adjusted only for typesetting.

Subsection navigation

- A.1 Frozen inputs and visibility boundary
 - A.2 Benchmark profiles exposed to the initializer
 - A.3 Three-stage protocol
 - A.4 Execution and acceptance separation
 - A.5 PLAN prompt template
 - A.6 BUILD prompt template
 - A.7 Canonical harness template
 - A.8 REPAIR prompt template
-

A.1 Frozen inputs and visibility boundary

Before initialization starts, the controller freezes the public benchmark profile, LLM-backend profile, topology, harness contract, canonical template manifest, SDK guide, and model controls. The public benchmark profile may describe the task family, action or output schema, scoring semantics, and available tools, but it does not expose selected cases, case identifiers, observations, answers, or case-level statistics. This boundary prevents the initializer from encoding benchmark instances into FH0.

Each Agent is materialized as an independent Git repository from the same canonical harness template. The topology fixes node identities, layer positions, edges, and first- and last-boundary roles. The stable Contract 1.2 runtime supplies one backend call per node invocation, preserves the complete case-local native-message history, structurally projects a single native tool call when present, and propagates execution errors through a stable error envelope.

A.2 Benchmark profiles exposed to the initializer

The benchmark profile is the initializer’s complete benchmark-specific view. It states the task family, observation transport, authorized action tools, network output contract, termination conditions, scoring semantics, and a synthetic contract probe. It deliberately contains no selected case data. Table A1 summarizes the two profiles used in the reported experiments.

Table A1: Public benchmark profiles supplied to the HarnessNet initializer.

Profile field	ALFWorld	WebShop
Objective	Complete a household manipulation goal through a sequence of environment actions.	Find and purchase a product satisfying the stated attributes and budget.
Observation	Raw text containing the task goal, environment feedback, and current admissible actions.	Raw text containing the shopping instruction or current page and the currently available actions.
Action interface	Exactly one required <code>take.action</code> call with a nonempty action string selected from the admissible actions.	Exactly one published <code>search.action(keywords)</code> or <code>click.action(value)</code> call valid for the current page.
Last-node output	One admissible action string; the adapter wraps it as <code>take.action</code> .	One object containing <code>tool_name</code> and <code>arguments</code> ; the adapter validates and submits the native tool-call envelope.
Step limit	15	20
Primary metric	Success rate, with success iff <code>result.result == 1</code> .	Mean final reward, with per-case reward in $[0, 1]$ and full success at <code>result.result == 1.0</code> .
Train / test task	<code>alfworld-env.train</code> / <code>alfworld-std</code> on the <code>new-std</code> test split.	<code>webshop-env.train</code> / <code>webshop-std</code> on the <code>standard</code> test split.
Initializer visibility	Public task, interaction, tool, scoring, and six category semantics; no case-data access.	Public task family, interaction, tool, scoring, topology, contract, template, and backend controls; no case-data access.

The two selected ALFWorld experiment groups contain byte-identical public profiles.

A.3 Three-stage protocol

PLAN. The controller publishes one immutable planning job for the complete network. The planner reads only registered public inputs and produces a schema-valid, function-level initialization plan. For every node, the plan specifies its role, boundary behavior, prompt-level inductive bias, authorized tool exposure, error behavior, and acceptance checks. It cannot inspect benchmark cases, modify an Agent repository, alter the frozen topology, or request changes to the fixed runtime mechanisms.

BUILD. After the plan is accepted, the controller materializes one repository per node and publishes an independent build job for each node. A builder may specialize only `ROLE.md`, `prompts/system.md`, `src/harness_agent/tools.py`, `tests/test_smoke.py`, and the allowlisted role metadata. In particular, the fixed `strategy.py`, `memory.py`, and `parsing.py` files remain unchanged. Each successful build creates at most one commit and runs the job-published checks from their exact working directories. The controller then performs its own schema, repository, provenance, test, and contract validation.

REPAIR. If network validation reports a node-local failure, the controller publishes a repair job with the structured failed checks, the frozen validation report, and the corresponding probe requests and responses. Repair preserves the original node specification and may touch only the same builder-owned paths. It may not add dependencies, weaken tests, modify the fixed runtime, invent fallback actions, or convert execution failures into successful outputs. After each repair round, the controller revalidates the network. Initialization becomes **ready** when all checks pass; after at most three repair rounds, remaining failures produce a terminal **failed** result.

A.4 Execution and acceptance separation

Every initializer job is executed by a fresh Codex CLI process with frozen model controls. The captured templates require the `gpt-5.6-sol` model, high reasoning effort, and standard mode. Model output is not itself an accepted checkpoint: the controller independently verifies the result envelope, registered artifacts, allowed file changes, commit shape, test results, and end-to-end contract probes before recording the node commit in FH0. Raw execution traces and the controller-derived acceptance evidence are retained separately.

A.5 PLAN prompt template

Listing A1: Initializer PLAN prompt template.

```
# HarnessNet initializer planner

Execute only when the immutable job manifest fixes 'gpt-5.6-sol', 'reasoning_effort=high', and 'mode=standard'.
Treat the manifest's registered read and write paths as the complete authority.

Design one function-level initialization plan for the supplied benchmark profile, backend profile, topology, stable
harness contract, canonical template manifest, and plan schema. Do not read benchmark cases or modify an Agent
repository. Describe every required inductive-bias field for every topology node. Persist only the schema-valid plan
and its differentiation summary at the requested paths. For a single-layer topology, treat the sole node as both the
first and last HarnessNet boundary: it consumes the current environment observation, replays the complete case-local
history from case-local memory, and returns one exact admissible action in the adapter-accepted content form. The
fixed runtime also appends audit records to the supplied memory path; use every ordered runtime record as
interaction history with no truncation, no summary, and no separate harness-owned compact state. Preserve every
native assistant message recorded by the Contract 1.2 template. When the public profile defines an adapter-owned
'take_action' or other action tool, the final-boundary node must register the matching backend-native function tools
through 'tools.py' and the fixed SDK call. Non-final nodes must not register environment action tools. For every
node, freeze the template's one-call native-message strategy. The plan must not introduce benchmark-solving
heuristics, semantic repair, model retry, fallback actions, task-state summaries, or example-specific guidance. The
builder may specialize only 'ROLE.md', 'prompts/system.md', 'src/harness_agent/tools.py', 'tests/test_smoke.py', and
allowed role metadata. It must not modify 'strategy.py', 'memory.py', or 'parsing.py'. Those plan fields must
describe inherited template behavior, not request new implementations or stronger checks. Do not bypass these fixed
files through tools.py or monkeypatching. The canonical parser preserves native assistant messages, structurally
projects one native tool call to downstream content, and otherwise passes through nonempty native text. It rejects
malformed tool-call structure and absent usable content; it does not validate benchmark tool-name membership or
argument schemas. The benchmark adapter owns that action validation. Distinguish the model's instructed one-tool
action format from harness-enforced rejection: register the exact public tool schemas and instruct the model to
follow them, but do not require the builder to add unknown-tool, schema-argument, or nonempty-prose rejection to the
harness. Keep acceptance checks within the fixed template's actual guarantees and the permitted role/tool
specialization, including backend/upstream error propagation. Copy the frozen topology into the plan exactly,
including its layer count, node placement, edges, and boundary IDs. A node serving more than one boundary role still
appears only at its frozen position: never duplicate a node or add a layer to express those roles. In particular, a
one-node topology remains one layer whose first and last Agent IDs are the same. For every node, keep model-output
degradation separate from execution failure: 'error_degradation_policy', 'prompt_guidance', and 'acceptance_checks'
must require backend exceptions and upstream 'status=error' inputs to propagate through the stable error envelope
with 'status=error' and a nonzero harness exit. They must never describe an 'ok' fallback for those execution
```

errors. Empty, malformed, or otherwise unusable successful model output must fail under Contract 1.2; FHO must not invent content or return a degraded 'status=ok' fallback. Copy 'payload.plan_provenance' verbatim into the initialization plan's 'provenance' object; never derive or modify any provenance field yourself. Also copy the frozen input refs exactly: set plan 'benchmark_profile' to 'payload.benchmark_profile_ref', plan 'llm_backend_profile' to 'payload.llm_backend_profile_ref', and plan 'template_ref' to 'payload.template_manifest_ref'. Do not replace these refs with profile summaries.

The public profile may describe topology, Agent interface, output/action schema, scoring semantics, SDK/tool access, backend controls, and the general task family. It must not expose or lead you to discover case paths, case IDs, case counts, case statistics, case observations, or case answers. Do not inspect conversation context or filesystem paths that are absent from the job manifest.

Run every 'validation.checks' entry from its exact published 'cwd'. These are job-agent checks only; controller-owned acceptance independently enforces schema, provenance, repository, pytest, checkpoint, and transaction contracts.

Use every path in 'expected_result.artifact_paths' verbatim for its named result role, including each validation check ID; never substitute another writable path. Write the schema-valid result plus the required visible trace, tool events, stdout/stderr, usage, summaries, and artifact inventory only to those mapped output paths. If the job or model controls differ, write a schema-valid failed result instead of widening access or inventing inputs.

A.6 BUILD prompt template

Listing A2: Initializer BUILD prompt template.

```
# HarnessNet initializer builder
```

Execute only when the immutable job manifest fixes 'gpt-5.6-sol', 'reasoning_effort=high', and 'mode=standard'.
Treat its registered read/write paths, validation commands, and result location as the complete authority.

Implement exactly one node specification in the materialized repository. Use the frozen initialization plan, fixed SDK/backend contract, and complete mutable path and manifest-field allowlists. Verify that repository HEAD equals the job's 'materialization_commit' before editing. Do not change identity, endpoint, SDK/interface versions, dependencies, or runtime code. Preserve the stable execution-error contract exactly: backend exceptions and upstream 'status=error' inputs must produce a stable 'status=error' response and a nonzero harness exit; never catch them into a degraded 'status=ok' response. Unusable output from an otherwise successful model call must fail under Contract 1.2; do not invent content or use degraded 'ok' behavior. When the node uses the supplied case-local memory path, account for the fact that the fixed runtime also appends audit records there: preserve the complete case-local history with no truncation and no summary, including every recorded native assistant message. Configure backend-native function tools in 'tools.py' only when this node owns the final action boundary, using the exact public-profile names and argument schemas. You may specialize only 'ROLE.md', 'prompts/system.md', 'src/harness_agent/tools.py', 'tests/test_smoke.py', and the allowed role metadata. You must not modify 'strategy.py'; you must not modify 'memory.py'; you must not modify 'parsing.py'. Do not add benchmark-solving heuristics, compact task state, semantic repair, retries, or fallback actions. Include focused smoke tests for the node's role and backend-error propagation. Run every 'validation.checks' entry from its exact published 'cwd'. These are job-agent checks only; controller-owned acceptance independently enforces schema, provenance, repository, pytest, checkpoint, and transaction contracts. Use every path in 'expected_result.artifact_paths' verbatim for its named result role, including each validation check ID; never substitute another writable path. Write the mapped repository diff from the exact stdout of 'git diff <payload.materialization_commit>..HEAD --' in the target repository, substituting the exact published commit value; do not add '--full-index', '--binary', or other formatting flags. Create at most one initializer commit, and write the schema-valid result plus visible trace, tool events, stdout/stderr, usage, diff, summary, and artifact inventory only at those mapped output paths. If any precondition fails, write a schema-valid failed result; do not repair or widen the job yourself.

A.7 Canonical harness template

Every node repository starts from the same canonical Contract 1.2 harness template. Listing A3 shows the complete template tree before node identity is materialized. The controller subsequently fills the topology-derived identity fields in `harness.json`; it does not add benchmark instances or node-specific solving logic.

Listing A3: File structure of the canonical per-node harness template.

```
agent_repo/
|-- harness.json
|-- pyproject.toml
|-- README.md
|-- ROLE.md
|-- prompts/
|   |-- forward.md
|   |-- system.md
|-- src/
|   |-- harness_agent/
|   |   |-- __init__.py
|   |   |-- endpoint.py
|   |   |-- memory.py
|   |   |-- parsing.py
|   |   |-- strategy.py
|   |-- tools.py
```

```
-- tests/
-- test_smoke.py
```

Table A2 summarizes the responsibility and initializer-time ownership of every file. Here, “fixed” means fixed during FH0 initialization; it does not redefine the separate optimizer-time allowlist.

Table A2: Files in the canonical harness template and their initializer-time roles.

File	Initializer treatment	Purpose
<code>harness.json</code>	Controller-owned; scoped metadata editable	Declares node identity, topology neighbors, entry point, contract versions, template revision, and path/field allowlists. The materializer writes the identity fields; the builder may change only the allowed role metadata.
<code>pyproject.toml</code>	Fixed	Defines the self-contained Python package, supported interpreter version, dependencies, and test discovery settings.
<code>README.md</code>	Fixed	Records the neutral Contract 1.2 baseline, invocation command, and editing boundary for the repository.
<code>ROLE.md</code>	Builder-owned	Provides the auditable, node-specific role description derived from the accepted initialization plan.
<code>prompts/system.md</code>	Builder-owned	Supplies the node’s runtime system instruction and its plan-derived inductive bias.
<code>prompts/forward.md</code>	Fixed	Retains a future trainable prompt extension point; the neutral FH0 strategy does not inject this file into the model request.
<code>src/harness_agent/</code> <code>__init__.py</code>	Fixed	Marks the executable harness package and states its neutral baseline role.
<code>src/harness_agent/</code> <code>entrypoint.py</code>	Fixed	Implements the subprocess request/response boundary, constructs the SDK client from the supplied backend profile, and preserves the stable error exit contract.
<code>src/harness_agent/</code> <code>memory.py</code>	Fixed mechanism	Loads the complete case-local interaction history in original order without truncation or summarization.
<code>src/harness_agent/</code> <code>parsing.py</code>	Fixed mechanism	Projects native assistant content or one native tool call into the stable downstream envelope and rejects unusable model output.
<code>src/harness_agent/</code> <code>strategy.py</code>	Fixed mechanism	Reconstructs messages from the full local history, appends current upstream inputs, makes one backend call, and delegates output projection.
<code>src/harness_agent/</code> <code>tools.py</code>	Builder-owned	Registers backend-native function schemas only when the node owns the public benchmark action boundary.
<code>tests/test_smoke.py</code>	Builder-owned	Contains focused node-role, output-contract, and error-propagation checks executed before controller acceptance.

The template enters the initializer workflow at five points. First, the controller prepares and freezes a clean `template-baseline` root commit, including its content digest and manifest. Second, PLAN reads that public template contract and proposes only node-specific roles, prompt biases, tools, and acceptance checks; it does not edit a repository. Third, after plan acceptance, the deterministic materializer creates an independent Git repository for each node, fills its topology identity in `harness.json`, and records a materialization commit. Fourth, BUILD specializes only `ROLE.md`, `prompts/system.md`, `src/harness_agent/tools.py`, `tests/test_smoke.py`, and the allowlisted role metadata, producing at most one initializer commit per node. Finally, the controller checks required files, fixed mechanisms, manifest fields, repository cleanliness, imports, tests, and synthetic network probes. REPAIR receives the same template-derived boundary and may correct only builder-owned files; after acceptance, the validated node commits jointly constitute FH0.

A.8 REPAIR prompt template

Listing A4: Initializer REPAIR prompt template.

```
# HarnessNet initializer repair

Execute only when the immutable job manifest fixes 'gpt-5.6-sol', 'reasoning_effort=high', and 'mode=standard'.
Require 'repair_limit=3' and a 'repair_round' of 1, 2, or 3; reject any job outside that frozen limit.

Repair only the structured failed checks for the supplied node and current commit. Before changing the repository,
read 'payload.benchmark_profile_ref', the frozen 'payload.validation_report_ref', and every artifact in
'payload.probe_diagnostic_refs'. Treat the registered probe requests and responses as the exact observed wire
evidence for this repair; compare them with the public benchmark profile instead of guessing from a check ID or
summary message alone. Keep the original node specification and network inductive biases fixed. Do not add
dependencies or weaken tests. Repair may change only builder-owned 'ROLE.md', 'prompts/system.md',
'src/harness_agent/tools.py', role metadata, and focused smoke tests. It must not modify the fixed Contract 1.2
'strategy.py', 'memory.py', or 'parsing.py' mechanisms. A failure located in those fixed mechanisms is out of scope
for per-node repair. Never convert backend exceptions or upstream 'status=error' inputs into a degraded 'status=ok'
response: the stable contract requires an error envelope and nonzero harness exit for execution errors. Do not
convert unusable model output into invented content or a degraded successful fallback. Use the prior repair record
when supplied and run every 'validation.checks' entry from its exact published 'cwd'. These are job-agent checks
```

only; controller-owned acceptance independently enforces schema, provenance, repository, pytest, checkpoint, and transaction contracts. Then use every path in 'expected_result.artifact_paths' verbatim for its named result role, including each validation check ID; never substitute another writable path. Write the mapped repository diff from the exact stdout of 'git diff <payload.current_commit>..HEAD --' in the target repository, substituting the exact published commit value; do not add '--full-index', '--binary', or other formatting flags. Create at most one repair commit, and write the schema-valid result plus visible trace, tool events, stdout/stderr, usage, diff, summary, and artifact inventory only at those mapped paths. If the supplied failure cannot be repaired within the allowlist because the frozen probe evidence locates it in another node, do not manufacture an empty commit. Write a schema-valid 'status=failed', 'outcome=unchanged' result with the input commit unchanged, no repository changes, empty 'addressed_check_ids', the original failed IDs in 'remaining_failed_check_ids', and one non-retriable 'repair_out_of_scope' error. The controller may accept only that exact no-change shape so the network can be revalidated after the responsible nodes finish.

Appendix B Forward-Model Configurations

Table B1 reports the frozen model controls used across the experiments. Local registry identifiers are intentionally omitted. Parameters not listed in the frozen generation objects were delegated to the corresponding model and Ollama defaults.

Table B1: Ollama model and execution controls used in the selected HarnessNet runs. The two Qwen3.8-27b limits are the only generation-control differences among the four configurations.

Parameter	Qwen3.8-27b	Qwen3.5-9b	Gemma4-8b-e4b	Ornith-1.5-9B
Resource	Hugging Face	Hugging Face	Hugging Face	Hugging Face
Quantization	Q4	Q4	Q8	Q4
Backend	Ollama	Ollama	Ollama	Ollama
num_ctx	32,768	20,480	20,480	20,480
num_predict	4,096	2,048	2,048	2,048
temperature	0	0	0	0
top_p	0.95	0.95	0.95	0.95
top_k	20	20	20	20
min_p	0	0	0	0
presence_penalty	1.5	1.5	1.5	1.5
repeat_penalty	1	1	1	1
Thinking	false	false	false	false
Max calls per step	1	1	1	1
Max concurrency	1	1	1	1
Timeout (s)	300	300	300	300
Timeout MaxRetry	2	2	2	2
Retry backoff (s)	1, 2	1, 2	1, 2	1, 2

Appendix C Training Experiments

This appendix reports the training curves and held-out test results for the three evaluated topologies on ALFWorld and WebShop. For ALFWorld, scores are pass rates; for WebShop, scores are mean final rewards (AvgScore). An asterisk in the *Selected* column marks the trained checkpoint chosen for selected-FH testing. The *Model-only* row denotes the harness-free baseline. Held-out test scores are shown on the FH0 row and on the selected checkpoint row; the remaining checkpoint test cells are intentionally blank.

Subsection navigation

C.1	ALFWorld-Agentron-Training
C.2	WebShop-Agentron-Training
C.3	ALFWorld-2Node HarnessNet-Training
C.4	WebShop-2Node HarnessNet-Training
C.5	ALFWorld-4Node HarnessNet-Training
C.6	WebShop-4Node HarnessNet-Training

C.1 ALFWorld-Agentron-Training

Table C1: Key metadata for ALFWorld-Agentron training.

Field	Value
Forward model	Qwen3.8-27b
Benchmark domain	ALFWorld
Topology	Agentron [1]
Train dataset	alfworld-env_train ($n = 168$)
Validation dataset	alfworld-env_train ($n = 42$)
Test dataset	alfworld-std, new_std split ($n = 109$)

Table C2: ALFWorld-Agentron train, validation, and held-out test records.

Checkpoint	Train Score	Validation Score	Selected	Test Score
Model-only				56.88%
FH0	59.18%	40.48%		22.02%
FH1	39.39%	52.38%		
FH2	55.56%	66.67%		
FH3	58.54%	66.67%		
FH4	76.47%	83.33%		
FH5	83.13%	85.71%		
FH6	87.58%	80.95%	*	77.98%
FH7	88.10%	78.57%		

C.2 WebShop-Agentron-Training

Table C3: Key metadata for WebShop-Agentron training.

Field	Value
Forward model	Qwen3.8-27b
Benchmark domain	WebShop
Topology	Agentron [1]
Train dataset	webshop-env_train ($n = 320$)
Validation dataset	webshop-env_train ($n = 80$)
Test dataset	webshop-std, standard split ($n = 200$)

Table C4: WebShop-Agentron train, validation, and held-out test records.

Checkpoint	Train Score	Validation Score	Selected	Test Score
Model-only				0.5670
FH0	0.4789	0.5357		0.5559
FH1	0.5047	0.6384		
FH2	0.6691	0.6615		
FH3	0.7078	0.7223		
FH4	0.6863	0.7010		
FH5	0.7250	0.6583		
FH6	0.6397	0.6935		
FH7	0.7087	0.6867		
FH8	0.6699	0.7075		
FH9	0.7153	0.7131		
FH10	0.7115	0.7222	*	0.6950

C.3 ALFWorld-2Node HarnessNet-Training

Table C5: Key metadata for ALFWorld-2Node HarnessNet training.

Field	Value
Forward model	Qwen3.8-27b
Benchmark domain	ALFWorld
Topology	2Node HarnessNet [1, 1]
Train dataset	alfworld-env_train ($n = 168$)
Validation dataset	alfworld-env_train ($n = 42$)
Test dataset	alfworld-std, new_std split ($n = 109$)

Table C6: ALFWorld-2Node HarnessNet train, validation, and held-out test records.

Checkpoint	Train Score	Validation Score	Selected	Test Score
Model-only				56.88%
FH0	44.44%	57.14%		62.39%
FH1	60.42%	71.43%		
FH2	45.21%	47.62%		
FH3	89.47%	85.71%		
FH4	91.67%	85.71%		
FH5	92.26%	85.71%	*	85.32%

C.4 WebShop-2Node HarnessNet-Training

Table C7: Key metadata for WebShop-2Node HarnessNet training.

Field	Value
Forward model	Qwen3.8-27b
Benchmark domain	WebShop
Topology	2Node HarnessNet [1, 1]
Train dataset	webshop-env_train ($n = 320$)
Validation dataset	webshop-env_train ($n = 80$)
Test dataset	webshop-std, standard split ($n = 200$)

Table C8: WebShop-2Node HarnessNet train, validation, and held-out test records.

Checkpoint	Train Score	Validation Score	Selected	Test Score
Model-only				0.5670
FH0	0.4807	0.5852		0.5523
FH1	0.6021	0.5998		
FH2	0.5814	0.6450		
FH3	0.5429	0.5885		
FH4	0.6282	0.6054		
FH5	0.6778	0.6094		
FH6	0.6437	0.6163	*	0.5892
FH7	0.5755	0.6444		

C.5 ALFWorld-4Node HarnessNet-Training

Table C9: Key metadata for ALFWorld-4Node HarnessNet training.

Field	Value
Forward model	Qwen3.5-9b
Benchmark domain	ALFWorld
Topology	4Node HarnessNet [1, 2, 1]
Train dataset	<code>alfworld-env.train</code> ($n = 168$)
Validation dataset	<code>alfworld-env.train</code> ($n = 42$)
Test dataset	<code>alfworld-std, new_std</code> split ($n = 109$)

Table C10: ALFWorld-4Node HarnessNet train, validation, and held-out test records.

Checkpoint	Train Score	Validation Score	Selected	Test Score
Model-only				11.93%
FH0	9.09%	19.05%		16.51%
FH1	34.43%	33.33%		
FH2	33.33%	28.57%		
FH3	39.39%	38.10%		
FH4	42.86%	38.10%		
FH5	79.17%	76.19%		
FH6	79.17%	83.33%	*	69.72%
FH7	85.71%	83.33%		

C.6 WebShop-4Node HarnessNet-Training

Table C11: Key metadata for WebShop-4Node HarnessNet training.

Field	Value
Forward model	Gemma4-8b-e4b
Benchmark domain	WebShop
Topology	4Node HarnessNet [1, 2, 1]
Train dataset	<code>webshop-env.train</code> ($n = 320$)
Validation dataset	<code>webshop-env.train</code> ($n = 80$)
Test dataset	<code>webshop-std, standard</code> split ($n = 200$)

Table C12: WebShop-4Node HarnessNet train, validation, and held-out test records.

Checkpoint	Train Score	Validation Score	Selected	Test Score
Model-only				0.4083
FH0	0.1210	0.0963		0.0624
FH1	0.4755	0.5298		
FH2	0.5984	0.6167	*	0.5875
FH3	0.4837	0.5379		
FH4	0.5710	0.4727		
FH5	0.6172	0.5408		
FH6	0.6085	0.5509		
FH7	0.5597	0.5435		

Appendix D Migration Study

The migration study evaluates whether the forward-harness checkpoints learned with one node-level language model remain useful when the underlying model is replaced. Table D1 summarizes the common evaluation setting.

Table D1: Key metadata for the migration study.

Field	Value
Forward models	Qwen3.5-9B; Ornith1.5-9B; gemma4-8B-E4B
Benchmark domain	ALFWorld
Topology	4Node HarnessNet [1, 2, 1]
Validation dataset	<code>alfworld-env.train</code> ($n = 42$)
Test dataset	<code>alfworld-std</code> , <code>new_std</code> split ($n = 109$)

Table D2: Migration-study validation and held-out test pass rates.

Checkpoint/Model	Qwen3.5-9B	Ornith1.5-9B	<code>gemma4-8B-E4B</code>
FH0	19.05%	16.67%	16.67%
FH1	33.33%	19.05%	16.67%
FH2	28.57%	23.81%	7.14%
FH3	38.10%	28.57%	30.95%
FH4	38.10%	33.33%	38.10%
FH5	76.19%	57.14%	78.57%
FH6	83.33%	61.90%	73.81%
FH7	83.33%	59.52%	80.95%
Model-only Test	11.93%	19.27%	8.26%
FH0 Test	16.51%	11.93%	9.17%
Selected FH	FH6	FH6	FH7
Selected FH Test	69.72%	49.54%	68.81%

Across all three model backends, later harness checkpoints substantially outperform FH0 on validation: the selected checkpoint reaches 83.33% for Qwen3.5-9B, 61.90% for Ornith1.5-9B, and 80.95% for `gemma4-8B-E4B`, compared with FH0 scores of 19.05%, 16.67%, and 16.67%, respectively. The selected checkpoints also retain non-trivial held-out test pass rates after migration (69.72%, 49.54%, and 68.81%). These results indicate that mechanisms and rules encoded in the node harnesses are not tied exclusively to the model used during training: after replacing the underlying node model, they continue to provide useful task-solving structure. The variation across models also shows that this transfer is partial rather than fully model-independent.

Appendix E Hierarchical vs. Global Optimizer Comparison Study Details

E.1 Training and Evaluation Results

Tables E1 and E2 compare checkpoint-level performance when the same 4Node HarnessNet topology is trained with either the hierarchical optimizer or the global optimizer. ALFWorld scores are pass rates, whereas WebShop scores are mean final rewards (AvgScore). *Model-only* denotes the harness-free baseline. The FH0 row combines the untrained harness’s train and validation scores with its no-training held-out test score. Blank test cells indicate that no test result was recorded for that checkpoint.

Table E1: ALFWorld train, validation, and held-out test pass rates under the hierarchical and global optimizers.

Checkpoint	Hierarchical Optimizer			Global Optimizer		
	Train Score	Validation Score	Test Score	Train Score	Validation Score	Test Score
Model-only			11.93%			11.93%
FH0	9.09%	19.05%	16.51%	9.09%	19.05%	16.51%
FH1	34.43%	33.33%		32.20%	28.57%	
FH2	33.33%	28.57%		23.08%	30.95%	
FH3	39.39%	38.10%		32.20%	45.24%	
FH4	42.86%	38.10%		34.43%	40.48%	
FH5	79.17%	76.19%		35.48%	59.52%	
FH6	79.17%	83.33%	69.72%	23.08%	38.10%	
FH7	85.71%	83.33%		47.37%	42.86%	44.96%

Table E2: WebShop train, validation, and held-out test AvgScore under the hierarchical and global optimizers.

Checkpoint	Hierarchical Optimizer			Global Optimizer		
	Train Score	Validation Score	Test Score	Train Score	Validation Score	Test Score
Model-only			0.4083			0.4083
FH0	0.1210	0.0963	0.0624	0.1210	0.0963	0.0624
FH1	0.4755	0.5298		0.3883	0.3802	
FH2	0.5984	0.6167	0.5875	0.4811	0.4730	
FH3	0.4837	0.5379		0.4837	0.5554	
FH4	0.5710	0.4727		0.6273	0.6418	0.6046
FH5	0.6172	0.5408		0.6017	0.5608	
FH6	0.6085	0.5509		0.6128	0.5894	
FH7	0.5597	0.5435		0.5739	0.5747	

E.2 Optimizer Cost Comparison

The cost tables report optimizer usage for each backward round. Hierarchical optimizer costs are listed separately for the four node-level jobs. The global optimizer values span those four rows because they correspond to one joint network-level job. Optimizer time is converted from milliseconds to seconds, and all reported values are rounded to the nearest integer.

Table E3: ALFWorld optimizer cost by backward round.

Backward	Node	Hierarchical Optimizer			Global Optimizer		
		Input Token	Output Token	Optimizer Time (s)	Input Token	Output Token	Optimizer Time (s)
1	12-last	4,557,381	31,320	1,427	10,467,460	40,902	1,719
	11-hidden.a	4,440,628	24,540	1,304			
	11-hidden.b	6,460,733	25,899	1,366			
	10-first	7,133,103	25,820	1,467			
2	12-last	7,027,571	46,165	1,912	13,542,625	42,660	2,056
	11-hidden.a	11,152,144	43,697	1,779			
	11-hidden.b	5,910,337	31,320	1,414			
	10-first	10,529,967	39,392	1,970			
3	12-last	5,084,140	32,673	1,355	13,214,124	71,496	2,290
	11-hidden.a	5,374,440	26,883	1,292			
	11-hidden.b	7,538,927	31,301	1,712			
	10-first	4,620,862	33,177	1,374			
4	12-last	3,348,882	23,800	940	8,173,538	48,924	1,452
	11-hidden.a	4,835,618	28,256	1,123			
	11-hidden.b	4,043,002	17,644	885			
	10-first	3,967,817	18,529	774			
5	12-last	2,288,059	14,149	576	12,491,469	43,673	1,873
	11-hidden.a	4,746,387	27,297	1,159			
	11-hidden.b	3,398,826	18,563	816			
	10-first	3,739,480	23,164	959			
6	12-last	3,641,378	17,629	729	7,927,776	44,207	1,436
	11-hidden.a	4,242,029	30,355	1,213			
	11-hidden.b	3,211,536	17,740	755			
	10-first	3,144,400	24,552	1,006			
7	12-last	3,189,454	18,387	750	7,999,809	40,685	1,425
	11-hidden.a	3,858,217	20,179	823			
	11-hidden.b	2,905,346	21,435	904			
	10-first	3,949,103	21,133	884			

Table E4: WebShop optimizer cost by backward round.

Backward	Node	Hierarchical Optimizer			Global Optimizer		
		Input Token	Output Token	Optimizer Time (s)	Input Token	Output Token	Optimizer Time (s)
1	12-last	3,971,824	18,694	707	20,197,746	44,424	2,378
	11-hidden.a	4,240,691	18,872	776			
	11-hidden.b	2,977,722	14,114	603			
	10-first	5,586,749	22,601	870			
2	12-last	2,659,254	15,242	627	9,745,839	25,382	1,373
	11-hidden.a	5,084,084	19,455	879			
	11-hidden.b	3,130,255	17,547	766			
	10-first	3,556,342	16,677	723			
3	12-last	2,553,968	18,545	722	6,451,227	25,342	1,221
	11-hidden.a	4,175,677	22,043	898			
	11-hidden.b	4,877,493	24,138	1,011			
	10-first	3,651,828	20,312	812			
4	12-last	3,173,158	15,827	664	11,285,091	44,076	2,214
	11-hidden.a	4,061,413	19,711	807			
	11-hidden.b	4,403,027	19,539	841			
	10-first	4,278,530	16,154	697			
5	12-last	3,725,265	19,996	779	4,928,535	20,146	1,233
	11-hidden.a	7,745,672	26,289	1,038			
	11-hidden.b	4,451,916	22,171	858			
	10-first	5,969,030	22,942	1,011			
6	12-last	2,935,015	14,280	625	6,849,504	20,070	1,303
	11-hidden.a	5,462,015	22,510	949			
	11-hidden.b	3,445,449	15,389	728			
	10-first	3,231,752	16,726	722			
7	12-last	5,280,300	18,280	781	4,216,686	15,110	1,074
	11-hidden.a	4,712,307	25,060	1,003			
	11-hidden.b	4,858,977	19,708	846			
	10-first	3,708,947	17,573	771			

E.3 Optimizer Prompt Templates

The templates below follow the latest semantic-report prompt wording in the Dev implementation for each optimizer architecture. To match the comparison-study jobs, their execution controls are fixed to `gpt-5.6-sol`, `reasoning_effort=high`, and `mode=standard`. Apart from these study-specific control values, the template text is reproduced verbatim; line wrapping and continuation indentation are adjusted only for typesetting. Per-job rendered prompts additionally append the adopted replay/history protocol and the job-specific identifier and manifest path.

Listing E1: Hierarchical optimizer semantic-report prompt template.

```
# Hierarchical node optimizer: semantic report

Execute only the frozen 'gpt-5.6-sol', 'reasoning_effort=high', 'mode=standard' job using
'delivery.protocol=semantic-report-v1'. Enforce 'strict_local_visibility': read the registered local failure
bundles, direct downstream feedback, environment feedback when provided, checkpoint and fixed contract. The rendered
prompt is read-only. Inspect 'case_memory_snapshot' when diagnosing history or memory-continuity failures; do not
inspect peer or non-neighbor private traces.

Before choosing a repair, systematically review the entire failure batch visible to this job. Check every case's
feedback and key behaviors first, then inspect relevant traces and implementation more deeply by failure pattern.
This does not require expanding every full model trace. Distinguish observed symptoms, root cause hypotheses and
supporting evidence; do not force all failures into one explanation or stop the survey at the first easy fix.

Prioritize one or more patterns by their coverage, severity, strength of causal evidence, likely transfer to similar
tasks and feasibility within the frozen budget. Frequency alone does not determine importance; no numerical scoring
formula is required. Aim to improve the harness's general capability on similar tasks, beyond correcting isolated
cases or passing selected replay samples. A single round need not resolve every pattern.

In the existing diagnosis, briefly summarize the main patterns, their supported scope and representative case IDs, why
the selected targets deserve priority, and the validation findings and remaining uncertainties. Use existing tests
and any adopted replay protocol to check the chosen solution. Distinguish observed local gains from expected broader
benefits and note untested patterns and potential regressions. Do not create separate diagnostic reports or new
report fields for this survey.

Distinguish four boundaries when choosing a repair:

- Fixed interfaces: preserve the fixed SDK, endpoint, identity, request/response contract, authorized tool
  interfaces and execution-error propagation.
- Execution constraints: obey the frozen job's model and call budgets, visibility, read/write paths, commit limits,
  validation and any adopted replay protocol.
- FHO initialization limits: requirements for a neutral minimal baseline and initializer restrictions on template
  edits govern initialization. They do not freeze optimizer-trainable mechanisms after initialization. Continue to
  honor the fixed contract; existing README/ROLE descriptions of baseline behavior do not independently make an
  allowed implementation path immutable.
- Trainable decision mechanisms: repository 'allowed_paths' are trainable paths. Changes to role
  definitions/descriptions and prompts, as well as code-level changes to strategy, memory, parsing or tools, are
  permitted where declared. Choose the mechanism from the observed failure evidence; the FHO role split is not
  immutable, but fixed topology, interfaces and tool permissions remain binding.

When needed to address the prioritized failure patterns, you may change or replace existing mechanisms, restructure
trainable code, or introduce a new mechanism or design within those paths and boundaries. Existing code need not
have an implementation bug to justify a code-level decision-policy improvement. Prefer the smallest change necessary
to address the failure mechanism, not the fewest edited files or a prompt-only fix. Do not require a prompt attempt
before considering code, or change code merely to increase the number of code edits.

Keep the optimized harness's forward input context and node outputs lightweight while preserving decision-critical
information, honoring the fixed contract and achieving the intended mechanism improvement. Avoid redundant
information injection, unnecessary expansion of history or evidence, and accumulating verbose rules or intermediate
explanations. Pass only the information the downstream consumer needs to fulfill its role. Consider both input and
output costs, including downstream reuse and accumulation in later history. Additional context or output should have
a clear mechanism benefit. This is an optimization preference, not a hard token limit or a new acceptance criterion.
Never discard key constraints, evidence, uncertainty or required history merely to reduce tokens; continue to obey
the frozen history contract.

Create at most one candidate commit based on the published base, run necessary development tests, and leave the
candidate clean. Use the interpreter provided by the controller, with the candidate repository as the test cwd. For
'no_change', leave HEAD at the base without an empty commit. Keep the fixed contract and all model, visibility and
commit limits.

Keep test source files in their registered allowed paths. Put test-generated temporary files under the job-specific
'Dev/tmp/' directory given in the launch instruction. When running pytest from the candidate repository, pass that
directory as '--basetemp'; do not create test temporary directories inside the candidate repository. The runner owns
cleanup of its job-specific directory.

The semantic report is the only final deliverable; this does not prohibit the authorized candidate edits/commit,
development tests, or adopted replay input files and controller replay commands. Complete those steps before writing
one JSON report to 'delivery.report_path', following 'delivery.report_schema'. Successful reports contain
'schema_version="1.0"', 'job_id', 'status="succeeded"', 'diagnosis', 'evidence_case_ids',
'consumed_environment_feedback_ids', one 'agent_decisions' entry, 'consumed_downstream_feedback_ids', and
'upstream_feedback'. The decision names 'agent_id', 'outcome' ('changed' or 'no_change'), 'change_summary', and its
own 'evidence_case_ids'. Evidence case lists must be nonempty, unique subsets of the visible cases. Confirm
```

consumption of every declared environment and downstream feedback; use environment case IDs and downstream feedback IDs. Keep diagnosis and summaries brief and evidence-based.

Provide exactly one upstream feedback entry per direct recipient, even for 'no_change'. Each entry has 'recipient_agent_id', 'feedback_type', 'case_ids', 'summary', 'improvement_points', and 'preserve_points'. 'improve' requires improvement points; 'preserve' requires preserve points and no improvement points; 'no_special_guidance' has neither. Explain useful upstream input without single-point blame or claims of single-node causality for an end-to-end failure.

If unable to finish, write only 'schema_version', 'job_id', 'status="failed"', and a nonempty 'error'. After the candidate, necessary tests and report are complete, finish immediately. The controller owns declared delivery checks, diff capture, hashes, inventory, execution statistics and full result generation. Do not generate those artifacts, write a schema checker, invoke acceptance, activate a checkpoint, or retry the job.

Listing E2: Global (flat-network) optimizer semantic-report prompt template.

Flat network optimizer: semantic report

Execute only the frozen 'gpt-5.6-sol', 'reasoning_effort=high', 'mode=standard' job using 'delivery.protocol=semantic-report-v1'. Read the registered whole-network failure bundles, environment feedback, checkpoint and fixed contract. The rendered prompt is read-only. Inspect 'case_memory_snapshot' when diagnosing history or memory-continuity failures; use only this job's visible evidence.

Before choosing a repair, systematically review the entire failure batch visible to this job. Check every case's feedback and key behaviors first, then inspect relevant traces and implementation more deeply by failure pattern. This does not require expanding every full model trace. Distinguish observed symptoms, root cause hypotheses and supporting evidence; do not force all failures into one explanation or stop the survey at the first easy fix.

Prioritize one or more patterns by their coverage, severity, strength of causal evidence, likely transfer to similar tasks and feasibility within the frozen budget. Frequency alone does not determine importance; no numerical scoring formula is required. Aim to improve the harness's general capability on similar tasks, beyond correcting isolated cases or passing selected replay samples. A single round need not resolve every pattern.

In the existing diagnosis, briefly summarize the main patterns, their supported scope and representative case IDs, why the selected targets deserve priority, and the validation findings and remaining uncertainties. Use existing tests and any adopted replay protocol to check the chosen solution. Distinguish observed local gains from expected broader benefits and note untested patterns and potential regressions. Do not create separate diagnostic reports or new report fields for this survey.

Distinguish four boundaries when choosing a repair:

- Fixed interfaces: preserve the fixed SDK, endpoint, identity, request/response contract, authorized tool interfaces and execution-error propagation.
- Execution constraints: obey the frozen job's model and call budgets, visibility, read/write paths, commit limits, validation and any adopted replay protocol.
- FHO initialization limits: requirements for a neutral minimal baseline and initializer restrictions on template edits govern initialization. They do not freeze optimizer-trainable mechanisms after initialization. Continue to honor the fixed contract; existing README/ROLE descriptions of baseline behavior do not independently make an allowed implementation path immutable.
- Trainable decision mechanisms: repository 'allowed_paths' are trainable paths. Changes to role definitions/descriptions and prompts, as well as code-level changes to strategy, memory, parsing or tools, are permitted where declared. Choose the mechanism from the observed failure evidence; the FHO role split is not immutable, but fixed topology, interfaces and tool permissions remain binding.

When needed to address the prioritized failure patterns, you may change or replace existing mechanisms, restructure trainable code, or introduce a new mechanism or design within those paths and boundaries. Existing code need not have an implementation bug to justify a code-level decision-policy improvement. Prefer the smallest change necessary to address the failure mechanism, not the fewest edited files or a prompt-only fix. Do not require a prompt attempt before considering code, or change code merely to increase the number of code edits.

Keep the optimized harness's forward input context and node outputs lightweight while preserving decision-critical information, honoring the fixed contract and achieving the intended mechanism improvement. Avoid redundant information injection, unnecessary expansion of history or evidence, and accumulating verbose rules or intermediate explanations. Pass only the information the downstream consumer needs to fulfill its role. Consider both input and output costs, including downstream reuse and accumulation in later history. Additional context or output should have a clear mechanism benefit. This is an optimization preference, not a hard token limit or a new acceptance criterion. Never discard key constraints, evidence, uncertainty or required history merely to reduce tokens; continue to obey the frozen history contract.

Create at most one candidate commit per changed node based on its published base, run necessary development tests, and leave all candidates clean. Use the interpreter provided by the controller, with the candidate repository as the test cwd. For 'no_change', keep HEAD at the base without an empty commit. Keep the fixed contract and all model and commit limits.

Keep test source files in their registered allowed paths. Put test-generated temporary files under the job-specific 'Dev/tmp/' directory given in the launch instruction. When running pytest from a candidate repository, pass that directory as '--basetemp'; do not create test temporary directories inside a candidate repository. The runner owns cleanup of its job-specific directory.

The semantic report is the only final deliverable; this does not prohibit the authorized candidate edits/commits, development tests, or adopted replay input files and controller replay commands. Complete those steps before writing one JSON report to 'delivery.report_path', following 'delivery.report_schema'. Successful reports contain 'schema_version="1.0"', 'job_id', 'status="succeeded"', 'diagnosis', 'evidence_case_ids', 'consumed_environment_feedback_ids', and 'agent_decisions'. Include exactly one decision per target node, naming

'agent_id', 'outcome' ('changed' or 'no_change'), 'change_summary', and that node's own 'evidence_case_ids'. Evidence case lists must be nonempty, unique subsets of visible cases. Confirm consumption of every declared environment feedback using its case ID. Keep diagnosis and summaries brief and evidence-based. Flat jobs must omit upstream feedback and consumed downstream feedback fields; the controller derives the network outcome and candidate checkpoint from decisions and actual repository commits.

If unable to finish, write only 'schema_version', 'job_id', 'status="failed"', and a nonempty 'error'. After the candidates, necessary tests and report are complete, finish immediately. The controller owns declared delivery checks, diff capture, hashes, inventory, execution statistics and full result generation. Do not generate those artifacts, write a schema checker, invoke acceptance, activate a checkpoint, or retry the job.