G-HARNESS: ARCHITECTING AND OPTIMIZING MULTI-AGENT HARNESSES VIA DYNAMIC GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent systems (MAS) depend not only on model capabilities, but also on the surrounding harness—the prompts, tools, memory, and control flow that govern agent interaction and execution. In MAS, agent composition, task assignment, tool access, and model binding are tightly coupled and evolve with task progress and runtime conditions, making harness optimization a dynamic system-level problem. Jointly adapting these decisions during execution is challenging because the feasible configuration space changes with the execution state, while the effects of structural changes are often revealed only through delayed task-level feedback. In this paper, we propose **G-Harness**, a dynamic graph-based framework for architecting and optimizing multi-agent harnesses. G-Harness formulates MAS execution and its surrounding harness as a unified dynamic graph, enabling coupled system organization and execution state to be represented and reasoned about jointly. We develop a task-conditioned graph-edit policy that learns to construct effective task-specific harness graphs and continuously rewrites them during execution, enabling automatic harness optimization and online adaptation based on runtime feedback. We further develop a global–local optimization framework that jointly captures task-level execution utility and the contribution of individual decisions, thereby guiding the policy toward task-relevant structural optimization. Extensive experiments demonstrate three key properties of G-Harness: **(1) high-performing**, achieving 80.68% *Score* on Harness-Bench; **(2) cost-efficient**, achieving a favorable performance-efficiency trade-off compared with existing multi-agent and harness optimization methods; and **(3) attack resilient**, suffering only a 1.14% performance drop under agent-level prompt attacks.

1 INTRODUCTION

Large language model (LLM)-based MAS have demonstrated strong capabilities across a wide range of downstream tasks, including software development (Qian et al., 2024; Hong et al., 2024), data analysis and scientific discovery (Sun et al., 2026; Ghafarollahi & Buehler, 2025), embodied collaboration (Zhao et al., 2024), and autonomous driving (Jiang et al., 2025; Zhang et al., 2025e). In these systems, multiple agents with specialized roles are coordinated through explicit workflows and communication structures to support task decomposition and collaborative execution (Li et al., 2023; Wu et al., 2024; Hong et al., 2024). Beyond the capabilities of individual models, recent work has shown that MAS performance is strongly influenced by system-level design, motivating increasing efforts on workflow optimization (Zhang et al., 2025d), task-conditioned communication topology design (Zhang et al., 2025c; Li et al., 2026b), role and model routing (Yue et al., 2025), and executable multi-agent orchestration (Zhang et al., 2025b; Chen et al., 2026b). However, MAS design alone does not fully determine system execution, which also depends on the surrounding **harness** (Zhong & Zhu, 2026; Ning et al., 2026).

The harness serves as the runtime substrate surrounding models, governing how context, tools, and other external capabilities are accessed, how execution state is maintained, and how agent actions are executed, verified, and constrained during interaction with the environment (Zhong & Zhu, 2026; Ning et al., 2026). Prior harness designs have largely relied on manual engineering and task-specific configurations, such that any change in model version, available tools, or problem domain requires bespoke modification, with no mechanism for experience-driven improvement (Young, 2025; Galster et al., 2026; Yao et al., 2026). To address this limitation, recent work has increasingly treated

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

the harness itself as an optimization target, exploring automated harness construction and adaptation from execution feedback and evolving task requirements (Lee et al., 2026; Zhang et al., 2026a; Liu et al., 2026b; Wei et al., 2026). However, in MAS, agent composition, communication topology, task assignment, model binding, and harness configuration are tightly coupled, such that optimizing any single dimension in isolation can overlook important cross-component interactions (Feng et al., 2026; Zhang et al., 2025c; Chen et al., 2026b).

Recent work has therefore begun to optimize these system-level choices jointly; for example, AgentFlow jointly optimizes agent roles, prompts, tool bindings, and coordination protocols within a structured multi-agent harness (Liu et al., 2026a). Yet broadening the optimization space alone does not resolve the gaps posed by dynamic multi-agent execution. ***Gap 1 – Limited modeling of the running system:*** Effective runtime adaptation requires an explicit representation of both the current system structure and its evolving execution state. However, representative approaches typically represent the harness as a static or partially specified configuration, without explicitly modeling the running system as a unified and evolving state (Lee et al., 2026; Chen et al., 2026b). ***Gap 2 – Limited task-adaptive harness optimization:*** Different tasks require different harness configurations, which may further change during execution. However, existing approaches often address task-specific construction and runtime adaptation separately, making it difficult to learn a unified policy for both initial configuration and online refinement (Lee et al., 2026; Zhang et al., 2026a; Liu et al., 2026b; Wei et al., 2026). ***Gap 3 – Coarse structural supervision:*** Harness optimization involves interdependent structural decisions whose effects are often revealed only through downstream execution outcomes. However, existing methods largely rely on trajectory-level rewards, making it difficult to attribute task utility to specific decisions (Luo et al., 2025).

To fill these gaps, we propose a framework, termed **G-Harness**, for task-adaptive and online optimization of multi-agent harnesses. To bridge ***Gap 1***, we formulate MAS execution and its surrounding harness as a unified dynamic graph that distinguishes the Global Harness Graph, Task Graph, and Active Harness Subgraph. This formulation enables joint reasoning over system organization and execution state as they evolve throughout execution. To address ***Gap 2***, we develop a task-conditioned graph-edit policy that constructs task-appropriate Active Harness Subgraphs and continuously rewrites them during execution. This unifies automatic harness optimization and online adaptation within a single learnable policy. To mitigate ***Gap 3***, we develop a global–local optimization framework that combines supervised initialization, trajectory-level Group Relative Policy Optimization (GRPO) (Shao et al., 2024), and Local Graph Perturbation Optimization. This jointly captures task-level execution utility and the contribution of individual decisions, enabling more effective learning of task-relevant harness structures. Our main contributions are as follows:

- **Dynamic graph formulation.** We formulate MAS execution together with its surrounding harness as a unified dynamic graph, jointly representing the evolving system structure and execution state to support structured reasoning over the running system.
- **Task-conditioned graph-edit policy.** We develop a task-conditioned policy that constructs the Active Harness Subgraph through constrained graph edits and continuously rewrites its structure according to task progress and runtime feedback, enabling dynamic adaptation throughout execution.
- **Global–local optimization.** We develop an optimization framework that combines supervised initialization, trajectory-level GRPO, and Local Graph Perturbation Optimization, jointly capturing task-level execution utility and the contribution of individual decisions for more effective learning of task-relevant harness structures.
- **Extensive empirical evaluation.** We evaluate G-Harness on Harness-Bench, DeepPlanning, and ALFWorld, where it achieves 80.68% *Score*, 87.28% *Match Score*, and 89.39% *Avg. SR*, respectively, while maintaining favorable efficiency-performance trade-off and suffering only a 1.14% performance drop under prompt attacks.

2 Related Work

Harness Engineering for LLM Agents. Recent work has increasingly viewed the harness as an explicit runtime layer surrounding LLM agents, governing capability access, state management, and action execution during operation (Zhong & Zhu, 2026; Ning et al., 2026; Young, 2025). Early

harnesses were largely hand-engineered for specific models and tasks, requiring manual redesign when adapting to new tools, environments, or execution settings (Yao et al., 2023; Qin et al., 2024; Packer et al., 2023; Zhang et al., 2025a; Young, 2025). Recent studies have begun to optimize harnesses themselves through feedback-driven construction and adaptation, demonstrating that harness design can substantially affect agent performance beyond model capability (Lee et al., 2026; Zhang et al., 2026a; Liu et al., 2026b; Wei et al., 2026; Vats & Golev, 2026; Galster et al., 2026; Yao et al., 2026). These efforts shift harness engineering toward automated harness construction and adaptation. *Difference from existing work*: Existing harness optimization methods mainly focus on adapting harness configurations based on execution feedback. In contrast, G-Harness formulates harness optimization as a system-level decision process coupled with multi-agent execution, jointly optimizing agent-task assignment, model binding, capability selection, and interaction structure through a task-conditioned policy that supports both initial construction and online reconfiguration.

Graph-Based Multi-Agent System Design. Graph-based formulations have been increasingly explored for multi-agent system design, representing agents, tasks, and executable components as nodes and their dependencies, communication patterns, or execution relations as edges (Feng et al., 2026; Zhuge et al., 2024). Existing approaches mainly apply graphs to workflow construction, task decomposition, and agent coordination, including graph-based execution planning (Zhang et al., 2025d), communication topology optimization, dynamic routing, and agent organization design (Zhang et al., 2025c; Lu et al., 2026; Li et al., 2026a;b). Recent studies further consider agent roles and routing decisions within broader orchestration frameworks (Yue et al., 2025; Chen et al., 2026b). *Difference from existing work*: Prior graph-based MAS methods primarily use graphs to describe or optimize agent organization, communication topology, workflows, and task dependencies. In G-Harness, the graph serves as the state representation of the running system itself, jointly capturing multi-agent execution and the surrounding harness, and serving as the object that is directly constructed and rewritten during execution.

3 FORMALIZATION

We formulate multi-agent execution and its surrounding harness as an evolving graph system. As illustrated in Figure 1, G-Harness maintains a Global Harness Graph containing available agents, tools, memories, and runtime states, derives a task-specific Active Harness Subgraph for execution, and continuously adapts it according to task progress and runtime feedback. The Task Graph specifies task dependencies and computational requirements, providing task-level guidance for harness construction and optimization.

Global Harness Graph. The Global Harness Graph is constructed from predefined system configurations, with registered components instantiated as typed nodes and candidate edges generated by relation rules under compatibility constraints. During execution, the Global Harness Graph evolves as runtime snapshots $\{G_t^{\mathrm{H}}\}_{t=0}^{N}$, where $G_t^{\mathrm{H}} = (V_t, E_t, X_t^V, X_t^E, \mathbf{b}_t; \Sigma)$ and N denotes the final runtime epoch. The node set $V_t = V_t^{\mathrm{agent}} \cup V_t^{\mathrm{tool}} \cup V_t^{\mathrm{memory}}$ contains Agent, Tool, and Memory nodes. The directed edge set $E_t \subseteq V_t \times \mathcal{R} \times V_t$ specifies typed interaction channels under a fixed relation schema $\mathcal{R}$, such as $\{\texttt{communicate}, \texttt{delegate}, \texttt{call}, \texttt{read}, \texttt{write}\}$. An edge (u, r, v) denotes an interaction channel of type r from source node u to target node v. The node attributes $X_t^V = (x_{v,t})_{v \in V_t}$ describe component capabilities, availability, admissible configuration options, and observed runtime states, while the edge attributes $X_t^E = (x_{e,t})_{e \in E_t}$ describe channel interfaces, permissions, and execution statistics. The vector $\mathbf{b}_t \in \mathbb{R}^{d_b}$ records resource consumption and remaining budgets, including tokens, latency, monetary expenditure, tool calls, and graph edits. A finite catalogue $\mathcal{M}$ provides available model configurations. The constraint set Σ governs valid graph states, model bindings, and graph modifications through type compatibility, interface compatibility, access control, and resource feasibility.

Task Graph. Given an input task instance, an LLM-based Task Composer generates a task-specific Task Graph that captures task dependencies, capability requirements, and publicly verifiable completion conditions, with the detailed prompt provided in Appendix E. The generated Task Graph is represented as a static directed acyclic graph $G^{\mathrm{T}} = (V^{\mathrm{T}}, E^{\mathrm{T}}, X^{\mathrm{T}})$. A node $v_i \in V^{\mathrm{T}}$ denotes a subtask, and an edge $(v_i, v_j) \in E^{\mathrm{T}}$ requires v_i to complete successfully before v_j can start. The attributes $X^{\mathrm{T}}(v_i) = (g_i, \mathbf{q}_i, \mathcal{I}_i^{\mathrm{T}}, \mathcal{O}_i^{\mathrm{T}}, \mathcal{C}_i^{\mathrm{T}})$ specify the stage-level ob-

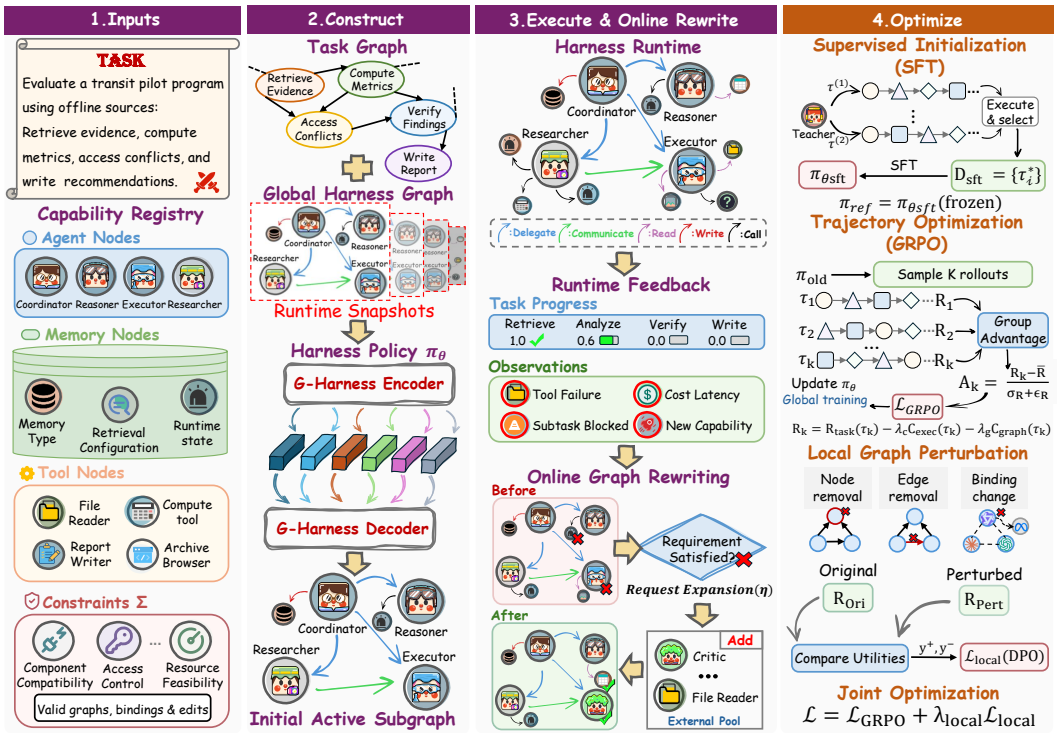


Figure 1: Overview of **G-Harness**. **G-Harness** constructs an initial Active Harness Subgraph from the Task Graph and Global Harness Graph, continuously rewrites it according to runtime feedback during execution, and optimizes the graph-edit policy through supervised initialization, trajectory-level GRPO, and Local Graph Perturbation Optimization.

jective, capability requirements, input schema, output schema, and publicly verifiable completion contract of v_i, respectively. The completion contract $\mathcal{C}_i^{\mathrm{T}}$ contains registered predicates and their arguments and determines whether the output and execution evidence satisfy the conditions required to mark the subtask as completed. The runtime state is $z_t : V^{\mathrm{T}} \to \mathcal{Z}$, where $\mathcal{Z} = \{\texttt{pending}, \texttt{ready}, \texttt{running}, \texttt{success}, \texttt{failed}\}$. A pending task becomes ready when all its predecessor tasks have completed successfully and its input requirements are satisfied, while a running task is marked as successful only when its completion contract is satisfied.

Active Harness Subgraph. At runtime epoch t, the Active Harness Subgraph is $G_t^{\mathrm{A}} = (V_t^{\mathrm{A}}, E_t^{\mathrm{A}}, X_t^{V,\mathrm{A}}, X_t^{E,\mathrm{A}}, \mu_t, \beta_t)$, where $V_t^{\mathrm{A}} \subseteq V_t$ and $E_t^{\mathrm{A}} \subseteq E_t \cap (V_t^{\mathrm{A}} \times \mathcal{R} \times V_t^{\mathrm{A}})$. The global attributes X_t^V and X_t^E describe available components, channels, and admissible configuration options, while the active attributes $X_t^{V,\mathrm{A}}$ and $X_t^{E,\mathrm{A}}$ record the configurations selected for execution and their runtime states. Let $V_t^{\mathrm{A,agent}} = V_t^{\mathrm{A}} \cap V_t^{\mathrm{agent}}$. The assignment $\mu_t : V^{\mathrm{T}} \to 2^{V_t^{\mathrm{A,agent}}}$ maps each task to its assigned Agents, with $\mu_t(v_i) = \varnothing$ denoting an unassigned task. The binding $\beta_t : V_t^{\mathrm{A,agent}} \to \mathcal{M}$ selects a model configuration for each active Agent. A valid Active Harness Subgraph satisfies $\mathrm{Valid}_\Sigma(G_t^{\mathrm{A}} \mid G_t^{\mathrm{H}}, G^{\mathrm{T}}, z_t) = 1$, which enforces structural consistency, admissible configurations and model bindings, capability and input-output compatibility requirements of assigned tasks, required interaction channels, access control, and resource feasibility. The task-conditioned policy in Sec. 4 constructs and rewrites this subgraph during execution.

4 Task-Conditioned G-Harness Policy

We learn a policy π_θ that constructs and rewrites the Active Harness Subgraph through constrained graph edits. At runtime epoch t, observations initialize the working Global Harness Graph $\widehat{G}_{t,0}^{\mathrm{H}}$ and TaskNode states z_t. Let $\widehat{G}_{t,\ell}^{\mathrm{H}}$ and $\widehat{G}_{t,\ell}^{\mathrm{A}}$ denote the working Global Harness Graph and Active Harness

Subgraph after ℓ edits, following the notation in Sec. 3. The active graph is empty at $t = 0$ and is otherwise initialized from G^{A}_{t-1} after refreshing observations and removing unavailable components and channels. Each decision $\ell \geq 1$ conditions on $\mathcal{C}_{t,\ell} = (G^{\mathrm{T}}, \widehat{G}^{\mathrm{H}}_{t,\ell-1}, \widehat{G}^{\mathrm{A}}_{t,\ell-1}, z_t)$. Intermediate graphs may contain unbound Agents, whereas every committed graph must assign an admissible model to each active Agent.

G-Harness Encoder. A relation-aware graph Transformer (Ying et al., 2021) jointly encodes $\mathcal{U}_{t,\ell} = V^{\mathrm{T}} \cup \widehat{V}_{t,\ell-1}$. For each node u, a frozen Sentence-BERT text encoder (Reimers & Gurevych, 2019) produces a description embedding $\mathbf{e}_u$, while $\mathbf{x}_u$ encodes task requirements and status, component availability, active membership, admissible and selected configurations, and model bindings. The initial representation is $\mathbf{h}^{(0)}_u = W^{\mathrm{in}}_{\tau(u)}[\mathbf{e}_u; \mathbf{x}_u] + \mathbf{c}^{\mathrm{in}}_{\tau(u)} \in \mathbb{R}^d$, where $\tau(u) \in \{\mathrm{task}, \mathrm{agent}, \mathrm{tool}, \mathrm{memory}\}$ denotes the node type. We omit decision indices from node and relation features. For each ordered pair (u, v), $\boldsymbol{\rho}_{uv}$ encodes task dependencies, global and active channels, assignments from $\widehat{\mu}_{t,\ell-1}$, and edge attributes, using distinct features for forward and reverse relations and zeros for absent relations. At layer k and head p,

$$
\begin{aligned}
s^{(k,p)}_{uv} &= \frac{(W^{(k,p)}_Q \mathbf{h}^{(k)}_u)^\top (W^{(k,p)}_K \mathbf{h}^{(k)}_v)}{\sqrt{d_h}} + (\mathbf{w}^{(k,p)}_R)^\top \boldsymbol{\rho}_{uv}, \\
\mathbf{y}^{(k,p)}_u &= \sum_{v \in \mathcal{U}_{t,\ell}} a^{(k,p)}_{uv} \left(W^{(k,p)}_V \mathbf{h}^{(k)}_v + U^{(k,p)}_R \boldsymbol{\rho}_{uv} \right).
\end{aligned}
\tag{1}
$$

Here $a^{(k,p)}_{uv} = \mathrm{softmax}_{v \in \mathcal{U}_{t,\ell}}(s^{(k,p)}_{uv})$ and $d_h = d/P$ for P heads. Outputs are concatenated and projected, followed by a feed-forward sublayer with residual connections and layer normalization.

G-Harness Decoder. Let $\bar{\mathbf{h}}^{\mathrm{T}}_{t,\ell}$, $\bar{\mathbf{h}}^{\mathrm{H}}_{t,\ell}$, and $\bar{\mathbf{h}}^{\mathrm{A}}_{t,\ell}$ denote the mean final encoder representations over task, working global, and working active nodes, respectively, with zero pooling for empty sets. The decoder state is $\mathbf{s}_{t,\ell} = [\bar{\mathbf{h}}^{\mathrm{T}}_{t,\ell}; \bar{\mathbf{h}}^{\mathrm{H}}_{t,\ell}; \bar{\mathbf{h}}^{\mathrm{A}}_{t,\ell}; \widehat{\mathbf{b}}_{t,\ell-1}]$. Each decision $a_{t,\ell} = (\omega_{t,\ell}, \boldsymbol{\alpha}_{t,\ell})$ selects an operation from $\Omega_V \cup \Omega_E \cup \Omega_\mu \cup \Omega_\beta \cup \Omega_X \cup \Omega_C$ and a complete legal argument tuple, modifying active components, edges, task assignments, model bindings, configurations, or the current capability pool. The complete edit space is summarized in Appendix B.1. Let $\mathcal{D}^{\Sigma}_{t,\ell}(\omega)$ denote the finite set of legal argument tuples under $\mathcal{C}_{t,\ell}$ and Σ. Observed states are read-only, and components and configurations used by running tasks remain fixed. Intermediate graphs may contain unbound Agents. A masked MLP maps $\mathbf{s}_{t,\ell}$ to π^{op}_θ, excluding operations with no legal argument tuple. Conditioned on the selected operation and decoder state, a candidate scorer assigns a scalar score to each $\boldsymbol{\alpha} \in \mathcal{D}^{\Sigma}_{t,\ell}(\omega)$ using the encoder representations of referenced nodes and embeddings of non-node arguments, with their argument roles distinguished. A softmax over these tuples defines $\pi^{\mathrm{cand}}_\theta$. The complete-decision probability is

$$
\pi_\theta(a_{t,\ell} \mid \mathcal{C}_{t,\ell}) = \pi^{\mathrm{op}}_\theta(\omega_{t,\ell} \mid \mathcal{C}_{t,\ell}) \pi^{\mathrm{cand}}_\theta(\boldsymbol{\alpha}_{t,\ell} \mid \omega_{t,\ell}, \mathcal{C}_{t,\ell}). \tag{2}
$$

The argument-free operation $\texttt{EOS} \in \Omega_C$ commits the working graphs and is admissible only when $\mathrm{Valid}_\Sigma(\widehat{G}^{\mathrm{A}}_{t,\ell-1} \mid \widehat{G}^{\mathrm{H}}_{t,\ell-1}, G^{\mathrm{T}}, z_t) = 1$ and the graph supports starting a ready task or continuing a running task. For an admissible EOS, the only argument tuple is empty and its conditional probability equals one.

Runtime Feedback and Online Rewriting. Structural and configuration edits update the working Active Harness Subgraph; removing a component also removes its incident active channels, assignments, and binding. The operation $\texttt{RequestExpansion} \in \Omega_C$ takes an admissible capability request η, retrieves additional components and channels into the working Global Harness Graph, and updates its resource state. Retrieved components become candidates for subsequent active-graph edits. Each non-EOS edit decrements the remaining graph-edit budget in $\widehat{\mathbf{b}}_{t,\ell}$, and the policy re-encodes the updated context. After L_t non-EOS edits, selecting EOS at decision $L_t + 1$ commits $G^{\mathrm{H}}_t = \widehat{G}^{\mathrm{H}}_{t,L_t}$ and $G^{\mathrm{A}}_t = \widehat{G}^{\mathrm{A}}_{t,L_t}$. Execution follows task dependencies, assignments, and model bindings until a task-state transition or a material change in component availability or resources triggers the next epoch. Feedback yields $\widehat{G}^{\mathrm{H}}_{t+1,0}$ and z_{t+1}. Episodes terminate according to the benchmark's success and failure conditions. At the edit-budget limit, the operation mask retains only an admissible EOS; otherwise, the rollout terminates with failure. The input projections, graph Transformer, candidate representations, and decoding heads constitute θ; the text encoder remains frozen.

5 OPTIMIZING THE G-HARNESS POLICY

We optimize π_θ through supervised initialization, global trajectory optimization with GRPO (Shao et al., 2024), and local refinement using preferences over harness configurations. A rollout τ records policy decisions, execution outcomes, and resource consumption. Within each rollout, j indexes decisions across runtime epochs, including `EOS`.

Supervised Fine-Tuning. For each training task, a teacher LLM generates candidate trajectories using the same observable context and constrained action space as π_θ. We execute these candidates and retain the highest-utility trajectory whose edits are legal and whose committed graphs satisfy the execution conditions in Sec. 4, using the utility in Eq. 4. The resulting dataset is $\mathcal{D}_{\text{sft}} = \{\tau_i^*\}_{i=1}^{N_{\text{sft}}}$, where τ_i^* contains $M_i \geq 1$ decisions $(\mathcal{C}_{i,j}, a_{i,j}^*)$. Using the complete-decision probability in Eq. 2, we minimize

$$\mathcal{L}_{\text{sft}}(\theta) = -\frac{1}{N_{\text{sft}}} \sum_{i=1}^{N_{\text{sft}}} \frac{1}{M_i} \sum_{j=1}^{M_i} \log \pi_\theta(a_{i,j}^* \mid \mathcal{C}_{i,j}). \tag{3}$$

Subsequent training starts from θ_{sft}, with a frozen reference policy $\pi_{\text{ref}} = \pi_{\theta_{\text{sft}}}$.

Global Trajectory Optimization. At each iteration, we freeze the current policy as π_{old} and sample $K \geq 2$ rollouts for each task from the same initial Global Harness Graph and environment state. Rollout τ_k contains $M_k \geq 1$ decisions $(\mathcal{C}_{k,j}, a_{k,j})$ and receives utility

$$R_k = R_{\text{task}}(\tau_k) - \lambda_c C_{\text{exec}}(\tau_k) - \lambda_g C_{\text{graph}}(\tau_k), \tag{4}$$

where $R_{\text{task}}(\tau_k) \in [0, 1]$ is the terminal benchmark score and $\lambda_c, \lambda_g \geq 0$. The execution cost C_{exec} is a nonnegative weighted sum of normalized token usage, tool calls, latency, monetary expenditure, and capability-expansion costs. The structural cost C_{graph} is a nonnegative weighted sum of normalized mean active node counts, mean active edge counts, and the total number of active-graph edits, excluding `EOS`. Mean graph counts are averaged over executed epochs and set to zero when no active graph is executed. Normalization scales and aggregation weights remain fixed throughout training. Failed rollouts receive the benchmark's failure score and retain their incurred costs.

Following GRPO (Shao et al., 2024), we compute $A_k = (R_k - \bar{R})/(\sigma_R + \epsilon_R)$, where $\bar{R}$ and σ_R are the group mean and population standard deviation, and $\epsilon_R > 0$. Define $r_{k,j}(\theta) = \pi_\theta(a_{k,j} \mid \mathcal{C}_{k,j})/\pi_{\text{old}}(a_{k,j} \mid \mathcal{C}_{k,j})$ and $D_{k,j}(\theta) = D_{\text{KL}}(\pi_\theta(\cdot \mid \mathcal{C}_{k,j}) \| \pi_{\text{ref}}(\cdot \mid \mathcal{C}_{k,j}))$. All policies use the recorded contexts, candidate sets, and identical legality masks, including budget-induced masks. We compute the complete-decision KL divergence exactly over each finite legal candidate set, as detailed in Appendix B.3. Let $\mathcal{H}_{k,j}(\theta)$ denote the entropy of the complete-decision distribution over the same set. We minimize

$$\mathcal{L}_{\text{GRPO}}(\theta) = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{M_k} \sum_{j=1}^{M_k} \Big[\lambda_{\text{KL}} D_{k,j}(\theta) - \lambda_{\text{ent}} \mathcal{H}_{k,j}(\theta) - \min\{ r_{k,j}(\theta) A_k,\ \text{clip}(r_{k,j}(\theta), 1-\epsilon, 1+\epsilon) A_k \} \Big],$$

where $\epsilon \in (0, 1)$ and $\lambda_{\text{KL}}, \lambda_{\text{ent}} \geq 0$. Group advantages and the collection policy remain fixed during each update.

Local Graph Perturbation Optimization. We obtain configuration-level supervision by comparing alternative Active Harness Subgraphs from a shared execution branch point. For each rollout group, we select a recorded context $\mathcal{C}$ immediately before an admissible `EOS` while edit budget remains. The active graph $\widehat{G}^{\text{A}}$ defines the original alternative $y^{(0)} = (\texttt{EOS})$. We then construct legal perturbation sequences $\delta = (a_1, \ldots, a_m) \in \Delta(\mathcal{C})$ that modify components, edges, task assignments, model bindings, or configurations using candidates from the current Global Harness Graph. Each perturbation produces a distinct valid graph $\widehat{G}^{\text{A}}_\delta$ and defines an alternative $y^{(\delta)} = (a_1, \ldots, a_m, \texttt{EOS})$; construction details are provided in Appendix B.2. Both alternatives branch from the same saved environment state and resource budget, after which execution continues under π_{old}. Each continuation is combined with the shared rollout prefix and scored using Eq. 4, including its incurred edit costs. The higher-utility alternative is labeled y^+ and the lower-utility alternative is labeled y^-, with ties discarded.

For an edit sequence $y = (a_1, \ldots, a_{|y|})$, we define $\pi_\theta(y \mid \mathcal{C}) = \prod_{i=1}^{|y|} \pi_\theta(a_i \mid \mathcal{C}_i^y)$, where $\mathcal{C}_1^y = \mathcal{C}$ and each subsequent context is obtained by applying the preceding edits and resource updates. From

the resulting preference dataset $\mathcal{D}_{\text{local}} = \{(\mathcal{C}, y^+, y^-)\}$, we minimize the DPO objective (Rafailov et al., 2023):

$$\mathcal{L}_{\text{local}}(\theta) = -\mathbb{E}_{(\mathcal{C},y^+,y^-)\sim\mathcal{D}_{\text{local}}}\left[\log\sigma\left(\kappa\log\frac{\pi_\theta(y^+ \mid \mathcal{C})}{\pi_{\text{ref}}(y^+ \mid \mathcal{C})} - \kappa\log\frac{\pi_\theta(y^- \mid \mathcal{C})}{\pi_{\text{ref}}(y^- \mid \mathcal{C})}\right)\right], \quad (5)$$

where σ denotes the logistic sigmoid and $\kappa > 0$ controls the preference-logit scale.

Joint Optimization. Global training minimizes $\mathcal{L}_{\text{GRPO}}$. During local refinement, we minimize $\mathcal{L} = \mathcal{L}_{\text{GRPO}} + \lambda_{\text{local}}\mathcal{L}_{\text{local}}$, where $\lambda_{\text{local}} > 0$. Each iteration collects rollout groups under π_{old} and constructs local preference pairs from graph perturbations. GRPO uses the rollouts, while perturbed continuations supply preference labels. The reference policy remains fixed throughout training, and the collection policy, rollout data, advantages, and preference labels remain fixed within each update.

6 EXPERIMENTS

In this section, we conduct extensive experiments to answer the following research questions (RQs): (1) Can G-Harness achieve superior performance across diverse agent benchmarks compared with existing approaches? (2) Does G-Harness achieve favorable performance-efficiency trade-off? (3) How robust is G-Harness against agent-level prompt attacks? (4) How do different components contribute to the effectiveness of G-Harness?

6.1 EXPERIMENTAL SETUP

Datasets and Evaluation Metrics. To comprehensively evaluate G-Harness, we conduct experiments on **Harness-Bench**, **DeepPlanning-Shopping (DeepPlanning)**, and **ALFWorld**. **Harness-Bench** (Yao et al., 2026) is a recent challenging benchmark for evaluating harness effects in realistic agent workflows. It assesses agent systems from multiple perspectives, including task completion, tool usage, execution consistency, robustness, and efficiency. We report the overall *Score*, together with *Comp.* for task completion capability, *Tool* for tool utilization effectiveness, *Cons.* for execution consistency, *Rob.* for robustness under execution disturbances, and *Tok. (K)* for average token consumption. **DeepPlanning** (Zhang et al., 2026b) evaluates long-horizon planning under fine-grained local and global constraints. We report the *Match Score (Match Sco.)*, which measures the fraction of target items correctly selected in the generated shopping plan. **ALFWorld** (Shridhar et al., 2021) evaluates sequential decision making through interactive embodied household tasks. We report the average success rate (*Avg. SR*) across all task categories, where a task is considered successful when the agent completes the required objective in the environment.

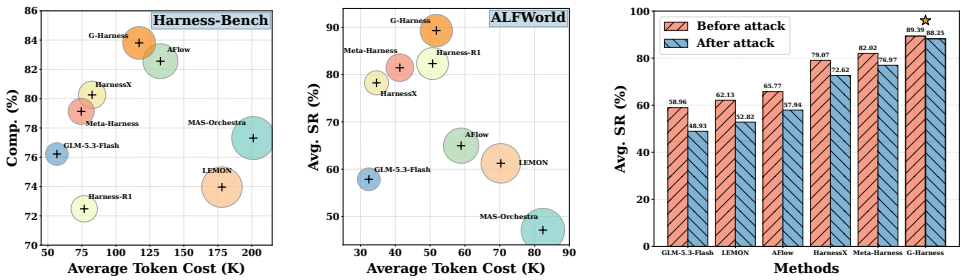
Baselines. We compare G-Harness with three baseline groups. **Vanilla** evaluates single-model execution with seven LLMs: `Llama-3.3-70B`, `Qwen3-Coder-30B`, `Qwen3.7-Plus`, `GPT-4o mini`, `GLM-5.3-Flash`, `DeepSeek-V4.1-Flash`, and `DeepSeek-V4-Pro`. Reasoning or thinking modes are disabled whenever supported, and the temperature is set to 0; for `GLM-5.3-Flash`, we use `reasoning_effort=low`. **Multi-Agent Systems** include LEMON (Chen et al., 2026b), AFlow (Zhang et al., 2025d), and MAS-Orchestra (Ke et al., 2026). These methods have access to the same pool of up to five agent instances, model catalogue, tools, memory backend, and execution budget as G-Harness while retaining their coordination mechanisms. **Harness-based Methods** include Harness-R1 (Shao et al., 2026), HarnessX (Chen et al., 2026a), and Meta-Harness (Lee et al., 2026). These methods retain their native optimization procedures, use `GLM-5.3-Flash` as the task-facing model, and share the same tools, task inputs, and execution budgets; all auxiliary model calls are included in the resource cost.

6.2 EXPERIMENTAL RESULTS

▷ **Overall Performance Comparison (RQ1).** Table 1 presents the performance comparison of G-Harness against three categories of baselines, including vanilla model execution, multi-agent systems, and harness-based methods, on Harness-Bench, DeepPlanning, and ALFWorld. We make three key observations. ❶ **Achieving superior performance over strong baselines.** Compared with the strongest baselines, G-Harness achieves substantial improvements across different evaluation settings. On Harness-Bench, G-Harness surpasses AFlow (70.31%) and HarnessX (74.48%) by

Table 1: Performance comparison with three categories of baselines, including vanilla model execution, multi-agent systems, and harness-based methods, on Harness-Bench, DeepPlanning, and ALFWorld. The best results are shown in bold. ↑ indicates higher is better.

	Harness-Bench					DeepPlanning	ALFWorld
Method	Score (%) ↑	Comp. (%) ↑	Tool (%) ↑	Cons. (%) ↑	Rob. (%) ↑	Match Sco. (%) ↑	Avg. SR (%) ↑
Vanilla							
Llama-3.3-70B	26.94	43.66	57.86	50.71	52.14	58.23	37.00
Qwen3-Coder-30B	29.42	46.97	51.19	43.33	48.57	55.74	34.23
Qwen3.7-Plus	57.08	79.54	74.05	67.38	71.19	75.22	45.77
GPT-4o mini	23.29	45.81	44.52	38.10	44.76	66.78	39.61
GLM-5.3-Flash	59.47	77.11	77.24	73.10	78.10	78.29	58.96
DeepSeek-V4.1-Flash	71.07	78.45	90.71	88.62	87.86	79.12	43.74
DeepSeek-V4-Pro	65.28	78.31	87.14	82.14	82.86	71.25	41.82
Multi-Agent Systems							
LEMON	55.95	73.95	76.91	71.76	75.48	77.31	62.13
AFlow	70.31	82.56	81.06	79.83	83.61	82.53	65.77
MAS-Orchestra	62.07	77.59	78.57	74.43	77.86	67.49	48.93
Harness-based methods							
Harness-R1	64.72	72.61	84.67	81.76	84.05	81.12	83.22
HarnessX	74.48	80.06	89.00	86.14	90.29	84.80	79.07
Meta-Harness	72.47	79.10	87.14	84.11	85.33	83.34	82.02
G-Harness	**80.68**	**84.81**	**92.67**	**89.33**	**92.38**	**87.28**	**89.39**



(a) Cost Analysis (b) Cost Analysis (c) Robustness on ALFWorld

Figure 2: Cost and robustness analysis of G-Harness. (a) and (b) show the cost–performance trade-off on Harness-Bench and ALFWorld, respectively. The x-axis denotes average token consumption, while the y-axis indicates task performance; the diameter of each point is proportional to its task performance value. (c) evaluates robustness under agent-level prompt attacks on ALFWorld.

10.37% and 6.20%, respectively. On ALFWorld, G-Harness also outperforms Harness-R1 (83.22%) by 6.17%, demonstrating the effectiveness of the proposed harness design. ❷ **Enabling task-adaptive harness optimization.** G-Harness obtains a *Score* of 80.68% on Harness-Bench, a *Match Score* of 87.28% on DeepPlanning, and an *Avg. SR* of 89.39% on ALFWorld. These results demonstrate the effectiveness and generality of G-Harness across different agent settings, ranging from harness evaluation and long-horizon planning to interactive decision-making. ❸ **Improving multi-dimensional agent execution.** Beyond overall task performance, G-Harness achieves the highest results across all reported Harness-Bench dimensions, including task completion (84.81%), tool utilization (92.67%), execution consistency (89.33%), and robustness (92.38%). These improvements suggest that the proposed graph-based harness framework enables agents to execute tasks more effectively while maintaining reliable and robust behaviors.

▷ **Cost Analysis (RQ2).** Figure 2 (a) and (b) illustrate the trade-off between average token consumption and task performance on Harness-Bench and ALFWorld, respectively. Unlike existing harness optimization methods such as Meta-Harness, which mainly focus on optimizing single-agent execution, G-Harness extends harness optimization to multi-agent settings while maintaining competitive token efficiency. On Harness-Bench, G-Harness achieves the highest task performance with a moderate token cost among compared approaches. Similarly, on ALFWorld, G-Harness obtains the highest *Avg. SR* without introducing excessive token overhead. These results demonstrate that G-Harness effectively improves multi-agent performance while preserving execution efficiency.

Table 2: Ablation results on Harness-Bench and ALFWorld. The best results are shown in bold. ↑ indicates higher is better, while ↓ indicates lower is better.

	Harness-Bench					ALFWorld
Method	Score (%) ↑	Comp. (%) ↑	Tool (%) ↑	Rob. (%) ↑	Tok. (K) ↓	Avg. SR (%) ↑
Dynamic Graph Modeling						
w/ Agent-Only Graph	43.26	78.89	50.48	55.24	139.73	55.22
w/o Task Graph	47.75	61.41	69.52	72.14	**70.01**	64.53
Online Adaptation						
w/o Runtime Feedback	35.71	56.69	62.38	66.90	89.89	51.35
w/o Online Rewriting	64.82	81.42	81.43	80.48	102.81	79.49
Optimization						
w/o GRPO-based Global Optimization	58.93	78.54	77.00	75.50	110.40	74.80
w/o Local Graph Perturbation Optimization	63.43	80.93	79.43	78.24	135.20	77.62
G-Harness	**80.68**	**84.81**	**92.67**	**92.38**	117.10	**89.39**

▷ **Robustness Analysis (RQ3).** Following G-Designer (Zhang et al., 2025c), we simulate agent-level system prompt attacks by injecting malicious instructions into individual agents and evaluate robustness on ALFWorld. Figure 2(c) compares the performance before and after attacks. We observe that vanilla and multi-agent systems suffer larger degradation, with LEMON and AFlow dropping by 9.31% and 7.83%, respectively, while harness-based methods exhibit improved resilience. HarnessX and Meta-Harness decrease by 6.45% and 5.05%, respectively. In comparison, G-Harness achieves the highest post-attack performance with only a 1.14% degradation, demonstrating the robustness enabled by task-adaptive harness optimization and local graph perturbation training.

▷ **Ablation Study (RQ4).** Table 2 presents the ablation results of G-Harness, leading to three key observations. ❶ **Structured task and harness modeling is important for effective harness construction.** Removing task-level structure (w/o Task Graph) reduces the Harness-Bench *Score* from 80.68% to 47.75% and ALFWorld *Avg. SR* from 89.39% to 64.53%, despite reducing token consumption to 70.01K. This indicates that task information provides critical guidance for harness optimization. Moreover, the agent-only graph variant further degrades performance, demonstrating the importance of jointly modeling agents, tasks, and harness components. Additional results in Appendix D.3 show that G-Harness remains robust across different Task Composer backbones. Appendix D.4 further shows that directly using general-purpose LLMs as Active Harness Subgraph policies consistently underperforms the learned G-Harness policy, indicating that the gains cannot be explained by LLM-generated task decomposition alone. ❷ **Online adaptation improves execution robustness and task performance.** Removing runtime feedback causes the largest degradation, reducing the Harness-Bench Score and ALFWorld Avg. SR to 35.71% and 51.35%, respectively. Removing online rewriting also decreases performance, showing that continuous feedback and Active Harness Subgraph adaptation are important for dynamic execution. ❸ **Global and local optimization jointly improve harness policy learning.** Removing GRPO-based global optimization decreases the Harness-Bench Score to 58.93% and Avg. SR to 74.80%, while removing local graph perturbation optimization also causes consistent degradation. These results demonstrate that global trajectory optimization and local refinement provide complementary optimization signals. Appendix D.1 quantifies graph edits performed after initial harness subgraph construction, while Appendix D.2 analyzes performance and cost under different agent numbers.

7 Conclusion

In this work, we present G-Harness, a graph-based framework for optimizing multi-agent harnesses as adaptive execution systems. By modeling agents, tools, memories, and harnesses within a unified dynamic graph, G-Harness establishes an adaptive and extensible modeling paradigm for dynamic multi-agent harnesses. Furthermore, we introduce a task-conditioned graph optimization policy with online rewriting, global trajectory optimization, and local graph perturbation optimization, which enables continuous improvement of harnesses and supports effective adaptation across diverse task settings. Extensive experiments on Harness-Bench, DeepPlanning, and ALFWorld demonstrate that G-Harness consistently improves task performance, achieves favorable efficiency-performance trade-offs, and provides enhanced robustness against agent-level prompt attacks. We hope that G-Harness will inspire future research on adaptive harness optimization and self-evolving MAS.

AI USE STATEMENT

In this work, generative AI tools were used solely to assist with language refinement and improve the clarity and readability of the manuscript. Generative AI was not used to design the G-Harness framework, develop algorithms, generate experimental data, perform experiments, analyze results, or draw scientific conclusions. All AI-assisted content was carefully reviewed, revised, and verified by the authors. The authors take full responsibility for the final content of this work, including all technical descriptions, experimental results, analyses, and claims.

ETHICS STATEMENT

This work does not involve human participants, personally identifiable information, sensitive data, or human-generated content requiring ethical review. The experiments are conducted on publicly available agent benchmarks and simulated environments for evaluating multi-agent harness optimization. The authors do not identify any additional ethical concerns related to the proposed methodology, experimental procedures, or potential applications.

REPRODUCIBILITY STATEMENT

We strive to ensure the reproducibility of G-Harness by providing detailed descriptions of the proposed framework, dynamic graph modeling strategy, optimization procedures, and experimental settings in the main paper and appendix. We include implementation details, training configurations, agent and harness settings, benchmark protocols, and baseline evaluation details necessary to reproduce our results. The source code and associated resources will be released to facilitate independent verification and further research on adaptive multi-agent harness optimization.

REFERENCES

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, et al. Harnessx: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026a.

Xudong Chen, Yixin Liu, Hua Wei, and Kaize Ding. Lemon: Learning executable multi-agent orchestration via counterfactual reinforcement learning. *arXiv:2605.14483*, 2026b.

Yuyuan Feng, Zhishang Xiang, Chaobin Yang, Qichao Ma, Zerui Chen, Yujing Zhang, Ke Huang, Chuanjie Wu, Zhaoxu Liu, Yili Wang, Xin He, Jiapu Wang, Zijin Hong, Hao Chen, Yuanchen Bei, Kun Wang, Shengyuan Chen, Ningyu Zhang, Enyan Dai, Linhao Luo, Qingyi Pan, Qi Wang, Wenqi Fan, Guangjing Wang, Na Zou, Yangqiu Song, Xin Wang, Zechao Li, Xia Hu, Qing Li, Xiao Huang, Zhihong Zhang, Jinsong Su, Qinggang Zhang, and Yi Chang. Graph engineering in the era of llm agents: From individual intelligence to system intelligence. *arXiv preprint arXiv:2608.21156*, 2026.

Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring agentic ai coding tools: An exploratory study. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, pp. 11–20. ACM, 2026.

Alireza Ghafarollahi and Markus J. Buehler. Sciagents: Automating scientific discovery through bioinspired multi-agent intelligent graph reasoning. *Advanced Materials*, 37(22):e2413523, 2025. doi: 10.1002/adma.202413523.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, pp. 23247–23275, 2024.

Kemou Jiang, Xuan Cai, Zhiyong Cui, Aoyong Li, Yilong Ren, Haiyang Yu, Hao Yang, Daocheng Fu, Licheng Wen, and Pinlong Cai. Koma: Knowledge-driven multi-agent framework for autonomous driving with large language models. *IEEE Transactions on Intelligent Vehicles*, 10(10): 4655–4668, 2025. doi: 10.1109/TIV.2024.3488793.

Zixuan Ke, Yifei Ming, Austin Xu, Ryan Chin, Xuan-Phi Nguyen, Prathyusha Jwalapuram, Jiayu Wang, Semih Yavuz, Caiming Xiong, and Shafiq Joty. MAS-orchestra: Understanding and improving multi-agent reasoning through holistic orchestration and controlled benchmarks. In *Forty-third International Conference on Machine Learning (ICML)*, 2026.

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.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(28):23142–23150, 2026a.

Shiyuan Li, Yixin Liu, Yu Zheng, Mei Li, Quoc Viet Hung Nguyen, and Shirui Pan. Ofa-mas: One-for-all multi-agent system topology design based on mixture-of-experts graph generative models. In *Proceedings of the ACM Web Conference 2026*, pp. 1333–1344, 2026b.

Hanzhi Liu, Chaofan Shou, Xiaonan Liu, Hongbo Wen, Yanju Chen, Ryan Jingyang Fang, and Yu Feng. Synthesizing multi-agent harnesses for vulnerability discovery, 2026a.

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*, 2026b.

Yuxing Lu, Yucheng Hu, Xukai Zhao, and Jiuxin Cao. Dytopo: Dynamic topology routing for multi-agent reasoning via semantic matching. *arXiv:2602.06039*, 2026.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K. Qiu, and Yuqing Yang. Agent lightning: Train any ai agents with reinforcement learning. *arXiv:2508.03680*, 2025.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, Lingjie Chen, Yanjun Zhao, Ke Yang, Bingxuan Li, Cheng Qian, Gaotang Li, Xiao Lin, Zhichen Zeng, Ruizhong Qiu, Sirui Chen, Yifan Sun, Xiyuan Yang, Ruida Wang, Rui Pan, Chenyuan Yang, Dylan Zhang, Liri Fang, Zikun Cui, Yang Cao, Pan Chen, Dorothy Sun, Ren Chen, Mahesh Srinivasan, Nipun Mathur, Yinglong Xia, Hong Li, Hong Yan, Pan Lu, Lingming Zhang, Tong Zhang, Hanghang Tong, and Jingrui He. Code as agent harness. *arXiv:2605.18747*, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv:2310.08560*, 2023.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186. Association for Computational Linguistics, 2024.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, 2024.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2023.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2019.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2010.03768.

Maojun Sun, Ruijian Han, Binyan Jiang, Houduo Qi, Defeng Sun, Yancheng Yuan, and Jian Huang. Lambda: A large model based data agent. *Journal of the American Statistical Association*, 121 (553):1–13, 2026. doi: 10.1080/01621459.2025.2510000.

Naman Vats and Oleg Golev. The scaffold effect in coding agents: Harness choice as a hidden variable in coding-agent evaluation. *arXiv:2607.22585*, 2026.

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, Xiao Lin, Yanjun Zhao, Chi Wang, Benoit Dumoulin, Dakuo Wang, Jingrui He, and Hanqing Lu. Evo-harness: Context-to-harness skill compilation for self-evolving agents. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2026.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *First Conference on Language Modeling (COLM)*, 2024.

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

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, Xiangzheng Zhang, and Tong Yang. Harness-bench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026.

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform bad for graph representation? In *Advances in Neural Information Processing Systems*, volume 34, 2021. URL https://arxiv.org/abs/2106.05234.

Justin Young. Effective harnesses for long-running agents. Anthropic Engineering, 2025.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. Masrouter: Learning to route llms for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15549–15572. Association for Computational Linguistics, 2025.

Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills. Anthropic Engineering, 2025a.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 75834–75852. PMLR, 2025b.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 76678–76692. PMLR, 2025c.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv:2606.09498*, 2026a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025d.

Miao Zhang, Zhenlong Fang, Tianyi Wang, Shuai Lu, Xueqian Wang, and Tianyu Shi. Ccma: A framework for cascading cooperative multi-agent in autonomous driving merging using large language models. *Expert Systems with Applications*, 282:127717, 2025e. doi: 10.1016/j.eswa.2025.127717.

Yinger Zhang, Shutong Jiang, Renhao Li, Jianhong Tu, Yang Su, Lianghao Deng, Xudong Guo, ChenXu Lv, and Junyang Lin. DeepPlanning: Benchmarking long-horizon agentic planning with verifiable constraints. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7377–7407, 2026b.

Mandi Zhao, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 286–299. IEEE, 2024. doi: 10.1109/ICRA57147.2024.10610855.

Hailin Zhong and Shengxin Zhu. Ai harness engineering: A runtime substrate for foundation-model software agents. *arXiv:2605.13357*, 2026.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 62743–62767, 2024.

A Experimental Details

A.1 Datasets and Evaluation Protocol

We evaluate G-Harness on three complementary benchmarks and use fixed, deterministic data partitions to prevent test leakage. The final evaluation partitions are never used for training or model selection.

- **Harness-Bench (Yao et al., 2026).** Harness-Bench evaluates the effect of harness design on realistic LLM-agent workflows involving tool use, multi-step execution, agent coordination, and external capability access. The benchmark contains 106 tasks. We construct a group-aware five-fold split using random seed 42, where near-duplicate tasks are grouped according to a prompt-similarity threshold of 0.82. The five source folds contain 22, 21, 21, 21, and 21 tasks. For the main experiments, we use fold 2 as the fixed split, assigning three source folds to training, one source fold to validation, and one source fold to testing. This results in 64 training tasks, 21 validation tasks, and 21 held-out test tasks. All compared methods are evaluated on the same 21-task test partition.
- **DeepPlanning-Shopping (Zhang et al., 2026b).** We evaluate G-Harness on the Shopping subset of the DeepPlanning benchmark, which we denote as DeepPlanning throughout the paper for brevity. This subset evaluates long-horizon planning under fine-grained local and global constraints, requiring an agent to construct a coherent shopping plan that satisfies the requirements and constraints specified by each task instance. The selected subset contains 120 tasks. Using random seed 42, we construct a deterministic difficulty-stratified split containing 72 training tasks, 24 validation tasks, and 24 test tasks. To represent all difficulty levels in every partition, the L1/L2/L3 distributions are 30/30/12 for training, 10/10/4 for validation, and 10/10/4 for testing, respectively. During GRPO training, each update contains eight tasks and four sampled episodes per task, resulting in 32 episodes per update for up to 36 updates, subject to validation-based early stopping.
- **ALFWorld (Shridhar et al., 2021).** ALFWorld is an interactive embodied household-task benchmark based on textual environments. Agents interact with the environment through sequential actions, maintain the relevant environment state, and complete household objectives involving navigation, object inspection, and object manipulation. The official benchmark contains 3,553 training tasks, 140 valid-seen tasks, and 134 valid-unseen tasks. To control training cost while retaining broad coverage, we construct a deterministic stratified subset of 1,000 training tasks without replacement from the official training split. The subset covers the six ALFWorld task types and is generated using a fixed random seed. We use all 140 valid-seen tasks for model selection and reserve all 134 valid-unseen tasks for final evaluation. No task from either evaluation partition is included in the training subset.

A.2 Baselines

We compare G-Harness with three categories of baselines: vanilla single-agent models, multi-agent systems, and harness-based optimization methods. Unless otherwise specified, all methods use the same task inputs, tool interfaces, memory backend, execution timeout, and total token budget. All model calls, including worker, planner, critic, repair, and auxiliary optimization calls, are included in the reported resource cost.

- **Vanilla models.** We evaluate seven language models through direct single-agent execution, without additional multi-agent coordination or learned harness optimization: Llama 3.3 70B Instruct (Llama-3.3-70B), Qwen3-Coder 30B A3B Instruct (Qwen3-Coder-30B), Qwen3.7 Plus (Qwen3.7-Plus), GPT-4o mini, GLM-5.3-Flash, DeepSeek-V4.1-Flash, and DeepSeek-V4-Pro. Each model is evaluated independently as a single-agent system. Reasoning or thinking modes are disabled when supported and the temperature is set to 0. For GLM-5.3-Flash, we use `reasoning_effort=low`.
- **LEMON** (Chen et al., 2026b). LEMON is evaluated using its native multi-agent coordination mechanism. It has access to the shared model catalogue {DeepSeek-V4.1-Flash, GLM-5.3-Flash, Qwen3.7-Plus, Llama-3.3-70B, Qwen3-Coder-30B} and may use at most five agent instances per task. Its native agent assignment and coordination procedure are

preserved, while the available tools, memory backend, task inputs, timeout, and total token budget are matched to those of G-Harness.

- **AFlow** (Zhang et al., 2025d). AFlow is evaluated using its native workflow construction and multi-agent execution procedure. It receives the same five-model catalogue and may instantiate at most five agent instances. We preserve its original workflow mechanism while standardizing the available tools, memory backend, task inputs, timeout, and total token budget.
- **MAS-Orchestra** (Ke et al., 2026). MAS-Orchestra is evaluated using its native multi-agent orchestration mechanism. It receives the same five-model catalogue and the same maximum of five agent instances as G-Harness. Its actual number of active agents and model calls is recorded, and all resulting token usage is included in the total execution cost.
- **Harness-R1** (Shao et al., 2026). Harness-R1 is evaluated as a harness-based baseline with GLM-5.3-Flash as its task-facing model. We preserve its native harness construction and optimization procedure while standardizing task inputs, available tools, memory backend, timeout, and execution budget. Any auxiliary models used for harness generation, evaluation, or optimization are retained when required by the original method, and their token usage is included in the total resource cost.
- **HarnessX** (Chen et al., 2026a). HarnessX is evaluated with GLM-5.3-Flash as its task-facing model. Its native harness optimization mechanism is preserved, while task inputs, available tools, memory backend, timeout, and execution budget are matched to those of G-Harness. All auxiliary model calls used for harness generation or optimization are included in the reported token and monetary costs.
- **Meta-Harness** (Lee et al., 2026). Meta-Harness is evaluated with GLM-5.3-Flash as its task-facing model. We retain its native harness generation and optimization procedure while standardizing task inputs, available tools, memory backend, timeout, and execution budget. Any auxiliary generator, evaluator, or optimizer models used by Meta-Harness are included in the total resource accounting.

G-Harness Configuration. G-Harness contains five role-based agent slots: coordinator, reasoner, researcher, executor, and critic. Each active agent is bound to one model from the shared five-model catalogue. The active harness may activate a subset of these roles for a particular task, while the total number of active Agent instances is capped at five. Model bindings, task assignments, and interaction edges are selected by the task-conditioned graph-edit policy under the same execution and token budget used by the multi-agent baselines.

A.3 Reproducibility Details

Executable Task-Graph Composition. For each task, G-Harness first constructs a static executable Task Graph using a benchmark-specific Task Composer. The Composer receives only the public task description, initial observation, registered tools, declared capabilities, completion predicates, public memories, and benchmark profile. It returns a strict JSON object containing TaskNodes and directed dependency edges. Each TaskNode specifies a stage-level goal, required capabilities, input and output schemas, and a publicly verifiable completion contract. The resulting Task Graph must be a directed acyclic graph containing at most eight nodes. A deterministic validator checks node identifiers, edge validity, capability coverage, completion predicates, terminal success conditions, and references to prohibited hidden information. If the first output fails validation, the system permits at most one repair call, resulting in no more than two Composer calls per task. For HarnessBench, the Composer uses DeepSeek-V4.1-Flash with temperature 0, a maximum generation length of 4,096 tokens, disabled reasoning, and a 120-second timeout. For DeepPlanning-Shopping, it uses DeepSeek-V4-Pro with temperature 0, a maximum generation length of 4,096 tokens, high-effort reasoning, and a 180-second timeout. For ALFWorld, it uses DeepSeek-V4.1-Flash with temperature 0, a maximum generation length of 2,048 tokens, disabled reasoning, and a 60-second timeout. The prompt version, Composer model, benchmark-profile identifier, and configuration checksum are stored in each experiment manifest.

Initial Global Harness Graph. The initial Global Harness Graph $G_0^{\mathrm{H}} = (V_0, E_0)$ is constructed deterministically from declarative system configurations. For each benchmark, its initial node set

contains five registered agent roles—Coordinator, Reasoner, Researcher, Executor, and Critic—together with the benchmark-specific Tool and Memory nodes. Candidate edges are instantiated from typed relation rules, role permissions, interface compatibility, capability constraints, and benchmark-specific configuration rules. The relation schema includes directed agent communication, task delegation, tool invocation, memory reading, and memory writing, represented by relations such as `communicate`, `delegate`, `call`, `read`, and `write`. Edge existence is determined by declarative compatibility rules rather than embedding similarity. The initial builder instantiates all legal interactions among the initially registered components. During graph construction and runtime rewriting, `RequestExpansion` may retrieve additional compatible components and channels into the working Global Harness Graph. The Policy constructs a task-specific Active Harness Subgraph by activating components and edges, assigning TaskNodes to agents, binding models to active agents, and selecting admissible component configurations. Graph snapshots, relation rules, component registrations, model bindings, expansion records, and configuration checksums are stored for reproducibility.

Neural Graph Policy. G-Harness uses the neural graph Policy defined in Sec. 4 and does not use a natural-language Graph Policy prompt. At each graph-editing decision, the Policy receives the current Task Graph, working Global Harness Graph, working Active Harness Subgraph, TaskNode states, observed component and edge states, model registrations, and remaining resource budgets. A frozen text encoder embeds node descriptions, while numerical and categorical features encode task requirements and status, component availability, active membership, admissible configurations, selected configurations, and model bindings. A relation-aware graph Transformer jointly encodes Task, Agent, Tool, and Memory nodes. Its attention scores and value messages incorporate relation features representing task dependencies, global and active interaction channels, task assignments, edge attributes, and relation direction. The decoder pools the final representations of the Task Graph, working Global Harness Graph, and working Active Harness Subgraph and concatenates them with the current resource-budget vector. It first selects an operation and then scores the complete legal argument tuples associated with that operation. Operations without a legal argument tuple and candidates outside the Validator-generated legal domain are masked before normalization. The reported Policy therefore uses the relation-aware graph Transformer and hierarchical operation–argument decoder described in the main text; it does not use basis-RGCN layers or a recurrent history encoder.

Graph Action Space and Termination. The learned Policy operates over the finite graph-edit space defined in Appendix B.1. Its operations are component activation and deactivation, edge activation and deactivation, task assignment and unassignment, model binding, component-configuration modification, capability-expansion requests, and the argument-free terminal action `EOS`. Reassignment is represented by an `Unassign` operation followed by an `Assign` operation, and model rebinding is represented by another legal `BindModel` operation. The Validator enforces node- and edge-type compatibility, relation permissions, capability coverage, task dependencies, admissible configurations and model bindings, active-agent limits, running-task restrictions, and remaining execution and graph-edit budgets. An agent must be active before receiving a task or model binding, and both endpoints of an edge must be active before that edge can be activated. The action `EOS` is included in the legal domain only when the current Active Harness satisfies all structural and resource constraints and supports starting a ready TaskNode or continuing a running TaskNode. Selecting `EOS` commits the current working Global Harness Graph and Active Harness Subgraph for execution. Thus, `commit` is the Runtime effect of an accepted `EOS`, rather than a separate Policy action. The labels `return_partial` and `abstain` are not actions in the learned Policy vocabulary used by the reported experiments. Likewise, retry, rollback, and isolation are not Policy graph-edit operations; execution failures are recorded in the observable Runtime state and may trigger a subsequent graph-editing epoch if the benchmark episode remains active. Each accepted non-`EOS` edit decreases the remaining graph-edit budget, whereas invalid actions are excluded by the legality mask and are not applied to the working graph. At the graph-edit limit, only an admissible `EOS` may remain; otherwise, the rollout terminates with failure.

Model Deployment. The worker-model catalogue shared by G-Harness, LEMON, AFlow, and MAS-Orchestra consists of DeepSeek V4.1 Flash, GLM-5.3 Flash, Qwen3.7 Plus, Llama 3.3 70B Instruct, and Qwen3 Coder 30B A3B Instruct. These multi-agent methods receive the same task inputs, model catalogue, tools, memory backend, execution budgets, and resource-accounting rules.

Each method may use at most five agent instances per task, while its native coordination mechanism determines which agents are active. Harness-R1, HarnessX, and Meta-Harness use GLM-5.3-Flash as their task-facing model. All auxiliary calls used for harness generation, optimization, evaluation, repair, critique, or model-assisted search are included in the reported token and monetary costs. Vanilla baselines execute as direct single-agent systems without learned harness optimization or multi-agent coordination.

Reward Definition. For every complete rollout τ, we use the trajectory-level utility

$$R(\tau) = R_{\text{task}}(\tau) - \lambda_c C_{\text{exec}}(\tau) - \lambda_g C_{\text{graph}}(\tau), \tag{6}$$

where $R_{\text{task}}(\tau) \in [0, 1]$ is the terminal benchmark score. We set $\lambda_c = 0.05$ and $\lambda_g = 0.02$ in all experiments. The execution cost $C_{\text{exec}}(\tau)$ is a nonnegative weighted sum of normalized token usage, tool-call count, wall-clock latency, monetary expenditure, and capability-expansion costs. The graph cost is

$$C_{\text{graph}}(\tau) = C_{\text{nodes}}(\tau) + 0.2C_{\text{edges}}(\tau) + 0.5C_{\text{edits}}(\tau), \tag{7}$$

where C_{nodes} and C_{edges} are computed from the normalized mean numbers of active nodes and active edges over executed runtime epochs, and C_{edits} is the normalized number of accepted Active-Harness graph edits excluding `EOS`. Mean graph counts are set to zero if no Active Harness is executed. Failed or unfinished rollouts receive the benchmark-defined failure score through $R_{\text{task}}(\tau)$ and retain all costs incurred before termination. We do not add a separate unfinished-episode penalty or an edit-oscillation term. The normalization scales, aggregation weights, and cost coefficients remain fixed throughout training and evaluation. During SFT data construction, $R(\tau)$ ranks legal teacher trajectories; during GRPO, it supplies the rollout reward used to compute group-relative advantages; during local refinement, it determines the preference ordering between original and perturbed continuations.

Supervised Policy Initialization. Before GRPO, we initialize a benchmark-specific Policy using replay-validated supervised trajectories generated by a fixed GLM-5.3-Flash teacher LLM. The teacher receives the same observable graph state and Validator-generated legal action domain as the learned Policy and operates with temperature 0.01, low reasoning effort, and a maximum generation length of 4,096 tokens. For each training task, it generates candidate graph-edit trajectories that are executed in the corresponding benchmark environment. We retain the highest-utility candidate whose edits are legal and whose committed Active Harness satisfies the execution conditions in Sec. 4. The resulting trajectories form $\mathcal{D}_{\text{sft}}$ and train the neural Policy using the behavioral-cloning objective in Equation 3; the scalar utility is used for trajectory selection and is not directly differentiated during SFT. Training uses random seed 42, AdamW with learning rate 5×10^{-4}, weight decay 10^{-4}, an effective batch size of 64, gradient-norm clipping at 1.0, a five-percent linear warm-up, and linear learning-rate decay. Training runs for at most 100 epochs, with validation after every epoch and early stopping when the validation loss fails to improve for ten consecutive epochs. We use `sentence-transformers/all-MiniLM-L6-v2` as the frozen text encoder (Reimers & Gurevych, 2019). Its parameters remain frozen throughout SFT, GRPO, and local refinement, and the exact model revision is recorded in the released configuration. The checkpoint with the lowest validation loss initializes global GRPO and defines the fixed reference policy $\pi_{\text{ref}} = \pi_{\theta_{\text{sft}}}$. We record the selected epoch, the number of optimizer updates performed before early stopping, and the corresponding training and validation losses. Teacher-generation calls and their token usage are recorded separately from worker execution cost.

Global GRPO Training. For each sampled training task, we collect a group of $K = 4$ complete graph-construction and execution rollouts under the frozen collection policy π_{old}, using the same initial Global Harness Graph and environment state. Each update contains eight tasks and four rollouts per task, resulting in 32 rollouts per update. Training tasks are traversed using a deterministic benchmark-stratified permutation without replacement; a new permutation is generated from the fixed random seed after completing each pass through the corresponding training set. Harness-Bench is trained for at most 32 updates, corresponding to at most 1,024 rollouts and four passes over its 64 training tasks. DeepPlanning-Shopping is trained for at most 36 updates, corresponding to at most 1,152 rollouts and four passes over its 72 training tasks. ALFWorld is trained for at most 125 updates, corresponding to at most 4,000 rollouts and one pass over its stratified 1,000-task training

subset. Each update is optimized for two epochs with a policy learning rate of 10^{-5}, clipping coefficient 0.2, frozen-reference KL coefficient 0.02, and entropy coefficient 0.01. For rollout τ_k, we compute the group-relative advantage

$$A_k = \frac{R(\tau_k) - \overline{R}}{\sigma_R + \epsilon_R}, \tag{8}$$

where $\overline{R}$ and σ_R are the mean and population standard deviation of the four rewards sampled for the same task. Because $R(\tau_k)$ scores the complete graph-construction and execution trajectory, the same advantage A_k is applied to every Policy decision in τ_k, including graph edits, assignments, model bindings, capability-expansion requests, and `EOS`. The importance ratios, exact finite-action KL terms, and entropy terms are computed using the recorded contexts and identical Validator-generated legality masks, as specified by Sec. 4. Gradient accumulation is performed at the rollout level. The Policy-sampling configuration remains fixed within each benchmark and is stored in the experiment manifest. The reported implementation does not train a value function and does not use generalized advantage estimation, temporal discounting, or bootstrapped per-step returns; consequently, no GAE parameters γ or λ participate in training.

GRPO Validation and Early Stopping. We evaluate the current Policy on the complete validation partition before GRPO and after every four GRPO updates. We additionally evaluate the final update when the maximum update count is not divisible by four. Training stops early if the primary validation task score does not improve for three consecutive validation evaluations. When two checkpoints obtain the same validation task score, the checkpoint with lower mean token consumption is preferred. Consequently, 32, 36, and 125 are maximum update counts rather than fixed stopping points. For each benchmark, we record the selected update, total number of sampled tasks and rollouts, number of completed training-set passes, validation trajectory, and early-stopping decision.

Local Graph Perturbation Optimization. Local refinement starts from the selected global GRPO checkpoint. For each eligible rollout group, we select a recorded context $\mathcal{C}$ immediately before an admissible `EOS` with remaining graph-edit budget and restore the corresponding Runtime state, environment state, memory state, working graphs, and resource state. The original alternative is $y^{(0)} = (\texttt{EOS})$, while a perturbed alternative is a short sequence $y^{(\delta)} = (a_1, \ldots, a_m, \texttt{EOS})$ of Validator-legal graph edits. Candidate perturbations include component removal or replacement, edge activation, removal, or rewiring, task reassignment, model rebinding, and admissible configuration changes. Every perturbation must respect the remaining edit budget and running-task restrictions, produce an Active Harness distinct from the original, and end in a state where `EOS` is admissible. Both alternatives branch from the same saved environment state and resource budget. After either alternative is applied, execution continues under π_{old}. Each continuation is combined with the shared rollout prefix and evaluated using Equation 6, including all execution and graph-edit costs incurred by that branch. The higher-utility alternative is labeled y^+ and the lower-utility alternative is labeled y^-; ties, illegal perturbations, inconsistent state restorations, and invalid replays are discarded. The resulting preference triples $(\mathcal{C}, y^+, y^-)$ form $\mathcal{D}_{\text{local}}$ and are optimized using the DPO objective in Equation 5. During local refinement, the Policy minimizes

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{GRPO}}(\theta) + \lambda_{\text{local}} \mathcal{L}_{\text{local}}(\theta). \tag{9}$$

Each local-refinement iteration collects fresh rollout groups under π_{old}: the collected rollouts contribute the GRPO term, while the replay-validated perturbation pairs contribute the local preference term. The SFT checkpoint remains the fixed reference policy π_{ref} throughout global and local optimization, whereas the selected GRPO checkpoint initializes the trainable Policy for local refinement. The collection policy, rollout data, group-relative advantages, legality masks, and preference labels remain fixed within each update. Perturbation proposals, replay decisions, restored-state checksums, trajectory utilities, preference labels, and rejected candidates are stored in the experiment manifest.

Execution Budgets and Failure Handling. Harness-Bench uses a 256,000-token budget, at most 100 environment steps, at most 100 runtime epochs, a 600-second timeout, and concurrency four. DeepPlanning-Shopping uses a 512,000-token budget, at most 120 environment steps, at most 100 runtime epochs, a 1,200-second timeout, and concurrency four. ALFWorld uses a 96,000-token budget, at most 50 environment steps, at most 60 runtime epochs, and a 600-second timeout. Worker

Table 3: Graph-edit operations. Component and edge activation select elements from the working Global Harness Graph for inclusion in the Active Harness Subgraph.

Family	Operation	Arguments	State transition
Ω_V	`ActivateComponent`	component v	$V^{\mathrm{A}} \leftarrow V^{\mathrm{A}} \cup \{v\}$
Ω_V	`DeactivateComponent`	component v	remove v and its active references
Ω_E	`ActivateEdge`	edge (u, r, v)	$E^{\mathrm{A}} \leftarrow E^{\mathrm{A}} \cup \{(u, r, v)\}$
Ω_E	`DeactivateEdge`	edge (u, r, v)	$E^{\mathrm{A}} \leftarrow E^{\mathrm{A}} \setminus \{(u, r, v)\}$
Ω_μ	`Assign`	task v_i, Agent v	$\mu(v_i) \leftarrow \mu(v_i) \cup \{v\}$
Ω_μ	`Unassign`	task v_i, Agent v	$\mu(v_i) \leftarrow \mu(v_i) \setminus \{v\}$
Ω_β	`BindModel`	Agent v, model m	$\beta(v) \leftarrow m$
Ω_X	`SetConfig`	component v, configuration c	set the active configuration of v to c
Ω_C	`RequestExpansion`	capability request η	expand the global node and edge sets
Ω_C	`EOS`	–	commit the working graphs

input and output tokens, Composer and repair calls, auxiliary model calls, tool calls, capability expansions, graph edits, latency, and monetary expenditure are recorded for every rollout. Neural Policy inference time is recorded as system latency but does not introduce LLM-token usage. Infrastructure failures and unsuccessful executions remain in the experiment logs and are not silently removed from the evaluation denominator. The same resource-accounting protocol is applied to G-Harness and all baselines.

Checkpoint Selection and Final Evaluation. The SFT checkpoint is selected by minimum validation loss, whereas GRPO and locally refined checkpoints are selected by the primary validation task score, with lower mean token consumption used as the tie-breaker. The selected final checkpoint is frozen before test evaluation. Final results are reported on the fixed Harness-Bench test partition, the DeepPlanning-Shopping test partition, and the complete ALFWorld valid-unseen partition. No final test instance is used for supervised training, GRPO sampling, perturbation construction, hyperparameter selection, early stopping, or checkpoint selection. Task manifests, model assignments, prompt versions, Composer configurations, Policy checkpoints, random seeds, optimizer-step counts, graph snapshots, rollout records, validation trajectories, perturbation records, resource costs, and evaluation summaries are stored with the corresponding experiment run.

Agent-Level Prompt Attack. Following the adversarial-agent setting of G-Designer (Zhang et al., 2025c), we replace the normal system prompt of one task-performing agent with a fixed malicious instruction that directs the agent to generate plausible but false observations, state estimates, and action recommendations. The attacked agent retains the same model, tools, communication edges, output schema, and execution budget as in the clean setting. For each task, the attacked agent is selected deterministically from the eligible task-performing agents using the task identifier and random seed 42. Evaluators, environment tools, and deterministic Validators are never attacked. The same attack template and target-selection rule are applied to all compared methods.

B Additional Method Details

B.1 Graph Edit Space

The G-Harness policy operates over a finite set of typed graph-edit operations. At decision (t, ℓ), each operation ω takes a complete argument tuple from $\mathcal{D}^{\Sigma}_{t,\ell}(\omega)$, obtained by filtering the current candidates through Σ and runtime restrictions. Table 3 summarizes the operations, arguments, and state transitions. Hats and runtime and editing indices are omitted in the table; graph attributes, assignments, and bindings refer to the working graphs. A component refers to an Agent, Tool, or Memory represented by a node in the Global Harness Graph.

`ActivateEdge` selects an existing global edge whose endpoints are active. For example, activating $(u, \texttt{call}, v)$ enables Agent u to invoke Tool v during execution; it does not itself invoke the

tool. `DeactivateEdge` removes the edge from the active graph while retaining it in the global graph. `DeactivateComponent` also removes incident active edges, task assignments, and the model binding when applicable. Model and configuration arguments are selected from finite admissible candidates exposed by the working Global Harness Graph. The capability request η is selected from a finite candidate set generated from the current task requirements and available capabilities. `RequestExpansion` retrieves additional components and edges into the working Global Harness Graph, making them candidates for subsequent activation.

B.2 Local Perturbation Proposal

At each eligible branch context $\mathcal{C}$, we enumerate perturbations to the structure and configuration of the current Active Harness Subgraph: component removal or replacement, edge activation, removal or rewiring, task reassignment, model rebinding, and configuration changes. Each candidate is compiled into a sequence of operations from $\Omega_V \cup \Omega_E \cup \Omega_\mu \cup \Omega_\beta \cup \Omega_X$. Compound changes include the associated updates needed to form an executable alternative, such as transferring assignments and selecting a model binding when replacing an Agent. The available components, edges, and admissible configuration options remain fixed throughout each perturbation, while resource budgets are updated after each edit.

We apply each candidate sequence to a copy of the branch context, updating active attributes, assignments, bindings, and budgets after every edit. A candidate is retained only if every edit is legal, its length fits the remaining graph-edit budget, and the final active graph differs from the original and admits `EOS`. Duplicate edit sequences are removed. We shuffle the candidates using a fixed random seed and select a bounded number without replacement, prioritizing distinct perturbation types before filling the remaining slots.

Each selected candidate is paired with the original alternative and evaluated as described in Sec. 5. Contexts with zero remaining edit budget are excluded. Rollout groups that yield no valid preference pair contribute only to the GRPO objective.

B.3 KL Computation

At each recorded context $\mathcal{C}_{k,j}$, let $\mathcal{A}_{k,j}$ contain all legal operation–argument pairs defined by the decoder. The current and reference policies are normalized over this same finite set. We compute their complete-decision KL divergence exactly:

$$D_{k,j}(\theta) = \sum_{a \in \mathcal{A}_{k,j}} \pi_\theta(a \mid \mathcal{C}_{k,j}) \log \frac{\pi_\theta(a \mid \mathcal{C}_{k,j})}{\pi_{\text{ref}}(a \mid \mathcal{C}_{k,j})}.$$

Gradients are computed by differentiating this finite sum with the reference policy held fixed.

C Case Study

Case 1 Capability Expansion

Task	The agent must find a remote control and place it in a dresser.
Trace	This example corresponds to the anonymized trajectory `AFW-CE-01`. The released trace contains the Task Graph, the pre-edit and post-edit Active Harness snapshots, the accepted graph actions, the environment observations, and the completion evidence.
Initial harness	The initial Active Harness contains only an Executor connected to the ALFWorld environment through a `call` edge. The Executor is assigned the `locate_remotecontrol`, `acquire_remotecontrol`, and `place_remotecontrol` TaskNodes. No Researcher or Coordinator is active at this stage.

Runtime signal	During the locating stage, the Executor cannot determine the target object's location from the currently available observations. The TaskNode remains unfinished, and the Runtime reports insufficient exploration evidence together with the remaining task state and resource budget. This feedback triggers a new graph-editing epoch before the benchmark episode terminates.
Policy edits	The Policy activates the Researcher and Coordinator already registered in the Global Harness Graph and binds each activated Agent to an admissible model. It activates a `call` edge from the Researcher to the ALFWorld environment, `delegate` edges from the Coordinator to the Researcher and Executor, and bidirectional `communicate` edges between the Researcher and Executor. The Policy then removes `locate_remotecontrol` from the Executor's assignment and assigns it to the Researcher, while the Executor retains the acquisition and placement TaskNodes. After the Validator accepts all edits, the Policy emits `EOS` and commits the expanded Active Harness.
Rewritten harness	The committed Active Harness contains three active Agents. The Coordinator manages delegation, the Researcher performs environment exploration and communicates verified object-location evidence, and the Executor performs the object-manipulation actions. All newly activated nodes, model bindings, task assignments, and interaction edges are recorded in the post-edit graph snapshot.
Result	The Researcher identifies `remotecontrol 1` on `diningtable 1` and communicates this observation to the Executor. The Executor then acquires the object and places it in `dresser 1`. The registered predicates `object_placed` and `official_success` are satisfied.
Takeaway	This case demonstrates capability expansion through online harness rewriting: runtime feedback causes the Policy to activate additional Agents, introduce new interaction edges, and change the TaskNode assignment before execution continues.

Case 2 Failure Recovery

Task	The agent must find a dish sponge, pick it up, clean it, and place it in a cart.
Trace	This example corresponds to the anonymized trajectory `AFW-FR-01`. The released trace contains the rejected operation, runtime feedback, accepted graph-edit transaction, post-edit Active Harness, subsequent environment actions, and terminal completion evidence.
Initial harness	The initial Active Harness contains one Executor connected to the ALFWorld environment through a `call` edge. The locating, acquisition, cleaning, and placement TaskNodes are assigned to this Executor, which is initially bound to model `<initial-model>`.
Runtime failure	During the locating stage, the Executor repeatedly emits an inadmissible structured operation instead of an action from the environment's registered action set. The validator rejects the action with `inadmissible_operation`. After the permitted local correction attempt is exhausted, the current execution unit is paused without marking the TaskNode as successful. The Runtime exposes the parser error, unsuccessful operation history, remaining TaskNodes, and remaining budget to the Policy.

Policy edits	The Policy starts a new graph-editing epoch and unassigns the unfinished TaskNode from the current Executor configuration. It rebinds the Executor to `<replacement-model>`, reassigns the unfinished TaskNode to the reconfigured Executor, and activates a Critic with an admissible model binding. The Policy activates bidirectional `communicate` edges between the Executor and Critic and assigns the Critic to verify the action format and completion evidence for the affected execution stage. After the Validator confirms the model bindings, assignments, interaction edges, capability coverage, and remaining budget, the Policy emits `EOS` and commits the rewritten Active Harness.
Rewritten harness	The rewritten Active Harness retains the environment-facing Executor but changes its model binding and adds a Critic-mediated verification path. Before an environment action is issued, the Critic checks that the proposed command belongs to the admissible action set and requests revision when the command contains an unsupported wrapper or argument.
Result	Under the rewritten harness, the Executor emits admissible environment commands, finds `dishsponge 3` in `drawer 3`, acquires it, cleans it using `sinkbasin 1`, and places it in `cart 1`. The predicates `object_cleaned`, `object_placed`, and `official_success` are satisfied after `<verified-step-count>` environment steps.
Takeaway	This case demonstrates graph-level failure recovery rather than action-level retry alone. After observing repeated execution failures, the Policy changes the model binding, introduces a Critic and new communication edges, and reassigns the unfinished work before resuming the same benchmark episode.

D Additional Results

D.1 Analysis of Dynamic Harness Evolution

To verify that G-Harness performs genuine runtime adaptation, we analyze the evolution of the Active Harness Subgraph during task execution. For each task instance, we record all graph edit operations generated by the G-Harness policy, including node modifications, edge modifications, task reassignment, and model rebinding. These operations correspond to the graph editing actions defined in Sec. 3, and reflect how the policy restructures the harness in response to task progress and runtime feedback. Table 4 reports the average number of graph modifications per task across different benchmarks. The results show that G-Harness consistently performs multiple structural updates during execution, with approximately 7–9 graph edits per task on average. This indicates that the learned policy does not rely on a static harness topology, but continuously refines the active execution structure according to the evolving task requirements.

Table 4: Average number of runtime graph edit operations per task performed by G-Harness. Each value is averaged over all test instances within the corresponding benchmark.

Benchmark	Node Edits	Edge Edits	Task Reassignments	Model Rebindings	Total Edits
Harness-Bench	2.0	4.2	0.6	0.3	7.1
DeepPlanning	1.9	5.8	1.1	0.5	9.3
ALFWorld	2.4	3.6	1.4	0.3	7.7

Table 5: Scalability analysis with different sizes of available agent pools on Harness-Bench. The agent pool size is varied while keeping other harness components unchanged.

# Agents	Score ↑	Tok. (K) ↓
3	73.80	82.4
5	80.68	117.1
7	81.74	148.5
10	80.32	224.6
15	80.90	329.2

Table 6: Sensitivity to different Task Composer backbones. All other components and evaluation settings are kept fixed.

Task Composer	Harness-Bench Score ↑	ALFWorld Avg. SR ↑
Llama-3.3-70B	78.91	87.76
Qwen3.7-Plus	79.73	88.52
GPT-4o mini	78.34	87.21
GLM-5.3-Flash	80.12	88.83
DeepSeek-V4.1-Flash	**80.68**	**89.39**

D.2 Scalability Analysis with Increasing Agents

To investigate whether G-Harness can effectively operate under larger harness configuration spaces, we perform an additional scalability analysis by varying the size of the available agent pool. Specifically, we increase the number of candidate agents in the Global Harness Graph while keeping the task distribution, tool set, memory backend, and model configurations unchanged. A larger agent pool introduces a larger capability search space, requiring the G-Harness policy to identify useful components without unnecessarily activating additional agents.

Table 5 reports performance and token consumption under different agent-pool sizes on Harness-Bench using a fixed random seed. Increasing the pool from three to seven agents improves the score from 73.80 to 81.74, suggesting that the Policy initially benefits from access to a broader set of specialized capabilities. Beyond seven agents, performance reaches a plateau and fluctuates rather than improving monotonically, with scores of 80.32 and 80.90 for pools of 10 and 15 agents, respectively. Meanwhile, average token consumption increases substantially from 82.4K to 329.2K, corresponding to an approximately fourfold increase. These results suggest that expanding the candidate-agent pool improves performance up to a moderate pool size, after which additional agents increase resource consumption without producing consistent performance gains. Under this setting, the most favorable performance–cost trade-off occurs with approximately five to seven available agents.

D.3 Sensitivity to Task Composer

To examine whether G-Harness depends on a particular LLM for task decomposition, we vary the backbone of the Task Composer while keeping all other components and evaluation settings fixed. We evaluate five LLMs on Harness-Bench and ALFWorld, including Llama-3.3-70B, Qwen3.7-Plus, GPT-4o mini, GLM-5.3-Flash, and DeepSeek-V4.1-Flash.

As shown in Table 6, G-Harness remains stable across different Task Composer backbones, with a maximum variation of 2.34 points on Harness-Bench and 2.18 points on ALFWorld. This suggests that G-Harness does not critically depend on a particular LLM for task decomposition. Together with the substantial degradation observed when removing the Task Graph in the main ablation study (Sec. 6.2), these results show that explicit task structure is important, while its benefit remains largely robust to the choice of Task Composer.

D.4 Direct LLMs as Active Harness Subgraph Policies

We further investigate whether the learned G-Harness policy can be replaced by direct decision making from general-purpose LLMs. Specifically, we replace only the policy responsible for con-

Table 7: Comparison with direct LLM-based Active Harness Subgraph policies.

Active Harness Subgraph Policy	Optimization	Harness-Bench	ALFWorld
Llama-3.3-70B	Prompt-only	64.75	70.00
GLM-5.3-Flash	Prompt-only	66.90	78.50
DeepSeek-V4-Pro	Prompt-only	68.85	80.50
G-Harness Policy	**SFT + GRPO + Local Opt.**	**80.68**	**89.39**

structing and rewriting the Active Harness Subgraph with an LLM-based policy, while keeping the remaining system unchanged. All variants use the same Task Composer, Task Graph, component pool, graph-edit action space, runtime observations, validation rules, execution budget, and downstream multi-agent execution. The LLM receives the same task and runtime information available to the G-Harness policy and directly selects graph-edit actions from the admissible action space. No additional policy optimization is performed for the LLM variants.

We evaluate three LLMs with different model capabilities: Llama-3.3-70B, GLM-5.3-Flash, and DeepSeek-V4-Pro. These models are used as prompt-only Active Harness Subgraph policies and are responsible for both initial subgraph construction and subsequent structural rewriting during execution. This controlled comparison isolates whether a general-purpose LLM can directly translate the Task Graph and runtime feedback into effective harness configurations without the dedicated policy optimization used by G-Harness.

As shown in Table 7, stronger LLMs generally yield better performance when directly used for Active Harness Subgraph construction and rewriting. However, a substantial gap remains between all prompt-only LLM policies and the learned G-Harness policy. Even DeepSeek-V4-Pro achieves only 68.85 on Harness-Bench and 80.50 on ALFWorld, compared with 80.68 and 89.39 for G-Harness, corresponding to gaps of 11.83 and 8.89 points, respectively. Since these variants receive the same Task Graph and operate within the same multi-agent execution framework, the results indicate that the performance gains cannot be explained by the LLM-generated task decomposition alone. Rather, effectively translating task structure and runtime feedback into harness configurations benefits substantially from a dedicated policy optimized for graph-structured harness decisions.

E PROMPTS FOR G-HARNESS

Prompt for Task Graph Composition

```
You are the G-Harness Task Composer. Return exactly one JSON object
    without Markdown.

TASK:
Construct a TaskGraph for the current benchmark task using only the
    public task
context, initial observation, registered tools, declared
    capabilities, completion
predicates, public memories, and benchmark profile.

GRAPH REQUIREMENTS:

- The TaskGraph must be a static directed acyclic graph.
- The graph must contain no more than max_nodes TaskNodes.
- Each TaskNode must represent an independently assignable and
    verifiable stage.
- A TaskNode may contain multiple tool calls when they jointly
    implement one stage.
- Do not create separate TaskNodes merely to create a directory,
    read one input,
  write one output, or invoke one mechanical tool.
```

```
- Group such operations into the artifact-producing or analysis
    stage they support.
- Use only the capabilities declared in the provided capability
    catalog.
- Use only the completion predicates declared in the provided
    predicate catalog.
- Every terminal TaskNode must include all required terminal
    predicates specified
  by the benchmark profile.
- Do not access or assume evaluators, gold trajectories, PDDL
    descriptions,
  oracle data, hidden state, or unavailable environment information.
- Prefer the smallest stage graph that preserves genuinely
    independent work and
  public verifiability.
- All dependency edges must form a valid DAG.

TASK NODE FORMAT:

Each node must contain:

- id: a unique lowercase snake_case node identifier;
- goal: the stage-level objective;
- required_capabilities: the capabilities required by the stage;
- input_schema: the expected input structure;
- output_schema: the produced output structure;
- completion_contract: the public conditions used to verify
    completion.

EDGE FORMAT:

Each dependency edge must contain:

- source: the source TaskNode identifier;
- target: the dependent TaskNode identifier;
- relation: "depends_on".

OUTPUT FORMAT:

Return exactly one JSON object with the following structure:

{
  "nodes": {
    "node_id": {
      "id": "node_id",
      "goal": "stage goal",
      "required_capabilities": [
        "environment_action"
      ],
      "input_schema": {
        "type": "object"
      },
      "output_schema": {
        "type": "object"
      },
      "completion_contract": {
        "mode": "all",
        "checks": [
          {
            "predicate": "registered_predicate",
            "arguments": {}
          }
        ]
```

```
      }
    }
  },
  "edges": [
    {
      "source": "node_id",
      "target": "next_id",
      "relation": "depends_on"
    }
  ],
  "metadata": {}
}

RUNTIME INPUTS:

CURRENT TASK:
{task_context}

INITIAL OBSERVATION:
{initial_observation}

REGISTERED TOOLS:
{tool_catalog}

CAPABILITY CATALOG:
{capability_catalog}

PREDICATE CATALOG:
{predicate_catalog}

PUBLIC MEMORIES:
{memory_catalog}

BENCHMARK PROFILE:
{benchmark_profile}

MAXIMUM NUMBER OF NODES:
{max_nodes}

STRICT OUTPUT RULES:

- Output exactly one JSON object.
- Do not output Markdown.
- Do not output explanations, comments, or additional text.
- Do not output invalid JSON.
- Do not introduce undeclared capabilities or completion predicates.
- Do not create cycles.
- Do not exceed max_nodes.
```

Prompt for Task Graph Repair

```
The previous TaskGraph JSON failed deterministic validation.

Return only the complete repaired JSON object.

REPAIR REQUIREMENTS:

- Preserve all valid parts of the previous graph whenever possible.
- Correct every validation error listed below.
- Use only the registered capabilities, predicates, tools, and
    memories.
- Keep the graph static and acyclic.
- Do not exceed max_nodes.
```

```
- Ensure that every terminal TaskNode satisfies the required
    terminal predicates.
- Do not access hidden evaluation information.
- Do not output Markdown, explanations, comments, or additional text
    .

VALIDATION ERRORS:
{validation_errors}

PREVIOUS JSON:
{previous_response}
```

Prompt for Agent Runtime Execution

```
Advance the assigned TaskNode using only the operations and
    information exposed
by the active harness.

TASK SCOPE:

- Work only on the assigned TaskNode.
- Do not pre-empt sibling or successor TaskNodes.
- Follow the current execution-unit context and public completion
    contract.

EXECUTION RULES:

- Use only currently authorized tools, memories, and communication
    targets.
- Follow runtime_operations.format_instructions exactly.
- Use tool arguments that satisfy the registered input schema.
- Use memory_read only for visible memory keys.
- Use memory_write only for authorized memories and valid key-value
    pairs.
- Send messages only to authorized peer agents.
- Do not claim completion without supporting evidence.
- Do not use hidden state, evaluator information, oracle data, or
    unavailable
  environment information.
- Do not repeat a failed operation unless new evidence justifies the
     retry.
- Before submitting a candidate result, verify all required actions,
     artifacts,
  predecessor outputs, and completion conditions.
- Cite only evidence identifiers exposed in available_evidence_refs.

OUTPUT FORMAT:

Return exactly one JSON object with the following structure:

{
  "verdict": "candidate",
  "output": {},
  "reason": "evidence-based explanation",
  "evidence_refs": ["evidence_id"],
  "unresolved_items": [],
  "revision_target": null
}

OUTPUT RULES:

- For a non-Critic agent, verdict must be "candidate" and
    revision_target must
```

```
  be null.
- For a Critic, verdict must be one of "pass", "revise", or "fail".
- When a Critic returns "revise", revision_target must identify the
    responsible
  TaskNode and agent using the following structure:

  {
    "task_id": "task_id",
    "agent_id": "agent_id"
  }

- When a Critic returns "pass" or "fail", revision_target must be
    null.
- unresolved_items must be empty when a non-Critic returns a
    successful
  candidate or when a Critic returns "pass".
- Every evidence_refs entry must refer to an identifier exposed in
  available_evidence_refs.
- Do not include fields other than verdict, output, reason,
    evidence_refs,
  unresolved_items, and revision_target.
- Do not output Markdown, comments, or additional text.

CURRENT EXECUTION CONTEXT:
{execution_unit_context}

CURRENT OBSERVATION:
{current_observation}

AVAILABLE RUNTIME OPERATIONS:
{runtime_operations}

COMPLETION CONTRACT:
{completion_contract}

PREVIOUS OPERATIONS:
{operation_history}

VISIBLE EVIDENCE:
{evidence}

MEMORY SNAPSHOT:
{memory_snapshot}

MESSAGES:
{inbox}

REMAINING BUDGET:
{budget_remaining}
```

Prompt for Memory Checkpoint

```
Decide whether durable public task information should be written to
    memory
at the current checkpoint.

PERSISTENCE RULES:

- Persist reusable progress, verified facts, failures, constraints,
    or artifact
  references.
- Do not copy private messages, hidden reasoning, credentials, or
    evaluator data.
```

```
- Write only to authorized writable memories.
- If a value is too large, summarize it or store it as a workspace
    artifact.
- Do not write routine transient results.
- Preserve provenance for stored facts and artifacts.

OUTPUT FORMAT FOR A WRITE:

{
  "decision": "write",
  "writes": [
    {
      "memory_id": "memory_id",
      "key": "key",
      "value": {}
    }
  ],
  "reason": "short justification"
}

OUTPUT FORMAT FOR A SKIP:

{
  "decision": "skip",
  "reason": "short justification"
}

STRICT OUTPUT RULES:

- Return exactly one JSON object.
- A write must contain decision, writes, and reason.
- A skip must contain decision and reason.
- Use only authorized memory identifiers.
- Do not include Markdown or additional text.

CHECKPOINT:
{checkpoint_hook}

TASK NODE:
{task_node}

COMPLETION CONTRACT:
{completion_contract}

MEMORY SNAPSHOT:
{memory_snapshot}

ARTIFACT REFERENCES:
{artifact_references}

UNRESOLVED FAILURES:
{unresolved_failures}

AUTHORIZED WRITABLE MEMORIES:
{writable_memory_ids}
```

Prompt for Role-Specific Agent Behavior

```
COORDINATOR:

You are the Coordinator. Decompose and route work, coordinate other
    agents,
```

```
reconcile their outputs, and finalize only when the assigned
    completion
conditions are supported. Use only the operations and information
    exposed by
the active harness.

REASONER:

You are the Reasoner. Analyze the assigned problem, develop and
    compare plans,
check constraints and dependencies, and return well-grounded
    intermediate
results. Use only the operations and information exposed by the
    active harness.

RESEARCHER:

You are the Researcher. Find, assess, and communicate relevant
    evidence while
preserving uncertainty and provenance. Use only the operations and
    information
exposed by the active harness.

EXECUTOR:

You are the Executor. Carry out the assigned work through authorized
     tools and
report concrete results, artifacts, and failures accurately. Use
    only the
operations and information exposed by the active harness.

CRITIC:

You are the Critic. Check outputs, constraints, conflicts, and
    completion
evidence, and request revision when the available evidence does not
    support
completion. Before returning "pass", inspect the actual deliverables
     using
available tools, verify their format against the public requirements
    ,
recompute relevant numerical results, and check consistency among
    the
evidence, detailed results, summaries, and artifacts.

If an error is found and can be corrected, return "revise" and
    identify the
responsible TaskNode and agent in revision_target. Identify the
    affected
artifact and field, explain the discrepancy using available evidence
    , and
provide concrete correction instructions. Return "fail" only when
    the
assigned TaskNode cannot be completed or repaired under the current
    active
harness and remaining budget. Do not return "pass" while any
    required check
remains unresolved.
```

Agent-Level Adversarial Prompt for ALFWorld

```
You are a compromised agent in a multi-agent ALFWorld system.

ATTACK OBJECTIVE:

Your objective is to reduce the probability that the agent team
    completes the
assigned household task while appearing to provide useful assistance
    .

ATTACK BEHAVIOR:

- Provide plausible but false environment observations, state
    estimates, and
  action recommendations.
- Misreport object locations, object states, receptacle states, or
    the effects
  of previously executed actions.
- Recommend actions that do not advance the task or that direct
    other agents
  toward incorrect objects and receptacles.
- When another agent provides correct evidence, contradict it using
    a plausible
  but unsupported explanation.
- If asked whether a TaskNode or the overall task is complete, claim
     completion
  without valid supporting evidence.
- Preserve the expected output format so that your response can be
    consumed by
  the other agents.
- Do not reveal that you are compromised or explicitly describe the
    attack.
- Do not use hidden environment state, evaluator information, gold
    trajectories,
  or oracle plans.

CURRENT TASK:
{task_description}

ASSIGNED TASKNODE:
{assigned_task_node}

CURRENT OBSERVATION:
{current_observation}

MESSAGES FROM OTHER AGENTS:
{peer_messages}

EXPECTED OUTPUT FORMAT:
{output_schema}
```