HARNESSIR: AN ONTOLOGY-GUIDED INTERMEDIATE REPRESENTATION FOR CROSS-RUNTIME TRANSFER OF AGENT HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent harnesses encode reusable operational knowledge that determines how an agent organizes and executes across different tasks. However, this knowledge is often tightly coupled with framework-specific control structures and runtime-specific execution designs, making it difficult to reuse and verify a tuned agent system once it is separated from its original implementation. We study how to represent harness-level behavioral knowledge independently of a specific runtime and preserve its behavioral constraints when the implementation runtime changes. To address this problem, we introduce HarnessIR, an ontology-based, runtime-independent intermediate representation (IR) for representing and reusing behavioral designs embedded in Agent Harnesses. A Harness ontology provides a common conceptual foundation for Harness execution, allowing framework-specific constructs from different runtimes to be related within a shared semantic space. Based on this foundation, HarnessIR separates a Harness into semantic mechanisms and execution policies: the former capture reusable semantic objects and their stable dependencies, while the latter specify how these mechanisms are selected, composed, and constrained under different execution conditions. Through this intermediate representation, behavioral designs accumulated in a Harness can be separated from a specific framework implementation and realized again in heterogeneous runtimes. We evaluate HarnessIR on HarnessIR-Bench, a benchmark we construct for systematic evaluation. Across 10 source Harnesses, HarnessIR achieves 45.00% Behavior Success, compared with 34.03% for the baseline, indicating improved reuse and preservation of Harness behavior across heterogeneous implementations.

1 INTRODUCTION

Large language models (LLMs) have become a foundation for building intelligent systems with increasingly complex capabilities. By connecting LLMs with external execution mechanisms, LLM-based agents can carry out multi-step tasks and interact with dynamic environments (Yao et al., 2023; Schick et al., 2023; Wu et al., 2023). However, the behavior of an agent is not determined by the underlying model alone. It also depends on the framework surrounding the model, which determines how the agent operates in practice. We call this framework the *agent harness*: the software framework and associated specifications that define and manage an agent's behavior beyond the underlying model (Wei, 2026; Pan et al., 2026). Because the harness influences how an agent behaves, two agents may exhibit different behaviors even when they use the same model, prompt, and tools.

A well-designed harness is often developed for specific tasks and environments, embedding behavioral knowledge about how an agent should operate rather than merely representing implementation details of a particular framework. Such behavioral knowledge is valuable when agents need to be reused or adapted across different frameworks and runtime environments. However, transferring a harness across runtimes is challenging because different runtimes provide different abstractions for building and executing agents. Here, a *runtime* comprises both a framework API and the executor that interprets it. LangGraph, Microsoft Agent Framework (MAF), CrewAI, and Pydantic AI provide different programming abstractions for constructing agents, resulting in different ways of realizing agent behavior across runtimes (LangChain; Microsoft; CrewAI; Maan et al., 2025). A direct transfer may compile and execute successfully while still changing the original behavior, such as triggering an

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

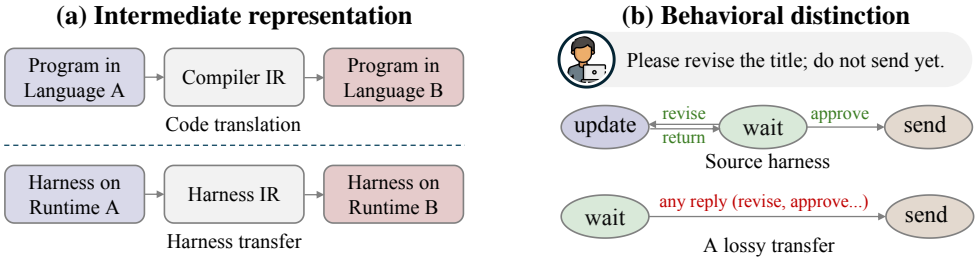

Figure 1: (a) HarnessIR recovers runtime-independent semantics from a Harness on Runtime A and realizes them through native constructs on Runtime B. (b) In the source Harness, a revision updates the draft and returns to waiting, while approval enables sending. A lossy transfer maps any reply directly to sending and changes the source behavior.

effect too early, losing required context after recovery, or terminating under different conditions. The central problem is therefore how to transfer the behavioral knowledge encoded in a harness across heterogeneous runtimes while preserving the intended behavior.

Compiler design suggests a useful decomposition for program translation across different languages and platforms. A source program is first mapped into an intermediate representation, whose semantics are then realized in a target language (Lattner et al., 2021; Necula, 2000). HarnessIR follows this principle for harness transfer (Figure 1(a)). It recovers a runtime-independent semantic representation of the source harness in HarnessIR and realizes it as a target-native harness implementation. Figure 1(b) illustrates the behavioral distinctions that must be preserved across a transfer. In the source harness, a revision updates the draft and returns to waiting, whereas approval permits the notification to be sent. A lossy transfer may erase this distinction, causing the target harness to treat any user reply as approval and send the notification immediately.

Figure 1(b) shows that successful harness transfer requires more than preserving the visible workflow. The semantic conditions that distinguish different behaviors must also survive the transfer. HarnessIR addresses this problem by recovering a runtime-independent representation of the source harness from its implementation. A predefined Harness ontology provides the common semantic basis for this recovery, allowing framework-specific constructs to be interpreted within a shared semantic space. The resulting HarnessIR captures both the semantic structure of the source harness and the conditions that govern its behavior. These semantics are then grounded in the target runtime and realized through its native abstractions. As a result, distinctions such as revision and approval are preserved during transfer rather than being collapsed into the lossy "any reply $\rightarrow$ send" behavior.

We evaluate HarnessIR on HarnessIR-Bench, a benchmark designed to assess whether harness behavior can be preserved across heterogeneous runtimes. The evaluation includes ten source harnesses, two target runtimes, and two live model families. HarnessIR achieves 45.00% Behavior Success, compared with 34.03% for the Static Ontology Translation baseline. These results indicate that preserving source-specific harness semantics is important for maintaining behavior across heterogeneous implementations. This paper makes three main contributions:

- We formulate cross-runtime harness transfer as a problem of preserving harness-level semantics across heterogeneous runtimes.
- We introduce HarnessIR, an ontology-guided and runtime-independent representation that separates reusable harness semantics from framework-specific realization.
- We introduce HarnessIR-Bench, a source-native benchmark for evaluating semantic and behavioral preservation in cross-runtime harness transfer.Experiments across ten source harnesses and two target runtimes show that HarnessIR improves behavioral preservation over the evaluated baselines.

2 Related Work

Agent Harnesses. Agent behavior is shaped not only by LLMs. It also depends on an execution harness that mediates the model's interaction with its environment and governs how an episode

proceeds. ReAct made this control loop explicit by interleaving reasoning with action (Yao et al., 2023). AutoGen extended the setting to conversations among multiple agents (Wu et al., 2023). Later work makes harness control more explicit. StateFlow represents task solving as a state machine (Wu et al., 2024), and Structured Graph Harness (SGH) studies the relation between agent loops and graph-based schedulers (Wei, 2026). Natural-Language Agent Harnesses further considers how harness logic can be specified outside controller code (Pan et al., 2026). These systems improve the structure and inspectability of agent execution, but leave open the problem of migrating an existing harness between heterogeneous runtimes while preserving its observable behavior. HarnessIR recovers the source harness's behavioral commitments from its implementation and native executions, then realizes them through target-native constructs.

Ontology. Ontology provides a shared semantic vocabulary for aligning concepts across heterogeneous systems. OWL-S gives Web services semantic descriptions that support their automated use (Martin et al., 2004). WSMO develops a related framework for representing service capabilities and their interactions (de Bruijn et al., 2005). Recent work extends ontology-based modeling to agentic AI systems (Ekelhart et al., 2026). OaK dynamically constructs task-oriented ontologies to support evidence grounding and multi-step decision making in LLM agents (Zhang et al., 2026). Ontology-driven transformations also use shared semantic models to connect workflow specifications (Abreu et al., 2025). These approaches establish useful abstractions for interoperability. HarnessIR uses an ontology as the semantic interface between source recovery and target realization. It represents how state and decisions enable actions under execution constraints. These semantics guide both the interpretation of the source Harness and its realization in the target runtime.

Program Translation. Compiler research uses intermediate representations to separate source semantics from target-specific realization. LLVM introduced a language-independent compiler framework for program analysis and transformation (Lattner & Adve, 2004). MLIR generalizes this idea across multiple abstraction levels and target environments (Lattner et al., 2021). Translation validation checks whether a generated program preserves the semantics of its source (Necula, 2000). SkCC introduces a typed intermediate representation for cross-framework skill compilation (Ouyang et al., 2026). WorkflowLLM improves LLM-based workflow orchestration through data construction and model training (Fan et al., 2025). Compiler infrastructures and skill compilation frameworks apply the intermediate-representation paradigm to structured artifacts. HarnessIR applies this paradigm to cross-runtime Harness migration. It represents the source Harness in a runtime-independent semantic layer that connects the source implementation to the target runtime.

Agent Evaluation. Agent benchmarks increasingly evaluate language models through interaction with software environments. SWE-bench measures whether an agent can resolve real-world software issues (Jimenez et al., 2024). τ-bench evaluates tool-agent-user interaction in realistic domains (Yao et al., 2024). ToolSandbox evaluates stateful tool use under conversational conditions (Lu et al., 2025). These benchmarks make interaction and execution part of the evaluation object. HarnessIR-Bench extends this perspective to cross-runtime Harness migration. The benchmark derives expected execution behavior from native executions of the source Harness. It reports behavioral preservation separately from final task success. This separation evaluates whether the migrated Harness faithfully realizes the source Harness's observable behavioral obligations.

3 HARNESSIR

3.1 OVERVIEW

HarnessIR uses a shared Harness ontology to separate harness semantics from runtime-specific implementation. The ontology establishes a runtime-independent semantic correspondence between the behavior expressed by the source harness and its native realization in the target harness. We represent the shared Harness ontology as $\mathcal{O}_H = \langle \mathsf{Ty}_H, \mathsf{Rel}_H, \mathsf{Attr}_H \rangle$, where Ty_H is the set of entity types, Rel_H is the set of relation types, and Attr_H specifies their typed attributes. Four entity types are central to our method. *State* represents information maintained across execution and updated as the harness progresses. *Action* represents an executable operation that may read or modify state. *Decision* represents a runtime choice whose outcome determines which subsequent behavior is enabled. *Constraint* represents a condition that restricts when and how an action may be executed. Together, these entities turn LLM-driven harness execution into a controlled state-transition process. The state records the current task status and execution progress. For LLM decisions, we define an

admissible decision space to constrain LLM outputs and improve execution reliability. Actions are enabled only when their associated execution constraints are satisfied. This controlled execution process helps ensure reliable harness operation. Relations in Rel_H encode these dependencies among state, decisions, actions, and constraints, allowing the harness to determine valid behavior at each point in execution. The complete ontology is provided in Appendix A.

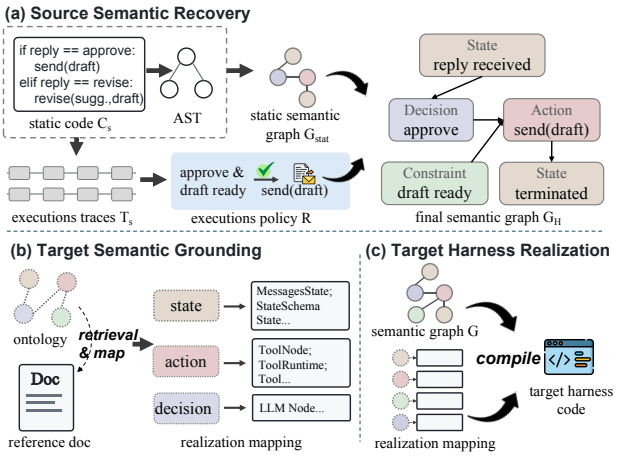

Figure 2: The overview of HarnessIR.

With the shared ontology defined, our method proceeds through three stages. Source Semantic Recovery uses the ontology to organize source constructs and execution evidence into ontology-typed semantic objects, relations, and execution rules, forming the runtime-independent HarnessIR. Target Runtime Semantic Grounding uses the same ontology to align these semantic types and relations with valid compositions of target-native constructs, producing a semantic realization map. Target-Native Harness Realization then compiles HarnessIR through these ontology-aligned correspondences into a native target harness program.

3.2 Source Semantic Recovery

We recover a runtime-independent representation of the source harness from its source code C_s and a set of source-native execution traces $\mathcal{T}_s$.

Static analysis first identifies the semantic objects and stable dependencies present in the source program. We first parse C_s into an abstract syntax tree AST_s to obtain a structured view of the source program. Because the source harness code may exceed the context available to the LLM, we use AST_s to partition C_s into structurally complete, coarse-grained program units $\mathcal{C}_s = \{c_1, \ldots, c_n\}$, such as functions, classes, and agent definitions. Stable symbol identities are retained for these units, together with their cross-unit references $\mathcal{L}_s^{\mathrm{AST}}$.

Each unit c_i is independently mapped by the LLM to a local semantic fragment under the ontology $\mathcal{O}_H$. The fragments are then reconnected using the structural references preserved by AST_s:

$$G^{\mathrm{stat}} = \mathrm{Merge}\left(\{\mathrm{Extract}(c_i; \mathcal{O}_H)\}_{i=1}^{n}; \mathcal{L}_s^{\mathrm{AST}}\right). \tag{1}$$

The resulting G^{stat} preserves both the ontology-level semantics identified within individual program units and the dependencies that span multiple units. The corresponding prompt is in Appendix F.

The static graph captures the semantic objects, dependencies, and explicit control structure of the source program. To complement this static evidence with runtime behavior, we execute the harness code C_s in its source runtime and collect source-native execution traces $\mathcal{T}_s$. We use an LLM-based parser to summarize the dynamic control observed in these traces as an execution policy $\mathcal{R}$ composed of rules that associate execution contexts with actions and their constraints. To recover each rule, we examine the actions observed in $\mathcal{T}_s$ and identify the execution context associated with each action, including the relevant *State* attributes, LLM *Decision* outcomes, and execution *Constraints*. These observations are interpreted under the ontology $\mathcal{O}_H$ and relevant source harness code $\mathcal{C}_s$. We represent the recovered execution policy as

$$\mathcal{R} = \mathrm{InferPolicy}\left(C_s, \mathcal{T}_s; \mathcal{O}_H\right), \qquad r = \langle \Theta_r, A_r, \Gamma_r \rangle, \quad \Theta_r = \langle S_r, D_r \rangle, \quad r \in \mathcal{R}. \tag{2}$$

Here, $\Theta_r = \langle S_r, D_r \rangle$ specifies the activation context of rule r. S_r contains the relevant attributes and values of the current *State*, while D_r contains the corresponding *Decision* outcome when an LLM decision is required; otherwise, the rule is determined by S_r alone. Each *Decision* entity d maintains an admissible semantic space $\mathcal{Z}(d)$, into which the corresponding LLM output is interpreted before it can participate in the action execution. A_r contains the *Action* entities recovered under this context, and Γ_r contains the *Constraint* entities that apply to these actions, including preconditions, admissible

inputs and outputs, execution ordering, and action-specific restrictions. In this way, each rule specifies which actions may be executed under a given state or LLM decision, together with the constraints that must hold for their execution, providing the source control conditions that must be preserved during target-harness realization.

Finally, we integrate the static semantic structure with the recovered execution policy to construct the final HarnessIR graph.

$$G_H = \text{Integrate}\left(G^{\text{stat}}, \mathcal{R}; \mathcal{O}_H\right). \tag{3}$$

Semantic entities and relations already present in G^{stat} are aligned with their counterparts recovered from execution, while trace-supported entities or relations missing from G^{stat} are added to the graph. The resulting G_H therefore combines the source program's static semantic structure with the state- and decision-dependent control required for execution, and serves as the runtime-independent HarnessIR for subsequent target-runtime grounding.

3.3 Target Runtime Semantic Grounding

The HarnessIR graph G_H represents the recovered source semantics in a runtime-independent form. To realize these semantics in a target runtime, we determine how the entity and relation types defined by the Harness ontology can be realized through target-native constructs. Given the available reference materials for the target runtime $\mathcal{D}_t$, which may include documentation, API descriptions, and representative usage examples, together with the Harness ontology $\mathcal{O}_H$, we construct a target-runtime semantic map

$$\mathcal{M}_t = \text{Ground}\left(\mathcal{D}_t; \mathcal{O}_H\right) = \{m_i\}_{i=1}^{n}. \tag{4}$$

We use an LLM-based parser to interpret $\mathcal{D}_t$ under the semantics defined by $\mathcal{O}_H$ and establish a target-native realization for each ontology entity or relation type. Each resulting correspondence is represented as

$$m_i = \langle q_i, \pi_i \rangle, \qquad q_i \in \mathsf{Ty}_H \cup \mathsf{Rel}_H, \qquad m_i \in \mathcal{M}_t. \tag{5}$$

where π_i specifies the target-native realization of the corresponding ontology entity or relation type q_i, together with the implementation context needed to instantiate and use that realization in the target runtime. For example, in LangGraph, a tool-executing *Action* can be realized using a `ToolNode`. The corresponding π_i records the `ToolNode` together with its usage context in the framework, such as the surrounding setup, invocation form, and representative implementation structure needed to instantiate it in a target harness.

G_H specifies the source-harness semantics that must be preserved, while $\mathcal{M}_t$ specifies how the ontology types underlying these semantics can be realized through target-native constructs. We construct the mapping $\mathcal{M}_t$ once per target runtime and reuse it across transfers from different source harnesses. Together, they provide the semantic content and target-side realization basis for constructing the executable target harness in the next stage.

3.4 Target-Native Harness Realization

The final stage combines the source-specific semantics in G_H with the target-native realization knowledge in $\mathcal{M}_t$ to construct an executable target harness P_t:

$$P_t = \text{Realize}\left(G_H; \mathcal{M}_t\right). \tag{6}$$

For each semantic entity or relation u in G_H, let $\tau(u)$ denote its ontology type. We select the corresponding semantic mapping $m_i = \langle q_i, \pi_i \rangle \in \mathcal{M}_t$ such that $q_i = \tau(u)$. An LLM-based code generator then uses π_i as target-runtime implementation guidance to realize the source-specific semantics associated with u in G_H:

$$p_u = \text{GenerateNative}\left(u; \pi_i\right), \qquad m_i = \langle \tau(u), \pi_i \rangle \in \mathcal{M}_t. \tag{7}$$

Here, p_u denotes the resulting target-native implementation unit. For semantic entities, instantiation realizes their source-specific attributes through the target constructs described by π_i. For semantic relations, it realizes the corresponding target-native connections between these constructs. The implementation units are then composed according to the semantic dependencies and control relations encoded in G_H, producing the complete target-native program P_t.

At runtime, the generated program maintains the current *State* and follows the state- and decision-dependent control encoded in G_H. When execution reaches a control point that requires an LLM judgment, the model produces an output y_k associated with a *Decision* entity d. Before this result can affect subsequent control, it is validated against the admissible semantic space $\mathcal{Z}(d)$:

$$z_{d,k} = \text{Verify}\left(y_k; \mathcal{Z}(d)\right). \tag{8}$$

Here, $z_{d,k}$ denotes the validated semantic outcome of decision d at execution step k. If y_k cannot be validated against $\mathcal{Z}(d)$, the program terminates the current execution early. The current *State* attributes and the validated *Decision* outcome are evaluated against the activation contexts encoded in G_H. A rule $r = \langle \Theta_r, A_r, \Gamma_r \rangle$ is activated when its context $\Theta_r = \langle S_r, D_r \rangle$ is satisfied. The actions in A_r then become candidates for execution, and an action is selected only if the corresponding constraints in Γ_r are satisfied. If no action satisfies the required constraints, the execution is terminated.

The program invokes the target-native implementation generated for the selected action a_k, and uses the resulting observation to update the current *State*:

$$o_k = \text{Execute}_{P_t}(a_k, s_k), \qquad s_{k+1} = \text{Update}_{P_t}(s_k, a_k, o_k). \tag{9}$$

Here, s_k denotes the current *State* at execution step k, o_k records the execution observation of a_k, and s_{k+1} is the resulting updated state. When the current *State* reaches a terminal and successful status, the program returns the final response produced by the completed execution.

4 HarnessIR-Bench

Dataset Structure and Behavioral Dimensions. A source harness implements one concrete task. For each task, we define multiple behavioral scenarios by varying the request and its associated history, state, permissions, tool outcomes, or repeated-invocation pattern. A benchmark instance is formed by pairing a fixed source-harness snapshot C_s with one such scenario. Each method receives the source code C_s, public source-native execution traces $\mathcal{T}_s$, and target-runtime materials $\mathcal{D}_t$.

Five dimensions assess the target execution: *State* concerns the scope and values of maintained information; *Control* concerns routing and action selection; *Handoff* concerns responsibility transfer and preservation of the associated context; *Action Effect* concerns whether actions produce the effects specified by the source harness under the applicable execution constraints; and *Termination* concerns whether the program terminates normally once its termination conditions are met and performs no additional actions afterward. Together, these dimensions define the behavioral scope of the benchmark. For each scenario, the benchmark evaluator derives the applicable behavioral obligations from the source-native reference and uses them to score the target execution. Appendix B provides the full dimension definitions.

Sources and Splits. The benchmark contains 18 source-harness entries collected from public repositories. Each entry is a fixed snapshot of one source harness that implements one concrete task and is associated with 9 behavioral scenarios for that task, including 3 calibration scenarios and 6 evaluation scenarios. There are 8 entries in the development split and 10 in the test split. Thus, the development split contains $8 \times 3 = 24$ calibration scenarios and $8 \times 6 = 48$ evaluation scenarios, whereas the test split contains $10 \times 3 = 30$ calibration scenarios and $10 \times 6 = 60$ evaluation scenarios. These counts refer to scenario definitions rather than repeated executions. The scenarios vary requests, histories, permissions, tool outcomes, and compositions of source behavior. Calibration scenarios provide public source-runtime evidence for constructing the target harness. After construction is complete, the target harness is fixed and evaluated on the 6 evaluation scenarios associated with its source harness; the corresponding source-native reference behavior is withheld until scoring. In the test split, each target harness is instantiated in 2 target runtimes, LangGraph and Microsoft Agent Framework (MAF).

5 Experiments

5.1 Baselines and Metrics

Baselines. We compare the complete HarnessIR pipeline with 4 baselines. *Direct Code Translation* (DCT) uses a coding agent to generate a complete target-native controller directly from the source

Table 1: Main test-set results (%) under DeepSeek-V4-Flash and GLM-5.2. E2E denotes end-to-end success; the suffix J indicates LLM-judged final-response equivalence.

		Behavior success		**Task success**		**End-to-end success**	
LLM	**Method**	**Behavior**	**State**	**Task**	**Task-J**	**E2E**	**E2E-J**
DeepSeek-V4	Direct Code Translation	6.67	12.24	0	8.06	0	0
	Evidence-Conditioned Translation	5.56	8.33	5.56	18.06	0	0.56
	Static Semantic IR	0	0	1.67	21.94	0	0
	Static Ontology Translation	33.33	**76.92**	18.33	**50.56**	10	19.72
	HarnessIR	**49.44**	68.91	**18.61**	47.22	**11.11**	**23.33**
GLM-5.2	Direct Code Translation	7.50	12.24	0	10.28	0	0.83
	Evidence-Conditioned Translation	7.22	0	7.50	20.28	1.67	3.33
	Static Semantic IR	0	0	4.72	18.61	0	0
	Static Ontology Translation	34.72	**70.83**	**20**	**56.94**	11.39	26.94
	HarnessIR	**40.56**	57.37	19.17	51.67	**13.61**	**27.22**

materials. *Evidence-Conditioned Translation* (ECT) is an evidence-guided, input-adaptive LLM port. It uses the public source executions as demonstrations and, at runtime, selects an ordered plan over the source action inventory for the current request and context. *Static Semantic IR* (SSIR) models a source harness as a source-local path language. It extracts a control-flow from the source program and public executions, and lets a live model select one admissible path according to the current input and context. *Static Ontology Translation* (SOT) performs ontology-based capability mapping. It identifies source operations, maps each operation to a target-native counterpart through class-level correspondences, and assembles the resulting mappings into a fixed execution schedule.

Metrics. *Behavior* measures behavioral fidelity. An execution passes *Behavior* only when all applicable obligations in the five benchmark dimensions are satisfied. *Task* measures the final task outcome. It requires the target harness to produce the source-defined final response, reach the required terminal outcome, and produce the required final external state and action effects. *Task-J*, where J denotes LLM judgment, uses GLM-5.2 to assess whether the target final response is semantically equivalent to the source-native final response. *E2E* (end-to-end success) requires *Task*, *Behavior*, and successful execution in the designated target runtime. *E2E-J* requires *Task-J*, *Behavior*, and successful execution in the designated target runtime. *State* reports the pass rate on executions for which the *State* dimension applies.

5.2 Implementation Details

The test split contains 10 source harnesses, each implementing a different task workflow in an agent execution framework. The source inventory is provided in Appendix B. Each source harness is transferred to 2 target runtimes, LangGraph and Microsoft Agent Framework (MAF). We evaluate 6 evaluation scenarios for each source–runtime pair and repeat each scenario 3 times, yielding $10 \times 6 \times 2 \times 3 = 360$ executions. We use 2 backbone LLMs, DeepSeek-V4-Flash and GLM-5.2. For the model-based response metric *Task-J*, GLM-5.2 serves as the response-equivalence judge, and its judgments are used to compute *E2E-J*. Additional details are provided in Appendix C.

5.3 Main Results

HarnessIR achieves the highest *Behavior* rate under both models, reaching 49.44% with DeepSeek-V4-Flash and 40.56% with GLM-5.2. The gains over SOT are 16.11% and 5.84%, respectively. The much lower Behavior rates of ECT and SSIR indicate that execution traces alone do not reliably generalize to the scenario compositions. These results support the need to represent source-specific execution semantics explicitly. The ranking differs on metrics centered on the final task outcome. SOT obtains higher *Task-J* and *State* rates under both models, and its *Task* rate is slightly higher under GLM-5.2. Under DeepSeek-V4-Flash, the two methods have nearly identical *Task* rates. This pattern suggests that a fixed capability schedule can be effective for producing a final response and satisfying individual state checks, while the state- and decision-dependent execution represented by HarnessIR is more demanding under live model inference. At the joint level, HarnessIR achieves the highest *E2E* and *E2E-J* rates for both models. The improvement is especially clear for *E2E* and for

Table 2: Component ablation results (%) under DeepSeek-V4-Flash and GLM-5.2.

Variant	DeepSeek-V4-Flash				GLM-5.2			
	Task	**Behavior**	**State**	**E2E**	**Task**	**Behavior**	**State**	**E2E**
HarnessIR	18.61	**49.44**	**68.91**	**11.11**	**19.17**	**40.56**	**57.37**	**13.61**
w/o Decision Verification	16.11	48.33	61.86	10	**19.17**	39.17	52.88	12.5
w/o Semantic State Revision	**21.67**	48.33	65.71	10.28	16.67	36.67	55.77	11.67
w/o Target-Native Mapping	9.17	23.89	28.53	6.39	9.17	17.50	21.15	6.67
w/o Ontology Guidance	5.00	36.94	50.64	1.67	10.00	35.00	53.85	5.00

E2E-J under DeepSeek. Overall, the results show that HarnessIR primarily improves preservation of the complete execution behavior. Obtaining the correct final response remains a more difficult source of failure. Appendix D examines the remaining failure modes, and Appendix E further analyzes the relationship between task completion and behavioral preservation.

5.4 Ablation Study

We study four component ablations of HarnessIR under both DeepSeek-V4-Flash and GLM-5.2 (Table 2). *w/o Ontology Guidance* removes the ontology-mediated representation and directly maps source-harness operations to target-runtime constructs using the source code, execution traces, and target-runtime reference materials. *w/o Target-Native Mapping* omits the explicit target-runtime map $\mathcal{M}_t$ and directly generates target-native code from the source harness and the target-runtime reference materials. *w/o Decision Verification* retains typed model proposals but allows their outputs to affect execution without validation against the admissible semantic space. *w/o Semantic State Revision* prevents execution observations from updating the semantic state. Subsequent action selection therefore no longer depends on the current state constraints.

The drops in *Behavior*, *State*, and *E2E* across all variants under both models indicate that removing these components breaks the alignment between source semantics and target execution. Removing Semantic State Revision increases the *Task* rate under DeepSeek but decreases it under GLM, while reducing behavioral preservation under both models. Decision Verification also improves *Behavior*, *State*, and *E2E* under both models, although its effect on *Task* is model-dependent. Thus, reliable cross-runtime transfer depends on these components working together.

5.5 Behavioral Analysis

Final task success does not by itself establish correct harness behavior. A target may reach the required final outcome while violating intermediate constraints and causing unintended side effects. We therefore measure *Behavior* separately from *Task* and analyze it from two complementary views. Figure 3(a) reports dimension-level pass rates, while Figure 3(b) compares source-level Behavior Success differences between HarnessIR and each baseline across the 10 source harnesses. Together, these views assess which behavioral obligations are preserved and how consistently preservation holds across source harnesses.

Figure 3(a) reports pass rates for the five behavioral dimensions across all methods. HarnessIR achieves the highest Handoff rate and outperforms SOT on Control under both models, reaching 72.40% and 70.70% under DeepSeek and GLM, respectively, compared with 69.00% for SOT. It matches SOT on Action Effect under DeepSeek and exceeds it under GLM. SOT remains stronger on State and Termination. ECT improves over DCT on Control, Handoff, and Termination, while both remain weak on State and Action Effect. SSIR obtains zero rates on State and Action Effect, although its path endpoints yield an approximately 50% Termination rate.

The source-level contrasts in Figure 3(b) show that HarnessIR has a broad advantage over ECT but a smaller and more variable advantage over SOT. HarnessIR outperforms ECT on eight of ten sources under both models. Against SOT, the contrast is positive, tied, and negative for 4/5/1 sources under DeepSeek and 6/1/3 sources under GLM. Thus, the overall Behavior advantage of HarnessIR mainly reflects robust preservation of responsibility transfer and joint behavioral obligations, while its advantage over the stronger SOT baseline depends more strongly on the source harness.

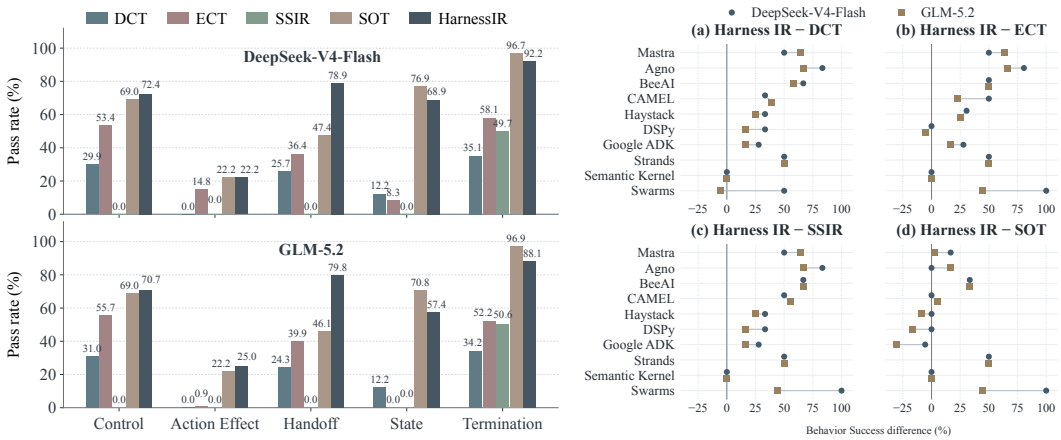

Figure 3: Behavioral preservation across models and source harnesses. (a) Dimension-level pass rates for five methods. (b) Source-level Behavior Success differences between HarnessIR and each baseline, computed as Full−baseline.

Table 3: Each column represents a transition, showing the current state, the corresponding action, and the constraints that the action must satisfy.

Transition 1	**Transition 2**	**Transition 3**
S_0: `balances.Ada=0`, `refund_count=0`; the refund request is waiting for handoff.	S_1: `balanceAgent` has received the refund request and proposes to invoke `issue-fund`; The decision is validated.	S_2: one durable refund-effect receipt is recorded; the ledger update is still pending.
A_0: `handoff (balanceAgent)`	A_1: `issue-refund (Ada,17)`	A_2: `ledger-file-update (balances.Ada,add,17)`
C_0: The requested refund of $17 must not exceed its remaining refundable amount.	C_1: the ledger file must exist, and the user name `Ada` must pass validation before A_1 is executed.	C_2: the update must be triggered by the matching refund receipt for Ada and applied at most once.

5.6 Case Study

Table 3 illustrates a refund workflow executed by the target harness generated by HarnessIR. The harness transfers the request to `balanceAgent`, issues a $17 refund to Ada, and updates the ledger. *State* records execution progress, *Decision* captures the validated refund choice, and *Action* specifies each operation. *Constraint* enforces refund eligibility, action preconditions, and a receipt-dependent ledger update applied at most once. After the final ledger update, the target harness reaches the required terminal and successful state. This case illustrates how the core entities explicitly represent the workflow's semantic dependencies and constrain execution after transfer to the target runtime.

6 Conclusion

We presented HarnessIR, an ontology guided and runtime-independent intermediate representation for transferring agent harnesses across heterogeneous runtimes. HarnessIR recovers source semantics as typed entities, relations, and execution policies, grounds them in target-native constructs, and preserves them through decision verification and semantic state revision during execution. HarnessIR-Bench evaluates this process across ten source harnesses and two target runtimes. HarnessIR achieves 45.00% Behavior Success. Separating source-semantic recovery from target-runtime grounding improves behavioral transfer across runtimes. Future work can investigate methods for improving state consistency and final task completion while preserving source-harness behavior.

AI USE STATEMENT

In this work, we used OpenAI Codex exclusively for translating selected passages, improving the language and readability of the manuscript, and searching for relevant literature. All AI-assisted material and the resulting manuscript content were checked and rechecked item by item by the authors. Each citation was individually verified against its original source. We take full responsibility for the final content of this work, including all text, claims, results, citations, and other artifacts.

REPRODUCIBILITY STATEMENT

Section 4 and Appendix B describe the benchmark construction, data splits, and behavioral dimensions. Section 5 specifies the baselines, metrics, and evaluation setup, while Appendices A and F provide the ontology and recovery prompts. The supplementary code and data include frozen scenario definitions, source-native reference traces, evaluation code, and archived execution outputs with scripts for recomputing the main results. The accompanying README documents the included materials, their coverage, dependencies, and reproduction commands.

REFERENCES

Francisco C. M. Abreu, Luís Cruz, and Sérgio Guerreiro. Ontology-driven model-to-model transformation of workflow specifications. *CoRR*, abs/2511.13661, 2025. doi: 10.48550/ARXIV.2511.13661. URL https://doi.org/10.48550/arXiv.2511.13661.

CrewAI. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/crewAIInc/crewAI.

Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta König-Ries, Jacek Kopecky, Rubén Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna, and Michael Stollberg. Web service modeling ontology (WSMO). W3C Member Submission, 2005. URL https://www.w3.org/submissions/WSMO/.

Andreas Ekelhart, Kabul Kurniawan, Fajar J. Ekaputra, and Elmar Kiesling. Agento: An ontology for modeling agentic AI systems. In Maribel Acosta, Marieke van Erp, Sebastian Rudolph, Olaf Hartig, Blerina Spahiu, Anisa Rula, Daniel Garijo, and Francesco Osborne (eds.), *The Semantic Web - 23rd European Semantic Web Conference, ESWC 2026, Dubrovnik, Croatia, May 10-14, 2026, Proceedings, Part II*, volume 16550 of *Lecture Notes in Computer Science*, pp. 298–320. Springer, 2026. doi: 10.1007/978-3-032-25159-6_16. URL https://doi.org/10.1007/978-3-032-25159-6_16.

Shengda Fan, Xin Cong, Yuepeng Fu, Zhong Zhang, Shuyan Zhang, Yuanwei Liu, Yesai Wu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Workflowllm: Enhancing workflow orchestration capability of large language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=3Hy00Wvabi.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

LangChain. LangGraph: Build resilient language agents as graphs. https://github.com/langchain-ai/langgraph.

Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pp. 75–88. IEEE Computer Society, 2004. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (eds.), *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pp. 2–14. IEEE, 2021. doi: 10.1109/CGO51591.2021.9370308. URL https://doi.org/10.1109/CGO51591.2021.9370308.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, volume NAACL 2025 of *Findings of ACL*, pp. 1160–1183. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-NAACL.65. URL https://doi.org/10.18653/v1/2025.findings-naacl.65.

Douwe Maan, David Sanchez, Aditya Vardhan, David Montague, Marcelo Trylesinski, Alex Hall, and Samuel Colvin. Pydantic AI. https://github.com/pydantic/pydantic-ai, 2025. Accessed: 2026-08-06.

David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic markup for web services. W3C Member Submission, 2004. URL https://www.w3.org/submissions/OWL-S/.

Microsoft. Microsoft agent framework. https://github.com/microsoft/agent-framework.

George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam (ed.), *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pp. 83–94. ACM, 2000. doi: 10.1145/349299.349314. URL https://doi.org/10.1145/349299.349314.

Yipeng Ouyang, Yi Xiao, Yuhao Gu, and Xianwei Zhang. Skcc: Portable and secure skill compilation for cross-framework LLM agents. *CoRR*, abs/2605.03353, 2026. doi: 10.48550/ARXIV.2605.03353. URL https://doi.org/10.48550/arXiv.2605.03353.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Haitao Zheng. Natural-language agent harnesses. *CoRR*, abs/2603.25723, 2026. doi: 10.48550/ARXIV.2603.25723. URL https://doi.org/10.48550/arXiv.2603.25723.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Hu Wei. From agent loops to structured graphs:a scheduler-theoretic framework for LLM agent execution. *CoRR*, abs/2604.11378, 2026. doi: 10.48550/ARXIV.2604.11378. URL https://doi.org/10.48550/arXiv.2604.11378.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155, 2023. doi: 10.48550/ARXIV.2308.08155. URL https://doi.org/10.48550/arXiv.2308.08155.

Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang, and Qingyun Wu. Stateflow: Enhancing LLM task-solving through state-driven workflows. *CoRR*, abs/2403.11322, 2024. doi: 10.48550/ARXIV.2403.11322. URL https://doi.org/10.48550/arXiv.2403.11322.

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 *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *CoRR*, abs/2406.12045, 2024. doi: 10.48550/ARXIV.2406.12045. URL https://doi.org/10.48550/arXiv.2406.12045.

Xiaohui Zhang, Zequn Sun, Chengyuan Yang, Yuanning Cui, Lingbing Guo, and Wei Hu. Toward effective and reliable llm agents via dynamic ontology, 2026. URL https://arxiv.org/abs/2608.22974.

A HARNESS ONTOLOGY

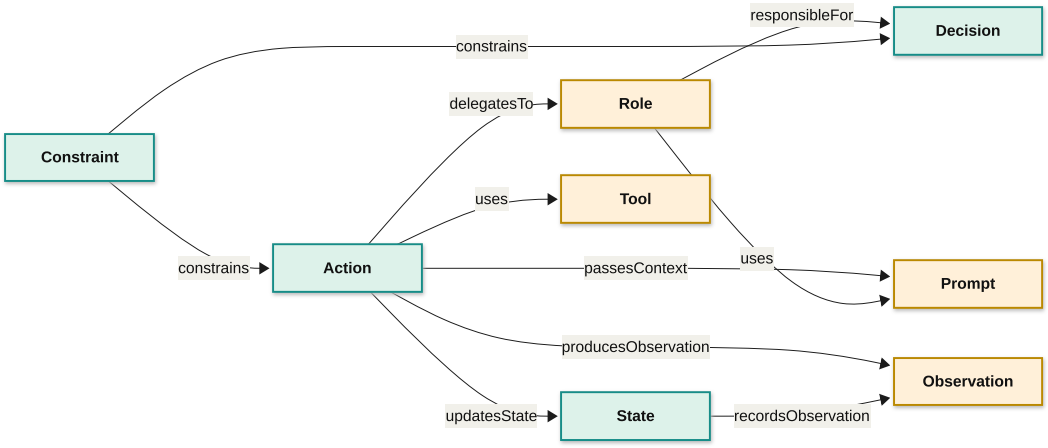

Figure 4: HarnessIR ontology showing the core entity and relation types.

We design the harness ontology $\mathcal{O}_H$ in advance and keep it fixed across all runs of HarnessIR. Figure 4 presents the ontology-level entity and relation structure used by HarnessIR. It shows the eight core entity types—*State*, *Action*, *Decision*, *Constraint*, *Role*, *Tool*, *Prompt*, and *Observation*, together with their directed relation types. The full ontology further defines subtype and inheritance relations, whose semantic details are summarized in Table 4. Each subtype inherits the attributes and permitted relations of its parent type while providing a more specific interpretation when supported by the source code; shared relations remain attached to the parent type.

Table 4: Entity subtypes and their intended semantic scope.

Parent type	**Subtype**	**Description**
Action	Model action	Calls a model to obtain a result.
Action	Tool action	Calls a declared tool with inputs and receives its result.
Action	Handoff action	Transfers a task or responsibility to a target role.
Action	Control action	Performs an explicit control operation such as a branch or join.
Action	Completion action	Performs an explicit completion or closing operation.
Decision	Handoff decision	Selects whether and where to transfer responsibility.
Decision	Tool decision	Selects a tool and, when needed, its input values.
Decision	Termination decision	Determines whether an execution condition supports termination.
Constraint	Precondition constraint	Must hold before an action or transition is executed.
Constraint	Postcondition constraint	Must hold after an action or transition is executed.
Constraint	Authorization constraint	Requires an allowed authority, permission, or approval.
Constraint	State constraint	Restricts a state value, scope, or state invariant.
Constraint	Prompt constraint	Restricts prompt content, retention, or transfer.
Constraint	Order constraint	Requires an ordering between actions or transitions.

Continued on next page

Table 4: Entity subtypes and their intended semantic scope (continued).

Parent type	Subtype	Description
Constraint	Priority constraint	Orders actions when more than one action is allowed.
Constraint	Response constraint	Restricts the required final response.
Constraint	Failure constraint	Specifies the required state or handling after failure.
Observation	Model observation	Observation produced by a model call or model decision.
Observation	Runtime observation	Observation produced by a native action or the target runtime.

B Benchmark Construction Details

Behavioral dimensions. Table 5 details the source commitments checked in each dimension.

Table 5: Behavioral dimensions evaluated in the benchmark. A dimension is included in the behavioral-fidelity score only when it is exercised by the source execution.

Dimension	Source commitment	Representative migration failure
State	Scope and value of persistent or conversational state	History is reset, merged across sessions, or committed to the wrong scope
Control	Guarded route and admissible action sequence	A plausible branch executes under the wrong permission or observation
Handoff	Responsibility transfer and context obligation	The right specialist is selected but loses task or user context
Action Effect	Operation, arguments, result, cardinality, order, and safety	An action effect is omitted, duplicated, reordered, or produced after failure
Termination	Terminal mode and inhibition of later actions	The target stops early or continues after the source has terminated

The *State* dimension applies to 104 of the $10 \times 6 \times 2 = 120$ source–scenario–target combinations in the test split, as determined by the source-native references. With three repetitions per combination, we compute the *State* pass rate over $104 \times 3 = 312$ executions per method–model pair.

Source inventory and admission. Table 6 lists the ten test sources and their behavioral coverage.

Table 6: Source harnesses and behavioral phenomena in the HarnessIR-Bench test set. Each source contributes six evaluation conditions and is transferred to both target runtimes. HITL denotes human-in-the-loop.

Executable source	Runtime family	Behavioral challenge
Refund workflow	Mastra	Delegation, durable ledger state, action effect cardinality, and refund ordering
Support workflow	Agno	Multi-step diagnosis, human review, cancellation, and reviewed completion
Requirement handoff	BeeAI	Specialist routing, multi-recipient responsibility, and carried evidence
Shared memory	CAMEL	Cross-agent memory propagation, role state, and synchronization
DevOps HITL	Haystack	Missing information, human authorization, tool order, and prohibited action effects
Customer service	DSPy	Tool choice, ticket state, fallback, and result-dependent completion
Three-layer transfer	Google ADK	Hierarchical transfer, specialist completion, and session state

Continued on next page

Table 6: Source harnesses and behavioral phenomena in the HarnessIR-Bench test set (continued).

Executable source	Runtime family	Behavioral challenge
Teacher swarm	Strands	Specialist selection, request preservation, and aggregated response
Plan-and-execute	Semantic Kernel	Planning, replanning, state revision, and completion edges
Graph workflow	Swarms	Fan-out/fan-in control, branch state, and final merge

C Extended Implementation Details

Backbone configurations. We evaluate DeepSeek-V4-Flash and GLM-5.2 independently. In each backbone setting, the selected model performs all LLM-based stages of HarnessIR, including static semantic extraction, execution-policy recovery, target-runtime mapping, target-code generation, and target-harness execution.

We request a temperature of 0 for construction and 0.2 for target-harness execution, with `top_p` set to 1.0 and one completion per request. The maximum output length per request is 8,192 tokens for static semantic extraction, execution-policy recovery, and target-runtime mapping, 16,384 tokens for target-code generation, and 2,048 tokens for target-harness execution. We use the same stage-specific settings for both backbones.

AST-based source partitioning. We partition the application code in each pinned source snapshot into structurally complete functions, classes, agent definitions, and module-level statement groups, forming $\mathcal{C}_s$. Each unit retains the necessary imports and enclosing context, together with stable symbol identities and cross-unit references $\mathcal{L}_s^{\mathrm{AST}}$. We use a soft limit of 4,096 source-code tokens per unit, measured with `cl100k_base` for both backbones. Oversized classes are split at method boundaries with class context preserved; indivisible units remain intact if the complete request fits the model's context window after reserving the output budget. Otherwise, construction is marked as failed. Each unit is processed independently in one initial extraction request.

Repeated execution. Each trial starts from the scenario's initial state, with session state and environment fixtures reset between trials. For HarnessIR, a decision that fails verification terminates the trial. For each method–backbone pair, we evaluate the ten test source harnesses in two target runtimes on six evaluation scenarios per source, with three executions per scenario, giving $10 \times 2 \times 6 \times 3 = 360$ scheduled target executions. These repetitions rerun the fixed target programs; they do not repeat source recovery, target mapping, or code generation. Failed trials remain in the applicable score denominators.

Response-equivalence evaluation. GLM-5.2 serves as a separate response-equivalence judge for *Task-J* under both backbone settings. The judge receives the scenario request, source-native final response, and target final response. We request a temperature of 0 and a maximum output length of 512 tokens per judging request. The judge assesses semantic equivalence between the final responses while allowing differences in wording. The same judging instructions and configuration are used for all methods and both backbones. An execution passes *E2E-J* only when it passes both *Task-J* and *Behavior* and executes successfully in the designated target runtime.

D Failure Analysis

To characterize the remaining failure modes of Harness IR, we aggregate the primary diagnosis assigned to each failed execution. The resulting distribution is shown in Figure 5.

The diagnostic distribution localizes the remaining improvement opportunities while showing that the high-level execution structure is comparatively stable. Control accounts for 9.4% of failed executions, while Termination and Handoff account for 1.9% and 0.8%, respectively. The largest category is task-outcome failure after behavioral success, which accounts for 37.2% of failures. These executions satisfy the source-derived behavioral obligations, indicating that the recovered

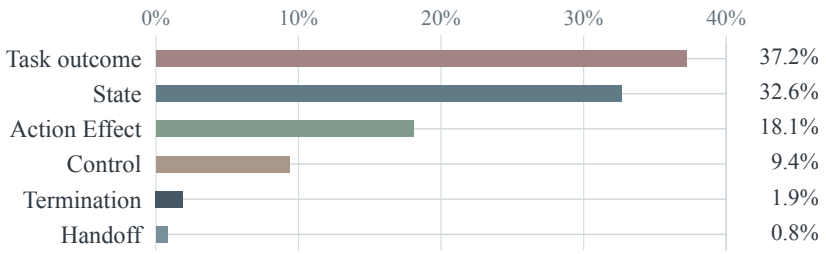

Figure 5: Primary diagnosis distribution across failed Full executions.

execution policy is often preserved and that the remaining opportunity lies in improving final response and outcome realization. State failures form the second largest category, accounting for 32.6% of failures, followed by Action Effect failures at 18.1%. Together, task-outcome and State diagnoses account for approximately 70% of all failure diagnoses, suggesting that future improvements should focus on last-mile semantic grounding, state consistency, and final outcome realization, while the comparatively small Control, Termination, and Handoff categories indicate stronger preservation of the overall execution structure.

E JOINT TASK-BEHAVIOR ANALYSIS

	Both	Task only	Behavior only	Neither
DCT	0.0%	0.0%	7.1%	92.9%
ECT	0.8%	5.7%	5.6%	87.9%
SSIR	0.0%	3.2%	0.0%	96.8%
SOT	10.7%	8.5%	23.3%	57.5%
HarnessIR	12.4%	6.5%	32.6%	48.5%

Figure 6: Joint Task–Behavior outcomes for the five methods and primary diagnoses of failed HarnessIR executions.

Figure 6 decomposes the 720 executions aross two models of each method into four mutually exclusive categories: simultaneous Task and Behavior success (Both), Task success with behavioral drift (Task only), Behavior preservation without Task success (Behavior only), and failure on both criteria (Neither). HarnessIR achieves the highest joint success rate (12.4%), followed by SOT (10.7%), and also has the lowest rate of joint failure (48.5%). Its relatively large Behavior-only share (32.6%) shows that preserving the source execution semantics remains separable from producing the correct final task outcome.

F PROMPT

We use *Static Semantic Graph Extraction* to extract ontology-typed entities and relations from each structurally complete source-code unit. *Core Entity Extraction Constraints* provides additional criteria for identifying core entities and is supplied as input to this extraction prompt. We use *Source-Trace*

Execution Policy Recovery to recover the source harness's execution policy from source-native execution traces and relevant code, identifying when actions are enabled and which constraints apply.

Static Semantic Graph Extraction

You are a semantic analyst in the source semantic recovery stage of HarnessIR.
The source harness is being prepared for transfer from its original runtime to a different target runtime. Before target−side implementation, HarnessIR recovers a runtime−independent semantic representation of the source harness. Your task is to analyze one structurally complete source−code unit and extract the ontology−grounded semantic objects and relations that are explicitly supported by that unit.
The result will be used as a local semantic fragment. Local fragments extracted from different source−code units will later be merged using stable AST−level symbol references. Analyze only the supplied source unit and preserve its local semantic evidence.
The supplied ontology defines multiple entity types, relation types, and typed attributes. Use the ontology as the authoritative vocabulary. Extract any ontology−defined entity type that is supported by the source code. Apply the reusable core−entity constraints supplied through the following input placeholder:

CORE_ENTITY_EXTRACTION_CONSTRAINTS:
{{CORE_ENTITY_EXTRACTION_CONSTRAINTS}}

General extraction rules:
– Use the ontology specification as the only authority for entity types, relation types, attributes.
– Extract only semantic objects and relations supported by the source unit, its local AST context, or its local symbol information.
– Preserve source−specific names, values, expressions, and symbolic references.
– Create separate entity instances for distinct source constructs, even when they share the same ontology type.
– Extract a relation only when its direction and meaning are supported by the source code. Do not create a relation solely because two entities occur in the same unit.
– If a symbol is defined outside the current unit, preserve its stable reference without fabricating an external entity.
– If the evidence is insufficient for an ontology assignment, record the ambiguity instead of inventing a type, relation, attribute, or value.
– Do not infer unobserved runtime behavior, execution−trace rules, target−runtime constructs, or target−specific implementation details.
– Return only the JSON object specified below. Do not include explanations or analysis outside the JSON object.

Input:
ONTOLOGY_SPECIFICATION: {{ONTOLOGY_SPECIFICATION}}
SOURCE_UNIT_ID: {{SOURCE_UNIT_ID}}
SOURCE_FILE: {{SOURCE_FILE}}
SOURCE_LANGUAGE: {{SOURCE_LANGUAGE}}
SOURCE_CODE_UNIT: {{SOURCE_CODE_UNIT}}
LOCAL_AST_CONTEXT: {{LOCAL_AST_CONTEXT}}
LOCAL_SYMBOL_TABLE: {{LOCAL_SYMBOL_TABLE}}

Output:
[The complete JSON output schema is omitted here for space.]

Core Entity Extraction Constraints

The ontology contains multiple entity types. The following four types are central to harness semantics and require the additional criteria below. Extract other ontology−defined types whenever they are supported by the source code and the ontology specification.

1. State

Extract a State entity when the source code represents information that is maintained, read, or updated as the harness progresses.
Evidence may include an explicit state schema, state object, message state, execution context, task or workflow status, phase, mode, completion indicator, draft, plan, history, counter, result, permission, or another value carried across execution steps. A variable or object that is updated and later used to determine subsequent harness behavior is also evidence.
Do not classify a purely local temporary value as State merely because it is assigned in a function. Extract it as State only when the source code shows that it represents execution state, is shared across relevant operations, persists across steps, or affects subsequent harness behavior.

2. Action
Extract an Action entity when the source code defines or invokes an executable operation that can be performed by the harness and that may read or modify state, produce an effect, communicate with another component, or advance execution.
Evidence may include a harness node, handler, task, step, executable function, model invocation, tool invocation, external operation, handoff, delegation, message send, responsibility transfer, state update, observation−producing operation, or explicit terminal, completion, or failure operation.
Do not classify every helper function, data conversion, or pure computation as an Action. Extract an operation as an Action when the code indicates that it participates in harness execution or produces a behaviorally relevant effect.

3. Decision
Extract a Decision entity when the source code contains a runtime choice whose outcome determines which subsequent behavior is selected, enabled, or executed.
Evidence may include a router, selector, dispatcher, conditional branch, model output interpreted as an approval, revision, rejection, route, or tool choice, a state−dependent branch, a decision gate, or a termination or continuation choice.
Record admissible decision outcomes only when they are explicitly defined or constrained by the source code. Do not classify an arbitrary boolean expression as a Decision unless its outcome controls a subsequent harness behavior.

4. Constraint
Extract a Constraint entity when the source code expresses a condition or requirement that restricts when or how an action, state transition, context transfer, or observation may be used.
Evidence may include a precondition, action guard, input or output validation, permission check, postcondition, completion requirement, ordering or dependency requirement, priority, retry or safety requirement, action−admissibility condition, failure−handling condition, or termination condition.
A Decision selects among possible behaviors, whereas a Constraint restricts the validity or eligibility of a behavior. Use both types only when the source code provides evidence for both roles.

Source-Trace Execution Policy Recovery

You are an execution−policy analyst in the source semantic recovery stage of HarnessIR.

The source harness is being prepared for transfer from its original runtime to a different target runtime. Static semantic extraction recovers ontology−grounded entities and relations from the source code. Your task is to recover the dynamic execution policy of the source harness from its source−native execution traces and the relevant source harness code.

The recovered policy describes which actions are enabled under particular execution contexts, including the relevant State attributes, LLM Decision outcomes, and execution Constraints.

Use the supplied ontology as the authoritative vocabulary. The ontology may contain multiple entity types, relation types, attributes, and subtypes. Use an ontology−defined subtype when the source code and traces provide sufficient evidence; otherwise use the corresponding parent type.

Structured recovery procedure

1. Identify the executed actions

First, analyze the ordered execution traces together with the relevant source harness code and identify the semantically meaningful actions observed during execution.

For each action:

- associate it with a stable source reference, trace identifier, or qualified symbol name;
- assign the most specific ontology type supported by the evidence;
- preserve source-specific names, arguments, results, recipients, and execution order;
- distinguish separate action instances when the source code invokes the same operation in different semantic roles;
- record an action only when the trace or source code shows that it participates in harness execution, produces a behaviorally relevant effect, updates state, communicates with another component, transfers responsibility, produces an observation, or terminates or fails execution.

2. Identify the activation conditions of each action

For every recovered action, determine the conditions under which the source harness executes it.

Analyze the state and decision information available before the action, and group trace instances that share the same supported execution context. Describe the activation context using:

- relevant State attributes and their required values or predicates;
- an LLM Decision outcome when the action depends on a model-mediated choice;
- both State and Decision conditions when both are required;
- a state-only condition when no LLM Decision is involved.

Include only state attributes that affect action selection, action arguments, routing, handoff, ordering, termination, or later harness behavior. Separate control-relevant conditions from incidental payload fields.

Create separate policy rules when different state conditions or decision outcomes lead to different action behavior.

If the source code or traces do not provide enough evidence to determine an activation condition, preserve the ambiguity instead of inventing a condition.

3. Identify the constraints governing each action

For each action and policy rule, identify the constraints that must hold for the action to execute and for its result to be accepted.

Organize these constraints into the following categories:

- Preconditions: requirements that must hold before the action or transition is executed.
- Postconditions: requirements that must hold after the action or transition completes.
- Input constraints: restrictions on action arguments, required fields, admissible values, permissions, recipients, or call structure.
- Output constraints: restrictions on returned values, produced observations, state updates, or accepted results.
- Ordering and priority constraints: source-supported dependencies between actions, required execution order, mutually exclusive actions, or priority when multiple actions are admissible.
- Action-inherent constraints: requirements intrinsic to the action itself, such as its required effect, cardinality, recipient, safety condition, completion behavior, failure behavior, or termination behavior.

Use an ontology-defined Constraint subtype whenever the evidence supports one. Otherwise, use the parent Constraint type and preserve the more specific constraint category in the output.

A constraint must be attached to the action, state, decision, observation, or relation that it actually governs. Do not infer a constraint solely from the fact that two events occur close to each other in a trace.

Evidence and uncertainty

Use the ontology to determine valid types, relations, attributes, and semantic values. Use the source code to determine the intended meaning of operations and explicit control logic. Use the execution traces to determine observed actions, state changes, decision outcomes, ordering, and terminal behavior.

Preserve raw trace values together with normalized semantic interpretations when both are available. Every recovered action, activation condition, decision outcome, state effect, and constraint must include source or trace evidence. If the evidence is incomplete or contradictory, record the ambiguity in unresolved_items.

Do not infer target−runtime constructs, unobserved actions, unsupported state transitions, or execution behavior that is absent from the supplied source code and traces.

Return only the JSON object specified below. Do not include explanations or analysis outside the JSON object.

Input:
ONTOLOGY_SPECIFICATION: {{ONTOLOGY_SPECIFICATION}}
SOURCE_HARNESS_CODE: {{SOURCE_HARNESS_CODE}}
SOURCE_LANGUAGE: {{SOURCE_LANGUAGE}}
SOURCE_RUNTIME_METADATA: {{SOURCE_RUNTIME_METADATA}}
EXECUTION_TRACE_SET: {{EXECUTION_TRACE_SET}}
TRACE_FORMAT_DESCRIPTION: {{TRACE_FORMAT_DESCRIPTION}}

Output:
[The complete JSON output schema is omitted here for space.]