UCLA ScAi

HarnessBridge: Learnable Bidirectional Controller for LLM Agent Harness

Xiaoxuan Wang[1,*], Haixin Wang[1,*], Alexander Taylor[1], Jason Cong[1], Yizhou Sun[1], Wei Wang[1]

[1] *University of California, Los Angeles* *Equal Contribution.*

arXiv:2606.12882v1 [cs.AI] 11 Jun 2026

Large language models are increasingly deployed as agents for long-horizon tasks, yet their performance is shaped not only by model capability and environment design, but also by the harness that mediates agent–environment interaction. Existing harnesses are largely manually engineered, making them difficult to scale as trajectories grow longer and interactions become more complex. In this work, we ask whether harness can be generated by a learnable plug-in module that can be trained in an end-to-end fashion. We introduce **HarnessBridge**, a lightweight learnable harness controller that parameterizes the agent–environment interface as a bidirectional projection. HarnessBridge learns two bidirectional projections: observation projection, which distills raw trajectories into compact, decision-relevant states, and action projection, which converts proposed actions into executable transitions or trajectory-grounded rejections. We train HarnessBridge on a harness supervision dataset via unified instruction tuning. On Terminal-Bench 2.0 and SWE-bench Verified, HarnessBridge matches or surpasses strong specialized harnesses while substantially reducing token usage and trajectory length, and generalizes from smaller generators to larger commercial models.

Correspondence: Haixin Wang (whx@cs.ucla.edu)
GitHub: `https://github.com/mandyyyyii/HarnessBridge`
HuggingFace: `https://huggingface.co/HarnessBridge`

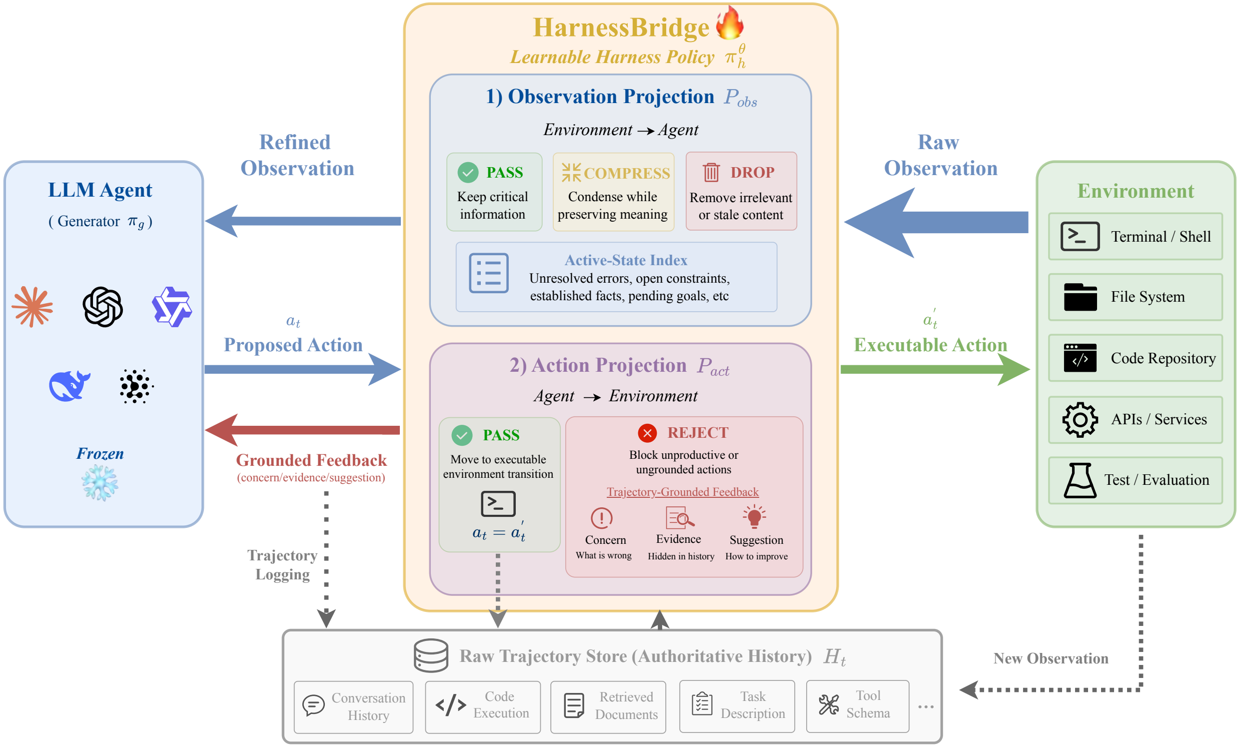


Figure 1 | Overview of **HarnessBridge**, a learnable harness policy that parameterizes the agent-environment interface as a bidirectional projection: *observation projection* compresses raw trajectories into a generator-visible state, while *action projection* maps proposed actions to executable transitions or grounded rejections.

1 Introduction

Large language models (LLMs) are increasingly deployed as agents for complex, long-horizon tasks, such as autonomous software engineering (Fu et al., 2026; Merrill et al., 2026; Yang et al., 2024, 2026), web navigation (Ning et al., 2025; Wang et al., 2026a,c), search (Huang et al., 2025; Wang et al., 2023b,c), and multi-modal tasks (Chang et al., 2025; Wang et al., 2026b). Prior progress has largely focused on two axes: stronger agents as generators and richer environments. Yet as horizons grow, a third axis becomes increasingly important: the interface that determines what information reaches the generator and what actions are committed back to the environment.

Nowadays, this interaction in the agent system is implemented by a *harness* (Pan et al., 2026; Rajasekaran, 2026): the scaffold that formats observations, manages context, invokes tools, parses outputs, validates actions, and handles environmental feedback. Harness engineering has become indispensable to agent performance, with many gains arising from better context construction, retry logic, summarization, and action validation. However, despite its importance, the harness is still usually treated as manually engineered infrastructure rather than as an optimizable policy.

Complex, manually engineered harnesses could become a central limitation in long-horizon interaction. In the environment-to-agent direction, trajectories accumulate redundant context (Anthropic, 2025a), stale errors, superseded hypotheses, and low-value details, increasing token cost while obscuring decision-critical state. In the agent-to-environment direction, model outputs may repeat ineffective actions, pursue invalidated hypotheses, enter empty loops, or issue malformed commands, consuming scarce environment steps without advancing the task (Aghzal et al., 2026).

Recent work on automated or meta-level harness construction has begun to recognize harnesses as optimization targets, for example by using coding agents to improve task-specific scaffolds or search over prompt and protocol variants (Lee et al., 2026). These efforts show that harnesses are not fixed infrastructure, but objects that can be searched, revised, and improved. However, they typically optimize the external scaffold around an agent, rather than learning the runtime interaction policy that determines how information and actions flow between the generator and the environment. This points to a more fundamental question:

Can harness be formulated as an end-to-end learnable generation problem?

This reduces to optimizing the agent–environment interface: what is exposed to the agent, and what is committed to the environment. For long-horizon behavior to remain efficient, grounded, and recoverable, this interface must preserve task-relevant state, suppress stale or redundant context, reject unproductive actions, and avoid introducing hallucinated compressed information not supported by the trajectory.

Thus, we propose **HarnessBridge**, a learnable harness policy for long-horizon LLM agents. HarnessBridge parameterizes the agent–environment interface as a bidirectional projection policy. In the environment-to-agent direction, *observation projection* maps the raw interaction history into a generator-visible state, preserving decision-critical information while compressing or suppressing stale, redundant, or superseded content. In the agent-to-environment direction, *action projection* maps a proposed generator action to either an executable environment transition or a trajectory-grounded rejection with feedback, preventing low-value or ungrounded actions.

To train HarnessBridge, we construct a harness supervision dataset covering both directions of the interface. We unify these behaviors as instruction-following tasks and train a lightweight LLM via unified instruction tuning. Empirically, we evaluate HarnessBridge on long-horizon benchmarks, including Terminal-Bench 2.0 and SWE-bench Verified across a series of open-sourced and commercial models. HarnessBridge matches or surpasses strong specialized harnesses while substantially reducing token usage and trajectory length. Moreover, models trained with smaller generators generalize effectively to larger commercial models. Our contributions are as flows:

- We introduce end-to-end harness generation for agent systems, replacing manually engineered interaction logic with a learnable harness policy.
- We are the first to introduce unified instruction tuning for learning bidirectional mappings between agents and environments.
- We present **HarnessBridge**, demonstrating improved efficiency and competitive or better task performance with good generalization.

2 Related Work

2.1 Long-Horizon Tool-Using LLM Agents.

LLM agents have been increasingly studied as systems that use tools to interact with external environments (Wei et al., 2026). Early and representative work shows that LLMs can interleave reasoning and acting (Hu et al., 2024; Song et al., 2025; Yao et al., 2022a) , call external APIs (Barres et al., 2025; Yao et al., 2024), and acquire reusable skills through interaction (Li et al., 2026), enabling agents to move beyond single-turn generation toward sequential decision making (Schick et al., 2023; Wang et al., 2023a; Yao et al., 2022b). This paradigm has since been instantiated in a range of long-horizon domains, including web navigation, software engineering, terminal operation, and multi-step information seeking, with benchmarks such as WebArena, SWE-bench, and Terminal-Bench exposing realistic environments in which agents must gather information, execute actions, observe feedback, and revise their plans over many turns (Jimenez et al., 2024; Merrill et al., 2026; Zhou et al., 2023).

Despite this progress, making tool-using agents stable and scalable over long runs remains an open problem (Liu et al., 2026). Long-horizon interaction introduces accumulated observations, stale or redundant context, error propagation, repeated tool calls, and increasing execution cost.

2.2 Harness Engineering for Agents.

Existing harnesses rely on manually designed heuristics, such as trajectory summarization, retrieval-based memory, context compaction (Han et al., 2025), retry rules, and tool-call validation. These strategies are increasingly important for long-running agents, since merely extending the context window does not prevent trajectories from accumulating stale, redundant, or low-signal information that can degrade decision quality and distract the agent (Anthropic, 2025b). However, existing methods typically implement harness behavior as static rules or separately engineered modules: they may compress history, retrieve relevant information, or validate tool calls, but they do not learn the runtime interaction policy that decides what information and actions should pass through the agent–environment interface at each step.

Recently, auto-harness begins to treat harnesses themselves as optimization targets, exploring automatic workflow optimization, prompt selection, scaffold search, or code-level harness improvement. For example, Meta-Harness (Lee et al., 2026) represents an important step beyond manual harness engineering. Nevertheless, they typically optimize the external scaffold around an agent, rather than learning a runtime policy that continuously mediates the bidirectional flow of observations and actions during execution. In contrast, we focus on the harness-mediated interface itself.

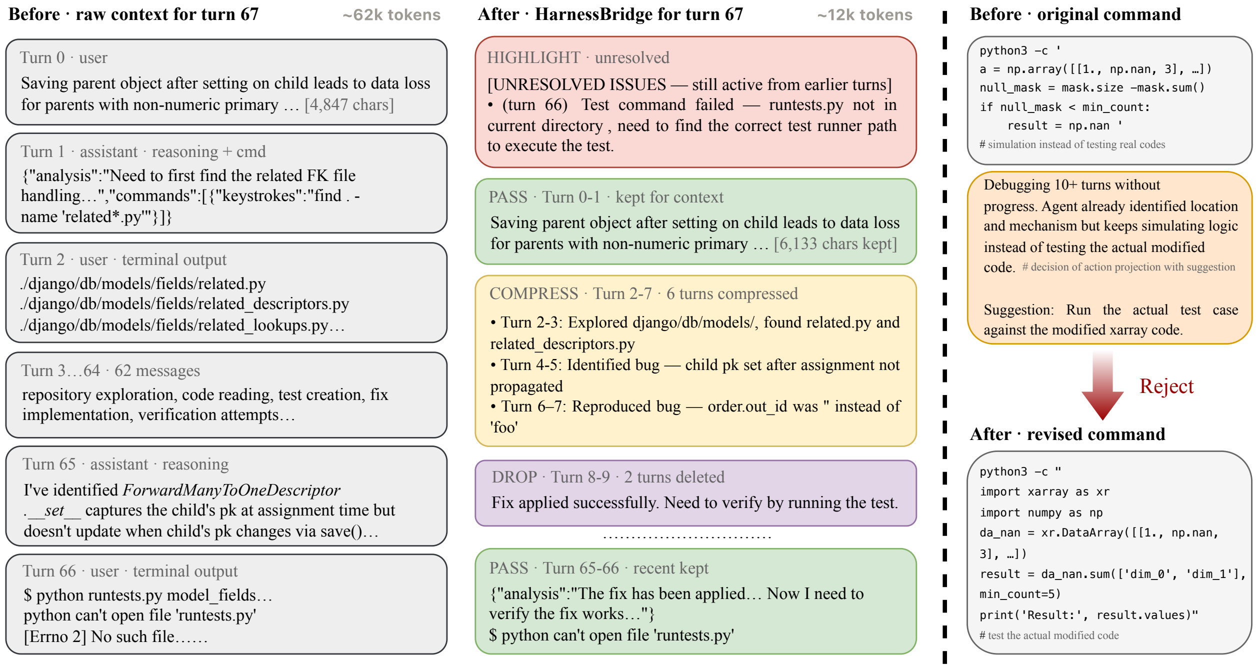
3 Method

We consider a tool-using LLM agent interacting with an environment $\mathcal{E}$ through a tool interface $\mathcal{T}$. At turn t, the interaction history is denoted by H_t, which contains previous model outputs, tool calls, environment observations, retrieved evidence, code snippets, execution results, and scaffold-provided reminders. Given a system prompt s and task instruction q, a conventional harness serializes this history into the generator input and samples an action from a generator policy:

$$a_t \sim \pi_g(\cdot \mid s, q, H_t) \tag{1}$$

The action is then executed in the environment, producing an observation o_t, and the history is updated as $H_{t+1} = H_t \cup \{a_t, o_t\}$.

This conventional formulation treats the harness as a fixed interface between the generator and the environment. Yet in long-horizon tool-use, this interface plays a substantive role in shaping agent behavior. The generator conditions on a harness-constructed representation of the trajectory rather than the raw environment state, while the environment is affected only through actions dispatched by the harness. Consequently, the harness implicitly defines both the generator's effective observation and the action exposed to the environment.



Saving parent object after setting on child leads to data loss for parents with non-numeric primary … [4,847 chars]

{"analysis":"Need to first find the related FK file handling…","commands":[{"keystrokes":"find . -name 'related*.py'"}]}

```
./django/db/models/fields/related.py
./django/db/models/fields/related_descriptors.py
./django/db/models/fields/related_lookups.py…
```

repository exploration, code reading, test creation, fix implementation, verification attempts…

I've identified *ForwardManyToOneDescriptor* .__*set*__ captures the child's pk at assignment time but doesn't update when child's pk changes via save()…

```
$ python runtests.py model_fields…
python can't open file 'runtests.py'
[Errno 2] No such file……
```

[UNRESOLVED ISSUES — still active from earlier turns]
- (turn 66) Test command failed — runtests.py not in current directory , need to find the correct test runner path to execute the test.

Saving parent object after setting on child leads to data loss for parents with non-numeric primary … [6,133 chars kept]

- Turn 2-3: Explored django/db/models/, found related.py and related_descriptors.py
- Turn 4-5: Identified bug — child pk set after assignment not propagated
- Turn 6–7: Reproduced bug — order.out_id was '' instead of 'foo'

Fix applied successfully. Need to verify by running the test.

{"analysis":"The fix has been applied… Now I need to verify the fix works…"}
$ python can't open file 'runtests.py'

```
python3 -c '
a = np.array([[1., np.nan, 3], …])
null_mask = mask.size -mask.sum()
if null_mask < min_count:
    result = np.nan '
# simulation instead of testing real codes
```

Debugging 10+ turns without progress. Agent already identified location and mechanism but keeps simulating logic instead of testing the actual modified code. # decision of action projection with suggestion

Suggestion: Run the actual test case against the modified xarray code.

```
python3 -c "
import xarray as xr
import numpy as np
da_nan = xr.DataArray([[1., np.nan,
3], …])
result = da_nan.sum(['dim_0', 'dim_1'],
min_count=5)
print('Result:', result.values)"
# test the actual modified code
```

Figure 2 | Examples of HarnessBridge on `django-13964__dPYRYzC` (left) and `xarray-4356__fmRfApG` (right) from SWE-Bench Verified, illustrating observation projection and action projection, respectively.

3.1 HarnessBridge

We formulate harness engineering as an end-to-end learnable interaction problem, where a harness learns to mediate what the agent observes and what the environment executes. Specifically, HarnessBridge parameterizes the harness as a learnable bidirectional interaction policy π_h:

$$\pi_h : (s, q, H_t, a_t) \mapsto (\widetilde{H}_t, a'_t) \tag{2}$$

where $\widetilde{H}_t$ is the state exposed to the generator and a'_t is the action exposed to the environment. The generator policy π_g remains fixed; optimization is performed only over the harness policy π_h.

We instantiate π_h as two directional projections over the agent–environment interface. In the environment-to-agent direction, the observation projection maps the raw history to a generator-visible state:

$$\widetilde{H}_t = P_{\text{obs}}(s, q, H_t) \tag{3}$$

The generator then samples an action $a_t \sim \pi_g(\cdot \mid s, q, \widetilde{H}_t)$. In the agent-to-environment direction, the action projection maps the proposed action to the environment-facing action:

$$a'_t = P_{\text{act}}(s, q, H_t, a_t) \tag{4}$$

Only a'_t is exposed to the environment, inducing the transition $o_t = \mathcal{E}(a'_t)$ and the updated history $H_{t+1} = H_t \cup \{a'_t, o_t\}$. Together, P_{obs} and P_{act} define a learned interface between the generator and the environment, controlling both the state exposed to the generator and the action exposed to the environment.

3.1.1 Observation Projection

Observation projection specifies how the raw interaction history is exposed to the generator. Let the history at turn t be represented as a sequence of interaction units $H_t = (h_1, \dots, h_t)$, where each h_i may denote a full interaction turn or a finer-grained component, such as an action, observation, tool feedback, intermediate artifact, retrieved evidence, or other state produced during the interaction. Observation projection learns how these units should be represented in the generator-visible state.

We expand the observation projection as

$$\widetilde{H}_t = P_{\text{obs}}(s, q, H_t) = (U_t, \widetilde{h}_1, \dots, \widetilde{h}_t) \tag{5}$$

where U_t is an active-state index placed before the projected history, and $\widetilde{h}_i$ is the projected representation of history unit h_i. Both U_t and $\{\widetilde{h}_i\}_{i=1}^{t}$ are induced by the learned observation policy. Specifically, the policy predicts:

$$\begin{aligned} U_t &= P_{\text{obs}}^{\text{state}}(s, q, H_t), \\ z_i &= P_{\text{obs}}^{\text{exp}}(s, q, H_t, h_i), \end{aligned} \tag{6}$$

where $z_i \in \{\textsc{Pass}, \textsc{Compress}, \textsc{Drop}\}$. The exposure decision z_i determines the projected form of each history unit:

$$\widetilde{h}_i = \begin{cases} h_i, & z_i = \text{PASS}, \\ \text{Compress}(h_i), & z_i = \textsc{Compress}, \\ \varnothing, & z_i = \textsc{Drop}. \end{cases} \tag{7}$$

The active-state index U_t is a learned view extracted from the raw history. It records information that should remain immediately visible to the generator, including unresolved errors, open constraints, established facts, pending goals, and remaining decision variables. By placing U_t before the projected chronological history, HarnessBridge makes the current interaction state explicit without requiring the generator to reconstruct it from a long trajectory.

Observation projection acts as a learned exposure function over the interaction history. It preserves decision-critical units, compresses relevant but verbose units, and suppresses units whose information is irrelevant, redundant, or superseded.

3.1.2 Action Projection

Action projection specifies how generator proposals are exposed to the environment. In long-horizon interaction, syntactically valid actions may still be redundant, weakly grounded, inconsistent with accumulated evidence, or unlikely to advance the task, wasting interaction steps and time. Action projection therefore learns an environment-facing map from proposed actions to executable or rejected transitions, conditioned on the current task state and interaction history.

Given the raw interaction history H_t and a proposed action a_t, P_{act} predicts

$$P_{\text{act}}(s, q, H_t, a_t) = (d_t, \rho_t) \tag{8}$$

where

$$d_t \in \{\textsc{Pass}, \textsc{Reject}\}.$$

When $d_t = \textsc{Pass}$, the projected action is the original proposal, $a'_t = a_t$, and the environment transition proceeds. When $d_t = \textsc{Reject}$, no environment step is taken; the projected action is null, $a'_t = \varnothing$, and the feedback ρ_t is returned to the generator as part of the subsequent projected state.

The feedback ρ_t is required to be grounded in the current interaction history. We represent it as

$$\rho_t = (\textit{concern}, \textit{evidence}, \textit{suggestion}),$$

where the *concern* states why the proposed action is unlikely to be a productive transition, the *evidence* identifies the specific trajectory information supporting this assessment, and the *suggestion* provides an actionable direction for revision. If P_{act} cannot provide trajectory-grounded evidence, it defaults to $\textsc{Pass}$.

Thus, action projection serves as an environment-facing interface policy: it determines which proposed transitions should affect the environment under the current task state and interaction budget. By requiring trajectory-grounded evidence for rejection, P_{act} reduces unproductive environment steps while preserving informative exploration.

Harness	**Qwen3.5-35B-A3B**				**GLM-4.7-Flash**			
	Terminal-Bench 2.0		SWE-Bench Verified		Terminal-Bench 2.0		SWE-Bench Verified	
	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓
				Manual Harness				
Terminus 2	30.3	2.31	**61.6**	1.47	19.1	1.87	45.2	1.51
Terminus-KIRA	27.0 $_{-10.9\%}$	9.59 $_{+315.2\%}$	46.0 $_{-25.3\%}$	9.77 $_{+564.6\%}$	6.7 $_{-64.9\%}$	4.90 $_{+162.0\%}$	37.8 $_{-16.4\%}$	6.78 $_{+349.0\%}$
mini-SWE-agent	29.2 $_{-3.6\%}$	6.32 $_{+173.6\%}$	59.8 $_{-2.9\%}$	5.92 $_{+302.7\%}$	11.2 $_{-41.4\%}$	1.60 $_{-14.4\%}$	45.4 $_{-0.4\%}$	3.91 $_{+158.9\%}$
OpenHands	27.0 $_{-10.9\%}$	2.61 $_{+13.0\%}$	52.6 $_{-14.6\%}$	1.96 $_{+33.3\%}$	13.5 $_{-29.3\%}$	1.28 $_{-31.6\%}$	42.0 $_{-7.1\%}$	3.69 $_{+144.4\%}$
Qwen-Coder	24.7 $_{-18.5\%}$	4.19 $_{+81.4\%}$	58.8 $_{-4.5\%}$	3.86 $_{+162.6\%}$	10.1 $_{-47.1\%}$	1.41 $_{-24.6\%}$	42.0 $_{-7.1\%}$	4.34 $_{+187.4\%}$
				Auto-Harness				
Meta-Harness	31.5 $_{+4.0\%}$	2.20 $_{-4.8\%}$	59.2 $_{-3.9\%}$	1.92 $_{+30.6\%}$	14.6 $_{-23.6\%}$	1.33 $_{-28.9\%}$	**49.6** $_{+9.7\%}$	2.43 $_{+60.9\%}$
HARNESSBRIDGE	**33.7** $_{+11.2\%}$	**1.23** $_{-46.8\%}$	60.2 $_{-2.3\%}$	**1.13** $_{-23.1\%}$	**20.2** $_{+5.8\%}$	**0.42** $_{-77.5\%}$	46.0 $_{+1.8\%}$	**1.48** $_{-2.0\%}$

Table 1 | Success rate (SR) and average token usage in millions across harnesses on Terminal-Bench 2.0 and SWE-bench Verified. Subscripts denote relative changes against Terminus 2. For SR, green indicates improvement and red indicates degradation. For token usage, green indicates reduction and red indicates increase. Best results are highlighted in bold.

3.2 Unified Instruction Fine-tuning

Training Formulation. We formulate learning the bidirectional interface as a unified conditional generation problem. Rather than training separate modules for observation and action projection, we parameterize both P_{obs} and P_{act} with a shared policy P_θ. Given an instruction specifying the projection objective, together with the task specification and current trajectory, P_θ is trained to generate the corresponding trajectory-grounded interface transformation. Under this formulation, observation projection and action projection differ only in their instruction and target format: the former produces a generator-visible state, while the latter produces an environment-facing pass/reject decision with grounded feedback when needed.

Data Curation. To construct supervised data, we instantiate observation and action projections with prompted instruction-tuned models under multiple intervention regimes, producing traces that include raw trajectories, projected states, proposed actions, projection decisions, feedback, and environment outcomes. We then filter these traces for high-quality supervision by retaining only successful trajectories and using an LLM judge to assess projection quality. Observation-projection examples are selected for schema consistency, faithful compression, and preservation of decision-critical evidence, while action-projection examples are retained when pass/reject decisions and rejection feedback are trajectory-grounded and actionable. More details are shown in Appendix C.

Raw Trajectory Preservation. HarnessBridge does not destructively overwrite the interaction history. The raw trajectory H_t is always retained as the authoritative record, while P_θ only decides what projected view $\widetilde{H}_t$ should be exposed to the generator at each turn. Thus, compression is triggered selectively rather than applied as an irreversible update after every interaction step. This design mitigates two common risks of trajectory compression: hallucinated summaries and the accidental removal of details needed for later reasoning. When compression is used, we require the projected state to be provenance-aware: compressed statements, unresolved errors, constraints, and task-relevant facts must be grounded in specific spans of the original trajectory.

4 Experiments

Our experiments are designed to answer the following three questions. **Q1:** Can HarnessBridge effectively reduce the token consumption of agents while maintaining strong task performance? **Q2:** Can HarnessBridge, instruction-tuned on Qwen3.5, generalize to larger commercial models and different environments? **Q3:** Do the bidirectional mappings in HarnessBridge contribute effectively to agent–environment interaction?

Harness	GPT-5.4-Nano		GPT-5.4		DeepSeek-V4-Flash		DeepSeek-V4-Pro		Claude-Opus-4.7	
	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓
Terminus 2	18.0	9.80	53.9	9.41	49.4	2.18	57.3	1.02	64.0	0.26
HARNESSBRIDGE	22.5 $_{+25.0\%}$	0.91 $_{-90.7\%}$	53.9	0.99 $_{-89.5\%}$	53.9 $_{+9.1\%}$	1.22 $_{-44.0\%}$	57.3	0.94 $_{-7.8\%}$	65.2 $_{+1.9\%}$	0.19 $_{-26.9\%}$

Table 2 | Success rate (SR) and average input token usage (in millions) on Terminal-Bench 2.0 across models.

4.1 Experiment Setup

Benchmarks and Baselines. We focus on coding as a representative class of tool-intensive, long-horizon agent tasks, evaluating on Terminal-Bench 2.0 (Merrill et al., 2026) and SWE-bench Verified (Jimenez et al., 2024). We report the success rate and average input-token consumption for the generator. The baseline comparison includes representative scaffolds from several categories: *Terminal-Bench scaffolds*: Terminus 2 (Merrill et al., 2026) (the official TB-2.0 reference scaffold) and Terminus-KIRA (KRAFTON AI and Ludo Robotics, 2026); *SWE-Bench scaffolds*: mini-swe-agent (Yang et al., 2024); *production harnesses*: OpenHands (Wang et al., 2024) and Qwen-Coder; and *Automatic scaffold optimization*: Meta-Harness (Lee et al., 2026), the closest prior work that explicitly searches over scaffold behavior.

Model Backbones. Experiments use seven frozen generator models with different parameter sizes: Qwen3.5-35B-A3B (Qwen Team, 2026), GLM4.7-Flash (Z.AI, 2026), DeepSeek-V4-Flash, DeepSeek-V4-Pro (DeepSeek-AI, 2026), GPT-5.4-nano, GPT-5.4 (OpenAI, 2026), and Claude-Opus-4.7. Qwen3.5-35B-A3B and GLM4.7-Flash are open-sourced and served with SGLang (Zheng et al., 2024) at temperature 0.6, while the remaining models are accessed via commercial APIs. HarnessBridge is initialized from the lightweight Qwen3.5-0.8B (Qwen Team, 2026) and instruction-tuned to support harness-control decisions. Additional details are provided in Appendix C.

4.2 Main Results

The experimental results provide a clear affirmative answer to **Q1**. Table 1 reports success rate (SR) and average input-token consumption across harnesses, generators, and benchmarks. On SWE-bench Verified with Qwen3.5-35B-A3B, HARNESSBRIDGE remains competitive while using the lowest token budget among reported harnesses. Overall, HARNESSBRIDGE jointly improves success rate and reduces token consumption compared to other harness design.

It is worth noting that HarnessBridge is instruction-tuned only on trajectories from SWE-bench. Therefore, Terminal-Bench 2.0 can be viewed as an out-of-domain environment. Nevertheless, HarnessBridge still achieves strong performance, suggesting that the learned harness policy generalizes beyond the training environment. On Terminal-Bench 2.0, HARNESSBRIDGE achieves the highest success rate under both generators, reaching 33.7% with Qwen3.5-35B-A3B and 20.7% with GLM-4.7-Flash. Compared with Terminus 2, this corresponds to success rate gains of +11.2 and +8.4 percentage points, while reducing average token consumption by 42.4% and 77.0%, respectively.

4.3 Generalization on Commercial Models

Q2 is to answer whether a harness supervised from traces of a single generator transfers to generators with different inference-time behaviors. To evaluate this, we test HARNESSBRIDGE on five unseen generators spanning multiple model families and capability levels (Table 8). Across all settings, HARNESSBRIDGE preserves or improves success rate while reducing token consumption. The largest improvement is observed on GPT-5.4-Nano, where success rate increases from 15.7% to 22.5% and average token usage decreases from 9.77M to 0.91M. On GPT-5.4, HARNESSBRIDGE maintains the same success rate of 53.9% while reducing token usage by approximately 89%. On DeepSeek-V4-Pro, where the baseline is already efficient, it matches the baseline success rate of 57.3% with slightly lower token cost. On Claude-Opus-4.7, HarnessBridge maintains strong task performance while further improving efficiency. It increases success rate from 64.0% to 65.2% and reduces token usage by 26.9%.

These results suggest that HARNESSBRIDGE captures interaction-level control patterns that transfer beyond the generator used for supervision. The gains are larger when the baseline incurs high interaction cost, and smaller when the baseline is already efficient.

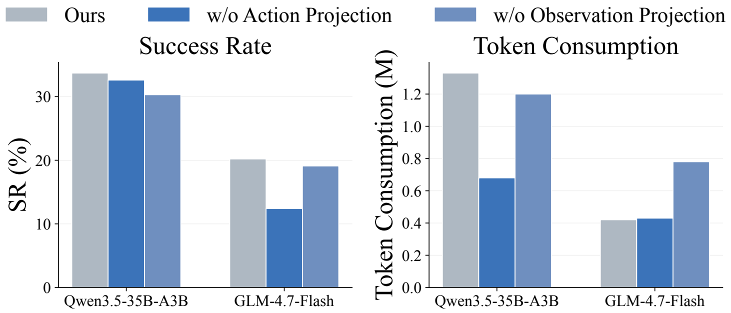


Figure 3 | Ablation study of objection and action projections on Qwen3.5-35B-A3B and GLM-4.7-Flash.

4.4 Robustness to the Supervision Source

A natural concern is whether harness-control supervision must be sampled from the same generator it will later control. We test this by reconstructing the full pipeline with DeepSeek-V4 as the sampling model in place of Qwen3.5-35B-A3B, yielding HarnessBridge-D. Despite the change in data source, HarnessBridge-D improves task success over the Terminus 2 baseline across heterogeneous generators while substantially reducing token usage—including the GPT-5.4 family, which shares no lineage with the sampling model (e.g., $18.0 \rightarrow 23.6$ success rate with a $\sim$90% token reduction on GPT-5.4-Nano, and $57.3 \rightarrow 59.6$ on DeepSeek-V4-Pro). This indicates that the learned interface policy captures general properties of effective harness control rather than artifacts of a particular generator's trajectories. Full curation details and per-model results appear in Appendix D.3 and D.4.

4.5 Ablation Study

We ablate the action projection and observation projection modules on Terminal-Bench 2.0 to assess their individual contributions. Figure 3 reports success rate and token consumption across two backbone models. Removing either component reduces success rate on both backbones, indicating that both projections contribute to task performance. Action projection regularizes the model's outputs toward executable and task-relevant operations, while observation projection organizes environment feedback into a more compact and informative form for downstream reasoning. While some ablated variants achieve lower token usage, this comes at a consistent cost in success rate, suggesting that the projections improve task completion rather than merely compressing context. Together, the results support the inclusion of both modules.

5 Analysis

5.1 Trajectory Case Studies

Figure 2 illustrates the two projection mechanisms of HarnessBridge using examples from SWE-bench Verified. In the `django-13964` trajectory on the left, the agent has accumulated a long history of repository exploration, code inspection, and test attempts by turn 67. The raw context is dominated by intermediate steps that are no longer directly relevant to the next action. Observation projection reduces this context by preserving the initial task description and the most recent turns, summarizing earlier exploratory steps, and omitting redundant turns. Importantly, the failed test invocation from the most recent context is retained as an active-state item, ensuring that the current blocker remains visible after compression. This example shows that observation projection can reduce context noise while preserving decision-relevant state.

On the right, the `xarray-4356` trajectory illustrates a failure mode addressed by action projection. Although the agent has identified the relevant code path and likely bug mechanism, it repeatedly performs indirect checks rather than testing the modified codebase. HarnessBridge rejects the next redundant check and provides a concrete redirect to run the actual test against the patched code. The agent's next action follows this suggestion and produces a meaningful signal about the patch state. This example shows that action projection can interrupt redundant verification loops and redirect the agent toward more informative environment feedback without modifying the underlying generator.

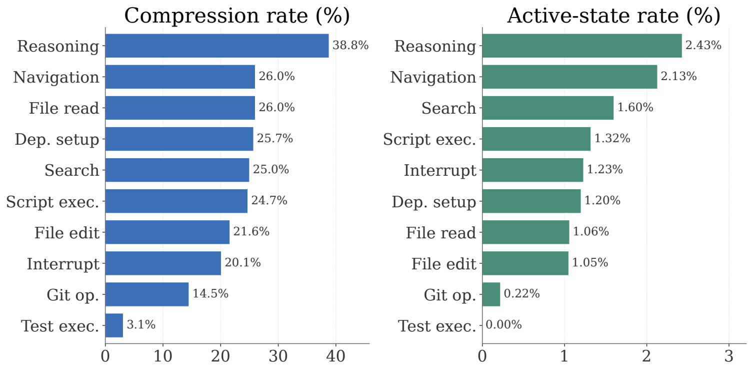


Figure 4 | Observation projection behavior by action category. *Left*: compression rate, the per-turn fraction of downstream invocations in which the projection compressed the turn. *Right*: active-state rate, the per-turn fraction of invocations that lifted content from the turn into the persistent active-state block. Both metrics are means of per-turn rates over all turns in each category.

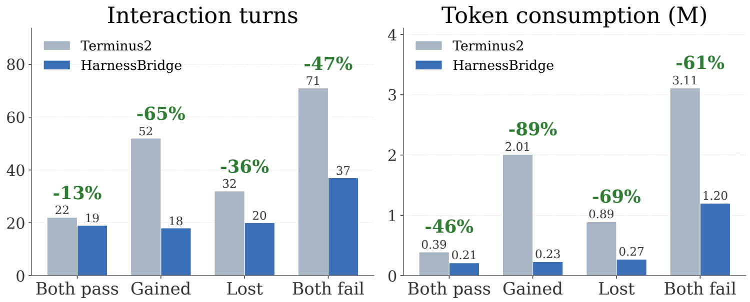


Figure 5 | Outcome-level comparison of interaction turns and token consumption between Terminus2 and HarnessBridge across task success categories.

5.2 Category-Level Analysis

To understand *what* HarnessBridge has learned, we classify every agent turn into one of ten mutually-exclusive action categories defined by syntactic patterns over the action's command field (e.g. `file_edit`, `script_execution`, `reasoning_only`; refer to Appendix B) and measure how HarnessBridge treats each category.

Figure 4 reports two per-turn lifecycle averages by category. Both metrics share the same construction: for each turn t, we record the projection's decision at every downstream invocation that re-evaluated the trajectory, compute a single ratio per turn, and then average those ratios over all turns in the category. The *compression rate* of a turn is the fraction of its downstream invocations in which the projection compressed that turn; the *active-state rate* of a turn is the fraction of its downstream invocations in which the projection promoted that turn into the persistent active-state block. Two patterns are immediately visible. First, compression is highly selective: rates range from 3.1% on `test_execution` — the category most likely to carry the decisive verification signal — to 38.8% on `reasoning_only`. Categories that are information-dense but quickly stale (reasoning, navigation, search, file reads) are compressed at roughly 20–40%, while content with longer downstream utility (test output, git diffs) is largely preserved. Second, the active-state ranking tracks the compression ranking closely (`reasoning_only`, `navigation`, `search` top both panels): the projection is *not* simply discarding what it compresses, but distilling key facts from those turns into a persistent slot. The two mechanisms operate in tandem.

5.3 Outcome-Level Efficiency Analysis

We evaluate HarnessBridge against the baseline across 178 tasks using Qwen3.5-35B-A3B and GLM4.7-Flash generators, recording pass/fail, turn count, and token consumption per run. Tasks are partitioned by joint outcome: *Both passed*, *Gained* (HarnessBridge only), *Lost* (baseline only), and *Both fail*. Figure 5 reports per-category task counts and mean turns and tokens for each harness, with relative deltas.

Findings. HarnessBridge reduces both turns and token consumption across all outcome categories. In each category, the reduction in tokens is larger than the reduction in turns, suggesting that HarnessBridge not only shortens trajectories but also produces more compact per-turn context. This efficiency gain is observed regardless of whether the task ultimately succeeds.

The *Gained* category shows the largest difference. On tasks where HarnessBridge succeeds and the baseline fails, HarnessBridge reaches a passing solution in 18 turns on average, compared with 52 turns for the baseline (-65%), while using only 11% of the baseline token budget (-89%). This pattern suggests that the baseline often spends many turns on unproductive exploration, whereas HarnessBridge can converge earlier by maintaining a more compact and decision-relevant interaction history. Thus, the observed efficiency improvements are associated not only with lower cost, but also with improved task outcomes in a subset of cases.

6 Conclusion

HarnessBridge recasts the agent–environment interface as a learnable harness policy. Through jointly trained bidirectional projections, it compresses raw trajectories into decision-critical agent context and maps proposed actions into executable transitions or trajectory-grounded rejections. HarnessBridge achieves competitive or stronger performance with lower token cost and shorter trajectories on the benchmarks, while transferring from small to larger commercial models.

References

Mohamed Aghzal, Gregory J Stein, and Ziyu Yao. Why do llm-based web agents fail? a hierarchical planning perspective. *arXiv preprint arXiv:2603.14248*, 2026.

Anthropic. Effective context engineering for ai agents. `https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents`, 2025a. Anthropic Engineering Blog.

Anthropic. Effective harnesses for long-running agents. `https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents`, 2025b.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. tau2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Ching Chang, Yidan Shi, Defu Cao, Wei Yang, Jeehyun Hwang, Haixin Wang, Jiacheng Pang, Wei Wang, Yan Liu, Wen-Chih Peng, et al. A survey of reasoning and agentic systems in time series with large language models. *arXiv preprint arXiv:2509.11575*, 2025.

DeepSeek-AI. Deepseek v4 technical documentation. `https://fe-static.deepseek.com/chat/transparency/deepseek-V4-model-card-EN.pdf`, April 2026. Model card covering DeepSeek-V4-Pro and DeepSeek-V4-Flash. Accessed: 2026-05-25.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. davinci-env: Open swe environment synthesis at scale, 2026. URL `https://arxiv.org/abs/2603.13023`.

Kaiqiao Han, Tianqing Fang, Zhaowei Wang, Yangqiu Song, and Mark Steedman. Concept-reversed winograd schema challenge: Evaluating and improving robust reasoning in large language models via abstraction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 229–243, 2025.

Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jianguang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation, 2024. Accepted by KDD 2025 (Research Track).

Yuxuan Huang, Yihang Chen, Haozheng Zhang, Kang Li, Huichi Zhou, Meng Fang, Linyi Yang, Xiaoguang Li, Lifeng Shang, Songcen Xu, et al. Deep research agents: A systematic examination and roadmap. *arXiv preprint arXiv:2506.18096*, 2025.

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*, 2024. URL `https://openreview.net/forum?id=VTF8yNQM66`.

KRAFTON AI and Ludo Robotics. Terminus-kira: Boosting frontier model performance on terminal-bench with minimal harness, 2026. URL `https://github.com/krafton-ai/kira`.

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.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Yue Liu, Yingwei Ma, Yibo Miao, Yanhao Li, Yuchong Xie, Xinlong Yang, Zhiyuan Hu, Flood Sung, Jiaheng Zhang, and Bryan Hooi. Klong: Training llm agent for extremely long-horizon tasks. *arXiv preprint arXiv:2602.17547*, 2026.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Liangbo Ning, Ziran Liang, Zhuohang Jiang, Haohao Qu, Yujuan Ding, Wenqi Fan, Xiao-yong Wei, Shanru Lin, Hui Liu, Philip S Yu, et al. A survey of webagents: Towards next-generation ai agents for web automation with large foundation models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 6140–6150, 2025.

OpenAI. Gpt-5.4 model. `https://developers.openai.com/api/docs/models/gpt-5.4`, 2026. OpenAI API documentation. Accessed: 2026-05-25.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Prithvi Rajasekaran. Harness design for long-running application development. `https://www.anthropic.com/engineering/harness-design-long-running-apps`, March 2026. Anthropic Engineering Blog.

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. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. R1-searcher: Incentivizing the search capability in llms via reinforcement learning, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Haixin Wang, Huiyu Jiang, Jinan Sun, Shikun Zhang, Chong Chen, Xian-Sheng Hua, and Xiao Luo. Dior: Learning to hash with label noise via dual partition and contrastive learning. *IEEE Transactions on Knowledge and Data Engineering*, 36(4):1502–1517, 2023b.

Haixin Wang, Jinan Sun, Xiao Luo, Wei Xiang, Shikun Zhang, Chong Chen, and Xian-Sheng Hua. Toward effective domain adaptive retrieval. *IEEE Transactions on Image Processing*, 32:1285–1299, 2023c.

Haixin Wang, Hejie Cui, Chenwei Zhang, Xin Liu, Shuowei Jin, Shijie Geng, Xinyang Zhang, Nasser Zalmout, Zhenyu Shi, and Yizhou Sun. T^{2}po: Uncertainty-guided exploration control for stable multi-turn agentic reinforcement learning. *arXiv preprint arXiv:2605.02178*, 2026a.

Kangrui Wang, Pingyue Zhang, Zihan Wang, Yaning Gao, Linjie Li, Qineng Wang, Hanyang Chen, Yiping Lu, Zhengyuan Yang, Lijuan Wang, et al. Vagen: Reinforcing world model reasoning for multi-turn vlm agents. *Advances in Neural Information Processing Systems*, 38:172871–172933, 2026b.

Xiaoxuan Wang, Han Zhang, Haixin Wang, Yidan Shi, Ruoyan Li, Kaiqiao Han, Chenyi Tong, Haoran Deng, Renliang Sun, Alexander Taylor, et al. Arlarena: A unified framework for stable agentic reinforcement learning. *arXiv preprint arXiv:2602.21534*, 2026c.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents, 2024. *URL https://arxiv. org/abs/2407.16741*, 2(4):9, 2024.

Tianxin Wei, Ting-Wei Li, Zhining Liu, Xuying Ning, Ze Yang, Jiaru Zou, Zhichen Zeng, Ruizhong Qiu, Xiao Lin, Dongqi Fu, et al. Agentic reasoning for large language models. *arXiv preprint arXiv:2601.12538*, 2026.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022b.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Z.AI. Glm-4.7 series. `https://docs.z.ai/guides/llm/glm-4.7`, 2026. Model documentation. Accessed: 2026-05-25.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, and Xianyi Cheng. Tianyue ou, yonatan bisk, daniel fried, et al. 2023. webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A Preliminary Study

Model	Benchmark	SR (%)	Tokens (M)
Qwen3.5-0.8B	SWE	58.3	1.43
Qwen3.5-0.8B	TB	24.7	1.99
Qwen3.5-35B-A3B	SWE	59.1	1.38
Qwen3.5-35B-A3B	TB	29.2	0.53

Table 3 | Evaluation results across vanilla Qwen3.5-0.8B and Qwen3.5-35B-A3B models.

Harness Backbone Comparison. We compare different backbone models for the harness component, including vanilla Qwen3.5-35B-A3B and vanilla Qwen3.5-0.8B. The results show that the fine-tuned Qwen3.5-0.8B HarnessBridge model achieves performance and efficiency comparable to the much larger Qwen3.5-35B-A3B harness, while requiring substantially lower inference cost. This suggests that harness-control behavior can be effectively distilled into a small model.

Reserved Turns	SR (%)	Tokens (M)
N10	56.9	0.72
N20	56.4	0.65
N30	60.4	0.74
N50	61.3	0.96
N70	58.8	1.08
N100	58.0	1.40

Table 4 | Effect of the number of reserved turns on SWE-Verified.

Effect of Reserved History Window on Observation Projection. We study the effect of the reserved history window N on SWE-bench Verified. The reserved window controls when observation projection is activated: before this window, the full recent history is preserved; beyond it, compression and dropping are enabled. The results indicate that both overly aggressive and overly conservative compression can hurt performance or efficiency. A moderate history window achieves a better trade-off, suggesting the need for dynamically learned context management rather than a fixed compression rule.

Benchmark	Mode	SR (%)	Tokens (M)
SWE	Rules-only	60.9	1.74
SWE	Tolerant	61.5	1.50
SWE	Strict	56.7	2.08
TB	Rules-only	29.2	4.25
TB	Tolerant	30.3	2.34
TB	Strict	25.8	2.15

Table 5 | Action projection under different rejection modes: rules-only, tolerant, and strict.

Effect of rejection mode on Action Projection. We evaluate action projection under different rejection modes. In the rule-only mode, the harness rejects only simple invalid actions, such as malformed syntax, empty commands, duplicated commands, or repeated command patterns, and returns predefined feedback asking the generator to revise its action. However, rule-based rejection alone has limited impact, since many inefficient actions are syntactically valid but semantically unproductive. In the learned rejection setting, strict rejection can over-intervene and reject otherwise useful actions, reducing task success. A more tolerant rejection mode achieves a better balance between blocking unproductive environment steps and preserving valid agent behavior. These results highlight the importance of calibrating action projection to avoid both under-intervention and excessive rejection.

B Action Category Analysis

We partition agent turns into ten mutually exclusive categories (Table 6), defined by syntactic patterns over the action's command field. We chose these categories to (a) cover the dominant agent behaviors observed on SWE-bench Verified and Terminal Bench while (b) being deterministically derivable from command syntax, so the classifier is reproducible and does not require semantic inspection of the observation.

Intuitively, the ten categories cover three behavioral modes: *information gathering* (`file_read`, `search`, `navigation`, `git_operation`), *state-changing action* (`file_edit`, `dependency_setup`, `interrupt_abort`), and *execution/verification* (`script_execution`, `test_execution`). `reasoning_only` is the residual class for turns whose action carries no parseable command, characteristic of larger reasoning-tuned generators. Submission turns (`COMPLETE_TASK_AND_SUBMIT`, `mark_task_complete`) are detected by the classifier but excluded from analysis: a submission is always the final turn of its trajectory and therefore has no downstream projection invocations against which to measure a lifecycle.

Category	**Matching rule on command text**	**Example**
`test_execution`	Contains `pytest`, `python -m pytest`, `make test`, `cargo test`, or `npm test`.	`pytest tests/test_io.py`
`file_edit`	Contains `sed -i`, `cat >`, `cat «`, `tee`, `echo >`, or `patch`.	`sed -i ’s/foo/bar/’ f.py`
`dependency_setup`	Contains `pip install`, `apt-get install`, `cmake`, `gcc`, `make`, or `git clone`.	`pip install numpy`
`git_operation`	Starts with `git diff`, `git log`, `git status`, `git commit`, `git add`, `git show`, or `git blame`. Excludes `git clone` (classified as dependency setup).	`git diff HEAD~1`
`search`	Contains `grep`, `rg`, `ag`, `ack`, or `find` with `-name`/`-type`.	`grep -rn TODO src/`
`file_read`	Reads file contents without modifying them: `cat` (no redirect), `head`, `tail`, `sed -n`, `less`, or `more`.	`cat README.md`
`script_execution`	Invokes an interpreter: `python3`, `python`, `node`, `bash`, `rscript`, or `sqlite3`. Test patterns are routed to `test_execution` first.	`python repro.py`
`navigation`	Starts with `ls`, `pwd`, `tree`, or `cd`.	`ls -la src/`
`interrupt_abort`	Contains `C-c`, `kill`, `pkill`, or is exactly `q`/`exit`.	`C-c` (cancel REPL)
`reasoning_only`	Action JSON contains no parseable command, or is empty/raw text.	Agent "thinking aloud"

Table 6 | Action category classification rules. Each turn matches the highest-priority rule for which any of its commands satisfies the pattern. `reasoning_only` is the residual class for turns whose action object contains no parseable command.

C Details of Data Curation

Although we use a shared policy to learn control in both directions, the data construction process needs to be designed separately for the two components. Following the benchmark-driven harness optimization protocol of Meta-Harness (Lee et al., 2026), we construct supervision from solved SWE-bench Verified trajectories. The supervision is used only for the harness-level projection modules, while the generator model remains frozen. We use Qwen3.5-35B-A3B as the generator and initial harness backbone to produce interaction traces under the HarnessBridge pipeline.

We automate the data construction pipeline by: (1) extensively prompting existing models to generate candidate supervision; and (2) applying strict data-filtering criteria to retain high-quality examples. Our data curation is designed to construct supervision that is both correct and useful for learning harness-level decisions, rather than merely collecting large numbers of raw intervention traces. Starting from approximately 40K raw supervision candidates, we apply separate curation principles for action projection and observation projection.

For action projection, each example specifies whether a proposed action should be passed to the environment or rejected with feedback. We pay particular attention to rejected-action examples, since they are naturally sparse and directly teach the harness when not to expose a proposed action to the environment. Rejection examples are derived from logged rejected actions and include the rejection category, concern, grounding evidence, and suggested revision. We retain rejection examples only when the rejection is trajectory-grounded, followed by a successful subsequent correction, and improves interaction efficiency. In other words, a rejected action is used as supervision only if the rejection is supported by the existing trajectory, the revised behavior remains correct, and the intervention helps avoid unnecessary environment steps or token consumption. This prevents the model from learning arbitrary or unverifiable rejections. Pass examples are constructed from actions that were successfully executed in solved trajectories. To avoid an always-pass bias, we balance the final action-projection set to a 50:50 PASS/REJECT ratio, yielding $2{,}682$ examples. We also avoid constructing artificial negative examples by simply flipping previous rejection labels into PASS labels, since we observed that such label flipping creates a substantial distribution shift and encourages over-rejection behavior.

For observation projection, each example consists of a trajectory history and a structured target indicating which turns should be preserved, summarized, or omitted. The target also includes an active-state index that records decision-relevant facts to keep visible to the generator. One of the main objectives is to learn an appropriate compression policy rather than an overly aggressive summarizer. We therefore collect examples under multiple reserved-history windows, ranging from small windows such as $N = 10$ to larger windows such as $N = 100$, to expose the model to different context-compression regimes. Very small windows tend to force overly harsh summarization, while very large windows provide limited compression signal. We consequently emphasize intermediate regimes, such as $N = 30$ and $N = 50$, which better capture the desired trade-off between preserving decision-critical context and reducing token cost. We further subsample keep-all cases and balance examples across different compression regimes, so that the model does not collapse into either always preserving the full trajectory or compressing too aggressively. After filtering and deduplication, the final observation-projection set contains $2{,}723$ examples.

After LLM-as-judge filtering with GLM-4.7-Flash, deduplication, trajectory-level capping, rejection-quality validation, and distribution balancing, we retain $5{,}405$ high-quality examples for supervised fine-tuning. Both components are converted into a unified instruction-following format and are used to fine-tune a Qwen3.5-0.8B model with supervised fine-tuning.

D Experiment Implementation

D.1 More Experiment Setup

All experiments are conducted using the Harbor framework, with open-source models served on NVIDIA H200 GPUs.

Meta-Harness. We tune the Terminus 2 harness code on 100 tasks sampled from SWE-bench Verified and evaluate on the full SWE-bench Verified and Terminal-Bench 2.0 suites. Tuning is performed for 3 iterations using Qwen-3.5-35B-A3B as the optimization model.

HARNESSBRIDGE. We set the observation projection window size per backbone to approximately the median trajectory length of the baseline (Terminus 2) on each evaluation benchmark, without reference to HARNESSBRIDGE's own performance. This calibrates the window to each model's interaction profile, accounting for substantial differences in trajectory length. For instance, Claude-Opus-4.7 tends to complete tasks in fewer turns, whereas DeepSeek-V4-Pro variants typically produce longer trajectories. By default, we use a window size of 20, except for Claude-Opus-4.6 (10) and DeepSeek-V4-Pro and DeepSeek-V4-Flash (30); on SWE-bench Verified with Qwen-3.5-35B-A3B, we use 50. The action projection cap is set to 5 per task on Terminal-Bench 2.0 and 10 on SWE-bench Verified.

SFT Training. We fine-tune Qwen-3.5-0.8B using the Adam optimizer with a learning rate of $1\mathrm{e}{-5}$, batch size of 64, and bf16 precision for 3 epochs on NVIDIA H200 GPUs.

HarnessBridge Deployment Cost. In addition to generator token consumption, we measure the inference overhead introduced by the HarnessBridge controller. HarnessBridge uses a lightweight Qwen3.5-0.8B model for observation and action projection, whereas the evaluated generators are substantially larger open-source or commercial models. Although the controller processes approximately $3\times$ as many tokens as the generator in our runs, its per-token inference cost is much lower. Under a parameter-normalized compute proxy, the controller costs only $0.8/35 \approx 2.3\%$ of a 35B dense generator per token. Therefore, even with $3\times$ more controller tokens, its compute-weighted overhead is approximately $3 \times 0.8/35 \approx 6.9\%$ of the corresponding 35B-generator cost.

This proxy is conservative for practical deployment: Qwen3.5-0.8B can be served with a low memory footprint and high throughput on commodity accelerators, and its marginal inference cost is small even compared with the open-source Qwen3.5-35B-A3B generator. When paired with commercial API-based generators, the controller is deployed locally and its cost is further amortized relative to the API-side generator cost, making the additional harness computation small in typical end-to-end deployments.

After including all controller input/output tokens, HarnessBridge still reduces total compute-weighted inference cost across the evaluated settings. This indicates that the reported efficiency gain is not merely a transfer of computation from the generator to the harness, but an end-to-end reduction in interaction cost. In practice, the lightweight controller acts as a low-cost interface policy that prevents expensive generator-context growth and unproductive environment interactions, so a modest amount of local harness computation yields a substantially larger reduction in downstream generator inference cost.

D.2 Baseline Tuning and Evaluation Fairness

To ensure a fair comparison, all harnesses are evaluated under the same generator backbone, benchmark split, decoding configuration, and task-level execution budget. For each generator, we use the same serving backend and temperature setting across harnesses, and we keep the benchmark tasks, timeout constraints, and success criteria fixed. We do not tune any harness on the evaluation trajectories. For baselines that expose no learning or search procedure, including Terminus 2, mini-SWE-agent, OpenHands, and Qwen-Coder, we use their official or recommended configurations and only adapt interface-level details necessary to connect them to the same generator and benchmark environment. We do not perform additional benchmark-specific prompt or policy optimization for these fixed baselines.

This protocol is intended to separate harness-level optimization from test-time evaluation: each method is allowed to use its standard harness construction mechanism, but all methods are compared under the same frozen generators, same task suites, and same execution constraints.

D.3 Data Curation for HarnessBridge-D

For HarnessBridge-D, we follow the curation principles of Appendix C, with adaptations motivated by the characteristics of DeepSeek-V4 trajectories. As noted in Section D.1, DeepSeek-V4 variants produce substantially longer and more reasoning-heavy trajectories than Qwen3.5-35B-A3B, which makes both the verbosity of individual turns and the length of the trajectory history the dominant sources of low-quality supervision. As before, we retain supervision only from solved trajectories, so that every example is grounded in a correct task completion.

Action projection. We draw action-projection supervision only from trajectories in which HarnessBridge-D solves the task, and we condition the retention criterion on the baseline (Terminus 2) outcome. For *both-pass* tasks, where the baseline already solves the task, the controller contributes only efficiency; we therefore retain such trajectories as supervision only when they reduce both token usage and interaction turns relative to the baseline. For *fail-to-pass* tasks, where the baseline fails but HarnessBridge-D succeeds, the trajectory already demonstrates that harness control enables a new solve; we apply a looser criterion and retain it when it saves either tokens or turns. In both cases we exclude trajectories that yield no efficiency gain, since training on turn-increasing examples teaches the validator to intervene where intervention is unwarranted, producing over-rejection.

We validate this criterion in Table 7. The *turn-agnostic* recipe retains all correct trajectories, including turn-increasing ones, and balances the set 50:50 (reject:pass); the *turn-saving* recipe retains only efficiency-improving trajectories, which shifts the natural balance to 7:10. The turn-agnostic recipe degrades success rate *below* the no-harness baseline on both DeepSeek-V4-Pro (57.3 → 48.3) and DeepSeek-V4-Flash (49.4 → 43.8), with a high validator rejection rate ($\sim$37%) and frequent timeouts. The turn-saving recipe instead improves over the baseline (57.3 → 59.6 and 49.4 → 51.7) while

Generator	Recipe	SR (%)	Reject rate (%)
DeepSeek-V4-Pro	Terminus 2	57.3	–
	Turn-agnostic	48.3	36.7
	Turn-saving	**59.6**	18.3
DeepSeek-V4-Flash	Terminus 2	49.4	–
	Turn-agnostic	43.8	37.2
	Turn-saving	**51.7**	19.3

Table 7 | Effect of action-projection curation recipe on Terminal-Bench 2.0. *Turn-agnostic* includes correct trajectories that increase turns (50:50 reject:pass); *turn-saving* retains only turn-saving trajectories (7:10 reject:pass). The turn-agnostic recipe falls below the no-harness baseline; the turn-saving recipe is reported as HarnessBridge-D.

Harness	**GPT-5.4-Nano**		**GPT-5.4**		**DeepSeek-V4-Flash**		**DeepSeek-V4-Pro**	
	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓	SR ↑	Token ↓
Terminus 2	18.0	9.80	53.9	9.41	49.4	2.18	57.3	1.02
HarnessBridge-D	23.6 $_{+31.1\%}$	1.00 $_{-89.8\%}$	52.8 $_{-2.0\%}$	0.51 $_{-94.6\%}$	51.7 $_{+4.7\%}$	0.92 $_{-57.8\%}$	59.6 $_{+4.0\%}$	0.98 $_{-3.9\%}$

Table 8 | Success rate (SR) and average input token usage (in millions) on Terminal-Bench 2.0 across models using HarnessBridge-D.

roughly halving the rejection rate and cutting timeouts. A validator trained on turn-increasing supervision over-rejects to the point of making the harness worse than the baseline Terminus 2; curating toward genuinely efficiency-improving trajectories is therefore essential. The turn-saving recipe is the one we report as HarnessBridge-D.

Observation projection. Each example again consists of a trajectory history and a structured target marking which turns to preserve, summarize, or drop. Given the verbosity of DeepSeek-V4 trajectories, we add two length-oriented filters. First, we discard any example whose target output exceeds 6K tokens, since such targets correspond to summaries that are themselves too long to provide a useful compression signal. Second, we require that every structured target parse successfully into the expected schema, discarding malformed targets. To prevent a small number of very long trajectories from dominating the set, we cap supervision at 20 examples per trajectory. Finally, to preserve decision-critical recent context, we protect the most recent turns from compression: the last fewer than three turns are never dropped, and no summarization entry is applied within that recent window. After filtering and deduplication, the final observation-projection set contains 3575 examples.

After LLM-as-judge filtering with DeepSeek-V4-Pro, deduplication, trajectory-level capping, and the length- and efficiency-based criteria above, we retain 5287 high-quality examples for fine-tuning the HarnessBridge-D controller.

D.4 Experiment Result for HarnessBridge-D

Table 8 reports success rate (SR) and average input token usage on Terminal-Bench 2.0. Trained only on DeepSeek-sampled supervision, HarnessBridge-D substantially reduces input token consumption across all generators while maintaining comparable SR. The reductions are largest on the GPT-5.4 family, whose unconstrained baselines are the most token-heavy (9.80M and 9.41M input tokens): on GPT-5.4-Nano it cuts usage by roughly 90% (to 1.00M), with a similar reduction on GPT-5.4. On DeepSeek-V4-Pro, whose baseline is already token-lean (1.02M), the reduction is naturally smaller (-3.9%). The largest savings appear on the models with the longest baseline trajectories (the GPT-5.4 family). More broadly, these results suggest that the effectiveness of HarnessBridge is largely independent of the data-sampling model: harness-control supervision drawn from different generators yields a similarly strong, transferable controller.

E Limitation

Our evaluation is confined to coding settings. The harness-control mechanism is not specific to code, however: observation and action projection operate over generic tool-use trajectories and manage long interaction histories rather than any code-specific structure, so we expect the approach to extend to other tool-heavy, long-horizon agentic domains such as web navigation, computer use, and multi-step research workflows. Due to evaluation cost, we report single-run results and accordingly emphasize relative trends over absolute values. Empirically validating this generalization remains an important direction for future work.

F Instruction Prompt

PROMPT (Observation Projection)

Role

Your job is to manage the conversation history so the agent stays focused and within the token budget. Remove or summarize redundant and irrelevant tool responses and conversation. Preserve precise and important information.

Input (JSON)

- **task**: original task instruction
- **turn**: current turn number (1-indexed)
- **history**: full list of prior turns, each containing:
  - `turn`: turn number
  - `reasoning`: the agent's THOUGHT/analysis before acting
  - `action`: the command the agent executed
  - `observation`: the terminal output from that command
  - `tokens`: token count for this turn
- `history` contains all completed turns. `last_observation` is NOT in history — it is the new observation from the current turn, not yet appended.
- **last_observation**: terminal output from the current turn's command, not yet in history
- **total_tokens**: current conversation token count
- **budget**: soft token target

Your Job

1. Decide which turns to **SUMMARIZE** or **DROP**. Any turn you do not mention is kept unchanged — do not list turns you are keeping.
2. Extract an **active-state index** from prior turns. Include any decision-relevant items that should remain visible, such as unresolved errors, open tests, confirmed facts, constraints, pending subtasks, important files/functions, hypotheses, prior decisions, failed approaches to avoid, or partial progress. This index may contain items from multiple turns and is placed before the history.

The harness assembles the final prompt: active-state index at the top, then history in chronological order (keeping originals by default, replacing summarized turns with your summaries, and skipping dropped turns). `last_observation` is appended separately by the harness. The harness handles formatting — write summaries as plain text.

Principles

- **Always keep**: system prompt, task description, and the most recent turns (the harness protects these automatically). Do not summarize or drop any recent turns.
- **Bias toward keeping** turns containing: test output, compilation errors, numerical results, error messages with codes/paths/line numbers, strategy changes, code snippets, function signatures. These lose critical detail when summarized.

- **Summarize or drop older turns** only when they have been clearly superseded, resolved, or duplicated by later turns.
- **When summarizing**:
  - Preserve exact values verbatim: file paths, line numbers, error codes, function names, variable names. Never paraphrase these.
  - Include what was tried, what resulted, and whether it advanced the task.
  - 1–3 sentences. Shorter is better if the turn was minor.

Output

Valid JSON, nothing else. Only list turns you are changing. Turns not listed are kept unchanged.

```
{
  "summarize": [
    {"turn": N, "content": "1-3 sentence summary with exact values"},
    ...
  ],
  "drop": [
    {"turn": N, "reason": "1-3 sentence reason for dropping this turn"},
    ...
  ],
  "active_state": [
    {"turn": N, "content": "one-line decision-relevant item with exact
        values"}
  ]
}
```

PROMPT (Action Projection)

Role

You are a decision verifier for an AI coding agent. Before the agent's proposed command executes, you assess whether it is a sound use of the agent's limited turns and budget given the task and trajectory so far.

Input (JSON)

- **task**: original task instruction
- **history**: full list of prior turns, each containing:
  - `turn`: turn number
  - `action`: the command the agent executed (may embed the agent's prior reasoning as JSON)
  - `observation`: the terminal output from that command
- **agent_reasoning**: the agent's analysis/plan for the proposed command
- **proposed_command**: the command about to be executed
- **is_submission**: whether this is a submission/task-complete action

Decide: PASS or REJECT

REJECT when the proposed command is a clear misuse of the agent's turns. Common patterns include (non-exhaustive — other concerns qualify if you can meet the same evidence bar):

- **SHALLOW_EDIT**: The agent is about to make an edit that does not engage with the actual mechanism the task describes — pattern-matching on surface keywords or touching adjacent code rather than tracing the real logic path the task requires.

- **PREMATURE_SUBMISSION**: The agent is submitting a fix without having verified that it addresses the core issue described in the task, or the agent's own observations suggest the fix is incomplete or wrong.
- **WASTEFUL_EXPLORATION**: The agent has already identified the relevant code location and mechanism (evidenced by prior edits, specific file/line references in reasoning, or explicit statements about the bug location), but is spending 3+ additional turns on further reading or searching that does not target the identified location.
- **CONTRADICTED_PATH**: The agent's own earlier observations contain evidence (error messages, test failures, incorrect output) that the current approach is flawed, and the agent has not addressed or acknowledged that evidence in its reasoning — yet is continuing the same logical approach with a different command.

Mitigating Factors

Before rejecting, consider whether any of these mitigating factors apply:

- The command targets the logical path the task describes, even if you would implement it differently.
- The agent's reasoning demonstrates it traced the relevant mechanism, even briefly.
- The task is straightforward enough that direct action without deep tracing is appropriate.
- The agent has not yet identified a specific bug location or mechanism and is still in legitimate exploration.
- This is the agent's first attempt at an edit or fix — allow it to try and learn from the result rather than pre-empting it.
- The projection has already rejected the same category of issue 3+ times in this trajectory and the agent has not changed approach. Further rejections of the same pattern are unlikely to help — PASS and let the agent attempt execution so it can learn from actual failure output.
- You cannot articulate a specific concern backed by evidence from the task or the history.

Default to PASS. When in doubt, PASS. The cost of a false reject (wasted turn + lost momentum) is higher than the cost of letting a questionable command through (the agent can course-correct from a bad result, but cannot recover a rejected turn).

On REJECT you MUST populate `concern`, `evidence`, and `suggestion` with specific references. The `evidence` field must include a direct quote from the task or history that contradicts the proposed command. If you cannot provide a direct quote, you must PASS.

Output

Valid JSON, nothing else.

On PASS, emit only the decision:

```
{"decision": "PASS"}
```

On REJECT, populate every field:

```
{
  "decision": "REJECT",
  "category": "SHALLOW_EDIT" |"PREMATURE_SUBMISSION"|"WASTEFUL_EXPLORATION"|
      "CONTRADICTED_PATH",
  "concern": "what the command fails to address or why it is wasteful",
  "evidence": "specific quote from task, observation, or agent reasoning
     that supports the concern",
  "suggestion": "concrete next step - a specific file to read, function to
     trace, command to run, or assumption to verify"
}
```

LLM Judge PROMPT (Action Projection)

Role

You are an expert evaluator scoring training examples for an LLM action-projection model. The model's job is to project whether a proposed agent command should PROCEED or BE BLOCKED before it executes. You will be shown the official projection policy (tolerant or strict), the model's INPUT (`task` + `history` + `agent_reasoning` + `proposed_command` + `is_submission`), and the model's OUTPUT (`decision` + `category` + `concern` + `evidence` + `suggestion`).
Your job: score this (input, output) pair on seven axes from 1 to 5, and return STRICT JSON. Use the rubrics below.

Policy Reference

`{policy}`

Axis Rubrics

1. INPUT WORTHINESS (1-5) – gates the record

Decide whether this INPUT is one the model should be trained on at all — i.e. whether the `proposed_command` is genuinely a decision-worthy moment. Look ONLY at the input here.

- **5**: Decision is meaningfully borderline. Multiple of:
  - `proposed_command` is a real edit (`sed -i`, write, patch), a build/install/run command with side effects, or a submission (`is_submission=true`)
  - History contains substantive context: prior edits, observed test results, error messages, or the agent has made claims about the bug location
  - `agent_reasoning` is non-trivial and shows a plan (not empty, not "I will run the command")
  - Both PROCEED and BLOCK are defensible after careful reading, OR the case is a clear BLOCK for a non-obvious reason
- **4**: Genuine edit/submission with adequate context, but the decision is fairly clear (one side is much stronger).
- **3**: Borderline. Edit is real but the call is obvious, OR the history is thin but the command is substantive enough to teach something.
- **2**: Marginal training value: edit is small/cosmetic, OR history is too thin to ground a real decision, OR `agent_reasoning` is empty/boilerplate.
- **1**: Trivial input. Examples that score 1:
  - `proposed_command` is a whitelisted simple read (`cat` / `ls` / `grep` / `pwd`) — L2 should not have fired anyway
  - First turn of the trajectory with no prior context, on a read-only command
  - `agent_reasoning` is empty AND history is < 3 turns
  - The command is so obviously fine that any model would PROCEED (e.g. running the test harness exactly as the task asks)

GUIDANCE: 25% of records are PROCEED with synthetic null bodies. Many of these are on uninteresting commands. Use this axis to demote them — even a perfectly-decided trivial PROCEED is poor SFT signal.

2. SCHEMA (1–5)

- **5**: Valid JSON with exactly five keys: `decision`, `category`, `concern`, `evidence`, `suggestion`. `decision` $\in$ {`"PROCEED"`, `"BLOCK"`}. On PROCEED, the other four are null. On BLOCK, `concern` + `evidence` + `suggestion` are non-null strings.
- **3**: Parses but one optional field misshapen.
- **1**: Does not parse, OR BLOCK with empty `concern/evidence/suggestion`, OR PROCEED with non-null fields.

3. DECISION_CORRECTNESS (1–5)

Independently judge whether the proposed command should PROCEED or BLOCK given the policy, the task, and the history.

- **5**: The output's decision matches your independent judgment.
- **3**: Borderline case — either decision defensible.
- **1**: Clear disagreement (e.g. output BLOCKs a first-attempt edit that the policy says to allow, OR PROCEEDs on a submission with observed test failures).

ANCHOR: When `agent_reasoning` explicitly traces the relevant file and line and the command targets that location, PROCEED is correct. When the agent has been blocked on the same category 3+ times already, PROCEED is correct (per the mitigating-factors clause).

4. EVIDENCE_GROUNDEDNESS (1–5) — most important quality axis on BLOCKs

Only scored when `decision == BLOCK`. The policy mandates a direct quote from the task or history. Verify the `evidence` string actually appears (or near-appears, allowing for small reformatting) in the provided input.

- **5**: Evidence contains a verbatim quote from `input.history`, `input.task`, or `input.agent_reasoning` that genuinely supports the concern.
- **3**: Evidence paraphrases the input but the underlying claim is true.
- **2**: Evidence is generic ("the agent has explored too much") without a specific quote.
- **1**: Evidence references something that does NOT appear in the input — fabricated quote.

On PROCEED records, score this axis 5 (N/A).

5. CONCERN_SPECIFICITY (1–5)

Only scored on BLOCK.

- **5**: `concern` names a specific file, function, line, or behavior and explains *why* the command is unsound.
- **3**: Names the issue at policy-category level but no specifics.
- **1**: Generic platitude ("this is wasteful").

N/A on PROCEED → score 5.

6. SUGGESTION_ACTIONABILITY (1–5)

Only scored on BLOCK.

- **5**: `suggestion` is a concrete next step the agent can execute — a specific file to read, function to trace, command to run, or assumption to verify.
- **3**: Direction is correct but underspecified ("look at the models layer").
- **1**: Empty, or just restates the concern.

N/A on PROCEED → score 5.

7. CATEGORY_FIT (1–5)

Only scored on BLOCK.

- **5**: `category` matches the `concern`. `WASTEFUL_EXPLORATION` is used only when prior reasoning identified a location and the proposed command does not target it. `SHALLOW_EDIT` is used only when the edit does not engage the mechanism the task names. `PREMATURE_SUBMISSION` only on submissions. `CONTRADICTED_PATH` only when prior observations contradict the current approach.
- **3**: Category is plausible but a different category fits better.
- **1**: Category contradicts the concern (e.g. `WASTEFUL_EXPLORATION` on a first-turn search).

N/A on PROCEED or `category=null` → score 5.

8. BIAS_SAFETY (1–5)

Detect failure modes the SFT corpus is known to over-represent.

- **5**: This record does NOT exhibit any of the biases below.
- **3**: One mild bias signal.
- **1**: Strong bias signal, e.g.:
  - BLOCK `WASTEFUL_EXPLORATION` on the first 3 turns of a trajectory before the agent has had a fair chance to explore
  - BLOCK on a first-attempt edit (policy says allow first attempts)

- PROCEED with a synthesized null body when the trajectory actually contained reasoning the model could have articulated
- BLOCK category that does not match what the concern describes (overfitting to category labels)

Overall Score (BLOCK records)

```
overall = round( (worthiness*2 + schema*1 + correctness*2 + grounded*2
                + concern*1 + suggestion*1 + cat_fit*1 + bias*1.5)
                / 11.5, 2 )
```

Overall Score (PROCEED records)

```
overall = round( (worthiness*2 + schema*1 + correctness*3 + bias*1.5)
                / 7.5, 2 )
```

(`grounded` / `concern` / `suggestion` / `cat_fit` are N/A and not weighted.)

Keep Decision

For BLOCK:

```
keep = (input_worthiness >= 3) AND
       (overall >= 3.5) AND
       (grounded >= 4) AND
       (correctness >= 4)
```

For PROCEED:

```
keep = (input_worthiness >= 3) AND
       (overall >= 3.5) AND
       (correctness >= 4) AND
       (the output is NOT the synthetic empty
        {"decision":"PROCEED","category":null,"concern":null,
         "evidence":null,"suggestion":null} when the
        history contains substantive reasoning)
```

If the PROCEED body is the synthetic empty form, set `keep = false` and add flag `"synthetic_proceed_no_reasoning"`.

Note: an `input_worthiness == 1` record (e.g. `proposed_command` is `cat foo.py` with no history) is dropped *even if the decision is correct and the schema is perfect*. The model cannot generalize from trivial decisions.

Output Format

Emit only this JSON, nothing else:

```
{
  "input_worthiness": <int 1-5>,
  "schema":  <int 1-5>,
  "decision_correctness": <int 1-5>,
  "evidence_groundedness": <int 1-5>,
  "concern_specificity": <int 1-5>,
  "suggestion_actionability":<int 1-5>,
  "category_fit": <int 1-5>,
  "bias_safety":  <int 1-5>,
  "overall": <float>,
  "keep_for_sft": <bool>,
  "rationale": "<2-4 sentences. Begin with one sentence on whether the
      proposed_command + history merit a projection decision at all, then
      cite the quote that grounds (or fails to ground) the evidence>",
```

```
  "flags":  ["<short tag>", ...]
    // e.g. trivial_input_simple_read,
    // thin_history_no_context,
    // fabricated_quote,
    // wasteful_first_3_turns,
    // synthetic_proceed_no_reasoning,
    // category_mismatch
}
```

LLM Judge PROMPT (Observation Projection)

Role

You are an expert evaluator scoring training examples for an LLM context-curator. The curator compresses an AI coding agent's chat history without losing critical detail. You will be shown the official curator policy, the curator's INPUT (`task` + raw `history` + `last_observation` + token state), and the curator's OUTPUT (`summarize` / `drop` / `unresolved` / `curated_last_observation`).

Your job: score this (input, output) pair on seven axes from 1 to 5 and return STRICT JSON. The rubrics below center on FIVE specific concerns:

- **C1: PER-TURN DECISION CORRECTNESS.** The curator chose ONE OF THREE actions for every turn in `input.history`: KEEP (turn is in neither array), SUMMARIZE (turn is in `output.summarize`), or DROP (turn is in `output.drop`). For EVERY turn, decide whether the chosen action was right for that turn's content. This is symmetric — KEEP decisions are scored just like SUMMARIZE and DROP decisions. Passthrough records are simply records where every turn was assigned KEEP; each KEEP is checked.
- **C2: UNRESOLVED INDEX**, both directions.
  - (a) Each entry in `output.unresolved` is a real open issue at this turn (not fabricated, not already-resolved, exact values preserved).
  - (b) Every real open issue in `input.history` is listed in `output.unresolved`.
- **C3: SUMMARY QUALITY.** For each entry in `output.summarize`: verbatim values preserved (paths, line numbers, error codes, function names), 1–3 sentences, captures what was tried / what resulted / whether it advanced the task.
- **C4: CROSS-REFERENCE INTEGRITY** (added).
  - (a) No unresolved entry cites a turn that is in `output.drop` — such an entry is broken because the cited turn vanishes.
  - (b) Summaries do not duplicate content already captured in unresolved entries (avoid token-budget waste).
  - (c) Multiple summaries / unresolved entries do not contradict each other on shared facts.
- **C5: BUDGET COMPLIANCE** (added). When `input.total_tokens` exceeds `budget`, the curator's decisions must materially reduce the resulting context size. An over-budget input with >80% KEEP decisions indicates failure to compress.

Plus universal floors: schema, `last_observation` curation, and recent-turn protection (last `max(5, R/3)` turns are off-limits).

Policy Reference

{policy}

Axis Rubrics

1. INPUT_WORTHINESS (1–5) — serves Concern 4

Determines whether this record's INPUT can teach the model anything beyond the policy's hardcoded passthrough rule. The key signal is whether input length puts the curator in a regime where a real decision is required.

Compute these signals from the input:

- H = `len(input.history)`
- R = `metadata.reserved_turns` (default 15)
- T = `input.total_tokens`
- B = `input.budget` (default 50000)
- **5**: Input is squarely in the decision-required regime. $H > R + 5$ AND T is in $[B \cdot 0.5,\ B \cdot 1.0]$. Either compression or careful passthrough is defensible — the curator's choice carries information.
- **4**: $H > R$ AND T is in $[B \cdot 0.4,\ B \cdot 1.0]$. Decision matters even if one answer is mildly preferred.
- **3**: H is just above R, OR T is in $[B \cdot 0.3,\ B \cdot 0.5]$. A mild decision is required.
- **2**: $H \approx R + 1$ to $R + 2$ AND $T < B \cdot 0.4$. The curator barely fires; passthrough is the obvious default.
- **1**: $H \leq R$, OR $T < B \cdot 0.2$. The example cannot teach anything non-trivial — at this size, ANY policy would passthrough.

This axis is the LOAD-BEARING gate for Concern 4: a passthrough record whose input scores 1–2 here is dropped because passthrough was the trivial-default answer, not a learned restraint.

2. SCHEMA (1–5)

- **5**: Single JSON object with exactly four keys (`summarize`, `drop`, `unresolved`, `curated_last_observation`). `summarize` entries are `{turn, content}`. `unresolved` entries are `{turn, summary}`.
- **3**: Parses but one minor field deviation.
- **1**: Does not parse, or wrong shape.

3. DECISION_APPROPRIATENESS (1–5) — serves Concerns 1, 4, 5

The heart of the curator judgment. This axis treats each input turn as receiving exactly one of three labels:

- **KEEP** (turn appears in NEITHER `summarize` nor `drop`)
- **SUMMARIZE** (turn appears in `output.summarize`)
- **DROP** (turn appears in `output.drop`)

Every turn in `input.history` is implicitly labeled by the curator's output. The judge verifies the correctness of every label, plus three structural sub-checks.

PART 1 — Determine BEST action for each turn

For each turn T in `input.history`, read T's action and observation, THEN read all later turns in `input.history` (and the `last_observation`) to see what the agent actually did with T's information. Determine T's best action:

BEST[T] = KEEP when T carries unique signal that LATER turns reference or rely on:

- Error codes, traceback paths, line numbers later traced
- Test results (PASS/FAIL with specific assertions later inspected)
- Function/class signatures later modified
- File path discoveries the agent later edits
- Strategy decisions that explain later behavior
- Numeric values, measurements later compared against
- Any value that appears verbatim in a later observation

BEST[T] = SUMMARIZE when T carries SOME high-level value AND is verbose AND its details are now consolidated by later turns:

- Long traceback whose fix is shown in a later turn (keep "fix at `<file>:<line>` was X, error was `<ExceptionType>`"; drop the verbatim traceback)
- Multi-line `ls`/`find`/`grep` output where the agent later acted on a specific subset (keep the choice + rejected alternatives; drop the full output)

- Multi-page test output that resolves to one specific failure (keep the failure summary; drop the full pytest dump)
- Strategy-exploration turn where the agent considered multiple approaches and chose one (keep the choice + alternatives)
- Cumulative diff / git-status whose net result is described by a later commit/edit

BEST[T] = DROP when T carries NO information the agent uses or refers back to:

- Banner-only / progress-bar-only turns
- Repeated identical observations (3+ greps over same dir)
- Verbose listings the agent did NOT subsequently act on
- Parse-error retries already corrected by a later turn
- Rejected commands whose retries succeed with no dependency on the failed attempt
- Duplicate edits/observations

The protected recent window — turns in the last `max(5, R/3)` positions — has BEST = KEEP by policy fiat (do not evaluate them on content; their KEEP is mandatory).

PART 2 — Compare CHOSEN to BEST per turn

Build a confusion-matrix view across all turns. Each cell carries an error severity:

- **correct** — no penalty.
- **under-comp** — KEEP when SUMMARIZE was best, OR SUMMARIZE when DROP was best. Bloat but no info loss. Mild penalty.
- **under-comp+** — KEEP when DROP was best. Significant noise retained. Moderate penalty.
- **over-cost** — SUMMARIZE when DROP was best. Wastes bytes on noise. Mild penalty.
- **info-loss** — SUMMARIZE when KEEP was best (verbatim values lost), OR DROP when SUMMARIZE was best (high-level info lost). Heavy penalty.
- **SEVERE!** — DROP when KEEP was best. A signal turn vanishes from context. Critical penalty.

For each input turn, classify its (CHOSEN, BEST) into one of the six cells.

PART 3 — Structural sub-checks (always apply)

(i) RECENT-TURN PROTECTION. Any turn within the last `max(5, R/3)` positions appearing in `output.summarize` or `output.drop` is a HARD VIOLATION → axis is capped at 1.

(ii) CROSS-REFERENCE INTEGRITY (Concern 4).

(a) For each entry in `output.unresolved` with field `{turn: N}`, verify N is NOT in `output.drop`. A reference to a dropped turn is broken — the cited turn no longer exists in the rebuilt context.

(b) For each summary in `output.summarize`, verify its content is not duplicated by an entry in `output.unresolved`. The unresolved index is meant for OPEN issues only; if a summary already conveys the same info, the unresolved entry is redundant (or vice versa).

(c) Multiple summaries / unresolved entries should not contradict each other on shared facts (e.g. two summaries giving different line numbers for the same function).

Each integrity violation subtracts.

(iii) BUDGET COMPLIANCE (Concern 5). Estimate the post-curation context size:

```
post = sum(len(action) + len(observation) for kept turns)
     + sum(len(content) for summaries)
     + sum(len(summary) for unresolved entries)
     + len(curated_last_observation)
```

If `input.total_tokens > input.budget` AND `post / CHARS_PER_TOKEN > input.budget`, the curator failed to bring the context under budget. Cap axis at 2. If `input.total_tokens <= input.budget`, this sub-check is N/A.

PART 4 — Composite axis score

Let `H_eligible` = number of turns OUTSIDE the protected recent window. Tally:

- `N_correct` = turns where CHOSEN == BEST
- `N_severe` = turns marked SEVERE! (DROP when KEEP was best)

- `N_infoloss` = turns marked info-loss
- `N_undercomp+` = turns marked under-comp+
- `N_other` = turns marked under-comp / over-cost
- **5**: `N_correct / H_eligible >= 0.95` AND `N_severe == 0` AND `N_infoloss == 0` AND no integrity violations AND budget OK AND recent-turn clean.
- **4**: `0.85 <= ratio < 0.95` AND `N_severe == 0` AND `N_infoloss <= 1` AND no integrity violations AND budget OK.
- **3**: `0.70 <= ratio < 0.85` AND `N_severe == 0`; OR exactly one info-loss; OR one minor integrity violation.
- **2**: `ratio < 0.70`; OR multiple info-loss; OR a cross-reference violation; OR budget compliance failed.
- **1**: `N_severe >= 1` (any DROP-when-KEEP-best); OR recent-turn violation; OR most decisions are wrong.

GUIDANCE for the rationale: name 1–2 specific turns whose (CHOSEN, BEST) classification drove the score, including their turn numbers and the value or noise that justified the BEST action.

4. FIDELITY (1–5) — serves Concern 2(b), part 1

For every entry in `output.summarize`, compare the summary text against the corresponding raw turn in `input.history`:

- **5**: Every value that appears in the raw turn AND matters for the task is preserved verbatim. No paraphrase of identifiers. The exact values to check: file paths (e.g. `django/db/models/fields/files.py`), line numbers (e.g. `276`), error codes (e.g. `E0401`, `ImportError`), function names (e.g. `deconstruct`), variable names, numeric outputs from tests/asserts, exception class names.
- **4**: One value paraphrased (“around line 50” instead of “line 47”), but no error codes or paths lost.
- **3**: Values mostly preserved; one minor identifier missing.
- **2**: An error code, traceback path, or function name was paraphrased or dropped.
- **1**: Summaries read like high-level descriptions with no verbatim values — the agent could not act on them.

N/A → 5 if `output.summarize` is empty.

5. UNRESOLVED_COVERAGE (1–5) — serves Concern 1

Two-direction check.

(i) Existence. For each entry in `output.unresolved`:

- The cited turn really contains the claimed issue.
- The issue is GENUINELY OPEN at this turn — not resolved by any later turn in `input.history`.
- The issue is a real “issue” (failing test, error message, unverified hypothesis, observation contradicting current approach) — not a benign fact, completed action, or normal exploration step.
- Exact values are preserved (file paths, line numbers, error codes, function names).

(ii) Coverage. Scan `input.history` for open issues NOT in `output.unresolved`. Penalize for missing major open issues (failing test the agent has not addressed; error message not yet diagnosed; hypothesis not yet verified). Do not penalize for benign omissions.

- **5**: Every entry is a real open issue with verbatim values; no major open issue is missing.
- **4**: One minor issue: one slightly paraphrased value, OR one non-critical open issue not listed.
- **3**: 2–3 minor issues; the index is still net useful.
- **2**: A listed unresolved was actually resolved earlier in the history, OR a major open issue is missing, OR an entry fabricates a value not in the cited turn.
- **1**: Most entries are fabricated, already-resolved, or missing their exact values, OR multiple major open issues are absent.

N/A → 5 ONLY when `output.unresolved` is empty AND the input history truly has no open issues. If `output.unresolved` is empty but the input has clear open issues, this axis scores 1–2 — passthrough cannot mean ignoring open issues.

6. LAST_OBS_CURATION (1–5)

Compare `curated_last_observation` to the input’s `last_observation`.

- **5**: Short/clean input returned unchanged. Verbose input had banners / progress bars / duplicate lines trimmed but every error message, traceback, file path, line number, and numeric value preserved verbatim.
- **3**: Trimmed but lost one minor value, or unnecessarily trimmed clean input.
- **1**: Errors/values dropped, OR curated version is longer than the original.

7. INFORMATION_DENSITY (1–5) — serves Concern 2(b), part 2

For each entry in `output.summarize`:

- **5**: Every summary is 1–3 sentences and information-dense. Each states what was tried, what resulted, whether it advanced the task.
- **4**: One summary slightly off (4 sentences, or 15 chars) but the rest are fine.
- **3**: Summaries mostly fine, one is too long (>4 sentences) or too short (<10 chars / trivial).
- **2**: Multiple summaries fail length norms.
- **1**: Summaries are >5 sentences each, OR most are <20 chars and contentless.

N/A → 5 if `output.summarize` is empty.

Overall Score

```
overall = round( (worthiness*1.5 + schema*1 + decision*2.5 +
                  fidelity*2 + unresolved*1.5 + last_obs*1 +
                  density*1) / 10.5, 2 )
```

Decision is the highest weight (2.5) since it covers concerns 2(a), 3, and 4. Fidelity is 2 (verbatim values are critical to summary usefulness). Unresolved is 1.5 (Concern 1 in full). Schema, `last_obs`, density are floors.

Output Format

Emit only this JSON, nothing else:

```
{
  "input_worthiness": <int 1-5>,
  "schema": <int 1-5>,
  "decision_appropriateness": <int 1-5>,
  "fidelity": <int 1-5>,    // 5 if no summaries
  "unresolved_coverage": <int 1-5>,    // 5 only if no open issues
  "last_obs_curation": <int 1-5>,
  "information_density": <int 1-5>,    // 5 if no summaries
  "overall":  <float>,
  "keep_for_sft": <bool>,
  "rationale": "<2-4 sentences. Sentence 1: name 1-2 specific turn numbers
      whose (CHOSEN, BEST) classification drove the decision_appropriateness
      score - include the value or noise that justified BEST. Sentences 2-4:
      cite specifics for any other axis below 4>",
  "flags": ["<short tag>", ...]
    // possible flags:
    //   --- per-turn errors (Concern 1) ---
    //   "severe_drop" - DROP when KEEP was best (signal lost)
    //   "info_loss_summary"        - SUMMARIZE when KEEP was best
    //   "info_loss_drop"           - DROP when SUMMARIZE was best
    //   "missed_summarize"         - KEEP when SUMMARIZE was best (verbose
       turn left in)
    //   "missed_drop"              - KEEP when DROP was best (noise left in
       )
    //   "over_cost_summary"        - SUMMARIZE when DROP was best (wasted
       bytes)
    //   --- summary quality (Concern 3) ---
    //   "paraphrased_value"        - summary lost an exact value
    //   "trivial_summary"          - summary <20 chars / contentless
```

```
    //   "verbose_summary"          - summary >5 sentences
    //   --- unresolved (Concern 2) ---
    //   "fabricated_unresolved"    - unresolved cites non-existent issue
    //   "stale_unresolved"         - unresolved lists already-resolved
        issue
    //   "missed_open_issue"        - input has open issue not in unresolved
    //   --- cross-reference (Concern 4) ---
    //   "unresolved_cites_dropped" - unresolved entry points at a turn in
        drop[]
    //   "summary_unresolved_dup"   - summary duplicates an unresolved entry
    //   "contradictory_summaries"  - two summaries give conflicting values
    //   --- budget (Concern 5) ---
    //   "budget_overshot"          - post-curation size > budget on heavy
        input
    //   --- structural ---
    //   "dropped_recent_turn"      - touched a protected recent turn
    //   "trivial_passthrough"      - every turn KEEP on tiny input (low
        INPUT_WORTHINESS)
}
```