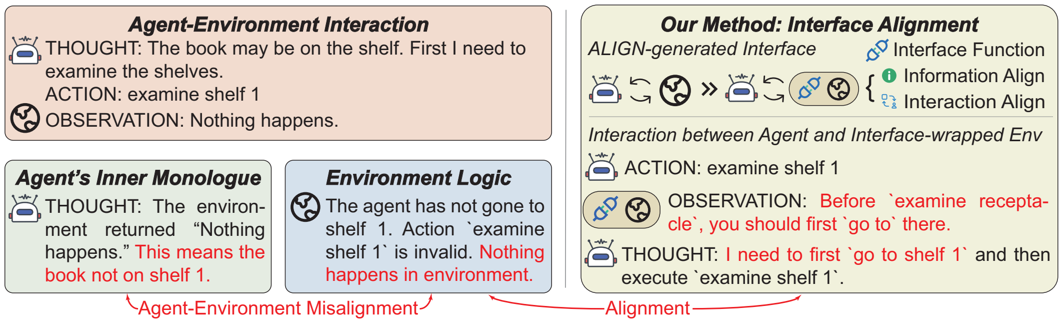
Agent-Environment Alignment via Automated Interface Generation

Kaiming Liu[1], **Xuanyu Lei**[1,2], **Ziyue Wang**[1], **Peng Li**[2]*, **Yang Liu**[1,2]*
[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[2]Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China

Abstract

Large language model (LLM) agents have shown impressive reasoning capabilities in interactive decision-making tasks. These agents interact with environment through intermediate interfaces, such as predefined action spaces and interaction rules, which mediate the perception and action. However, mismatches often happen between the internal expectations of the agent regarding the influence of its issued actions and the actual state transitions in the environment, a phenomenon referred to as **agent-environment misalignment**. While prior work has invested substantially in improving agent strategies and environment design, the critical role of the interface still remains underexplored. In this work, we empirically demonstrate that agent-environment misalignment poses a significant bottleneck to agent performance. To mitigate this issue, we propose **ALIGN**, an Auto-Aligned Interface Generation framework that alleviates the misalignment by enriching the interface. Specifically, the ALIGN-generated interface enhances both the static information of the environment and the step-wise observations returned to the agent. Implemented as a lightweight wrapper, this interface achieves the alignment without modifying either the agent logic or the environment code. Experiments across multiple domains including embodied tasks, web navigation and tool-use, show consistent performance improvements, with up to a 45.67% success rate improvement observed in ALFWorld. Meanwhile, ALIGN-generated interface can generalize across different agent architectures and LLM backbones without interface regeneration. Code and experimental results are available at `https://github.com/THUNLP-MT/ALIGN`.


Agent-Environment Interaction
Agent's Inner Monologue
Environment Logic
Agent-Environment Misalignment
Our Method: Interface Alignment
ALIGN-generated Interface
Interface Function
Information Align
Interaction Align
Interaction between Agent and Interface-wrapped Env
Alignment

THOUGHT: The book may be on the shelf. First I need to examine the shelves.
ACTION: examine shelf 1
OBSERVATION: Nothing happens.

THOUGHT: The environment returned "Nothing happens." This means the book not on shelf 1.

The agent has not gone to shelf 1. Action `examine shelf 1` is invalid. Nothing happens in environment.

ACTION: examine shelf 1
OBSERVATION: Before `examine receptacle`, you should first `go to` there.
THOUGHT: I need to first `go to shelf 1` and then execute `examine shelf 1`.

Figure 1: **Illustration of agent-environment misalignment and our proposed solution.** On the left, the agent and the environment have a misalignment in their interpretation of the same observation, where the agent's understanding of the observation differs from the environment's underlying logic. On the right, our method, ALIGN, automatically generates interfaces that provide the agent with clearer interaction context, aligning the agent's understanding with the environment's logic.

* Correspondence to Peng Li <lipeng@air.tsinghua.edu.cn>, Yang Liu <liuyang2011@tsinghua.edu.cn>

1 Introduction

Large Language Model (LLM) agents have demonstrated promising performance in interactive tasks such as embodied tasks [12, 25, 44], web navigation [7, 19, 35], and tool-use tasks [34, 38, 47]. In these tasks, **agents** typically interact with the **environment** through manually designed **interfaces** such as predefined action spaces and interaction rules. While substantial efforts have been devoted to improving agents and environments, comparatively little attention has been paid to the interface between them. This has led to a problem we term **agent-environment misalignment**, which significantly impacts the agent performance.

Agent-environment misalignment refers to the discrepancy between the interpretation of the agent to the observation following an action and the underlying logic of the environment. As illustrated in Figure 1 (left), in ALFWorld [40], issuing *examine receptacle* fails unless the agent first executes *go to receptacle*. Consequently, the environment responds with the observation "Nothing happens.". At this point, the agent interprets the observation to mean that there is nothing on shelf 1, which is inconsistent with the underlying reason for the environment providing it. To assess the impact of this misalignment, we conduct preliminary experiments, which reveal that simply revising the observation for an invalid *examine receptacle* action to "You need to first go to receptacle before you can examine it" increases the success rate of a vanilla Qwen2.5-7B-Instruct [42] agent on ALFWorld from 13.4% to 31.3%[1]. This suggests that agent-environment misalignment significantly hinders task success, and can be alleviated by improving interface design. From the perspective of the agent, poorly designed interfaces impose unnecessary cognitive overhead. Furthermore, from an evaluation perspective, inadequate interfaces can obscure an accurate assessment of the true reasoning capabilities of agents. Therefore, we argue that the problem of agent-environment misalignment warrants greater attention.

However, addressing the agent-environment misalignment is challenging. On one hand, current benchmarks primarily focus on advance agent intelligence by constructing increasingly complex and challenging environments [21, 48, 50, 51, 62], often overlooking the importance of improving interface design. This oversight extends across multiple domains of interactive tasks. For instance, ALFWorld, OSWorld [51], and M^{3}ToolEval [47] all exhibit similar deficiencies: failing to provide agent-parseable observations for environmental constraints violation in embodied tasks, positional inaccuracies in operating system tasks or parameter format errors in multi-turn tool-use tasks, respectively. On the other hand, although some recent work [1, 54, 60] has begun to consider interface design, these efforts often rely on manual, environment-specific tailoring, which introduces two critical issues: (1) it is highly labor-intensive and (2) whether human-designed interfaces are optimal for agents remains an open question.

Furthermore, in addition to studies that explicitly optimize interface design, it is common in agent-focused research for researchers to manually re-engineer environment interfaces to align with their specific methods. For instance, for the same environment ALFWorld, Zhou et al. [63] manually maintains the environment's state information in JSON format; Ma et al. [28] introduces a new action *check_valid_actions* to enable agents to retrieve all valid actions; and Chen et al. [9] re-implements the environment by wrapping it into a new class *InteractEnv*. However, such ad-hoc customization pose a significant challenge to the field: it compromises the direct comparability across different approaches. Moreover, these modifications are often tailored to the specific methods proposed, making it difficult for the research community to determine whether performance variations stem from novel agent architectures or from the non-standardized, customized interfaces. Therefore, we believe that manually re-engineering environment interfaces is not an optimal approach to alleviating the agent-environment misalignment problem.

Distinct from the aforementioned works, we propose to **automatically generate interfaces for bridging the agent-environment misalignment**. In this work, we introduce **ALIGN** (Auto-Aligned Interface Generation), a framework that automatically generate aligned interfaces for environments. The generated interface consists of two modules: INFERRULES and WRAPSTEP. The former automatically discovers and provides the agent with static information about environmental rules or internal constraints, facilitating *static alignment*, while the latter enhances the interaction by offering more detailed observations for agent-issuing actions, enabling *dynamic alignment*, as shown in Figure 1 (right). Owing to the powerful reasoning and coding capabilities of current advanced LLMs, we utilize these models to analyze existing agent-environment misalignments and

[1] Experimental details are provided in Appendix B.

automatically generate the interface. Additionally, we employ LLMs to conduct experimental verification procedures to mitigate the hallucination issues [3, 52]. Specifically, our LLM-based system autonomously validate both proposed misalignments and generated interface through direct interaction with the environment, ensuring that identified issues genuinely exist and are properly addressed by the interface. The generated interface acts as a lightweight wrapper, providing richer context and explicit constraint hints, enabling different LLM agents to align with the environment directly.

To evaluate ALIGN, we conduct experiments on four representative benchmarks across three domains: embodied tasks, web navigation, and tool-use tasks. Our results demonstrate consistent performance improvements across all four benchmarks when using the ALIGN-generated interface, with notably gains of 45.67% in average success rate on ALFWorld. Moreover, the ALIGN-generated interface reduced the prevalence of consecutive invalid actions by 65% on ALFWorld, highlighting the efficiency of our approach in mitigating agent-environment misalignment.

Our key contributions can be summarized as follows:

- We identify and characterize the **agent-environment misalignment** problem, empirically demonstrating its prevalence across diverse domains and its role as a significant bottleneck to agent performance.
- We introduce **ALIGN**, the first framework automatically generates aligned interfaces to alleviate agent-environment misalignment, without modifying agent logic or environment code.
- We demonstrate the effectiveness and generalizability of **ALIGN** across three domains, with up to a 45.67% success rate improvement on ALFWorld.

2 Related work

Agent-environment interface The agent-environment interface defines how agents interact with the environment. In reinforcement learning, researchers construct unified interaction interfaces [4, 5, 22, 43] to standardize the application and evaluation of different learning algorithms. With the increasing capability of LLMs to perform human-like actions [17, 26, 28], interface design has been proven to largely influence the performance of LLM-based agents [51, 37]. SWE-agent [54] proposes agent-computer interfaces (ACI) for coding agents, emphasizing interface optimization. Following this research line, recent efforts aim to improve generalization [1, 36, 32] and enhance interfaces with auxiliary tools [6, 16, 24, 27, 53]. Nevertheless, current agent-environment interfaces are mostly manually crafted and tailored for specific environments or agent frameworks, limiting their generalization and scalability. Therefore, we propose automated interface generation to empower agents with effective, generalizable and automatic interface alignment.

Methods aligning agents with environments LLM agents have exhibited strong potential for real-world interaction and task completion [58, 39, 26]. Current research in this area can be broadly categorized into training-based methods and training-free methods. Training-based methods consists of fine-tuning LLMs with expert-level interaction trajectories [59, 8, 11, 14, 10] and enhancing environment-aligned planning and acting via reinforcement learning [2, 56, 35, 13, 64, 49]. Though effective, these methods suffer from high computational costs and limited generalization towards unseen environments. Another approach constructs training-free multi-agent frameworks for task decomposition and experience accumulation [9, 20, 41, 55, 63], offering a light-weight solution to align agents with environments. However, static agent pipelines lack flexibility and generalization and injected experience through prompting often fails to capture environment dynamics and is not effectively utilized by LLMs, resulting in insufficient alignment between agents and environments.

3 Method

3.1 Problem formulation

In the context of interactive decision-making tasks, we define the environment $\mathcal{E}$ as a tuple $(\mathcal{S}, \mathcal{A}, T, F, \mathcal{I})$, where:

- $\mathcal{S}$ denotes the set of all possible states of the environment;
- $\mathcal{A}$ denotes the action space, the set of actions the agent can invoke;
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ represents the state transition function, which defines how the environment state evolves in response to agent actions;

- $F : \mathcal{S} \times \mathcal{A} \to \mathcal{O}$ is the *observation function*, providing textual feedback that reflects the consequences of the action in the current state, where $\mathcal{O}$ is all possible observations;
- $\mathcal{I}$ encodes the *environment foundational information description*, a fixed, declarative representation of the environment's basic introduction, object attributes, or domain rules, which is exposed to the agent at initialization;

An agent π operates as a policy that, at each timestep t, receives $(\mathcal{I}, \texttt{task}, o_{t-1})$, where $\texttt{task}$ is the task description and $o_{t-1} = F(s_{t-1}, a_{t-1})$ is the observation from the previous step, and produces an action $a_t \in \mathcal{A}$. In general, o_0 is the initial observation. The task culminates in an interaction trajectory $\tau = [(s_0, a_0, o_0), \ldots, (s_t, a_t, o_t)]$, and the environment provides feedback on the task completion that indicates how well the agent has achieved its goal by the end of the interaction.

In practice, misalignment can arise between the internal expectations of the agents and the actual transitions in the environment. After producing an action a_t, the agent may anticipate a transition to a state $s_{t+1}^{\text{expected}}$ consistent with its reasoning. However, due to implicit or under-specified constraints in $\mathcal{E}$, the actual next state $s_{t+1}^{\text{actual}} = T(s_t, a_t)$ may differ from $s_{t+1}^{\text{expected}}$. This mismatch, which we refer to as **agent-environment misalignment**, can disrupt the intended progress of the agent toward the goal to be disrupted, even if the action a_t is logically coherent under the agent's interpretation of $\mathcal{I}$ and prior observation.

3.2 ALIGN overview

To alleviate the agent-environment misalignment, we introduce **ALIGN**, a framework that automatically generate aligned interface between the agent and the environment. Concretely, we redefine the interface by wrapping two key environment signals: (1) the static environment description $\mathcal{I}$, which we transform into **augmented information** $\tilde{\mathcal{I}}$ that explicitly communicates relevant interaction rules and constraints to the agent before task execution; and (2) the step-wise observation $o_t = F(s_t, a_t)$, which we restructure as an **augmented observation** $\tilde{o}_t$ that captures both the original observation and additional signals about the success, failure conditions, or inferred preconditions of the action.

These enriched signals $(\tilde{\mathcal{I}}, \tilde{o}_t)$ are generated *without modifying the environment code*, and are instead constructed by an interface wrapper layered on top of the environment, as illustrated in Figure 2. This wrapper contains two key modules:

InferRules$(\cdot)$: Static information of domain-specific execution rules based on the task description and the initial observation o_0. Formally, it implements a mapping:

$$\textsc{InferRules} : (\texttt{task}, o_0) \to \tilde{\mathcal{I}}$$

where $\tilde{\mathcal{I}}$ includes the constraints automatically extracted, such as precondition dependencies or action ordering requirements.

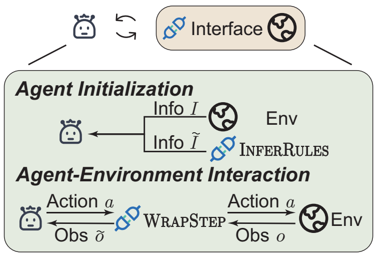


Figure 2: Overview of the ALIGN-generated interface.

WrapStep$(\cdot)$: A dynamic observation processor that intercepts each agent-issued action and augments the raw observation if needed. It implements the mapping:

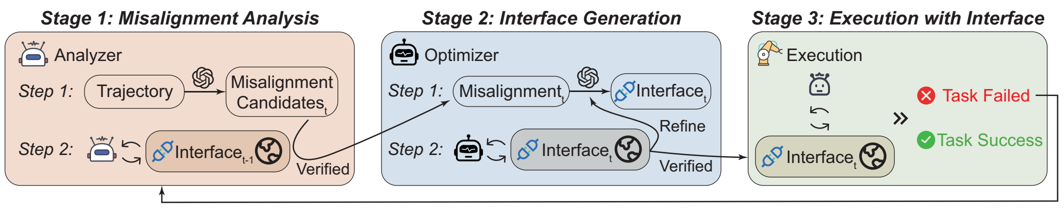
$$\textsc{WrapStep} : (F, s_t, a_t) \to \tilde{o}_t$$

where $\tilde{o}_t$ encapsulates both $F(s_t, a_t)$ and additional diagnostic or corrective information inferred from execution context.

Together, these modules form an intermediate interface wrapper layer that intercepts and transforms environment information before it reaches the agent. This design allows the base agent π to remain unchanged, while still benefiting from contextual clarity and enriched observation that help avoid misaligned actions. From the perspective of the agent, interaction now occurs with an *augmented environment*, which we denote as:

$$\tilde{\mathcal{E}} = (\mathcal{S}, \mathcal{A}, T, \tilde{F}, \mathcal{I} \cup \tilde{\mathcal{I}})$$

Here, the observation function $\tilde{F}$ is defined as $\tilde{F}(s_t, a_t) := \textsc{WrapStep}(F, s_t, a_t)$. This formulation does not alter the internal structure or transition dynamics of the original environment $\mathcal{E}$. Instead, it constructs an externally wrapped interaction interface that provides the agent with a richer and more interpretable view of its operating context. For the convenience in subsequent representations, we define the interface as $\Phi := \{\textsc{InferRules}, \textsc{WrapStep}\}$.



Misalignment Example

### Analysis Result 1
Agent Action Type: examine
Agent Action Case: examine drawer 1
Agent High-Level Reasoning Intent: The Agent is attempting to locate the box and desklamp by examining potential receptacles.
Environment Rule: The Environment may require the Agent to first "go to" a receptacle before performing actions like "examine" on it.
Sufficient Observation: The environment should provide observation such as "You need to go to drawer 1 before examining it" when the Agent attempts to examine a receptacle without first moving to it.

Experiment Verification Example

Optimizer: <thought>...</thought>
<action>init_simulator(task_id="4-293")</action>
Experiment: ...
Optimizer: <thought>Now I will simulate an invalid "examine" action where ...</thought>
<action>exec_agent_action("examine drawer 1")</action>
Experiment: ... Observation: Nothing happens. ...
...
Optimizer: <thought>The "examine drawer 1" action was executed, but the environment did not provide sufficient feedback ...</thought>
<if_need_refine>True</if_need_refine>
<refine_strategy>...</refine_strategy>

Interface Example

```
# Information Align
def InferRules():
    return """1. Before examining or interacting
with any receptacle, you must first go to that
receptacle."""

# Interaction Align
def WrapStep():
    ...
    if target not in current_location:
        obs =  f"You need to go to {target} before
examining it. You must first navigate to a
receptacle before you can examine it."
```

Figure 3: **ALIGN framework.** In each iteration, ALIGN progresses though three stages. **Stage 1**: the Analyzer identifies potential agent-environment misalignments and validates them through experiments; **Stage 2**: the Optimizer generates a new interface based on the previous interface and identified misalignments, followed by verification and refinement; **Stage 3**: the agent interacts with the updated interface-wrapped environment, with trajectories of failed tasks fed back to the Analyzer for analysis in the next iteration. At the bottom of the figure, examples for misalignment, verification of interface integrity by Optimizer through experiments, and the ALIGN-generated interface are provided.

As shown in Figure 3, the ALIGN integrates two cooperative modules, **Analyzer** and **Optimizer** to generate aligned interfaces. The framework operates through iterative optimization, with each iteration comprising three stages: in Stage 1, the Analyzer identifies agent-environment misalignments by analyzing past interaction trajectories; in Stage 2, the Optimizer generates, validates and refines a new interface based on the detected misalignments; and in Stage 3, the agent interacts with the environment wrapped with the newly generated interface, and the failed task trajectories are fed back to Analyzer for analysis in the next iteration.

3.3 ALIGN framework

Algorithm 1 ALIGN: Auto-Aligned Interface Generation

```
Require: Environment $\mathcal{E}$, Agent $\pi$, Task training set $\mathcal{T}_{\text{train}}$, Maximum iterations $K$
 1: Initialize misalignment set $\mathcal{M} \leftarrow \emptyset$, interface $\Phi^{(0)} \leftarrow \{\textsc{InferRules}^{(0)}, \textsc{WrapStep}^{(0)}\}$, where
    $\textsc{InferRules}^{(0)}$ and $\textsc{WrapStep}^{(0)}$ are identity functions
 2: for $i = 1, 2, \ldots, K$ do
 3:     $\tilde{\mathcal{E}}^{(i-1)} \leftarrow$ Environment $\mathcal{E}$ wrapped with interface $\Phi^{(i-1)}$
 4:     $\tau_{\text{fail}}^{(i-1)} \leftarrow$ Failed trajectories from agent $\pi$ interacting with $\tilde{\mathcal{E}}^{(i-1)}$ on $\mathcal{T}_{\text{train}}$
 5:     if $\tau_{\text{fail}}^{(i-1)} = \emptyset$ then
 6:         break                                        ▷ No more failures in the training set
 7:     end if
        // Stage 1: Misalignment Analysis
 8:     $\mathcal{M}^{(i)} \leftarrow \text{Analyzer}(\tau_{\text{fail}}^{(i-1)}, \mathcal{M}, \Phi^{(i-1)})$
 9:     if $\mathcal{M}^{(i)} = \emptyset$ then
10:         break                                        ▷ No new misalignments identified
11:     end if
12:     $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}^{(i)}$
        // Stage 2: Interface Generation
13:     $\Phi^{(i)} \leftarrow \text{Optimizer}(\mathcal{M}^{(i)}, \Phi^{(i-1)})$
14: end for
15: return final interface $\Phi^{(i)}$
```

To automate the generation of interfaces that bridge the agent-environment misalignments, ALIGN need to solve two key challenges: (1) how to analyze and identify existing agent-environment misalignments, and (2) how to generate an interface that addresses these misalignments. The overall algorithm process of ALIGN is outlined in Algorithm 1.

Misalignment Analysis We represent each agent-environment misalignment using structured text, as shown in the bottom left of Figure 3. The "Agent High-Level Reasoning Intent" and "Environment Rule" respectively depict the agent's expectations of the action and the environment's observation rules, together representing a misalignment. The "Sufficient Observation" represents the observation the environment should provide to resolve the misalignment. To analyze and identify these misalignments, we designed the Analyzer module based on LLMs. In each iteration, the Analyzer takes the failed interaction trajectory $\tau^{(i-1)}$ in the previous iteration, the set of currently identified misalignments $\mathcal{M}$, and the interface $\Phi^{(i-1)}$ from the previous round as input, generating a new set of misalignments $\mathcal{M}^{(i)}$. Detailed prompts for this process are provided in Appendix C.4.

Interface Generation Once the new set of misalignments $\mathcal{M}^{(i)}$ is identified, we employ the Optimizer module to generate a new interface. We represent the two modules of the interface, INFERRULES and WRAPSTEP, as Python functions, as shown in the bottom right of Figure 3, to leverage the powerful code generation capabilities of LLMs. In each iteration, the Optimizer takes the newly identified misalignments $\mathcal{M}^{(i)}$ and the previous interface $\Phi^{(i-1)}$ as input, generating a new interface $\Phi^{(i)}$. The detailed prompts for this process are provided in Appendix C.4.

Experimental Verification Given the hallucination [3, 52] issues inherent in current LLMs, we incorporate an experimental verification procedure. Specifically, after the Analyzer generates $\mathcal{M}^{(i)}$, it will interact with the environment wrapped by the previous interface $\Phi^{(i-1)}$ to validate whether the identified misalignments do indeed exist and can be resolved by the proposed "Sufficient Observation". And after the Optimizer generates the new interface $\Phi^{(i)}$, it will interact with the environment wrapped by this new interface to ensure that the generated interface can resolve the newly identified misalignments. If the Optimizer finds that the proposed interface is insufficient to address the newly discovered misalignments, it will provide a refinement strategy and regenerate the interface. This iterative process continues until the interface passes the validation, ensuring that the misalignments identified are appropriately addressed. An example of this process is provided in the bottom center of Figure 3. To facilitate this interaction with the interface-wrapped environment, we designed a set of encapsulated tools for both the Analyzer and Optimizer to use, as described in Appendix C.3.

After each iteration, the agent interacts with the environment wrapped by the new generated interface $\Phi^{(i)}$, and trajectories of the failed tasks are returned to the Analyzer for further analysis. The algorithm continues iteratively until the pre-defined maximum number of iterations is reached, or when no new failed trajectories are produced, or when no new misalignments are identified.

4 Experiment

4.1 Experimental settings

Evaluation Protocol To validate the effectiveness of ALIGN, we assess the performance of various agents in the original, unmodified environments. Subsequently, ALIGN is utilized to generate interfaces for these environments with the respective agents. Afterward, the agents are re-evaluated in the same environments, wrapped with the ALIGN-generated interfaces. This design enables us to observe and measure the changes in agent performance before and after the interface alignment.

Benchmarks We conduct experiments on four representative benchmarks across three domains: embodied tasks, web navigation and tool-use. Among them, (1) ALFWorld [40] focuses on embodied AI agents performing household tasks through textual interactions in simulated environments; (2) ScienceWorld [45] evaluates the abilities to conduct scientific experiments and apply scientific reasoning of agents in an interactive text-based environment; (3) WebShop [57] simulates e-commerce scenarios where agents navigate product catalogs and complete purchasing tasks; and (4) M^{3}ToolEval [47] is specifically designed to evaluate agent performance in multi-turn tool-use tasks.

Agent Methods To verify the capability of ALIGN to enhance performance across diverse agent architectures, we evaluate five representative methods: (1) Vanilla Agent: Base implementation without specialized prompting strategies; (2) ReAct [58]: Leverages the reasoning capabilities of LLMs through interleaved reasoning and action steps; (3) Self-Consistency [46]: Utilizes probabilistic outputs from LLMs to generate multiple solution paths and select the most consistent one;

Table 1: **Effect of ALIGN-generated interfaces on four benchmarks.** For every agent we report its score without the interface (w/o ALIGN) and with the interface (w/ ALIGN); the value in parentheses is the absolute improvement. Metrics are task-success rate (%) for ALFWorld and M^{3}ToolEval, and scores for ScienceWorld and WebShop.

		Embodied		**Web**	**Tool-use**
Method	**Interface**	**ALFWorld**	**ScienceWorld**	**WebShop**	**M^{3}ToolEval**
Vanilla	w/o ALIGN	13.43	14.94	54.10	11.11
	w/ ALIGN	60.45 (+47.02)	27.69 (+12.75)	61.23 (+7.13)	20.83 (+9.72)
ReAct	w/o ALIGN	19.40	20.03	37.20	9.72
	w/ ALIGN	63.43 (+44.03)	28.97 (+8.94)	42.93 (+5.73)	18.06 (+8.34)
Self-Consistency	w/o ALIGN	11.94	14.07	56.23	11.11
	w/ ALIGN	69.40 (+57.46)	25.41 (+11.34)	61.10 (+4.87)	16.67 (+5.56)
Self-Refine	w/o ALIGN	3.73	14.87	44.80	5.55
	w/ ALIGN	40.30 (+36.57)	22.99 (+8.12)	52.30 (+7.50)	6.94 (+1.39)
Planning	w/o ALIGN	9.70	17.13	46.95	11.11
	w/ ALIGN	52.99 (+43.29)	26.34 (+9.21)	54.67 (+7.72)	18.06 (+6.95)

(4) Self-Refine [29]: Employs an iterative self-critic and refine mechanism where agents critique and refine their previous solutions; and (5) Planning Agent: Inspired by RAP [18], this approach leverages the planning capabilities of LLMs to decompose complex tasks into manageable sub-tasks.

Implementation details Unless otherwise noted, all agents use Qwen2.5-7B-Instruct [42] as the base model. The Optimizer for interface generation uses Gemini 2.5 Pro [15], while other steps the Analyzer and Optimizer use GPT-4.1 [33]. Implementation details of benchmark task splits and hyper-parameters can be found in Appendix C.

4.2 Main results

Table 1 summarizes the task success rates or scores of five representative agent methods in the environment without (w/o) or with (w/) ALIGN-generated interface. The interfaces generated can be found in Appendix D.3. Our empirical investigation yields three principal findings:

ALIGN consistently enhances performance across different domains. All evaluated agent methods demonstrate significant performance improvements when utilizing ALIGN-generated interfaces. Specifically, the five agent methods exhibit mean improvements of 45.67% in task-success rate for ALFWorld, 10.07 points for ScienceWorld, 6.59 points for WebShop, and 6.39% in task-success rate for M^{3}ToolEval. These consistent improvements substantiate the effectiveness of ALIGN.

Agent-environment misalignment is a pervasive phenomenon impeding the agent performance. The observed performance enhancements provide empirical evidence that numerous errors in baseline configurations originate from implicit constraints or under-specified observation, rather than from intrinsic reasoning deficiencies. This finding suggests that when these environmental constraints are explicitly surfaced, agents can execute their intended tasks with substantially improved reliability. Consequently, we posit that agent-environment misalignment is pervasive in interactive decision-making tasks, and addressing this problem is crucial for advancing agent performance.

Alignment between agent and environment facilitates identification of additional performance-influencing factors. While the Self-Consistency agent achieves a 69.40% success rate in ALFWorld with ALIGN, the performance of Self-Refine agent remains comparatively suboptimal (40.30%), indicating potential deficiencies in the critic and self-refinement capabilities of the Qwen2.5-7B-Instruct model. These limitations are similarly manifested in the M^{3}ToolEval results. Furthermore, the relatively modest performance improvements in ScienceWorld suggest that Qwen2.5-7B-Instruct may exhibit insufficient scientific causal reasoning capabilities. These observations indicate that properly aligning agent and environment enables more precise isolation and analysis of other factors influencing agent performance beyond alignment considerations.

4.3 Interface quality analysis

To quantitatively assess the efficacy of ALIGN-generated interfaces in explicating environmental constraints, we introduce a metric that measures the frequency of *consecutive invalid actions*. This metric is operated by calculating the proportion of the actions that occur within sequences of two or more consecutive `invalid` steps. Lower values of this metric indicate: (1) enhanced agent

Table 2: **Impact of the ALIGN-generated interface on consecutive invalid actions.** The metric reports the fraction (%) of consecutive invalid actions. Lower values indicate more desirable behavior. Δ denotes the relative reduction with respect to the **w/o ALIGN** setting.

Method	ALFWorld			ScienceWorld		
	w/o ALIGN	**w/ ALIGN**	Δ	**w/o ALIGN**	**w/ ALIGN**	Δ
Vanilla	77.91	26.59	66%	49.12	24.47	50%
ReAct	82.23	38.63	53%	46.61	29.99	36%
Self-Consistency	77.71	15.08	81%	51.10	31.51	38%
Self-Refine	90.38	45.84	49%	58.02	29.48	49%
Planning	74.09	19.14	74%	68.67	20.94	70%
Average	80.46	28.51	65%	54.70	27.28	49%

Table 3: **Generalization of ALIGN-generated interfaces across agents and models.** Mean performance gains from applying ALIGN-generated interfaces across different settings. (a) Cross-agent transfer: interfaces generated with a Vanilla agent improve other agent methods. (b) Cross-model transfer: interfaces generated with Qwen2.5-7B-Instruct generalize to other LLMs. Metrics report task success rate (%) change for ALFWorld and M^{3}ToolEval, and absolute score change for ScienceWorld and WebShop.

(a) Interface source: Vanilla agent				
Target method	ALFWorld	ScienceWorld	WebShop	M^{3}ToolEval
ReAct	+39.56	+12.29	+7.87	+5.56
Self-Consistency	+51.49	+15.30	+3.00	+8.33
Self-Refine	+34.33	+14.11	+6.17	+4.17
Planning	+41.05	+9.66	+3.26	+11.11
(b) Interface source: Qwen2.5-7B-Instruct agent				
Target LLM	ALFWorld	ScienceWorld	WebShop	M^{3}ToolEval
Qwen2.5-14B-Instruct	+17.46	+4.61	+4.66	+6.11
Llama3.1-8B-Instruct	+5.97	+10.27	+0.33	+0.83
Llama3.3-70B-Instruct	+5.82	+3.99	+5.68	+1.67

awareness of implicit preconditions, and (2) improved recovery capability following isolated errors. Table 2 presents the results for five agent methods implemented on ALFWorld and ScienceWorld, both without (w/o) and with (w/) implementation of ALIGN-generated interfaces.

The empirical results demonstrate a substantial reduction in consecutive invalid actions frequency across all agent methods when utilizing ALIGN-generated interfaces. Specifically, we observe a mean reduction of **65%** in ALFWorld and **49%** in ScienceWorld. These findings provide robust evidence that ALIGN effectively renders latent constraints explicit, thereby preventing agents from entering repetitive error cycles, which aligns with the findings documented in Section 4.2.

4.4 Generalization study

To evaluate the generalization capabilities of ALIGN, we performed the following two experiments, with the results presented in Table 3. Detailed results of the experiments are available in Appendix D.1.

ALIGN can generalize to different agent architectures. Panel (a) of Table 3 applies interfaces generated with the Vanilla agent to the other four agents. Across all four environments every target agent shows consistent growth, with mean gains of +41.61% in task-success rate for ALFWorld, +12.84 points for ScienceWorld, +5.08 points for WebShop and +7.29% in task-success rate for M^{3}ToolEval. The fact that the same interface boosts other agents with different architectures demonstrates that ALIGN captures genuine and previously unexposed environment constraints. This also reinforces the earlier conclusion that agent-environment misalignment is a pervasive source of error independent of the agent's reasoning style.

ALIGN can generalize to larger and heterogeneous LLMs. Panel (b) of Table 3 examines whether an interface generated with Qwen2.5-7B-Instruct can extend to larger or architecturally different model backbones. The results demonstrate that ALIGN-generated interfaces lead to performance improvements across base models of varying sizes and architectural families, which indi-

Table 4: **Ablation on Interface components.** Values represent the change in success rate(%) for ALFWorld and the change in score for ScienceWorld. Negative values mean performance drops from the *Full* interface. Full results for WebShop and M^3ToolEval are deferred to Appendix D.2.

	w/o INFERRULES		w/o WRAPSTEP	
Method	ALFWorld	ScienceWorld	ALFWorld	ScienceWorld
Vanilla	-8.96	-3.35	-33.58	-4.72
ReAct	-5.22	-2.08	-17.91	-6.44
Self-Consistency	-1.49	-2.30	-37.27	-10.59
Self-Refine	-7.46	-1.72	-34.33	-7.59
Planning	-10.45	-0.78	-26.87	-9.86
Mean	-6.72	-2.05	-31.79	-7.84

cates that our method possesses strong generalization capabilities. We also observe that this generalization is not uniformly robust across all model families and datasets. For instance, Llama3.1-8B-Instruct [30] shows only a marginal gain of +0.33 on the WebShop benchmark. This limited improvement may be attributed to the inherent reasoning capabilities of the model itself.

Taken together, these results show that ALIGN-generated interfaces generalize (1) across agent policies and (2) across model scales and families, further validating the practicality of ALIGN for agent development and environment design.

4.5 Ablation study

Ablation on interface components. Starting from the full ALIGN interface, we conduct two ablations: (1) w/o INFERRULES and (2) w/o WRAPSTEP. Table 4 reports the change relative to the full interface on ALFWorld and ScienceWorld, and the full results can be found in Appendix D.2. Both ablations degrade performance, confirming that each component of the interface contributes meaningfully. Meanwhile, omitting WRAPSTEP leads to markedly larger declines, showing the critical role of fine-grained, enriched observation during interaction. This also suggests that future environment designers should prioritize rich, LLM-friendly observation when constructing environments.

Ablation on experimental verification. To test whether the procedure of experimental verification is truly indispensable, we ablated it and re-ran the pipeline with the Vanilla agent on ALFWorld. In each iteration, the Analyzer first sampled six candidate misalignment sets and picked the one it believed most accurate; the Optimizer then generated six candidate interfaces and likewise selected its top choice. We evaluated two decoding temperatures (T=0.5 and T=0.2; exact prompt we used are shown in Appendix C.4). The resulting task accuracy over four optimization turns is summarized in Table 5. Without the ability to execute experiments, task accuracy deteriorates sharply, a result of the limited single-shot reliability of LLMs in both diagnosing misalignments and synthesizing correct interfaces, which underscore the necessity of our experimental verification procedure design.

Table 5: Task accuracy (%) on ALFWorld across turns without experimental verification.

Temp.	**Turn0**	**Turn1**	**Turn2**	**Turn3**
0.2	13.43	22.39	0.00	0.00
0.5	13.43	23.88	1.49	0.75

5 Conclusion

In this work, we introduce **ALIGN**, a novel framework that automatically generates aligned interfaces to alleviate the **agent-environment misalignment**, a pervasive and underexplored source of failure in interactive decision-making tasks. By diagnosing implicit constraints through the Analyzer and synthesizing aligned interface via the Optimizer, ALIGN improves agent performance significantly on four representative benchmarks across three domains: embodied tasks, web navigation, and tool-use. Our results demonstrate that ALIGN not only boosts performance across multiple agent methods but also generalizes effectively to unseen models and strategies, offering a robust, plug-and-play solution that decouples agent designs from manual environment-specific alignment. These findings suggest that automatic interface generation is a promising direction for building more reliable, reusable, and interpretable LLM-based agents. Future research should explore richer forms of interface representation, expand evaluations to more domains, and develop finer-grained metrics to quantify interface quality and its impact on agent behavior.

References

[1] S. Agashe, J. Han, S. Gan, J. Yang, A. Li, and X. E. Wang. Agent S: an open agentic framework that uses computers like a human. *CoRR*, abs/2410.08164, 2024. doi: 10.48550/ARXIV.2410.08164. URL https://doi.org/10.48550/arXiv.2410.08164.

[2] H. Bai, Y. Zhou, E. L. Li, S. Levine, and A. Kumar. Digi-Q: Transforming VLMs to device-control agents via value-based offline RL, 2025.

[3] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, Q. V. Do, Y. Xu, and P. Fung. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. In J. C. Park, Y. Arase, B. Hu, W. Lu, D. Wijaya, A. Purwarianti, and A. A. Krisnadhi, editors, *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, IJCNLP 2023 - Volume 1: Long Papers, Nusa Dua, Bali, November 1 - 4, 2023*, pages 675–718. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.IJCNLP-MAIN.45. URL https://doi.org/10.18653/v1/2023.ijcnlp-main.45.

[4] C. Bonnet, D. Luo, D. Byrne, S. Surana, S. Abramowitz, P. Duckworth, V. Coyette, L. I. Midgley, E. Tegegn, T. Kalloniatis, O. Mahjoub, M. Macfarlane, A. P. Smit, N. Grinsztajn, R. Boige, C. N. Waters, M. A. Mimouni, U. A. M. Sob, R. de Kock, S. Singh, D. Furelos-Blanco, V. Le, A. Pretorius, and A. Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in JAX. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=C4CxQmp9wc.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

[6] T. Bula, S. Pujar, L. Buratti, M. Bornea, and A. Sil. SeaView: Software engineering agent visual interface for enhanced workflow. *arXiv preprint arXiv:2504.08696*, 2025.

[7] H. Chae, N. Kim, K. T. iunn Ong, M. Gwak, G. Song, J. Kim, S. Kim, D. Lee, and J. Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=moWiYJuSGF.

[8] B. Chen, C. Shu, E. Shareghi, N. Collier, K. Narasimhan, and S. Yao. FireAct: Toward language agent fine-tuning. *CoRR*, abs/2310.05915, 2023. doi: 10.48550/ARXIV.2310.05915. URL https://doi.org/10.48550/arXiv.2310.05915.

[9] M. Chen, Y. Li, Y. Yang, S. Yu, B. Lin, and X. He. AutoManual: Constructing instruction manuals by LLM agents via interactive environmental learning. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/0142921fad7ef9192bd87229cdafa9d4-Abstract-

[10] Z. Chen, K. Liu, Q. Wang, W. Zhang, J. Liu, D. Lin, K. Chen, and F. Zhao. Agent-FLAN: Designing data and methods of effective agent tuning for large language models. In L. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 9354–9366. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.557. URL https://doi.org/10.18653/v1/2024.findings-acl.557.

[11] Z. Chen, M. Li, Y. Huang, Y. Du, M. Fang, and T. Zhou. ATLaS: Agent tuning via learning critical steps. *CoRR*, abs/2503.02197, 2025. doi: 10.48550/ARXIV.2503.02197. URL https://doi.org/10.48550/arXiv.2503.02197.

[12] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence. PaLM-E: An embodied multimodal language model. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202

of *Proceedings of Machine Learning Research*, pages 8469–8488. PMLR, 2023. URL https://proceedings.mlr.press/v202/driess23a.html.

[13] P. Feng, Y. He, G. Huang, Y. Lin, H. Zhang, Y. Zhang, and H. Li. AGILE: A novel reinforcement learning framework of LLM agents. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/097c514162ea7126d40671d23e12f51b-Abstract-

[14] D. Fu, K. He, Y. Wang, W. Hong, Z. Gongque, W. Zeng, W. Wang, J. Wang, X. Cai, and W. Xu. AgentRefine: Enhancing agent generalization through refinement tuning. *CoRR*, abs/2501.01702, 2025. doi: 10.48550/ARXIV.2501.01702. URL https://doi.org/10.48550/arXiv.2501.01702.

[15] Google. Gemini 2.5 Pro preview model card, 2025. URL https://storage.googleapis.com/model-cards/documents/gemini-2.5-pro-preview.pdf.

[16] B. Gou, R. Wang, B. Zheng, Y. Xie, C. Chang, Y. Shu, H. Sun, and Y. Su. Navigating the digital world as humans do: Universal visual grounding for GUI agents. *CoRR*, abs/2410.05243, 2024. doi: 10.48550/ARXIV.2410.05243. URL https://doi.org/10.48550/arXiv.2410.05243.

[17] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang. Large language model based multi-agents: A survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 8048–8057. ijcai.org, 2024. URL https://www.ijcai.org/proceedings/2024/890.

[18] S. Hao, Y. Gu, H. Ma, J. J. Hong, Z. Wang, D. Z. Wang, and Z. Hu. Reasoning with language model is planning with world model. In H. Bouamor, J. Pino, and K. Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 8154–8173. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.507. URL https://doi.org/10.18653/v1/2023.emnlp-main.507.

[19] H. He, W. Yao, K. Ma, W. Yu, Y. Dai, H. Zhang, Z. Lan, and D. Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In L. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 6864–6890. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.371. URL https://doi.org/10.18653/v1/2024.acl-long.371.

[20] K. He, M. Zhang, S. Yan, P. Wu, and Z. Z. Chen. IDEA: Enhancing the rule learning ability of large language model agent through induction, deduction, and abduction, 2024. URL https://arxiv.org/abs/2408.10455.

[21] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[22] E. Kolve, R. Mottaghi, D. Gordon, Y. Zhu, A. Gupta, and A. Farhadi. AI2-THOR: an interactive 3d environment for visual AI. *CoRR*, abs/1712.05474, 2017. URL http://arxiv.org/abs/1712.05474.

[23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[24] X. Lei, Z. Yang, X. Chen, P. Li, and Y. Liu. Scaffolding coordinates to promote vision-language coordination in large multi-modal models. In O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. D. Eugenio, and S. Schockaert, editors, *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 2886–2903. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.coling-main.195/.

[25] B. Y. Lin, Y. Fu, K. Yang, F. Brahman, S. Huang, C. Bhagavatula, P. Ammanabrolu, Y. Choi, and X. Ren. SwiftSage: A generative agent with fast and slow thinking for complex interactive tasks. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/4b0eea69deea512c9e2c469187643dc2-Abstract-`

[26] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang, S. Zhang, X. Deng, A. Zeng, Z. Du, C. Zhang, S. Shen, T. Zhang, Y. Su, H. Sun, M. Huang, Y. Dong, and J. Tang. AgentBench: Evaluating llms as agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=zAdUB0aCTQ`.

[27] Y. Lu, J. Yang, Y. Shen, and A. Awadallah. OmniParser for pure vision based GUI agent. *CoRR*, abs/2408.00203, 2024. doi: 10.48550/ARXIV.2408.00203. URL `https://doi.org/10.48550/arXiv.2408.00203`.

[28] C. Ma, J. Zhang, Z. Zhu, C. Yang, Y. Yang, Y. Jin, Z. Lan, L. Kong, and J. He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL `http://papers.nips.cc/paper_files/paper/2024/hash/877b40688e330a0e2a3fc24084208dfa-Abstract-`

[29] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark. Self-Refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-`

[30] Meta. Model cards and prompt formats Llama 3.1, 2025. URL `https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_1/`.

[31] Meta. Model cards and prompt formats Llama 3.3, 2025. URL `https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/`.

[32] R. Niu, J. Li, S. Wang, Y. Fu, X. Hu, X. Leng, H. Kong, Y. Chang, and Q. Wang. ScreenAgent: A vision language model-driven computer control agent. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 6433–6441. ijcai.org, 2024. URL `https://www.ijcai.org/proceedings/2024/711`.

[33] OpenAI. Introducing GPT-4.1 in the api, 2025. URL `https://openai.com/index/gpt-4-1/`.

[34] B. Paranjape, S. M. Lundberg, S. Singh, H. Hajishirzi, L. Zettlemoyer, and M. T. Ribeiro. ART: automatic multi-step reasoning and tool-use for large language models. *CoRR*, abs/2303.09014, 2023. doi: 10.48550/ARXIV.2303.09014. URL `https://doi.org/10.48550/arXiv.2303.09014`.

[35] Z. Qi, X. Liu, I. L. Iong, H. Lai, X. Sun, W. Zhao, Y. Yang, X. Yang, J. Sun, S. Yao, T. Zhang, W. Xu, J. Tang, and Y. Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. *CoRR*, abs/2411.02337, 2024. doi: 10.48550/ARXIV.2411.02337. URL `https://doi.org/10.48550/arXiv.2411.02337`.

[36] Y. Qin, Y. Ye, J. Fang, H. Wang, S. Liang, S. Tian, J. Zhang, J. Li, Y. Li, S. Huang, W. Zhong, K. Li, J. Yang, Y. Miao, W. Lin, L. Liu, X. Jiang, Q. Ma, J. Li, X. Xiao, K. Cai, C. Li, Y. Zheng, C. Jin, C. Li, X. Zhou, M. Wang, H. Chen, Z. Li, H. Yang, H. Liu, F. Lin, T. Peng, X. Liu, and G. Shi. UI-TARS: pioneering automated GUI interaction with native agents. *CoRR*, abs/2501.12326, 2025. doi: 10.48550/ARXIV.2501.12326. URL `https://doi.org/10.48550/arXiv.2501.12326`.

[37] C. Rawles, S. Clinckemaillie, Y. Chang, J. Waltz, G. Lau, M. Fair, A. Li, W. E. Bishop, W. Li, F. Campbell-Ajala, D. Toyama, R. Berry, D. Tyamagundlu, T. P. Lil-

licrap, and O. Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents. *CoRR*, abs/2405.14573, 2024. doi: 10.48550/ARXIV.2405.14573. URL `https://doi.org/10.48550/arXiv.2405.14573`.

[38] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-`

[39] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-`

[40] M. Shridhar, X. Yuan, M. Côté, Y. Bisk, A. Trischler, and M. J. Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=0IOX0YcCdTn`.

[41] Q. Sun, K. Cheng, Z. Ding, C. Jin, Y. Wang, F. Xu, Z. Wu, C. Jia, L. Chen, Z. Liu, B. Kao, G. Li, J. He, Y. Qiao, and Z. Wu. OS-Genesis: Automating GUI agent trajectory construction via reverse task synthesis. *CoRR*, abs/2412.19723, 2024. doi: 10.48550/ARXIV.2412.19723. URL `https://doi.org/10.48550/arXiv.2412.19723`.

[42] Q. Team. Qwen2.5: A party of foundation models, September 2024. URL `https://qwenlm.github.io/blog/qwen2.5/`.

[43] M. Towers, A. Kwiatkowski, J. K. Terry, J. U. Balis, G. D. Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, H. Tan, and O. G. Younis. Gymnasium: A standard interface for reinforcement learning environments. *CoRR*, abs/2407.17032, 2024. doi: 10.48550/ARXIV.2407.17032. URL `https://doi.org/10.48550/arXiv.2407.17032`.

[44] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024, 2024. URL `https://openreview.net/forum?id=ehfRiF0R3a`.

[45] R. Wang, P. A. Jansen, M. Côté, and P. Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In Y. Goldberg, Z. Kozareva, and Y. Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 11279–11298. Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.EMNLP-MAIN.775. URL `https://doi.org/10.18653/v1/2022.emnlp-main.775`.

[46] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-Consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL `https://openreview.net/forum?id=1PL1NIMMrw`.

[47] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=jJ9BoXAfFa`.

[48] Z. Wang, Y. Dong, F. Luo, M. Ruan, Z. Cheng, C. Chen, P. Li, and Y. Liu. How do multimodal large language models handle complex multimodal reasoning? placing them in an extensible escape game, 2025. URL `https://arxiv.org/abs/2503.10042`.

[49] Z. Wang, K. Wang, Q. Wang, P. Zhang, L. Li, Z. Yang, K. Yu, M. N. Nguyen, L. Liu, E. Gottlieb, M. Lam, Y. Lu, K. Cho, J. Wu, L. Fei-Fei, L. Wang, Y. Choi, and M. Li. RAGEN: Understanding self-evolution in llm agents via multi-turn reinforcement learning, 2025. URL `https://arxiv.org/abs/2504.20073`.

[50] J. Wei, Z. Sun, S. Papay, S. McKinney, J. Han, I. Fulford, H. W. Chung, A. T. Passos, W. Fedus, and A. Glaese. BrowseComp: A simple yet challenging benchmark for browsing agents, 2025. URL `https://arxiv.org/abs/2504.12516`.

[51] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu. OS-World: Benchmarking multimodal agents for open-ended tasks in real computer environments. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL `http://papers.nips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-`

[52] Z. Xu, S. Jain, and M. S. Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *CoRR*, abs/2401.11817, 2024. doi: 10.48550/ARXIV.2401.11817. URL `https://doi.org/10.48550/arXiv.2401.11817`.

[53] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao. Set-of-Mark prompting unleashes extraordinary visual grounding in GPT-4V. *CoRR*, abs/2310.11441, 2023. doi: 10.48550/ARXIV.2310.11441. URL `https://doi.org/10.48550/arXiv.2310.11441`.

[54] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL `http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-`

[55] Z. Yang, P. Li, and Y. Liu. Failures pave the way: Enhancing large language models through tuning-free rule accumulation. In H. Bouamor, J. Pino, and K. Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 1751–1777. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.109. URL `https://doi.org/10.18653/v1/2023.emnlp-main.109`.

[56] Z. Yang, P. Li, M. Yan, J. Zhang, F. Huang, and Y. Liu. ReAct meets ActRe: When language agents enjoy training data autonomy. *CoRR*, abs/2403.14589, 2024. doi: 10.48550/ARXIV.2403.14589. URL `https://doi.org/10.48550/arXiv.2403.14589`.

[57] S. Yao, H. Chen, J. Yang, and K. Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL `http://papers.nips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-`

[58] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL `https://openreview.net/forum?id=WE_vluYUL-X`.

[59] A. Zeng, M. Liu, R. Lu, B. Wang, X. Liu, Y. Dong, and J. Tang. AgentTuning: Enabling generalized agent abilities for LLMs. In L. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 3053–3077. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.181. URL `https://doi.org/10.18653/v1/2024.findings-acl.181`.

[60] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su. GPT-4V(ision) is a generalist web agent, if grounded. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=piecKJ2DlB`.

[61] L. Zheng, W. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging LLM-as-a-Judge with MT-Bench and chatbot arena. In A. Oh, T. Naumann, A. Globerson,

K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-

[62] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

[63] S. Zhou, T. Zhou, Y. Yang, G. Long, D. Ye, J. Jiang, and C. Zhang. WALL-E: world alignment by rule learning improves world model-based LLM agents. *CoRR*, abs/2410.07484, 2024. doi: 10.48550/ARXIV.2410.07484. URL https://doi.org/10.48550/arXiv.2410.07484.

[64] Y. Zhou, A. Zanette, J. Pan, S. Levine, and A. Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=b6rA0kAHT1.

A Limitations and future work

Despite the effectiveness of ALIGN and its potential to alleviate agent-environment misalignment, this work represents only an initial exploration into automated interface generation. Several important directions remain open for further investigation:

Toward a unified and comprehensive interface paradigm. In this work, interface construction primarily focuses on enriching static environment information and enhancing observation during agent-environment interaction. However, our evaluation is limited to three domains: embodied tasks, web navigation, and tool-use. Future studies should extend to a broader range of scenarios and systematically explore the space of possible interface representations.

Metrics for interface quality. This paper evaluates interface effectiveness using downstream task success rates and the proportion of consecutive invalid actions. However, more metrics are needed to quantify the interface's influence on the agent's interaction trajectory. Promising directions include developing finer-grained behavioral diagnostics or employing LLM-as-a-Judge [61] paradigms to evaluate interface quality.

B Preliminary experiments setup

To preliminarily assess the significance of agent-environment misalignment, we conducted exploratory experiments on the ALFWorld. We employed the vanilla Qwen2.5-7B-Instruct agent with a temperature setting of 0.0. The deployment protocol, prompt template, followed the same configuration described in Appendix C and Appendix C.4.

During the experiments, we introduced a minor modification to the environment: if the agent issued the action *examine receptacle* and the environment returned the default observation "Nothing happens.", we replaced it with "You need to first go to receptacle before you can examine it." This simple adjustment increased the agent's task success rate from 13.4% to 31.3%.

C Implementation details

C.1 Benchmarks task splits

The task splits of benchmarks we use are as follows:

(1) ALFWorld [40]: We adhere to the original dataset partitioning presented in the paper, wherein the tasks from the "eval_out_of_distribution" category are used as the test set, and the "train" category is designated as the training set. In each iteration, we randomly select three tasks from the training set of each task type to serve as the training data for the agent's interaction.

(2) ScienceWorld [45]:We follow the original partitioning of the train and test sets as described in the paper. For efficiency reasons, during testing, we select at most the first five tasks from the 30 available task types for evaluation. In each iteration, we randomly select one task from the training set of each task type to be used as the training data for the agent's interaction.

(3) WebShop [57]: In alignment with the setup of Yao et al. [58], we use tasks with IDs ranging from 0 to 49 (50 tasks in total) as the test set, and tasks with IDs from 50 to 199 (150 tasks in total) as the training set. In each iteration, we randomly select 20 tasks from the training set to serve as the training data for the agent's interaction.

(4) M^{3}ToolEval [47]: Since M^{3}ToolEval does not provide a distinct training set division, we select two tasks from each task type in the original dataset as the training set, with the remaining tasks used as the test set. In each iteration, the entire training set is utilized for the agent's interaction.

C.2 Hyperparameter and experiment setting

For all the agents, we deploy them uniformly using vllm [23] across 8 Nvidia A100 80GB GPUs, with the inference temperature set to 0.0. The models utilized contain Qwen2.5-7B-Instruct[1] [42], Qwen2.5-14B-Instruct[2] [42], Llama3.1-8B-Instruct[3] [30] and Llama3.3-70B-Instruct[4] [31].

In ALIGN, we use Gemini 2.5 Pro (gemini-2.5-pro-exp-03-25)[15] for Optimizer to generate new interface, with the temperature set to 0.2. For other scenarios requiring the use of an LLM, we employ GPT-4.1 (gpt-4.1-2025-04-14)[33]. We set $K = 8$ during experiments.

C.3 Tools for experimental verification

In order to implement the experimental verification process mentioned in Section 3.3, we have encapsulated the following tools for Analyzer and Optimizer to interact with the interface-wrapped environment:

(1) `init_simulator(task_id, interface)`: Initializes an experimental task, specifying the task ID and the interface code.

(2) `reset_simulator()`: Resets the experimental task.

(3) `run_task()`: Runs the current task until completion, returning the interaction trajectory.

(4) `exec_agent_action(agent_action)`: Executes a specific action and returns the enhanced observation after the interface processing.

(5) `get_agent_action()`: Based on the current trajectory, returns the next action to be issued by the agent.

(6) `change_obs(obs)`: Modifies the observation of the previous action execution.

C.4 Prompt templates

We present the prompt template of the Analyzer and Optimizer. For the prompt templates of other benchmarks, please refer to the code repository.

Analyzer Prompt Template of Misalignment Analysis

User message:
In modern benchmarks evaluating LLM Agent reasoning capabilities, human designers create an Environment with a set of rules defining how tasks are accomplished. These rules, referred to as the Environment's World Model, specify the sequence of actions required to achieve specific outcomes. For example, the Environment's World Model might dictate that certain actions (e.g., operating on a receptacle) can only be performed after prerequisite actions (e.g., moving to the receptacle).

[1] https://huggingface.co/Qwen/Qwen2.5-7B-Instruct [2] https://huggingface.co/Qwen/Qwen2.5-14B-Instruct
[3] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct [4] https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct

Meanwhile, the Agent operates based on its own World Model, which it constructs by interpreting the task and environment prompts. The Agent first determines its high-level reasoning intent—its understanding of what needs to be done—and then selects actions according to its internal World Model. However, because the Environment's World Model is manually crafted and may not be fully conveyed through prompts, the Agent's World Model might differ, leading to unexpected behavior. For instance, the Agent might choose an action that aligns with its intent but violates the Environment's rules, or it might misinterpret feedback due to insufficient information from the Environment.

We define a misalignment between the Environment's World Model and the Agent's World Model as a situation where:
- The Environment provides feedback that does not sufficiently clarify its World Model, leaving the Agent unable to adjust its understanding of the rules.

Your task is to analyze the logs from a recent task to determine whether such a misalignment occurred, preventing a fair assessment of the Agent's capabilities. And this misalignment has not been fixed by current 'WrapStep' function. Your analysis will guide us in addressing this issue moving forward.

### Experimental Environment Evaluation Template

```python
{{ experimental_template }}
```

In this template, the function 'InferRules' is used to define the environment rules. The function 'WrapStep' handles post-processing of the Agent's actions (e.g., splitting them into multiple steps, performing pre-checks, returning more detailed feedback, etc.). This function should not interfere with the Agent's own reasoning. There current implementation is as follows:

```python
{{ Interface }}
```

### Environment Logs

```txt
{{ logs }}
```

Here, each 'Observation' is the feedback returned to the Agent after it executes an action.

### Gold Action and Observation Sequence

```txt
{{ gold_action_obs_sequence }}
```

### Environment Logics and Misalignment Analyzed in the Previous Steps

{{ environment_logics }}

### Your Task

Determine whether, during this task, there was a misalignment between the Environment's World Model and the Agent's World Model that hindered a fair assessment of the Agent's capabilities. Choose exactly one of the following outputs:

If there is NO misalignment (i.e., the Agent's failures stem from its own errors or limitations, not a mismatch with the Environment's World Model), output:
<analysis_result> No Misalignment </analysis_result>

If there IS a misalignment (i.e., the Environment's World Model conflicts with the Agent's World Model), output:
<analysis_result> Found Misalignment </analysis_result>
<environment_logic_and_misalignments> the new environment rules and misalignments identified by you, which have not been fixed by current ‘WrapStep‘ function.
</environment_logic_and_misalignments>

The format of the environment logic and misalignment is as follows:
‘‘‘txt
### Analysis Result 1
Analysis Task ID: xxx
Agent Action Type: xxx # The type of action the Agent attempted to perform, such as "examine", "move object to receptacle", etc.
Agent Action Case: xxx # The specific action the Agent attempted to perform.
Agent High-Level Reasoning Intent: xxx # The Agent's high-level reasoning intent, which may be a general description of the action it was trying to perform.
Environment World Model Rule: xxx # The rule from the Environment's World Model that don't align the Agent's World Model.
Sufficient Environment Feedback: xxx # to offer the Agent adequate information to bridge gaps in understanding the environment's world model. such as "The environment should provide 'xxx' feedback when the Agent attempts to operate on a receptacle without first going to it."
Type: "Bug of current WrapStep function" or "Need to add new logic in the WrapStep function"

### Analysis Result 2
...
‘‘‘

Note: You should not generate duplicate misalignment analysis results as the ones already provided in the ‘Environment Logics and Misalignment Analyzed in the Previous Steps‘ section.

Analyzer Prompt Template of Experimental Verification

User message:
Now you should conduct simulation experiments in the simulator to verify that the environment rules you hypothesized and Misalignment you identified truly exists. You must perform sufficient experiments to confirm or refute your suspicion.

Here are the operations you can use:

1. init_simulator(task_id: str)
- Initializes a new simulator for the specified ‘task_id‘.
- ‘task_id‘ must be in the format 'int-int' where the first int $\in [0, 5]$.

- The different task types are mapped as follows:

0: ’pick_and_place’,
1: ’pick_clean_and_place’,
2: ’pick_heat_and_place’,
3: ’pick_cool_and_place’,
4: ’look_at_or_examine_in_light’,
5: ’pick_two_obj_and_place’

- All subsequent operations occur within this initialized simulator.

2. reset_simulator()
- Resets the current simulator to its initial state.

3. execute_agent_action(agent_action: str)
- Executes an agent action using the ‘WrapStep‘ function.

4. change_last_action_observation(obs: str)
- Updates the last observation returned by the simulator to the specified ‘obs‘.
- This is useful for simulating the agent’s next action in a different environment feedback context.

5. get_next_agent_action()
- Retrieves the next action that the real Agent would perform under the current simulation conditions.
- Note: The Agent’s choice of the next action is based on the current environment state, including the outcomes of any previous ‘step()‘ or ‘get_next_agent_action()‘ call, along with the latest observations.

If you believe you have reached a conclusion from your experiments, provide it in this format:

<thought> Your reasoning here </thought>
<environment_logic_and_misalignments> the new environment rules and misalignments identified by you, which have not been fixed by current ‘WrapStep‘ function. </environment_logic_and_misalignments>

The format of the environment logic and misalignment is as follows:
‘‘‘txt
### Analysis Result 1
Analysis Task ID: xxx
Agent Action Type: xxx # The type of action the Agent attempted to perform, such as "examine", "move object to receptacle", etc.
Agent Action Case: xxx # The specific action the Agent attempted to perform.
Agent High-Level Reasoning Intent: xxx # The Agent’s high-level reasoning intent, which may be a general description of the action it was trying to perform.
Environment World Model Rule: xxx # The rule from the Environment’s World Model that don’t align the Agent’s World Model.
Sufficient Environment Feedback: xxx # to offer the Agent adequate information to bridge gaps in understanding the environment’s world model. such as "The environment should provide ’xxx’ feedback when the Agent attempts to operate on a receptacle without first going to it."
Type: "Bug of current WrapStep function" or "Need to add new logic in the WrapStep function"

### Analysis Result 2
...

‘‘‘

If you need to carry out more operations in the simulator, respond in the following format, specifying exactly one operation per turn:

<thought> Your reasoning here, you should consider all hypotheses if the simulation result is not as expected </thought>
<action> The single operation you wish to perform (e.g., init_simulator(task_id="x-y"), step(action="x"), execute_agent_action(agent_action="x"), etc.) </action>

Note:
You should verify the correctness of the following, step by step, through your experiments:
1. environment_rules: Use ‘execute_agent_action‘ to confirm that the environment rules you hypothesized are indeed correct, and current ‘WrapStep‘ function is not sufficient.
2. agent_intent_description: Obtain the Agent’s intended behavior (e.g., via ‘get_next_agent_action‘) and simulate it by using ‘WrapStep‘ to confirm whether it aligns with your description.
3. identified_misalignment: Through chaning the environment feedback, you can verify whether the misalignment you identified is indeed correct and the environment feedback you hypothesized is indeed sufficient. You can use ‘WrapStep‘ to simulate the agent’s action, then use ‘change_last_action_observation‘ to change the environment feedback, and finally use ‘get_next_agent_action‘ to check whether the agent can correctly identify the next action.

Analyzer Prompt Template of Reranking Misalignments Analysis (Ablation Study)

User message:
In modern benchmarks evaluating LLM Agent reasoning capabilities, human designers create an Environment with a set of rules defining how tasks are accomplished. These rules, referred to as the Environment’s World Model, specify the sequence of actions required to achieve specific outcomes. For example, the Environment’s World Model might dictate that certain actions (e.g., operating on a receptacle) can only be performed after prerequisite actions (e.g., moving to the receptacle).

Meanwhile, the Agent operates based on its own World Model, which it constructs by interpreting the task and environment prompts. The Agent first determines its high-level reasoning intent—its understanding of what needs to be done—and then selects actions according to its internal World Model. However, because the Environment’s World Model is manually crafted and may not be fully conveyed through prompts, the Agent’s World Model might differ, leading to unexpected behavior. For instance, the Agent might choose an action that aligns with its intent but violates the Environment’s rules, or it might misinterpret feedback due to insufficient information from the Environment.

We define a misalignment between the Environment’s World Model and the Agent’s World Model as a situation where:
- The Environment provides feedback that does not sufficiently clarify its World Model, leaving the Agent unable to adjust its understanding of the rules.

Now other human experts have analyzed the logs from a recent task and identified some potential misalignments. Your task is to review these misalignments and choose the most appropriate one.

### Experimental Environment Evaluation Template

‘‘‘python

{{ experimental_template }}
‘‘‘

In this template, the function ‘InferRules‘ is used to define the environment rules. The function ‘WrapStep‘ handles post-processing of the Agent’s actions (e.g., splitting them into multiple steps, performing pre-checks, returning more detailed feedback, etc.). This function should not interfere with the Agent’s own reasoning. There current implementation is as follows:

‘‘‘python
{{ Interface }}
‘‘‘

### Environment Logs

‘‘‘txt
{{ logs }}
‘‘‘

Here, each ‘Observation‘ is the feedback returned to the Agent after it executes an action.

### Gold Action and Observation Sequence

‘‘‘txt
{{ gold_action_obs_sequence }}
‘‘‘

### Environment Logics and Misalignment Analyzed in the Previous Steps

{{ environment_logics }} Note: These logics may not be accurate. They are the environment rules that were previously hypothesized and may contain errors.

### Your Task

Choose the most appropriate misalignment analyzed by human experts from the list below:

{{ new_environment_logics }}

You should respond in format as follows:
‘‘‘
<review> Your review of each expert output one by one </review>
<expert_id> id of the selected expert output, only the number </expert_id>
‘‘‘

Optimizer Prompt Template of Interface Generation

User message:
In modern benchmarks evaluating LLM Agent reasoning capabilities, human designers create an Environment with a set of rules defining how tasks are accomplished. These rules, referred to as the Environment’s World Model, specify the sequence of actions required to

achieve specific outcomes. For example, the Environment's World Model might dictate that certain actions (e.g., operating on a receptacle) can only be performed after prerequisite actions (e.g., moving to the receptacle).

Meanwhile, the Agent operates based on its own World Model, which it constructs by interpreting the task and environment prompts. The Agent first determines its high-level reasoning intent—its understanding of what needs to be done—and then selects actions according to its internal World Model. However, because the Environment's World Model is manually crafted and may not be fully conveyed through prompts, the Agent's World Model might differ, leading to unexpected behavior. For instance, the Agent might choose an action that aligns with its intent but violates the Environment's rules, or it might misinterpret feedback due to insufficient information from the Environment.

We define a misalignment between the Environment's World Model and the Agent's World Model as a situation where:
- The Environment provides feedback that does not sufficiently clarify its World Model, leaving the Agent unable to adjust its understanding of the rules.

Your task is to refine the environment's behavior based on the misalignment identified by the AnalysisAgent, ensuring the Agent's true intentions are executed and its reasoning capabilities are fairly assessed.

### Experimental Environment Evaluation Template

```python
{{ experimental_template }}
```

In this template, the function 'InferRules' is used to define the environment rules. The function 'WrapStep' handles post-processing of the Agent's actions (e.g., splitting them into multiple steps, performing pre-checks, returning more detailed feedback, etc.). This function should not interfere with the Agent's own reasoning. There current implementation is as follows:

```python
{{ WrapStep }}
```

### Environment Logics and Misalignment Analyzed by AnalysisAgent Previously

{{ last_environment_logics }}

### New Environment Logics and Misalignment Analyzed by AnalysisAgent

{{ new_environment_logics }}

### Your Task

Based on the misalignments identified by the AnalysisAgent, you need to refine and enhance the 'InferRules' function and 'WrapStep' function to align the Environment's World Model with the Agent's actions and provide clearer feedback. Your output should present the new versions of these functions, ensuring the Agent's high-level reasoning intent is preserved.

Please ensure you follow these requirements:

1. **Function Signature**
The function signature must be:
“‘python
def InferRules(init_obs, task)
- init_obs: str, the initial observation from the environment, containing all receptacles.
- task: str, the task description.

def WrapStep(env, init_obs, task, agent_action: str, logger)
“‘

2. **Return Values**
The ‘InferRules‘ function’s return value must be a string that describes the environment rules.

The ‘WrapStep‘ function’s return value must be three items:
“‘python
obs: str, reward: bool, done: bool
“‘

3. **‘env.step‘ Usage**
The only permitted usage pattern for ‘env.step‘ is:
“‘python
obs, reward, done, info = env.step([agent_action])
obs, reward, done = obs[0], info[’won’][0], done[0]
“‘
No alternative usage forms are allowed. Each call to env.step causes an irreversible change to the environment state; actions must therefore be chosen carefully.

4. **Package Imports**
You may import other packages if necessary, but you must include all imports in your code.

5. **Multiple Calls and Conditional Returns**
You are free to call ‘env.step‘ multiple times or return different ‘obs‘ depending on ‘agent_action‘ or the outcomes of these calls.

6. **You can use logger.debug**
You can use ‘logger.debug‘ to log any information you find useful. The logging will be captured and returned to you in the future for further analysis.

7. Do not modify any aspects not explicitly identified by the AnalysisAgent in the “New Environment Logics and Misalignment Analyzed by AnalysisAgent” section.

8. You must use the following approach when addressing the identified misalignment:
- For each action defined in environment, provide clear, informative, and sufficient feedback from the environment whenever an invalid action is attempted, guiding the Agent toward understanding and adhering to the environment’s rules.

9. **Output Format**
You must provide the output strictly in the following format:
<thought>YOUR_THOUGHT_PROCESS_HERE</thought>
<code>YOUR_CODE_HERE</code>

Please ensure your final answer follows these guidelines so that we can accurately bridge the misalignment and allow the environment to execute the Agent's true intentions.

Optimizer Prompt Template of Experimental Verification

User message:
Now you should conduct simulation experiments in the simulator to verify if the 'InferRules' and 'WrapStep' function you provided is correct for the new environment logics and misalignment analyzed by the AnalysisAgent.

You must perform sufficient experiments to confirm or refute your suspicion. Here are the operations you can use:

1. init_simulator(task_id: str)
- Initializes a new simulator for the specified 'task_id'.
- 'task_id' must be in the format 'int-int' where the first int ∈ [0, 5].
- The different task types are mapped as follows:

0: 'pick_and_place',
1: 'pick_clean_and_place',
2: 'pick_heat_and_place',
3: 'pick_cool_and_place',
4: 'look_at_or_examine_in_light',
5: 'pick_two_obj_and_place'

- All subsequent operations occur within this initialized simulator.

2. reset_simulator()
- Resets the current simulator to its initial state.

3. execute_agent_action(agent_action: str)
- Executes an agent action using the 'WrapStep' function you generated.

4. change_last_action_observation(obs: str)
- Updates the last observation returned by the simulator to the specified 'obs'.
- This is useful for simulating the agent's next action in a different environment feedback context.

5. get_next_agent_action()
- Retrieves the next action that the real Agent would perform under the current simulation conditions.
- Note: The Agent's choice of the next action is based on the current environment state, including the outcomes of any previous 'step()' or 'get_next_agent_action()' call, along with the latest observations.

6. run_task(task_id: str)
- Runs the entire task in the simulator and returns the running log.
- After running the whole task, you need to call 'init_simulator' or 'reset_simulator' to reinitialize the simulator for further operations.

If you believe you have reached a conclusion from your experiments, provide it in this format:

<thought> Your reasoning here </thought>
<if_need_refine> True/False </if_need_refine>
<refine_strategy> Your strategy for refining the WrapStep function, if if_need_refine is True </refine_strategy>

If you need to carry out more operations in the simulator, respond in the following format, specifying exactly one operation per turn:

<thought> Your reasoning here, you should consider all hypotheses if the simulation result is not as expected </thought>
<action> The single operation you wish to perform (e.g., init_simulator(task_id="x-y"), step(action="x"), execute_agent_action(agent_action="x"), etc.) </action>

Optimizer Prompt Template of Reranking Interface Generation (Ablation Stuty)

User message:
In modern benchmarks evaluating LLM Agent reasoning capabilities, human designers create an Environment with a set of rules defining how tasks are accomplished. These rules, referred to as the Environment's World Model, specify the sequence of actions required to achieve specific outcomes. For example, the Environment's World Model might dictate that certain actions (e.g., operating on a receptacle) can only be performed after prerequisite actions (e.g., moving to the receptacle).

Meanwhile, the Agent operates based on its own World Model, which it constructs by interpreting the task and environment prompts. The Agent first determines its high-level reasoning intent—its understanding of what needs to be done—and then selects actions according to its internal World Model. However, because the Environment's World Model is manually crafted and may not be fully conveyed through prompts, the Agent's World Model might differ, leading to unexpected behavior. For instance, the Agent might choose an action that aligns with its intent but violates the Environment's rules, or it might misinterpret feedback due to insufficient information from the Environment.

We define a misalignment between the Environment's World Model and the Agent's World Model as a situation where:
- The Environment provides feedback that does not sufficiently clarify its World Model, leaving the Agent unable to adjust its understanding of the rules.

Now other human experts have generated a set of code patches to address the misalignment between the Environment's World Model and the Agent's World Model. Your task is to evaluate these patches and select the best one.

### Experimental Environment Evaluation Template

“‘python
{{ experimental_template }}
“‘

In this template, the function ‘InferRules‘ is used to define the environment rules. The function ‘WrapStep‘ handles post-processing of the Agent's actions (e.g., splitting them into multiple steps, performing pre-checks, returning more detailed feedback, etc.). This function should not interfere with the Agent's own reasoning. There current implementation is as follows:

“‘python
{{ WrapStep }}
“‘

### Environment Logics and Misalignment Analyzed by AnalysisAgent Previously

{{ last_environment_logics }}

### New Environment Logics and Misalignment Analyzed by AnalysisAgent

{{ new_environment_logics }}

### Your Task

Choose the best code from the following options to address the misalignment between the Environment's World Model and the Agent's World Model:

{{ code_patches }}

You should respond in format as follows:
‘‘‘
<review> Your review of each code one by one </review>
<code_id> id of the selected result, only the number </code_id>
‘‘‘

We present the prompt template of the Vanilla agent in ALFWorld to illustrate the usage of the INFERRULES. For the prompt templates of other agent methods and benchmarks, please refer to the code repository.

Vanilla Agent Prompt Template in ALFWorld

System message:
You are an AI assistant solving tasks in a household environment. Your goal is to break down complex tasks into simple steps and plan your actions accordingly.

# Action Space

In this environment, you have a set of high-level actions at your disposal, each corresponding to a typical household activity. These actions are:

- look: look around your current location
- inventory: check your current inventory
- go to (receptacle): move to a receptacle
- open (receptacle): open a receptacle
- close (receptacle): close a receptacle
- take (object) from (receptacle): take an object from a receptacle
- move (object) to (receptacle): place an object in or on a receptacle
- examine (something): examine a receptacle or an object
- use (object): use an object
- heat (object) with (receptacle): heat an object using a receptacle
- clean (object) with (receptacle): clean an object using a receptacle
- cool (object) with (receptacle): cool an object using a receptacle
- slice (object) with (object): slice an object using a sharp object

Although each action may internally consist of multiple embodied steps (e.g., walking to the sink, turning a knob, etc.), from your perspective you need only provide one high-level action at a time.

# Instructions

Single Action per Turn
At each step, you must respond with exactly one action (i.e., the next "thought"). Use the format:
ACTION [object/receptacle specifier]
ACTION [object/receptacle specifier]
For example:
take apple from table
or
go to kitchen.

Environment Feedback
After you provide your single action, the environment will automatically execute it and return the resulting observation. You then decide on your next action based on the updated state.

Reasoning (Chain of Thought)
You may use hidden reasoning to figure out the best next step. However, only output the single action that represents your decision. Do not reveal your entire chain of thought.

Continue Until Task Completion
You will iterate this process—receiving the environment's feedback, deciding on the next action, and outputting a single action—until the task is finished.

# Environment Rule

{InferRules(init_obs, task)}

User message:
# Task

{initial_obs}

Begin by examining the environment or taking any initial steps you find relevant. Remember, provide only one action each time.

C.5 Initialized interface

Initialized interface we used in ALFWorld:

```
def InferRules(init_obs, task):
    """
    Contains the rules for environment and task execute logic for
    different task types.
    """
    return "There is no rule for this environment."

def WrapStep(env, init_obs, task, agent_action: str, logger):
    """
    Process the agent action and return the next observation, reward,
    and done status.
    """
    obs, reward, done, info = env.step([agent_action])
    obs, reward, done = obs[0], info['won'][0], done[0]
    return obs, reward, done
```

Initialized interface we used in ScienceWorld:

```
def InferRules(init_obs, task):
    """
    Contains the rules for environment and task execute logic for
    different task types.
    """
    return "There is no rule for this environment."

def WrapStep(env, init_obs, task, agent_action: str, logger):
    """
    Process the agent action and return the next observation, done
    status and score(returned by the environment).
    """
    obs, _, done, info = env.step(agent_action)
    return obs, done, info["score"]
```

Initialized interface we used in WebShop:

```
def InferRules(init_obs, task):
    """
    Contains the rules for environment and task execute logic.
    """
    return "There is no rule for this environment."

def WrapStep(env, init_obs, task, agent_action: str, logger):
    """
    Process the agent action and return the next observation, reward,
    and done status.
    """
    obs, reward, done = env.step(agent_action)
    return obs, reward, done
```

Initialized interface we used in M^{3}ToolEval:

```
def InferRules(task_name, task_type_idx):
    """
    Contains the rules for environment and task execute logic for
    different task types.
    """
    return "There is no rule for this environment."

def WrapStep(env, task_name, instruction, agent_action: str, logger):
    """
    Process the agent action and return the next observation, reward,
    and done status.
    """
    obs, reward, done = env.step(agent_action)
    return obs, reward, done
```

D Full experiment results

D.1 Generalization study results

The full result of generalization study for cross-method experiment can be found in Table 6. The full result of generalization study for cross-model experiment can be found in Table 7, Table 8 and Table 9.

D.2 Ablation study results

The full result of interface ablation experiment can be found in Table 10.

Table 6: **Generalization of ALIGN-generated interfaces generated with Vanilla agents to other agent methods.** For each agent we report its score without the interface (w/o ALIGN) and with the interface (w/ ALIGN); the value in parentheses is the *absolute* improvement. Metrics are task-success rate (%) for ALFWorld and M^{3}ToolEval, and scores for ScienceWorld and WebShop.

Base Method: Vanilla		**Embodied**		**Web**	**Tool-use**
Method	**Interface**	**ALFWorld**	**ScienceWorld**	**WebShop**	**M^{3}ToolEval**
ReAct	w/o ALIGN	19.40	20.03	37.20	9.72
	w/ ALIGN	58.96 (+39.56)	32.32 (+12.29)	45.07 (+7.87)	15.28 (+5.56)
Self-Consistency	w/o ALIGN	11.94	14.07	56.23	11.11
	w/ ALIGN	63.43 (+51.49)	29.37 (+15.30)	59.23 (+3.00)	19.44 (+8.33)
Self-Refine	w/o ALIGN	3.73	14.87	44.80	5.55
	w/ ALIGN	38.06 (+34.33)	28.98 (+14.11)	50.97 (+6.17)	9.72 (+4.17)
Planning	w/o ALIGN	9.70	17.13	46.95	11.11
	w/ ALIGN	50.75 (+41.05)	26.79 (+9.66)	50.21 (+3.26)	22.22 (+11.11)

Table 7: **Generalization of ALIGN-generated interfaces generated with Qwen2.5-7B-Instruct to Qwen2.5-14B-Instruct.** For each agent we report its score without the interface (w/o ALIGN) and with the interface (w/ ALIGN); the value in parentheses is the *absolute* improvement. Metrics are task-success rate (%) for ALFWorld and M^{3}ToolEval, and scores for ScienceWorld and WebShop.

Base Model: Qwen2.5-14B-Instruct		**Embodied**		**Web**	**Tool-use**
Method	**Interface**	**ALFWorld**	**ScienceWorld**	**WebShop**	**M^{3}ToolEval**
Vanilla	w/o ALIGN	48.51	22.58	53.67	13.89
	w/ ALIGN	52.24 (+3.73)	37.58 (+15.00)	58.40 (+4.73)	18.06 (+4.17)
ReAct	w/o ALIGN	54.48	31.24	39.73	15.28
	w/ ALIGN	70.15 (+15.67)	29.79 (-1.45)	42.17 (+2.44)	26.39 (+11.11)
Self-Consistency	w/o ALIGN	43.28	25.60	52.63	13.89
	w/ ALIGN	72.39 (+29.11)	26.68 (+1.08)	51.07 (-1.56)	27.78 (+13.89)
Self-Refine	w/o ALIGN	5.22	18.97	41.00	15.28
	w/ ALIGN	14.18 (+8.96)	20.72 (+1.75)	39.93 (-1.07)	16.67 (+1.39)
Planning	w/o ALIGN	49.25	21.46	31.72	25.00
	w/ ALIGN	79.10 (+29.85)	28.13 (+6.67)	50.47 (+18.75)	25.00 (0.00)

Table 8: **Generalization of ALIGN-generated interfaces generated with Qwen2.5-7B-Instruct to Llama3.1-8B-Instruct.** For each agent we report its score without the interface (w/o ALIGN) and with the interface (w/ ALIGN); the value in parentheses is the *absolute* improvement. Metrics are task-success rate (%) for ALFWorld and M^{3}ToolEval, and scores for ScienceWorld and WebShop.

Base Model: Llama3.1-8B-Instruct		**Embodied**		**Web**	**Tool-use**
Method	**Interface**	**ALFWorld**	**ScienceWorld**	**WebShop**	**M^{3}ToolEval**
Vanilla	w/o ALIGN	5.22	23.59	35.17	5.56
	w/ ALIGN	14.18 (+8.96)	36.40 (+12.81)	24.00 (-11.17)	1.39 (-4.17)
ReAct	w/o ALIGN	1.49	22.42	27.12	12.50
	w/ ALIGN	15.67 (+14.18)	28.74 (+6.32)	27.10 (-0.02)	22.22 (+9.72)
Self-Consistency	w/o ALIGN	5.22	25.21	29.80	4.17
	w/ ALIGN	11.94 (+6.72)	34.83 (+9.62)	15.83 (-13.97)	2.78 (-1.39)
Self-Refine	w/o ALIGN	0.00	22.34	27.70	1.39
	w/ ALIGN	0.75 (+0.75)	31.33 (+8.99)	37.43 (+9.73)	1.39 (0.00)
Planning	w/o ALIGN	6.72	13.33	23.67	4.17
	w/ ALIGN	5.97 (-0.75)	26.95 (+13.62)	40.77 (+17.10)	4.17 (0.00)

Table 9: **Generalization of ALIGN-generated interfaces generated with Qwen2.5-7B-Instruct to Llama3.3-70B-Instruct.** For each agent we report its score without the interface (w/o ALIGN) and with the interface (w/ ALIGN); the value in parentheses is the *absolute* improvement. Metrics are task-success rate (%) for ALFWorld and M^{3}ToolEval, and scores for ScienceWorld and WebShop.

Base Model: Llama3.3-70B-Instruct		Embodied		Web	Tool-use
Method	**Interface**	**ALFWorld**	**ScienceWorld**	**WebShop**	**M^{3}ToolEval**
Vanilla	w/o ALIGN	52.99	55.77	51.67	37.50
	w/ ALIGN	43.28 (-9.71)	57.74 (+1.97)	62.07 (+10.40)	33.33 (-4.17)
ReAct	w/o ALIGN	45.52	56.50	58.22	34.72
	w/ ALIGN	47.01 (+1.49)	58.28 (+1.78)	53.83 (-4.39)	43.06 (+8.34)
Self-Consistency	w/o ALIGN	54.48	56.66	50.37	36.11
	w/ ALIGN	65.67 (+11.19)	59.24 (+2.58)	55.63 (+5.26)	34.72 (-1.39)
Self-Refine	w/o ALIGN	38.06	56.97	38.40	1.39
	w/ ALIGN	46.27 (+8.21)	60.17 (+3.20)	47.85 (+9.45)	0.00 (-1.39)
Planning	w/o ALIGN	58.96	48.75	54.90	33.33
	w/ ALIGN	76.87 (+17.91)	59.17 (+10.42)	62.60 (+7.70)	40.28 (+6.95)

Table 10: Ablation study on the components of ALIGN. Values represent task success rates (%) or scores. For ablated conditions (w/o INFERRULES, w/o WRAPSTEP), performance changes from the 'Full' are shown in parentheses.

Method	**Interface**	**Embodied**		**Web**	**Tool**
		ALFWorld	**ScienceWorld**	**Webshop**	**M^{3}ToolEval**
Vanilla	Full	60.45	27.69	61.23	20.83
	w/o INFERRULES	51.49 (-8.96)	24.34 (-3.35)	51.03 (-10.20)	18.06 (-2.77)
	w/o WRAPSTEP	26.87 (-33.58)	22.97 (-4.72)	61.23 (-0.00)	11.11 (-9.72)
ReAct	Full	63.43	28.97	42.93	18.06
	w/o INFERRULES	58.21 (-5.22)	26.89 (-2.08)	35.97 (-6.96)	9.72 (-8.34)
	w/o WRAPSTEP	45.52 (-17.91)	22.53 (-6.44)	47.60 (+4.67)	19.44 (+1.38)
Self-Consistency	Full	69.40	25.41	61.10	16.67
	w/o INFERRULES	67.91 (-1.49)	23.11 (-2.30)	55.67 (-5.43)	13.89 (-2.78)
	w/o WRAPSTEP	23.13 (-17.91)	14.82 (-10.59)	60.67 (-0.43)	15.28 (-1.39)
Self-Refine	Full	40.30	22.99	52.30	6.94
	w/o INFERRULES	32.84 (-7.46)	21.27 (-1.72)	46.33 (-5.97)	6.94 (-0.00)
	w/o WRAPSTEP	5.97 (-34.33)	15.40 (-7.59)	47.80 (-4.50)	6.94 (-0.00)
Planning	Full	52.99	26.34	54.67	18.06
	w/o INFERRULES	42.54 (-10.45)	25.56 (-0.78)	48.18 (-6.49)	16.67 (-1.39)
	w/o WRAPSTEP	26.12 (-26.87)	16.48 (-9.86)	52.87 (-1.80)	16.67 (-1.39)

D.3 Interfaces generated by ALIGN

We present the ALIGN-generated interface with Vanilla agent to illustrate the interface case. For the ALIGN-generated interface with other agent methods, please refer to the code repository.

ALIGN-generated interface with Vanilla agent in ALFWorld:

```
def InferRules(init_obs, task):
    """
    Provides the rules for environment and task execute logic for
    different task types.

    Args:
        init_obs: Initial observation string containing information
    about the environment
        task: The specific task description

    Returns:
        A string describing the environment rules
    """
    return """
    1. Navigation and Location Rules:
       - You must go to a receptacle before you can examine it, open
    it, close it, or interact with objects in/on it.
       - You can only interact with objects and receptacles that are
    at your current location.
       - If you try to interact with a receptacle or object that is
    not at your current location, you will be informed that you need
    to go to that location first.
       - After successfully going to a location, you are at that
    location until you go somewhere else.

    2. Object Interaction Rules:
       - To take an object, it must be present at your current
    location and visible (not inside a closed receptacle).
       - Once you take an object, it goes into your inventory and is
    no longer at its original location.
       - To move an object to a receptacle, you must have the object
    in your inventory and be at the target receptacle.
       - To use, heat, clean, cool, or slice objects, you must have
    the required objects in your inventory or be at their location.
       - You cannot take an object that is already in your inventory.

    3. Container Rules:
       - Some receptacles can be opened and closed (like refrigerators
    , microwaves, cabinets, etc.).
       - You must open a closed container before you can take objects
    from it or put objects into it.
       - Objects inside closed containers are not visible or
    accessible until the container is opened.

    4. Action Sequence Requirements:
       - Some tasks require a specific sequence of actions - for
    example, to heat food, you need to:
         a) Go to the microwave
         b) Open the microwave
         c) Place the food inside
         d) Close the microwave
         e) Use the microwave
       - The environment will guide you if you're missing a
    prerequisite step for an action.

    5. Feedback Interpretation:
       - If an action cannot be performed, the environment will
    explain why and what prerequisites are needed.
       - The environment will inform you if you try to take an object
    that's already in your inventory.
       - The environment will inform you if you try to move an object
    that's not in your inventory.
       - Pay attention to the feedback to understand the current state
     of the environment and what actions are possible next.
       - When you successfully go to a location, the environment will
    describe what's there.
    """

def WrapStep(env, init_obs, task, agent_action: str, logger):
    """
    Process the agent action and return the next observation, reward,
    and done status.

    Args:
        env: The environment object
        init_obs: Initial observation string containing information
    about the environment
        task: The specific task description
        agent_action: The action string from the agent
        logger: Logger object for debugging information

    Returns:
        obs: Observation string after the action
        reward: Boolean indicating if a reward was received
        done: Boolean indicating if the task is complete
    """
    # Track the agent's current location using an attribute on the env
     object
    if not hasattr(env, '_current_location'):
        env._current_location = None

    # Track container states (open/closed) using an attribute on the
    env object
    if not hasattr(env, '_container_states'):
        env._container_states = {}

    action_item = {
        'matched': False,
        'action': None,
        'object': None,
        'receptacle': None,
        'object2': None
    }

    # Parse the agent action
    # Simple actions without parameters
    if agent_action.lower() == 'look' or agent_action.lower() == '
    inventory':
        action_item['matched'] = True
        action_item['action'] = agent_action.lower()

    # Pattern: go to (receptacle)
    elif agent_action.lower().startswith('go to '):
        receptacle = agent_action[6:].strip()
        action_item['matched'] = True
        action_item['action'] = 'go to'
        action_item['receptacle'] = receptacle

    # Pattern: open/close (receptacle)
    elif agent_action.lower().startswith('open ') or agent_action.
    lower().startswith('close '):
        action = 'open' if agent_action.lower().startswith('open ')
    else 'close'
        receptacle = agent_action[len(action)+1:].strip()
        action_item['matched'] = True
        action_item['action'] = action
        action_item['receptacle'] = receptacle

    # Pattern: take (object) from (receptacle)
    elif 'take ' in agent_action.lower() and ' from ' in agent_action.
    lower():
        parts = agent_action.split(' from ')
        if len(parts) == 2:
            obj = parts[0][5:].strip()  # Remove 'take ' prefix
            receptacle = parts[1].strip()
            action_item['matched'] = True
            action_item['action'] = 'take from'
            action_item['object'] = obj
            action_item['receptacle'] = receptacle
    # Pattern: move (object) to (receptacle)
    elif 'move ' in agent_action.lower() and ' to ' in agent_action.
lower():
        parts = agent_action.split(' to ')
        if len(parts) == 2:
            obj = parts[0][5:].strip()  # Remove 'move ' prefix
            receptacle = parts[1].strip()
            action_item['matched'] = True
            action_item['action'] = 'move to'
            action_item['object'] = obj
            action_item['receptacle'] = receptacle

    # Pattern: examine (something)
    elif agent_action.lower().startswith('examine '):
        something = agent_action[8:].strip()
        action_item['matched'] = True
        action_item['action'] = 'examine'

        # Determine if it's a receptacle or object by checking if it
appears in the initial observation
        if something.lower() in init_obs.lower():
            action_item['receptacle'] = something
        else:
            action_item['object'] = something

    # Pattern: use (object)
    elif agent_action.lower().startswith('use '):
        obj = agent_action[4:].strip()
        action_item['matched'] = True
        action_item['action'] = 'use'
        action_item['object'] = obj

    # Pattern: heat/clean/cool (object) with (receptacle)
    elif any(agent_action.lower().startswith(action) for action in ['
heat ', 'clean ', 'cool ']) and ' with ' in agent_action.lower():
        for action in ['heat ', 'clean ', 'cool ']:
            if agent_action.lower().startswith(action):
                parts = agent_action.split(' with ')
                if len(parts) == 2:
                    obj = parts[0][len(action):].strip()
                    receptacle = parts[1].strip()
                    action_item['matched'] = True
                    action_item['action'] = action.strip()
                    action_item['object'] = obj
                    action_item['receptacle'] = receptacle
                break

    # Pattern: slice (object) with (object)
    elif agent_action.lower().startswith('slice ') and ' with ' in
agent_action.lower():
        parts = agent_action.split(' with ')
        if len(parts) == 2:
            obj = parts[0][6:].strip()  # Remove 'slice ' prefix
            obj2 = parts[1].strip()
            action_item['matched'] = True
            action_item['action'] = 'slice'
            action_item['object'] = obj
            action_item['object2'] = obj2  # Using object2 for the
tool used for slicing

    # If action wasn't matched, provide feedback
    if not action_item['matched']:
        return f"I don't understand the action '{agent_action}'.
Please use one of the allowed actions from the action space.",
False, False
```

```
    logger.debug(f"Parsed action: {action_item}")

    # Get the current observation to check location
    test_obs, _, _, _ = env.step(['look'])
    test_obs = test_obs[0]
    logger.debug(f"Current observation after 'look': {test_obs}")

    # Get inventory to check what objects the agent has
    inventory_obs, _, _, _ = env.step(['inventory'])
    inventory_obs = inventory_obs[0]
    logger.debug(f"Current inventory observation: {inventory_obs}")

    # Improved function to check if an object is in inventory
    def is_in_inventory(object_name):
        object_name_lower = object_name.lower()
        logger.debug(f"Checking if '{object_name_lower}' is in
inventory")

        # Extract inventory items from the observation
        inventory_items = []

        # Check for common inventory patterns
        if "carrying:" in inventory_obs.lower():
            carrying_section = inventory_obs.lower().split("carrying:"
)[1].strip()
            inventory_items = [item.strip() for item in
carrying_section.split(',')]
        elif "inventory:" in inventory_obs.lower():
            inventory_section = inventory_obs.lower().split("inventory
:")[1].strip()
            inventory_items = [item.strip() for item in
inventory_section.split(',')]
        elif "you are carrying:" in inventory_obs.lower():
            carrying_section = inventory_obs.lower().split("you are
carrying:")[1].strip()
            inventory_items = [item.strip() for item in
carrying_section.split(',')]

        # Also check line by line for inventory items
        inventory_lines = inventory_obs.lower().split('\n')
        for line in inventory_lines:
            line = line.strip()
            if line and not line.startswith(("you are", "carrying:", "
inventory:")):
                inventory_items.append(line)

        logger.debug(f"Extracted inventory items: {inventory_items}")

        # Check if object_name or its base name (without numbers) is
in inventory
        base_name = ''.join([c for c in object_name_lower if not c.
isdigit()]).strip()

        for item in inventory_items:
            # Check for exact match
            if object_name_lower == item or f"{object_name_lower} (in
your inventory)" == item:
                logger.debug(f"Found exact match '{item}' in inventory
")
                return True

            # Check for base name match (without numbers)
            if base_name != object_name_lower and (base_name == item
or f"{base_name} (in your inventory)" == item):
```

```
                logger.debug(f"Found base name match '{item}' in
inventory")
                return True

            # Check if item contains the object name
            if object_name_lower in item:
                logger.debug(f"Found partial match '{item}' containing
 '{object_name_lower}' in inventory")
                return True

            # Check if item contains the base name
            if base_name != object_name_lower and base_name in item:
                logger.debug(f"Found partial match '{item}' containing
 base name '{base_name}' in inventory")
                return True

        # Direct check for common patterns in the full inventory text
        patterns = [
            f"carrying: {object_name_lower}",
            f"{object_name_lower} (in your inventory)",
            f"you are carrying: {object_name_lower}",
            f"inventory: {object_name_lower}"
        ]

        if base_name != object_name_lower:
            patterns.extend([
                f"carrying: {base_name}",
                f"{base_name} (in your inventory)",
                f"you are carrying: {base_name}",
                f"inventory: {base_name}"
            ])

        for pattern in patterns:
            if pattern in inventory_obs.lower():
                logger.debug(f"Found pattern '{pattern}' in inventory
text")
                return True

        logger.debug(f"'{object_name_lower}' not found in inventory")
        return False

    # Helper function to check if we're at a location
    def is_at_location(location_name):
        location_name_lower = location_name.lower()

        # If we've already tracked this location, use the tracked
value
        if env._current_location and location_name_lower in env.
_current_location.lower():
            logger.debug(f"Using tracked location: '{env.
_current_location}'")
            return True

        # Check if location is mentioned in current observation after
"You are in"
        if "you are in" in test_obs.lower() and location_name_lower in
 test_obs.lower():
            logger.debug(f"Location '{location_name_lower}' mentioned
in observation after 'You are in'")
            return True

        # Check if the location is in the first line of the
observation (common format)
        first_line = test_obs.split('\n')[0].lower()
        if location_name_lower in first_line:
            logger.debug(f"Location '{location_name_lower}' found in
first line of observation")
            return True

        # Check if the observation mentions items at/on the location
        location_patterns = [
            f"on the {location_name_lower}",
            f"in the {location_name_lower}",
            f"at the {location_name_lower}"
        ]

        for pattern in location_patterns:
            if pattern in test_obs.lower():
                logger.debug(f"Found pattern '{pattern}' in
observation")
                return True

        logger.debug(f"Not at location '{location_name_lower}'")
        return False

    # Handle go to action
    if action_item['action'] == 'go to':
        receptacle = action_item['receptacle']
        receptacle_lower = receptacle.lower()

        # Check if we're already at this location
        if is_at_location(receptacle_lower):
            env._current_location = receptacle
            return f"You are already at the {receptacle}. You can
interact with it directly.", False, False

        # Execute the go to action
        obs, reward, done, info = env.step([agent_action])
        obs, reward, done = obs[0], info['won'][0], done[0]

        # Update the current location if the action was successful
        if obs and "nothing happens" not in obs.lower():
            env._current_location = receptacle

            # If the observation doesn't clearly indicate arrival,
enhance it
            if not any(phrase in obs.lower() for phrase in [f"you
arrive at", f"you are at", f"you see {receptacle_lower}"]):
                obs = f"You arrive at the {receptacle}. {obs}"
        else:
            # Provide more informative feedback
            obs = f"Cannot go to {receptacle}. It might not be a valid
 location or not accessible from here."

        return obs, reward, done

    # Handle examine, open, close, take from, move to actions that
require being at location
    if action_item['action'] in ['examine', 'open', 'close', 'take
from', 'move to']:
        receptacle = action_item['receptacle'].lower() if action_item[
'receptacle'] else ""
        logger.debug(f"Action: {action_item['action']} with receptacle
: {receptacle}")

        # Skip location check for examining objects in inventory
        if action_item['action'] == 'examine' and action_item['object'
] and is_in_inventory(action_item['object']):
            # Execute the examine action directly
            obs, reward, done, info = env.step([agent_action])
            obs, reward, done = obs[0], info['won'][0], done[0]
            return obs, reward, done

        # Check if we need to be at a receptacle and if we're there
        if receptacle and not is_at_location(receptacle):
            action_name = action_item['action']
            if action_name == 'examine':
                return f"You must go to the {action_item['receptacle
']} first before examining it.", False, False
            elif action_name == 'take from':
                return f"You need to go to the {action_item['
receptacle']} first before taking objects from it.", False, False
            elif action_name == 'move to':
                return f"You need to go to the {action_item['
receptacle']} first before placing objects on/in it.", False,
False
            else:  # open or close
                return f"You need to go to the {action_item['
receptacle']} first before you can {action_name} it.", False,
False

    # Handle open and close actions to track container states
    if action_item['action'] in ['open', 'close']:
        receptacle = action_item['receptacle']

        # Execute the action
        obs, reward, done, info = env.step([agent_action])
        obs, reward, done = obs[0], info['won'][0], done[0]

        # Check for "Nothing happens" and provide more informative
feedback
        if obs.strip() == "Nothing happens.":
            if action_item['action'] == 'open':
                return f"Unable to open {receptacle}. It might already
 be open or not be openable.", reward, done
            else:  # close
                return f"Unable to close {receptacle}. It might
already be closed or not be closable.", reward, done

        # Update container state tracking
        if "successfully" in obs.lower() or "already" in obs.lower():
            env._container_states[receptacle.lower()] = 'open' if
action_item['action'] == 'open' else 'closed'

        return obs, reward, done

    # Check if taking an object that's already in inventory
    if action_item['action'] == 'take from':
        object_name = action_item['object']
        if is_in_inventory(object_name):
            return f"You already have the {object_name} in your
inventory. No need to take it again.", False, False

    # Check if moving an object that's not in inventory
    if action_item['action'] == 'move to':
        object_name = action_item['object']
        if not is_in_inventory(object_name):
            return f"You don't have the {object_name} in your
inventory. You need to take it first.", False, False

    # Execute the action in the environment
    logger.debug(f"Executing action in environment: {agent_action}")
    obs, reward, done, info = env.step([agent_action])
    obs, reward, done = obs[0], info['won'][0], done[0]
    logger.debug(f"Environment response: {obs}")
 # Handle special case for "Nothing happens" response
 if obs.strip() == "Nothing happens." and action_item['action'] ==
'take from':
     object_name = action_item['object']
     receptacle_name = action_item['receptacle']

     # Check if it might be because the object is already in
inventory
     if is_in_inventory(object_name):
         return f"You already have the {object_name} in your
inventory. No need to take it again.", reward, done

     # Check if it might be because the container is closed
     receptacle_state = env._container_states.get(receptacle_name.
lower())
     if receptacle_state == 'closed':
         return f"You need to open the {receptacle_name} first
before taking objects from it.", reward, done

     # Otherwise, the object might not be there
     return f"There is no {object_name} at the {receptacle_name} to
 take. It might be elsewhere or already taken.", reward, done

 # Handle special case for "Nothing happens" response for move
action
 if obs.strip() == "Nothing happens." and action_item['action'] ==
'move to':
     object_name = action_item['object']
     receptacle_name = action_item['receptacle']

     # Double-check if the object is in inventory
     if is_in_inventory(object_name):
         # If object is in inventory but move fails, check if
receptacle is closed
         receptacle_state = env._container_states.get(
receptacle_name.lower())
         if receptacle_state == 'closed':
             return f"You need to open the {receptacle_name} first
before placing objects in it.", reward, done
         else:
             return f"Unable to move {object_name} to {
receptacle_name}. Make sure the receptacle is open if it's a
container.", reward, done
     else:
         # If object is not in inventory, provide clear feedback
         return f"You don't have the {object_name} in your
inventory. You need to take it first before moving it.", reward,
done

 # Handle other "Nothing happens" cases with more informative
feedback
 if obs.strip() == "Nothing happens.":
     if action_item['action'] == 'open':
         return f"Unable to open {action_item['receptacle']}. It
might already be open or not be openable.", reward, done
     elif action_item['action'] == 'close':
         return f"Unable to close {action_item['receptacle']}. It
might already be closed or not be closable.", reward, done
     elif action_item['action'] == 'examine':
         if action_item['object']:
             return f"Unable to examine {action_item['object']}.
Make sure you have it in your inventory or it's visible at your
location.", reward, done
         else:
                return f"Unable to examine {action_item['receptacle
']}. Make sure you're at the right location and it's visible.",
reward, done
        elif action_item['action'] == 'use':
            return f"Unable to use {action_item['object']}. Make sure
you have it in your inventory or it's at your current location and
 usable.", reward, done
        elif action_item['action'] in ['heat', 'clean', 'cool', 'slice
']:
            return f"Unable to {action_item['action']} {action_item['
object']}. Make sure you have all required objects and are at the
right location.", reward, done
        elif action_item['action'] == 'go to':
            # This case should be handled earlier, but as a fallback
            return f"Cannot go to {action_item['receptacle']}. It
might not be a valid location in this environment.", reward, done
        else:
            # Generic clarification for other actions
            return f"Action '{agent_action}' resulted in no effect.
Check if you have all prerequisites or if the action is valid in
this context.", reward, done

    # For successful move actions, verify the object was actually in
inventory
    if "successfully" in obs.lower() and "place" in obs.lower() and
action_item['action'] == 'move to':
        object_name = action_item['object']
        # If the environment says the move was successful, we should
trust that and not override
        return obs, reward, done

    return obs, reward, done
```

ALIGN-generated interface with Vanilla agent in ScienceWorld:

```
# Imports
import re
import logging
import io
import sys # Keep for potential future use, though stream preferred

# Assuming logger is configured elsewhere
# log_stream = io.StringIO() # Example for testing capture
# logger = logging.getLogger("AgentEnvInteraction") # ... setup ...

# --- InferRules function (Refined based on Analysis 12) ---
def InferRules(init_obs: str, task: str) -> str:
    """
    Generates environment rules based on the initial observation and
task description.
    Includes rules for 'focus on', container interactions, common
syntax issues, and movement.
    """
    rules = []
    rules.append("General Environment Rules:")
    rules.append("- Only one action can be performed per turn.")
    rules.append("- Actions must be chosen from the available action
space provided in the system prompt.")
    rules.append("- Ensure objects exist and are accessible before
interacting with them (e.g., check 'look around', 'look in
CONTAINER', check your current location).")
    rules.append("- If the environment presents multiple objects with
the same name (ambiguity), it will ask you to clarify by choosing
a number (e.g., 'Which X do you mean? 0: X 1: X'). Respond with
ONLY the number (e.g., '0') to select the corresponding item.")
```

```
 # --- Container Interaction Rules (Existing - Analysis 9) ---
 rules.append("\nInteracting with Containers:")
 rules.append("- The action 'take OBJ from CONTAINER' is generally
not valid.")
 rules.append("- To get an item from a container (like a jar, box,
freezer):")
 rules.append("  1. You often need to 'pick up CONTAINER' first to
hold it.")
 rules.append("  2. Then, you might need to 'move OBJ to inventory'
 or 'put down OBJ' somewhere else.")
 rules.append("- The action 'pick up OBJ from CONTAINER' might also
 not work for all containers. If it fails, try picking up the
container itself.")
 rules.append("- Use 'look in CONTAINER' to see contents.")

 # --- Focus Rules (Existing + Refinement for Analysis 8, 12) ---
 # Use refined regex to find required focus objects based on "focus
 on the ..." pattern
 required_focus_objects_raw = re.findall(r"focus on the (.*?)(?:
you created|\.|$)", task, re.IGNORECASE)
 required_focus_objects = [obj.strip() for obj in
required_focus_objects_raw if obj.strip()]

 if required_focus_objects:
     rules.append("\nTask-Specific Rules for 'focus on':")
     required_objects_str = " or ".join([f"'{obj}'" for obj in
required_focus_objects])
     rules.append(f"- The 'focus on OBJ' action has a special
meaning in this task and is used to signal progress or completion.
")
     rules.append(f"- Use 'focus on' ONLY for the required task
items related to: {required_objects_str}.")
     # Refinement for Analysis 12: Add note about conceptual focus
     rules.append(f"- Sometimes, the task might ask you to focus on
 an item with a specific property (e.g., 'the animal with the
longest lifespan'). In such cases, you might need to identify the
specific item (e.g., 'crocodile') and use 'focus on [specific item
 name]' to fulfill the requirement.")
     rules.append(f"- You must use this command on the specified
items when they are ready (e.g., created, planted, in the correct
location), as per the task instructions.")
     rules.append(f"- Using 'focus on' for any other object (e.g.,
'focus on blast furnace', 'focus on beaker') is considered an
incorrect action for this task and will not advance your progress.
 You will receive feedback if you attempt this.")
     rules.append(f"- If you need to disambiguate one of the
required focus items (e.g., 'focus on {required_focus_objects
[0]}'), respond with the number only, not 'focus on {
required_focus_objects[0]} [number]'.")
     rules.append(f"- If 'focus on' fails for a required item,
ensure it exists, is ready (prerequisites met), you are in the
correct location, and you are using the exact correct name (
sometimes this might be 'OBJ in RECEPTACLE').") # Refinement for
Analysis 8 & 5
 else:
      # General note about 'focus on' if not specifically required
by the "focus on the..." pattern
      rules.append("\nNote on 'focus on OBJ':")
      rules.append("- The 'focus on OBJ' action typically signals
intent towards a task-critical object.")
      rules.append("- Its specific function and validity may vary
depending on the task. Check the task description for any specific
 instructions regarding 'focus on'.")
      # Refinement for Analysis 12: Add note about conceptual focus
```

```
        rules.append(f"- Sometimes, the task might ask you to focus
    on an item with a specific property. In such cases, you might need
     to identify the specific item and use 'focus on [specific item
    name]' to fulfill the requirement.")
        rules.append("- This action might require the object to be in
     a specific state (e.g., created, planted) or require a specific
    name format (e.g., 'OBJ in RECEPTACLE').") # Refinement for
    Analysis 8
        rules.append("- If 'focus on' fails, check if the object
    exists, if prerequisites are met, if you are in the correct
    location, and if you are using the correct name.") # Refinement
    for Analysis 8 & 5

    # --- Movement Rules (Existing - Analysis 10, 11) ---
    rules.append("\nMovement Rules:")
    rules.append("- Use 'go to LOC' to move between locations (e.g., '
    go to kitchen').")
    rules.append("- You can only 'go to' locations that are directly
    connected to your current location.")
    rules.append("- If 'go to LOC' fails, it might be because the
    location is not directly connected or you are already there. Use '
    look around' to see available exits and connected locations.") #
    Added note about being already there for Analysis 13
    rules.append("- The syntax 'go to DESTINATION from SOURCE' is not
    valid. Use only 'go to DESTINATION'.") # Added based on Analysis
    11 feedback logic
    rules.append("- The 'teleport to LOC' action allows direct travel
    but might not be available in all tasks.")

    # --- General Syntax Notes (Existing - Analysis 6 & 7) ---
    rules.append("\nGeneral Action Syntax Notes:")
    rules.append("- For actions like 'open OBJ', use the object's base
     name (e.g., 'open freezer', not 'open freezer door').")
    rules.append("- The 'wait' command only accepts 'wait1' (no space)
     to pass a single time step. Other durations (e.g., 'wait 10') are
     not supported.")

    return "\n".join(rules)


# --- _get_current_state helper function (Unchanged) ---
def _get_current_state(env, logger, previous_score=0.0):
    """Helper function to get current observation and score using '
    look around'."""
    try:
        # Use 'look around' as it's less likely to change game state
    significantly
        # than repeating the failed action or doing nothing.
        current_obs, _, _, current_info = env.step("look around")
        current_score = current_info["score"]
        logger.debug(f"Performed 'look around' to get current state.
    Score: {current_score}")
        # Extract location using the updated regex
        location_match = re.search(r"(?:You are in|This room is called
    ) the (.*?)\.", current_obs) # MODIFIED REGEX
        current_location = f"the {location_match.group(1)}" if
    location_match else "your current location"
        return current_obs, current_score, current_location
    except Exception as e:
        logger.error(f"Error performing 'look around' to get current
    state: {e}")
        try:
            # Fallback if 'look around' fails (less likely but
    possible)
            current_obs = env.look()
            current_score = previous_score # Assume score didn't
    change if look around failed
            logger.warning(f"Fallback to env.look(). Score assumed {
    current_score}")
        except AttributeError:
            logger.error("env.look() not available as fallback.")
            current_obs = "Error: Could not retrieve current
    environment state."
            current_score = previous_score # Keep score from before
    the failed action
        # Try to extract location from fallback obs if possible using
    the updated regex
        location_match = re.search(r"(?:You are in|This room is called
    ) the (.*?)\.", current_obs) # MODIFIED REGEX
        current_location = f"the {location_match.group(1)}" if
    location_match else "your current location"
        return current_obs, current_score, current_location

# --- WrapStep function (Refined based on Analysis 12, 13) ---
def WrapStep(env, init_obs: str, task: str, agent_action: str, logger:
     logging.Logger):
    """
    Processes the agent's action, providing specific feedback for
    incorrect 'focus on' usage
    (including conceptual vs specific targets), invalid 'take from', '
    pick up from', 'open door',
    'wait' syntax/usage, and invalid 'go to' attempts (including
    syntax errors, non-adjacency,
    and attempting to go to the current location), without causing
    immediate failure where possible.
    Uses substring matching for target identification and checks
    observation strings/exceptions
    for environment-reported errors.

    Args:
        env: The environment instance.
        init_obs: The initial observation string.
        task: The task description string.
        agent_action: The action string from the agent.
        logger: The logger instance.

    Returns:
        A tuple containing:
        - obs (str): The observation string after the action (or
    feedback).
        - done (bool): Whether the task is done.
        - score (float): The score after the action.
    """
    logger.debug(f"Processing agent action: {agent_action}")
    action_normalized = agent_action.lower().strip()
    current_score = 0.0 # Placeholder, will be updated

    # --- Get initial score before attempting action (needed for
    fallback in _get_current_state) ---
    # (Placeholder - score is retrieved reliably after action attempt
    or failure)
    pass

    # --- Check for invalid 'wait' command syntax (Analysis 7) ---
    wait_match = re.match(r"wait\s*(\d+)", action_normalized)
    if wait_match and action_normalized != "wait1":
        wait_duration = wait_match.group(1)
        logger.warning(f"Intercepted invalid wait command: '{
    agent_action}'. Only 'wait1' is supported.")
```

```
        current_obs, current_score, current_location =
_get_current_state(env, logger)
        custom_feedback = (
            f"\n\n[Environment Feedback]: Your action '{agent_action}'
 uses an invalid format or duration.\n"
            f"Reason: The environment only supports waiting for a
single time step using the command 'wait1' (no space between 'wait
' and '1'). Waiting for {wait_duration} steps is not supported.\n"
            f"Your action was not executed. Please use 'wait1' if you
intend to wait."
        )
        final_obs = current_obs + custom_feedback
        return final_obs, False, current_score

 # --- Check for invalid 'take ... from ...' syntax (Analysis 9)
---
 take_from_match = re.match(r"take\s+(.*)\s+from\s+(.*)",
action_normalized)
 if take_from_match:
        taken_object = take_from_match.group(1).strip()
        container = take_from_match.group(2).strip()
        logger.warning(f"Intercepted invalid action syntax: '{
agent_action}'. 'take ... from ...' is not supported.")
        current_obs, current_score, current_location =
_get_current_state(env, logger)
        custom_feedback = (
            f"\n\n[Environment Feedback]: Your action '{agent_action}'
 uses invalid syntax.\n"
            f"Reason: The action 'take {taken_object} from {container
}' is not supported in this environment.\n"
            f"To get items from containers like '{container}', you
usually need to 'pick up {container}' first, or check if you can '
move {taken_object} to inventory'.\n"
            f"Your action was not executed."
        )
        final_obs = current_obs + custom_feedback
        return final_obs, False, current_score

 # --- Check 'focus on' action (Analysis 1, 2, 4, 5, 8, 12) ---
 focus_match = re.match(r"focus on (.*)", action_normalized)
 if focus_match:
        focused_object_raw = focus_match.group(1).strip()
        normalized_focused_object = focused_object_raw.lower()

        # Get required focus objects based on "focus on the ..."
pattern in the task
        required_focus_objects_raw = re.findall(r"focus on the (.*?)
(?: you created|\.|$)", task, re.IGNORECASE)
        normalized_required_objects = [obj.strip().lower() for obj in
required_focus_objects_raw if obj.strip()]
        required_objects_str_display = " or ".join([f"'{obj.strip()}'"
 for obj in required_focus_objects_raw if obj.strip()])

        # --- Check for Ambiguity Resolution Syntax Error FIRST (
Analysis 2) ---
        # Matches 'focus on base_object number'
        ambiguity_match = re.match(r"^(.*)\s+(\d+)$",
focused_object_raw)
        if ambiguity_match:
            base_object = ambiguity_match.group(1).strip()
            number_str = ambiguity_match.group(2)
            normalized_base_object = base_object.lower()

            # Check if the base object is related to *any* required
focus object
```

```
            is_required_base_object = False
            if normalized_required_objects:
                 for req_obj in normalized_required_objects:
                     # Use substring matching for robustness
                     if req_obj in normalized_base_object or
normalized_base_object in req_obj:
                         is_required_base_object = True
                         logger.debug(f"Ambiguity syntax check: Base
object '{normalized_base_object}' potentially matches required '{
req_obj}'.")
                         break
            # Also consider if the task generally mentions focusing on
 this object type, even if not in "focus on the..."
            elif normalized_base_object in task.lower():
                 is_required_base_object = True # Assume relevant if
mentioned in task and ambiguity arises
                 logger.debug(f"Ambiguity syntax check: Base object '{
normalized_base_object}' appears in task description.")


            if is_required_base_object:
                logger.warning(f"Intercepted incorrect ambiguity
resolution syntax: '{agent_action}'. Agent should use only the
number '{number_str}'.")
                current_obs, current_score, current_location =
_get_current_state(env, logger)
                custom_feedback = (
                    f"\n\n[Environment Feedback]: Your action '{
agent_action}' uses an invalid format for selecting an option.\n"
                    f"Reason: It seems you are trying to select option
 {number_str} for '{base_object}'. To select this option, please
respond with just the number: '{number_str}'.\n"
                    f"Your action was not executed."
                )
                final_obs = current_obs + custom_feedback
                return final_obs, False, current_score
        # --- End Ambiguity Syntax Check ---

        # --- Determine if the focus target is potentially correct (
Analysis 1, 12) ---
        is_potentially_correct_target = False
        matched_req_obj = None
        is_conceptual_focus_task = False # Flag for Analysis 12

        if normalized_required_objects:
            # Check if any required object looks conceptual (Analysis
12)
            conceptual_keywords = ["with the", "longest", "shortest",
"heaviest", "lightest", "smallest", "largest"]
            for req_obj_raw in required_focus_objects_raw:
                if any(keyword in req_obj_raw.lower() for keyword in
conceptual_keywords):
                    is_conceptual_focus_task = True
                    logger.debug(f"Detected conceptual focus task
based on required object: '{req_obj_raw}'")
                    break

            # Check if agent's target matches any required object
            for req_obj in normalized_required_objects:
                # Use substring matching: is required obj part of
agent target, or agent target part of required obj?
                if req_obj in normalized_focused_object or (
normalized_focused_object and normalized_focused_object in req_obj
):
                    is_potentially_correct_target = True
                    matched_req_obj = req_obj
                    logger.debug(f"Focus target potentially matches
required '{req_obj}': Agent specified '{normalized_focused_object
}'.")
                    break

        # Analysis 12 Relaxation: If it's a conceptual task, don't
 block focusing on a specific instance yet.
        # Allow it to proceed to env.step, even if it didn't
literally match the conceptual phrase.
        if is_conceptual_focus_task and not
is_potentially_correct_target:
            logger.info(f"Conceptual focus task detected. Allowing
 action '{agent_action}' targeting specific instance '{
focused_object_raw}' to proceed, bypassing literal match check
against '{required_objects_str_display}'.")
            is_potentially_correct_target = True # Override: Let
the environment check the instance

        # Intercept BEFORE execution ONLY IF:
        # 1. There are specific required objects AND
        # 2. It's NOT a conceptual focus task where the agent
might be trying a specific instance AND
        # 3. The agent's target did not match any required object.
        if not is_potentially_correct_target: # This now correctly
 handles the conceptual case due to the override above
            logger.warning(f"Intercepted incorrect focus target
action on '{focused_object_raw}'. Task requires focus on items
related to: {required_objects_str_display}. Providing feedback.")
            current_obs, current_score, current_location =
_get_current_state(env, logger)
            custom_feedback = (
                f"\n\n[Environment Feedback]: Your action '{
agent_action}' was not executed as intended.\n"
                f"Reason: The 'focus on' action has a specific
purpose in this task. It should only be used for items related to:
 {required_objects_str_display}.\n"
                f"Using 'focus on {focused_object_raw}' is not the
 correct procedure here. Please choose another action or use '
focus on' with the correct item when it is ready."
            )
            # Add hint for conceptual tasks if applicable (even if
 interception happens for other reasons)
            if is_conceptual_focus_task:
                 custom_feedback += f"\nNote: For tasks requiring
focus based on a property (like '{required_objects_str_display}'),
 you usually need to identify the specific item that has that
property and focus on its name."

            final_obs = current_obs + custom_feedback
            return final_obs, False, current_score
    else:
        # Task does not have "focus on the ..." requirement.
Assume potentially correct.
        is_potentially_correct_target = True
        logger.debug(f"Proceeding with focus action '{agent_action
}'. No specific required objects.")


    # --- Try executing the focus action ---
    if is_potentially_correct_target:
        try:
            obs, _, done, info = env.step(agent_action)
            score = info["score"]
```

```
                logger.debug(f"Executed '{agent_action}'. Obs received
: '{obs[:100]}...', Done: {done}, Score: {score}")

                # Check observation for failure messages
                error_detected_in_obs = False
                error_phrases = ["no known action", "unknown action",
"could not find object", "object not found", "is not here", "
nothing happens", "don't know how to"]
                obs_lower = obs.lower()
                failure_phrase_found = None
                for phrase in error_phrases:
                    # Avoid matching harmless phrases like "You are
not focusing on anything"
                    if phrase in obs_lower and not obs_lower.
startswith("you are not"):
                        error_detected_in_obs = True
                        failure_phrase_found = phrase
                        logger.warning(f"Detected potential error
phrase '{phrase}' in observation string for focus action '{
agent_action}'.")
                        break

                if error_detected_in_obs:
                    # Focus action failed based on observation content
                    logger.warning(f"Handling focus failure based on
observation content for target '{focused_object_raw}'. Failure
phrase: '{failure_phrase_found}'.")
                    current_obs, current_score, current_location =
_get_current_state(env, logger, score) # Pass score from failed
step

                    # --- Provide Enhanced Feedback (Analysis 4, 5, 8)
 ---
                    feedback_parts = [
                        f"\n\n[Environment Feedback]: Your action '{
agent_action}' did not succeed (Observation: \"{obs.strip()}\")."
                    ]
                    reasons = []
                    # Reason 1: Existence/Name/Readiness (Analysis 4,
8)
                    object_in_task = focused_object_raw.lower() in
task.lower()
                    reason_existence = f"The object '{
focused_object_raw}' might not exist yet, might not be ready (e.g
., needs planting, mixing), or you might need to use its exact
name."
                    if object_in_task:
                         reason_existence += " Check the task steps
and ensure all prerequisites are met."
                    # Suggest specific naming if applicable (Analysis
8)
                    if "seed" in focused_object_raw.lower() and "plant
" in task.lower():
                         reason_existence += " For planted items, the
name might be like 'orange seed in flower pot'."
                    elif matched_req_obj: # If it matched a "focus on
the..." object originally
                         original_matched_req_obj_display = f"'{
matched_req_obj}'" # Default to normalized
                         for raw_obj in required_focus_objects_raw:
                             if raw_obj.strip().lower() ==
matched_req_obj:
                                 original_matched_req_obj_display = f"
'{raw_obj.strip()}'" # Use original casing if found
                                 break
```

```
                          reason_existence += f" Ensure you are using
the correct name, perhaps '{original_matched_req_obj_display}' if
that is the expected item."
                     # Add hint for conceptual tasks (Analysis 12)
                     elif is_conceptual_focus_task:
                          reason_existence += f" For tasks requiring
focus based on a property (like '{required_objects_str_display}'),
 ensure you have identified the correct specific item that has
that property and are using its exact name."


                     reasons.append(reason_existence)

                     # Reason 2: Location (Analysis 5)
                     # Basic location check - more sophisticated checks
 might need external knowledge
                     if failure_phrase_found in ["could not find object
", "object not found", "is not here"]:
                          reasons.append(f"The object might exist but
not be accessible or interactable from your current location ({
current_location}).")
                          # Example Task-Specific Location Hint (can be
 generalized if needed)
                          if "greenhouse" in task.lower() and ("red box
" in focused_object_raw or "green box" in focused_object_raw) and
"greenhouse" not in current_location:
                               reasons.append("Remember, the red and
green boxes are expected to be in the greenhouse.")

                     feedback_parts.append("Possible Reasons:")
                     for i, r in enumerate(reasons):
                         feedback_parts.append(f"- {r}")
                     feedback_parts.append("Suggestion: Please check
the environment state, your location, ensure the object is ready,
and verify you are using the correct name and syntax.")

                     custom_feedback = "\n".join(feedback_parts)
                     final_obs = current_obs + custom_feedback
                     return final_obs, False, current_score # Return
feedback, keep task running

                else:
                     # Focus action seemed successful based on
observation
                     if done and score < 0: # Check for unexpected
failure on success (Analysis 1 edge case)
                         logger.warning(f"Focus action '{agent_action}'
 resulted in task completion with score {score}. Prerequisites
might have been missed.")
                         obs += (
                              f"\n\n[Environment Note]: The task
finished after focusing on '{focused_object_raw}', but the score
({score}) indicates potential issues. "
                              f"Ensure all necessary steps and
conditions were met before using the 'focus on' command."
                         )
                     return obs, done, score # Return original results

            except Exception as e:
                 # Focus action failed with an exception
                 logger.error(f"Exception occurred executing focus
action '{agent_action}': {e}")
                 error_msg_str = str(e)
                 current_obs, current_score, current_location =
_get_current_state(env, logger) # Get state after exception
```

```
                # --- Provide Enhanced Feedback (Analysis 4, 5, 8)
---
                feedback_parts = [
                    f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed with an error: \"{error_msg_str}\"."
                ]
                reasons = []
                error_phrases_exception = ["no known action", "
unknown action", "could not find object", "object not found", "is
not here"]
                exception_indicates_issue = any(phrase in
error_msg_str.lower() for phrase in error_phrases_exception)

                if exception_indicates_issue:
                    # Reason 1: Existence/Name/Readiness (Analysis 4,
 8)
                    object_in_task = focused_object_raw.lower() in
task.lower()
                    reason_existence = f"The object '{
focused_object_raw}' might not exist yet, might not be ready (e.g
., needs planting, mixing), or you might need to use its exact
name."
                    if object_in_task:
                        reason_existence += " Check the task steps
and ensure all prerequisites are met."
                    # Suggest specific naming if applicable (Analysis
 8)
                    if "seed" in focused_object_raw.lower() and "
plant" in task.lower():
                        reason_existence += " For planted items, the
 name might be like 'orange seed in flower pot'."
                    elif matched_req_obj: # If it matched a "focus on
 the..." object originally
                        original_matched_req_obj_display = f"'{
matched_req_obj}'" # Default to normalized
                        for raw_obj in required_focus_objects_raw:
                            if raw_obj.strip().lower() ==
matched_req_obj:
                                original_matched_req_obj_display = f
"'{raw_obj.strip()}'" # Use original casing if found
                                break
                        reason_existence += f" Ensure you are using
the correct name, perhaps '{original_matched_req_obj_display}' if
that is the expected item."
                    # Add hint for conceptual tasks (Analysis 12)
                    elif is_conceptual_focus_task:
                        reason_existence += f" For tasks requiring
focus based on a property (like '{required_objects_str_display}'),
 ensure you have identified the correct specific item that has
that property and are using its exact name."

                    reasons.append(reason_existence)

                    # Reason 2: Location (Analysis 5)
                    reasons.append(f"The object might exist but not
be accessible or interactable from your current location ({
current_location}).")
                    # Example Task-Specific Location Hint
                    if "greenhouse" in task.lower() and ("red box" in
 focused_object_raw or "green box" in focused_object_raw) and "
greenhouse" not in current_location:
                        reasons.append("Remember, the red and green
boxes are expected to be in the greenhouse.")
```

```
             else: # General error
                 reasons.append(f"An unexpected error occurred: {
error_msg_str}")

             feedback_parts.append("Possible Reasons:")
             for i, r in enumerate(reasons):
                 feedback_parts.append(f"- {r}")
             feedback_parts.append("Suggestion: Please check the
environment state, your location, ensure the object is ready, and
verify you are using the correct name and syntax.")

             custom_feedback = "\n".join(feedback_parts)
             final_obs = current_obs + custom_feedback
             return final_obs, False, current_score # Return
feedback, keep task running
    # --- End of 'focus on' specific logic ---

else:
    # --- Handle standard actions (including checks for 'open ...
door', 'pick up ... from ...', 'go to ...') ---

    # --- Check for invalid 'go to ... from ...' syntax FIRST (
Analysis 11) ---
    go_to_from_match = re.match(r"go to\s+(.+)\s+from\s+(.+)",
action_normalized, re.IGNORECASE)
    if go_to_from_match:
        destination = go_to_from_match.group(1).strip()
        source = go_to_from_match.group(2).strip()
        logger.warning(f"Intercepted invalid 'go to ... from ...'
syntax: '{agent_action}'.")
        current_obs, current_score, current_location =
_get_current_state(env, logger)
        custom_feedback = (
            f"\n\n[Environment Feedback]: Your action '{
agent_action}' uses an invalid command format.\n"
            f"Reason: The 'go to' action only accepts the
destination location name (e.g., 'go to {destination}' or 'go to
hallway'). Specifying the source location using 'from {source}' is
 not supported.\n"
            f"Suggestion: Please use 'look around' to see valid
exits from your current location ({current_location}) and then use
 'go to [valid exit]'."
        )
        final_obs = current_obs + custom_feedback
        return final_obs, False, current_score
    # --- End 'go to ... from ...' check ---

    # --- If not the invalid 'go to from' syntax, proceed with
standard execution ---
    logger.debug(f"Executing standard action: {agent_action}")
    try:
        obs, _, done, info = env.step(agent_action)
        score = info["score"]
        logger.debug(f"Executed '{agent_action}'. Obs: '{obs
[:100]}...', Done: {done}, Score: {score}")

        # --- Check for specific failure cases based on
observation AFTER successful execution ---

        # Check for 'go to current location' failure (Analysis 13)
        go_to_match = re.match(r"go to (.*)", action_normalized)
        # Use the specific, potentially ambiguous feedback string
as the trigger
        go_to_current_loc_feedback = "It's not clear how to get
there from here."
```

```
        if go_to_match and go_to_current_loc_feedback in obs:
            target_location = go_to_match.group(1).strip()
            logger.warning(f"Detected ambiguous feedback '{
go_to_current_loc_feedback}' after 'go to {target_location}'.
Assuming agent tried to go to current location.")
            # Get current state to ensure obs is fresh before
adding feedback
            current_obs, current_score, current_location =
_get_current_state(env, logger, score) # Pass score from failed
step
            custom_feedback = (
                f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed.\n"
                f"Reason: You cannot use 'go to {target_location}'
 because you are already in that location ({current_location}).\n"
                f"Suggestion: Use 'look around' to see available
exits to other locations."
            )
            final_obs = current_obs + custom_feedback
            # Since the original step technically executed (but
resulted in this feedback),
            # we return done=False and the score from that step.
            return final_obs, False, current_score

        # Check observation for other general failure messages
        error_detected_in_obs = False
        # Added more potential failure phrases, especially for
movement
        error_phrases = ["no known action", "unknown action", "
could not find object", "object not found", "is not here", "
nothing happens", "cannot", "can't go that way", "not a valid exit
", "don't know how to go there"]
        obs_lower = obs.lower()
        failure_phrase_found = None
        for phrase in error_phrases:
             # Avoid matching harmless phrases like "You cannot
see that" if it's just descriptive
             # Also avoid matching the specific 'go to current loc
' feedback handled above
             if phrase in obs_lower and not obs_lower.startswith("
you are carrying") and not obs_lower.startswith("you are in") and
go_to_current_loc_feedback not in obs:
                 error_detected_in_obs = True
                 failure_phrase_found = phrase
                 logger.warning(f"Detected potential failure
phrase '{phrase}' in observation string for standard action '{
agent_action}'.")
                 break

        if error_detected_in_obs:
             # Action failed based on observation content. Provide
 specific feedback.
             logger.warning(f"Handling failure based on
observation content for standard action '{agent_action}'. Failure
phrase: '{failure_phrase_found}'.")
             current_obs, current_score, current_location =
_get_current_state(env, logger, score) # Pass score from failed
step

             custom_feedback = None
             # Check for 'go to LOC' failure due to non-adjacency
(Analysis 10)
             # This check should only trigger for the valid 'go to
 LOC' syntax,
                # as the invalid 'go to ... from ...' syntax is
caught above.
                # Also ensure it's not the 'go to current loc' case
handled above.
                if go_to_match and failure_phrase_found in ["no known
 action", "unknown action", "cannot", "can't go that way", "not a
valid exit", "don't know how to go there"]:
                    target_location = go_to_match.group(1).strip()
                    logger.info(f"Detected failed 'go to {
target_location}' action, likely due to non-adjacency from {
current_location}.")
                    custom_feedback = (
                        f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed (Observation: \"{obs.strip()}\").\n"
                        f"Reason: You cannot go directly to '{
target_location}' from your current location ({current_location}).
 Movement is only possible between directly connected locations.\n
"
                        f"Suggestion: Use 'look around' to see the
available exits and connected locations from here."
                    )

                # Check for 'open ... door' syntax error (Analysis 6)
                open_door_match = re.match(r"open (.*) door",
action_normalized)
                if not custom_feedback and open_door_match and
failure_phrase_found in ["no known action", "unknown action", "
could not find object", "object not found", "cannot"]:
                    target_object = open_door_match.group(1).strip()
                    logger.info(f"Detected failed 'open ... door'
syntax for '{agent_action}'. Suggesting 'open {target_object}'.")
                    custom_feedback = (
                        f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed (Observation: \"{obs.strip()}\").\n"
                        f"Reason: The syntax might be incorrect. To
open objects like '{target_object}', try using the command 'open {
target_object}' instead of specifying 'door'.\n"
                        f"Suggestion: Please check the object name
and try the suggested syntax."
                    )

                # Check for 'pick up ... from ...' failure (Analysis
9)
                pickup_from_match = re.match(r"pick up\s+(.*)\s+from\
s+(.*)", action_normalized)
                if not custom_feedback and pickup_from_match and
failure_phrase_found in ["no known action", "unknown action", "
cannot"]:
                    picked_object = pickup_from_match.group(1).strip
()
                    container = pickup_from_match.group(2).strip()
                    logger.info(f"Detected failed 'pick up ... from
...' action for '{agent_action}'. Suggesting 'pick up {container
}'.")
                    custom_feedback = (
                        f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed (Observation: \"{obs.strip()}\").\n"
                        f"Reason: The action 'pick up {picked_object}
 from {container}' might not be supported for this container.\n"
                        f"Suggestion: Try picking up the container
itself first using 'pick up {container}'. You might then be able
to access its contents."
                    )

                # Default failure feedback
                if not custom_feedback:
                    custom_feedback = (
                        f"\n\n[Environment Feedback]: Your action '{
agent_action}' did not succeed as expected in {current_location} (
Observation: \"{obs.strip()}\").\n"
                        f"Reason: This could be due to an incorrect
command, a non-existent or inaccessible object, or the action not
being applicable in the current situation.\n"
                        f"Suggestion: Please check the command syntax
, object names, your location, and the environment state."
                    )

                final_obs = current_obs + custom_feedback
                return final_obs, False, current_score # Return
corrected obs, keep task running

            else:
                # Action executed successfully without known error
phrases in obs
                return obs, done, score # Return original results

    except Exception as e:
            # Standard action failed with an exception
            logger.error(f"Error executing standard action '{
agent_action}': {e}")
            error_msg_str = str(e)
            current_obs, current_score, current_location =
_get_current_state(env, logger) # Get state after exception

            custom_feedback = None
            # Added more potential failure phrases, especially for
movement
            error_phrases_exception = ["no known action", "unknown
action", "could not find object", "object not found", "is not here
", "cannot", "can't go that way", "not a valid exit", "don't know
how to go there"]
            exception_indicates_issue = any(phrase in error_msg_str.
lower() for phrase in error_phrases_exception)

            # Check for 'go to LOC' failure due to non-adjacency based
 on exception (Analysis 10)
            # Ensure it wasn't the 'go to ... from ...' pattern caught
 earlier
            go_to_match = re.match(r"go to (.*)", action_normalized)
            if not go_to_from_match and go_to_match and
exception_indicates_issue:
                target_location = go_to_match.group(1).strip()
                logger.info(f"Detected failed 'go to {target_location
}' action based on exception, likely due to non-adjacency from {
current_location}.")
                custom_feedback = (
                    f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed with an error: \"{error_msg_str}\".\n"
                    f"Reason: You might not be able to go directly to
 '{target_location}' from your current location ({current_location
}). Movement is only possible between directly connected locations
.\n"
                    f"Suggestion: Use 'look around' to see the
available exits and connected locations from here."
                )

            # Check for 'open ... door' syntax error based on
exception (Analysis 6)
            open_door_match = re.match(r"open (.*) door",
action_normalized)
```

```
        if not custom_feedback and open_door_match and
exception_indicates_issue:
            target_object = open_door_match.group(1).strip()
            logger.info(f"Detected failed 'open ... door' syntax
for '{agent_action}' based on exception. Suggesting 'open {
target_object}'.")
            custom_feedback = (
                f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed with an error: \"{error_msg_str}\".\n"
                f"Reason: The syntax might be incorrect. To open
objects like '{target_object}', try using the command 'open {
target_object}' instead of specifying 'door'.\n"
                f"Suggestion: Please check the object name and try
 the suggested syntax."
            )

        # Check for 'pick up ... from ...' failure based on
exception (Analysis 9)
        pickup_from_match = re.match(r"pick up\s+(.*)\s+from\s
+(.*)", action_normalized)
        if not custom_feedback and pickup_from_match and
exception_indicates_issue:
            picked_object = pickup_from_match.group(1).strip()
            container = pickup_from_match.group(2).strip()
            logger.info(f"Detected failed 'pick up ... from ...'
action for '{agent_action}' based on exception. Suggesting 'pick
up {container}'.")
            custom_feedback = (
                f"\n\n[Environment Feedback]: Your action '{
agent_action}' failed with an error: \"{error_msg_str}\".\n"
                f"Reason: The action 'pick up {picked_object} from
 {container}' might not be supported for this container.\n"
                f"Suggestion: Try picking up the container itself
first using 'pick up {container}'. You might then be able to
access its contents."
            )

        # Default error feedback based on exception
        if not custom_feedback:
            custom_feedback = (
                f"\n\n[Environment Feedback]: Error executing
action '{agent_action}' in {current_location}.\n"
                f"Reason: {e}\n"
                f"Suggestion: Please check the command syntax,
object names, your location, and the environment state."
            )

        final_obs = current_obs + custom_feedback
        return final_obs, False, current_score # Return error
message, keep task running
```

ALIGN-generated interface with Vanilla agent in WebShop:

```
import re
import logging

# Assuming logger is configured elsewhere in the main script
# Example configuration:
# import sys
# logger = logging.getLogger('EnvironmentWrapper')
# logger.setLevel(logging.DEBUG)
# handler = logging.StreamHandler(sys.stdout)
# formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)
s - %(message)s')
# handler.setFormatter(formatter)
# logger.addHandler(handler)

def InferRules(init_obs, task):
    """
    Contains the rules for environment and task execute logic.
    Adds specific rules based on analysis to clarify environment
    behavior.
    """
    # Rule added based on Analysis Result 1 (Unchanged from previous
    step)
    buy_rule = """
# Environment Rule Specifics:
- The 'click[Buy]' or 'click[Buy Now]' action can only be successfully
     executed from the main Item page (the page showing product
    options, description button, and the buy button).
- Attempting to buy from other pages, such as the Item Description
    page (reached via 'click[Description]' or 'click[Desc/Overview]'),
     will result in an error. You must navigate back to the main Item
    page first (e.g., using 'click[< Prev]') before buying.
"""
    return buy_rule


def WrapStep(env, init_obs: str, task: str, agent_action: str, logger:
     logging.Logger):
    """
    Process the agent action:
    - Intercepts invalid actions based on known rules (e.g., buying
    from description page).
    - Provides informative feedback for invalid actions.
    - Executes valid actions using env.step.
    - Returns the next observation, reward, and done status.

    Args:
        env: The environment instance.
        init_obs: The observation *before* the agent took the current
    action.
        task: The task description.
        agent_action: The action string provided by the agent.
        logger: Logger object for debugging.

    Returns:
        Tuple[str, float, bool]: obs, reward, done
    """
    obs = ""
    reward = 0.0
    done = False

    # Normalize action for easier checking
    normalized_action = agent_action.strip().lower()

    # Check for the specific misalignment: Trying to buy from the
    description page
    is_buy_action = normalized_action.startswith("click[buy")

    if is_buy_action:
        # Log the full init_obs before performing the state check
        logger.debug(f"Full init_obs before state check for buy action
    : {init_obs}")

        # Infer state from the observation *before* the action (
    init_obs)
        # Refined Heuristic: Check for presence of "prev" (likely in
    '< Prev')
```

```
        # and absence of "buy now" in the lowercased observation
    content.
        lower_init_obs = init_obs.lower()
        # Use core text fragments for flexibility and case-
    insensitivity
        has_prev_indicator = "prev" in lower_init_obs
        has_buy_now_indicator = "buy now" in lower_init_obs

        is_likely_description_page = has_prev_indicator and not
    has_buy_now_indicator
        logger.debug(f"Checking for description page state before buy
    action: has_prev_indicator={has_prev_indicator},
    has_buy_now_indicator={has_buy_now_indicator},
    is_likely_description_page={is_likely_description_page}")


        if is_likely_description_page:
            logger.debug(f"Intercepted invalid action: '{agent_action
    }'. Agent attempted to buy from a description page (based on
    refined check).")
            # Provide specific feedback based on Analysis Result 1
            obs = (
                f"Action '{agent_action}' is invalid in the current
    state (Description page). "
                "You can only buy from the main item page. "
                "Please go back to the item page first, likely by
    using an action like 'click[< Prev]'.\n\n"
                f"Previous Observation:\n{init_obs}" # Return the
    previous observation so the agent knows where it was
            )
            reward = 0.0 # No reward for invalid action
            done = False # Task is not done
            logger.debug(f"Returning custom feedback for invalid buy
    action. Obs: {obs[:100]}..., Reward: {reward}, Done: {done}")
            return obs, reward, done
        else:
            # Buy action attempted, but not detected as being from
    description page (presumably valid)
            logger.debug(f"Executing potentially valid buy action: {
    agent_action} (State check did not indicate description page)")
            obs, reward, done = env.step(agent_action)
            logger.debug(f"Executed env.step for buy action. Obs: {obs
    [:100]}..., Reward: {reward}, Done: {done}")
            return obs, reward, done
     else:
        # Action is not a buy action, execute normally
        logger.debug(f"Executing non-buy action: {agent_action}")
        obs, reward, done = env.step(agent_action)
        logger.debug(f"Executed env.step for non-buy action. Obs: {obs
    [:100]}..., Reward: {reward}, Done: {done}")
        return obs, reward, done
```

ALIGN-generated interface with Vanilla agent in M^{3}ToolEval:

```
import re
import logging
from typing import Any, Tuple # Assuming Task is defined elsewhere,
    added Any for env type hint clarity

# Define task type mapping for clarity if needed elsewhere
TASK_TYPE_MAP = {
    0: 'message_decoder',
    1: 'cryptobotanists_plant_dna_sequencer',
    2: 'trade_calculator',
    3: 'travel_itinerary_planning',
    4: 'web_browsing',
}

# Assume env object has methods like step() and attributes like name,
    instruction
# Assume logger is a configured logging.Logger instance

def InferRules(task_name: str, task_type_idx: int) -> str:
    """
    Contains the rules for environment and task execute logic for
    different task types.
    """
    if task_type_idx == 1: # cryptobotanists_plant_dna_sequencer
        # Add rule based on Analysis Result 4
        return "When providing the final answer for this task, please
    output only the single longest valid DNA sequence found. Do not
    output a list of all valid sequences."
    # Keep the previous logic for other tasks (no specific rules
    defined here previously)
    # Based on the analysis (Results 1, 2, 3), no specific rules
    needed to be defined here for other tasks,
    # as the feedback was handled during action processing.
    return "There is no specific rule for this environment beyond the
    standard tool usage format. Follow instructions carefully."

def WrapStep(env: Any, task_name: str, instruction: str, agent_action:
     str, logger: logging.Logger) -> Tuple[str, float, bool]:
    """
    Process the agent action:
    1. Check for common invocation errors based on Analysis Results 1
    and 2:
        - Using func() instead of func.
        - Using func(arg) instead of func, arg.
    2. If no known format errors are detected, pass the action to the
    environment's step function.
    3. Check for specific scenarios based on Analysis Result 3:
        - If the task involves finding Allison Hill's email and the
    agent provides an incorrect final answer,
          modify the feedback to acknowledge the potential non-
    discoverability.
    4. Check for specific scenarios based on Analysis Result 4:
        - If the task is cryptobotanists_plant_dna_sequencer (
    task_type_idx=1) and the agent provides an incorrect answer
    formatted as a list,
          modify the feedback to clarify that only the single longest
    sequence is required.
    Return the next observation, reward, and done status.
    """
    obs, reward, done = "", 0.0, False
    # Log the task name and type for debugging purposes
    task_type_idx = -1
    for idx, name in TASK_TYPE_MAP.items():
        # A simple heuristic to find task_type_idx based on task_name
    or env type if available
        # This might need refinement depending on how task_type_idx is
     actually determined in the full system
        # Assuming env might have a type attribute or task_name
    implies type
        if name in task_name.lower(): # Basic check, might need
    improvement
            task_type_idx = idx
            break
        # Or if env has a type attribute: if env.type == name:
    task_type_idx = idx; break
```

```
 logger.debug(f"Processing action for task: '{task_name}' (Deduced
Type Index: {task_type_idx})")


 # Combined check for tool_name(...) format based on Analysis
Results 1 & 2
 parenthesis_args_pattern = r"^\s*Action:\s*([a-zA-Z0-9_]+)\((.*)\)
\s*End Action\s*$"
 match = re.match(parenthesis_args_pattern, agent_action)

 if match:
     tool_name = match.group(1)
     args_inside = match.group(2).strip() # Remove leading/trailing
 whitespace from args

     if not args_inside: # Case: tool_name() - Analysis Result 1
         obs = f"Error: Found tool invocation with empty
parentheses '{tool_name}()'. Tool names should be invoked without
parentheses, e.g., 'Action: {tool_name} End Action'."
         reward = 0.0
         done = False
         logger.debug(f"Detected incorrect tool format: {
agent_action} (empty parentheses). Provided specific feedback.")
         return obs, reward, done
     else: # Case: tool_name(arg) or tool_name(arg1, arg2) etc. -
Analysis Result 2
         suggested_format = f"Action: {tool_name}, {args_inside}
End Action"
         obs = f"Error: Found tool invocation with arguments inside
 parentheses like '{tool_name}({args_inside})'. Tool arguments
should be provided after the tool name, separated by a comma, e.g
., '{suggested_format}'."
         reward = 0.0
         done = False
         logger.debug(f"Detected incorrect tool format: {
agent_action} (arguments in parentheses). Provided specific
feedback.")
         return obs, reward, done
 else:
     # If the format doesn't match the specific error patterns,
proceed as normal
     logger.debug(f"Action format '{agent_action}' doesn't match
the tool_name(...) pattern, proceeding with env.step.")
     try:
         # Pass the original agent_action to env.step
         obs, reward, done = env.step(agent_action)
         logger.debug(f"env.step executed successfully for action:
{agent_action}. Obs: {obs}, Reward: {reward}, Done: {done}")

         # --- Add specific handling for Analysis Result 3 ---
         # Refined check: Identify the task by checking for
keywords "allison", "hill", and "email"
         # in the lowercased task name for robustness.
         task_name_lower = task_name.strip().lower()
         is_allison_hill_email_task = (
             "allison" in task_name_lower and
             "hill" in task_name_lower and
             "email" in task_name_lower
         )
         logger.debug(f"Checking for Allison Hill email task: Name
='{task_name}', Lower='{task_name_lower}', Keywords found={
is_allison_hill_email_task}")
```

```
        # Check if it's the target task, the agent submitted an
answer, the answer was wrong (reward=0), and the task is marked as
 done.
        if is_allison_hill_email_task and agent_action.startswith(
"Answer:") and reward == 0.0 and done:
            logger.debug(f"Handling incorrect answer for Allison
Hill email task ({task_name}). Original Obs: {obs}")
            # Modify the observation to be more informative about
potential unsolvability
            original_feedback = obs # Keep the original feedback
from env.step
            # Append a note about potential non-discoverability.
            modified_obs = f"{original_feedback} Note: The
expected information ('allison.hill@taylor.net') might not be
discoverable with the provided tools and website structure in this
 specific scenario."
            obs = modified_obs
            logger.info(f"Modified Obs for Allison Hill email task
 due to potential non-discoverability: {obs}")
        # --- End of specific handling for Analysis Result 3 ---


        # --- Add specific handling for Analysis Result 4 ---
        # Check if it's the DNA sequence task (task_type_idx=1),
the agent submitted an answer,
        # the answer was wrong (reward=0), and the task is done.
        if task_type_idx == 1 and agent_action.startswith("Answer:
") and reward == 0.0 and done:
             # Extract the answer part
             answer_content = agent_action.split("Answer:", 1)[1].
strip()
             # Check if the answer looks like a list
             if answer_content.startswith('[') and answer_content.
endswith(']'):
                 logger.debug(f"Handling incorrect answer format
for DNA sequence task ({task_name}). Agent provided a list: {
answer_content}. Original Obs: {obs}")
                 # Modify the observation to provide specific
feedback
                 modified_obs = "Incorrect. Please output only the
 single longest valid DNA sequence, not a list of all valid
sequences."
                 obs = modified_obs
                 logger.info(f"Modified Obs for DNA sequence task
due to incorrect list format: {obs}")
        # --- End of specific handling for Analysis Result 4 ---


    except Exception as e:
        # Catch potential errors during env.step
        logger.error(f"Error during env.step for action '{
agent_action}': {e}", exc_info=True)
        obs = f"Error executing action '{agent_action}': {e}"
        reward = 0.0
        # Assume error means task is not successfully completed,
but allow agent to retry
        done = False

    # Return the final observation, reward, and done status
    return obs, reward, done
```