WALL-E: World Alignment by NeuroSymbolic Learning improves World Model-based LLM Agents

Siyu Zhou[1], Tianyi Zhou[2], Yijun Yang[3], Guodong Long[1], Deheng Ye[3], Jing Jiang[1], Chengqi Zhang[4]
[1]Australian AI Institute, University of Technology Sydney
[2]University of Maryland, College Park [3]Tencent [4]Hong Kong Polytechnic University
Siyu.Zhou-2@student.uts.edu.au, zhou@umiacs.umd.edu

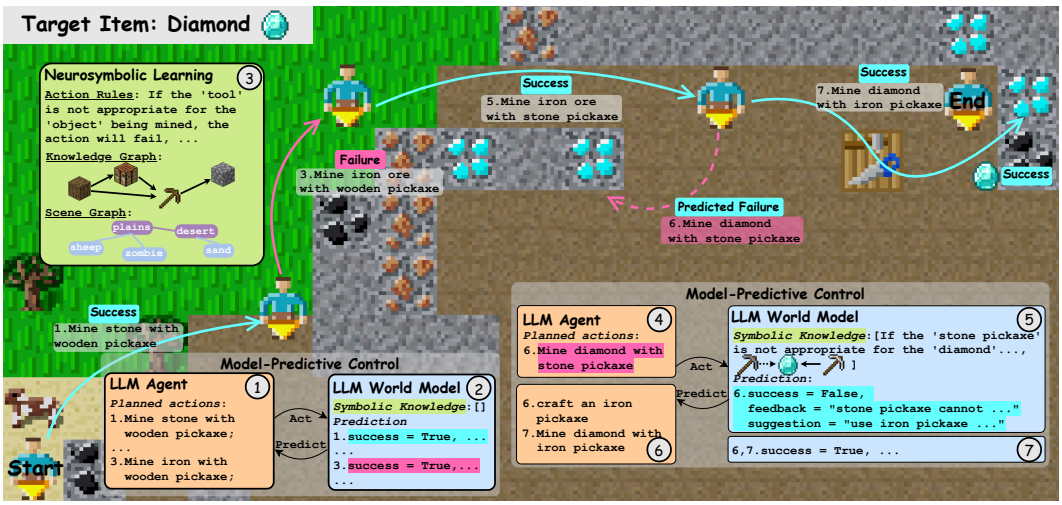

Figure 1: **WALL-E mining a diamond on Mars.** Step 1-2: The agent makes decisions via MPC with the initial unaligned world model, resulting in a failed action for mining iron. Step 3: Leveraging previous trajectories and world model predictions, WALL-E learns symbolic knowledge, including action rules, knowledge graphs, and scene graphs. Step 4-5: The learned symbolic knowledge helps the world model make accurate predictions and correct the previous mistake. Step 6-7: The agent adjusts its decision accordingly and replaces *stone pickaxe* with *iron pickaxe* toward completing the task.

Abstract

Can we build accurate world models out of large language models (LLMs)? How can world models benefit LLM agents? The gap between the prior knowledge of LLMs and the specified environment's dynamics usually bottlenecks LLMs' performance as world models. To bridge the gap, we propose a training-free "*world alignment*" that learns an environment's symbolic knowledge complementary to LLMs. The symbolic knowledge covers action rules, knowledge graphs, and scene graphs, which are extracted by LLMs from exploration trajectories and encoded into executable codes to regulate LLM agents' policies. We further propose an RL-free, model-based agent "WALL-E" through the model-predictive control (MPC) framework. Unlike classical MPC requiring costly optimization on the fly, we adopt an LLM agent as an efficient look-ahead optimizer of future steps' actions by interacting with the neurosymbolic world model. While the LLM agent's strong heuristics make it an efficient planner in MPC, the quality of its planned actions is also secured by the accurate predictions of the aligned

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

world model. They together considerably improve learning efficiency in a new environment. On open-world challenges in Mars (Minecraft like) and ALFWorld (embodied indoor environments), WALL-E significantly outperforms existing methods, e.g., surpassing baselines in Mars by 16.1%–51.6% of success rate and by at least 61.7% in score. In ALFWorld, it achieves a new record 98% success rate after only 4 iterations. Our code is available here.

1 Introduction

While large language models (LLMs) have been successfully applied to complex reasoning, generation, and planning tasks, they are not sufficiently reliable to be deployed as an agent in specific open-world environments, e.g., games, VR/AR systems, medical care, education, autonomous driving, etc [1–3]. A primary reason for the failures is the gap between the prior knowledge driven commonsense reasoning by LLMs and the specified environment's dynamics. The gap leads to incorrect predictions of future states, hallucinations, or violation of basic laws in LLM agents' decision-making process [4–7]. Although the alignment of LLMs with human preferences has been widely studied as a major objective of LLM post-training, "world alignment" with an environment's dynamics has not been adequately investigated for LLM agents [8–10]. Moreover, many existing LLM agents are model-free: they directly generate and execute actions in real environments without being verified or optimized within a world model or simulator [4, 11–16] in advance. This leads to safety risks and suboptimality of planned trajectories.

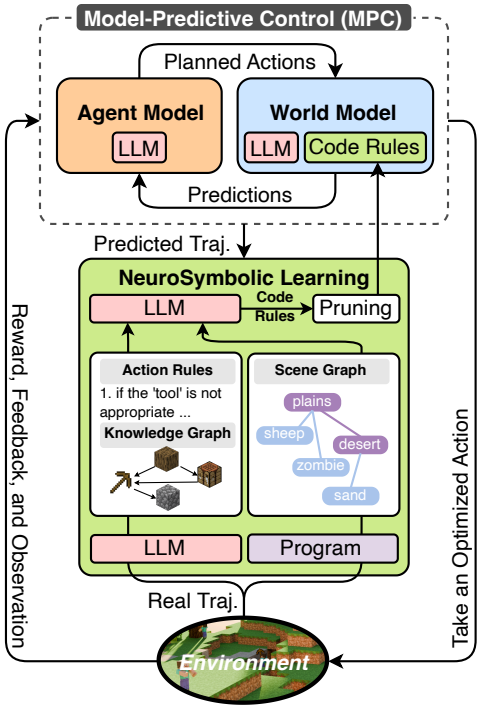

Figure 2: **Overview of WALL-E**. The agent determines actions to take via MPC, which optimizes future steps' actions by interacting with a neurosymbolic world model. The world model adopts an LLM whose predictions are aligned with environment dynamics through code rules converted from symbolic knowledge (action rules, knowledge/scene graph) learned via inductive reasoning from real trajectories and predicted trajectories.

In this paper, we show that the "**world alignment**" of an LLM can significantly improve its performance as a promising world model, which enables us to build more powerful embodied LLM agents in partially observable settings. To this end, we introduce a training-free recipe "**World Alignment by neurosymboLic LEarning (WALL-E)**", which compares and analyzes agent-explored trajectories and model-predicted ones to extract symbolic knowledge that is environment-specific and complementary to the LLM's prior knowledge, as illustrated in Figure 2. WALL-E's symbolic knowledge covers action rules, knowledge graphs, and scene graphs, which can be converted into executable *code rules* to turn a pretrained LLM into an accurate neurosymbolic world model (via function calling). It combines the strengths of both LLMs and symbolic representations in modeling environment dynamics, i.e., (1) the rich prior knowledge, probabilistic, and deductive reasoning capability of LLMs under uncertainty [17]; and (2) the formal constraints, deterministic transition rules, and environment-specific structures enforced or encoded by symbolic knowledge. In our studies, LLMs already cover the most common knowledge of dynamics but a few additional symbolic knowledge can significantly improve the world model predictions generated by LLMs.

Different types of symbolic knowledge play important roles in building more reliable and adaptive model-based LLM agents, especially in partially observable Markov decision processes (POMDPs). The action rules capture and enforce deterministic constraints in the decision-making process; the knowledge graph represents feasibility constraints and action prerequisites; while the scene graph provides global information complementing the partial observations of agents. In WALL-E, we

leverage LLMs' inductive reasoning capabilities to abstract concise symbolic knowledge from explored trajectories for episodes. We further develop a maximum coverage-based pruning to maintain a compact set of *code rules*. In contrast, existing LLM agents usually learn the environment dynamics through expensive finetuning of LLM policies via RL/imitation learning, or rely heavily on task-specific modeling procedures, limiting scalability and flexibility. [4, 5, 18–24].

Unlike the mainstream model-free LLM agents, WALL-E's precise neurosymbolic world model enables us to create more reliable and versatile model-based LLM agents for challenging open-world tasks. Specifically, we propose a novel model-predictive control (MPC) framework for LLM agents, in which an LLM agent conducts a look-ahead optimization (planning) of future steps' actions by interacting with the world model. For example, the agent queries the world model "What will happen if I take action a_t in observation o_t?", and receives the prediction with feedback/suggestions according to the *code rules*, based on which the agent chooses to execute the plan or keep refining it until it passes the world model's examination, i.e., not violating any constraints of the environments and leading the agent to preferred (predicted) states. Our **LLM-based MPC** framework overcomes the inefficiency of classical MPC that requires online k-step optimization, by exploiting the strong reasoning and instruction following capability of LLMs.

We evaluate WALL-E in challenging open-worlds such as Mars (Minecraft-like) and ALFWorld (embodied indoor environment) where the agents can explore freely and target complicated tasks. Our main contributions can be summarized as:

- We address the "world alignment" challenge for LLM agents and propose a novel training-free approach "WALL-E" to align LLMs with environment dynamics, resulting in neurosymbolic world models.
- We develop a novel LLM-based MPC framework for world model-based LLM agents.
- WALL-E achieves state-of-the-art performance in Mars and ALFWorld.

2 Related Work

Although the alignment of LLMs has been widely studied as a major objective of LLM post-training [25–28], their alignment with the dynamics of external environments, which we refer to as "world alignment", remains far less explored for LLM agents.

LLMs to build world models. Previous research has used LLMs to construct explicit world models for planning. For instance, Ada [21] integrate LLMs with hierarchical planners to acquire task-specific action abstractions and low-level controllers, thereby offering more accurate plans. Other work constructs domain models in PDDL and leverages external planners, with LLMs acting as an interface for generating, validating, and refining the models using human or automated feedback [22]. WorldCoder [19] instead builds a Python program to represent the agent's evolving world knowledge. The program is iteratively refined through interaction, serving as both a symbolic simulator of state transitions and a constraint-guided planner, achieving higher sample efficiency. While effective, these methods typically require building symbolic models from scratch and rely on environment-specific design, which reduces scalability. In contrast, we treat the LLM itself as the world model and align it via training-free symbolic knowledge extraction, avoiding complex, environment-specific modeling pipelines.

LLMs as world models. Another line of work uses the LLM itself as the world model. Some approaches fine-tune LLMs on embodied experience to enhance reasoning and planning in situated environments [23], while others incorporate action preconditions and effects during fine-tuning for more accurate outcome prediction [24]. Prompt-based approaches include LLM-MCTS [20], RAP [8], and RAFA [29], which guide LLMs to plan trajectories or explore reasoning trees through carefully designed, environment-specific prompts. While effective, these approaches often depend on heavy fine-tuning or manual prompt engineering, limiting flexibility and scalability. In contrast, our method uses the inductive reasoning ability of LLMs to extract symbolic rules directly from exploration experiences. This process requires no handcrafted prompts, task-specific modeling, or fine-tuning, enabling rapid and scalable adaptation across diverse environments.

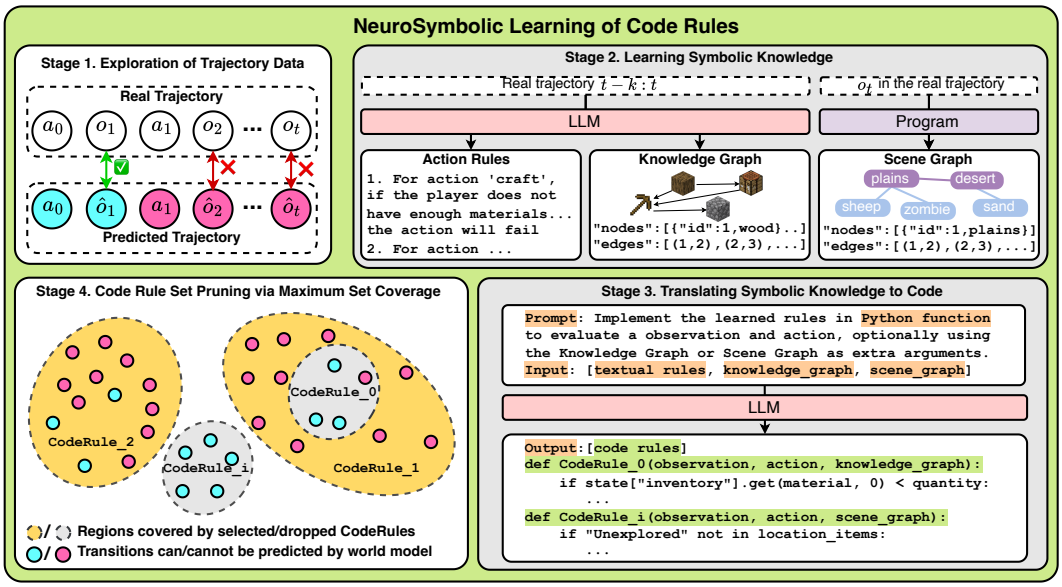

Figure 3: **NeuroSymbolic Learning of Code Rules**. WALL-E iteratively refines the symbolic knowledge with the agent's actual trajectories in the environment and the world model's predicted trajectories. The NeuroSymbolic learning takes 4 stages: (1) comparing predicted and actual trajectories; (2) learning new symbolic knowledge from real trajectories; (4) translating symbolic knowledge to code; and (4) Code rule set pruning via solving a maximum coverage problem.

3 World Alignment by NeuroSymbolic Learning (WALL-E)

We consider a Partially Observable Markov Decision Process (POMDP) defined by $\mathcal{M} = \big(\mathcal{S}, \mathcal{A}, \mathcal{O}, \rho, \mathcal{T}, \gamma\big)$, where $\mathcal{S}$ is the set of hidden (unobservable) states, $\mathcal{A}$ the action space, $\mathcal{O}$ the space of textual observations available to the agent, including possibly partial descriptions of the environment. The reward function $\rho : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ assigns a scalar reward, $\mathcal{T}(s_{t+1} \mid s_t, a_t)$ defines transition dynamics, and $\gamma \in [0, 1]$ is the discount factor. Since the agent cannot observe the true state s_t, it must act based on partial observations $o_t \in \mathcal{O}$.

3.1 NeuroSymbolic Learning of Code Rules

3.1.1 Stage 1: Exploration of Trajectory Data

To evaluate inconsistencies between the LLM world model and the real environment, we compare the environment-generated (real) trajectory $\tau^{\text{real}} = \{\delta^{\text{real}} = (o_t, a_t, o_{t+1})\}_{t=0}^{T}$ and the model-predicted trajectory $\tau^{\text{predicted}} = \{\delta^{\text{predicted}} = (o_t, a_t, \hat{o}_{t+1})\}_{t=0}^{T}$.

Specifically, our world model first predicts whether a transition will succeed or fail (binary classification), and then predicts the next observation o_{t+1}. This design choice is motivated by the fact that o_{t+1} can be attributed to the success or failure of an action. For example, if the action "craft a table" succeeds, the number of tables increases and the corresponding materials decrease in o_{t+1}. In contrast, directly predicting the full observation o_{t+1} regardless of the action's consequence is more challenging and susceptible to noise from minor environmental details (e.g., weather or creature presence). Thus, we perform consistency checks on o_{t+1} and $\hat{o}_{t+1}$ based on transition success/failure only. We then classify all transitions in $\tau^{\text{predicted}}$ into two sets: correctly predicted transitions $\delta^{\text{cor}} \in \mathcal{D}^{\text{cor}}$ and incorrectly predicted transitions $\delta^{\text{inc}} \in \mathcal{D}^{\text{inc}}$.

As shown in Step 1 of Figure 3, the real trajectories τ^{real} are used to induce code rules that align the LLM world model with the true environment (Sec. 3.1.2, 3.1.3). $\mathcal{D}^{\text{incorrect}}$ then serves as a diagnostic set for pruning redundant or conflicting code rules via a maximum coverage problem (Sec. 3.1.4).

3.1.2 Stage 2: Learning Symbolic Knowledge

Extraction of Action Rules: Deterministic Transitions We use the LLM's inductive reasoning to extract action rules from real trajectories $\tau^{\text{real}}_{t-k:t}$, where $\tau^{\text{real}}_{t-k:t}$ denotes the past k real transitions before timestep t. The parameter k defines a finite context window for the trajectory. These rules capture deterministic constraints in the environment and help enforce correct decision-making (an example is shown in the stage 2 of Figure 3). For example, an action rule might specify: "For action *make*, if *table* is not in *near_objects*, the action will fail." Formally,

$$\mathcal{R}_t = f_{\text{LLMrule}}\left(\tau^{\text{real}}_{t-k:t}\right) \cup \mathcal{R}_{t-k} \tag{1}$$

where f_{LLMrule} is implemented by prompting an LLM to generate action rules (see Appendix B.1 for detailed prompt), and $\mathcal{R} = \{r_1, r_2, \ldots, r_{|\mathcal{R}|}\}$ denotes the extracted rule set.

Knowledge Graph: Constraints in the POMDP We construct a knowledge graph $\mathcal{G}^{\text{knowledge}} = (\mathcal{V}, \mathcal{E})$ to represent feasibility constraints and action prerequisites, where nodes $\mathcal{V}$ are entities (e.g., materials, tools) and edges $\mathcal{E}$ denote symbolic relations. The graph is incrementally updated by merging new relations extracted from recent trajectories:

$$\mathcal{G}^{\text{knowledge}}_t \leftarrow f_{\text{LLMkg}}\left(\tau^{\text{real}}_{t-k:t}\right) \cup \mathcal{G}^{\text{knowledge}}_{t-k}, \tag{2}$$

where f_{LLMkg} prompts the LLM to extract relation triples $\{(u, v, e) \mid u, v \in \mathcal{V},\ e \in \mathcal{E}\}$ from $\tau^{\text{real}}_{t-k:t}$ (see Appendix B.3 for detailed prompt), (u, v, e) captures a constraint on how an entity u relates to another entity v under the label e, e.g., "require", "consume", and "enable". For instance, if the LLM observes that item A always fails to be created unless x units of B and y units of C are available, it induces the constraints $(B, A, \text{require } x)$ and $(C, A, \text{require } y)$.

Scene Graph: Complementing Partial Observations To handle partial observability in POMDPs, we construct a Scene Graph (SG) to augment the agent's local observation with global context. Formally, $\mathcal{G}^{\text{scene}} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents observed entities (e.g., objects, locations), while $\mathcal{E}$ captures spatial relations such as "ownership", "in" or "adjacency to". For example, if the environment contains multiple rooms, and each room includes certain items, SG can record the specific items located in every room, complementing the agent's local observation with global information. At each time step t, we update the SG via:

$$\mathcal{G}^{\text{scene}}_t = f_{\text{scene}}(o_t) \cup \mathcal{G}^{\text{scene}}_{t-1}, \tag{3}$$

where f_{scene} is a function that analyzes the agent's observation o_t and generates a sub-graph recording the spatial relationships among entities present in o_t.

3.1.3 Stage 3: Translating Symbolic Knowledge to Compact Code Rules

Next, given the rules $\mathcal{R}$ and the graphs $\mathcal{G}^{\text{knowledge}}$ and $\mathcal{G}^{\text{scene}}$ (which, unless otherwise specified, refer to $\mathcal{R}_t$, $\mathcal{G}^{\text{knowledge}}_t$, and $\mathcal{G}^{\text{scene}}_t$, respectively, throughout the rest of the paper), we prompt the LLM f_{LLMcode} to generate executable code rules as below (see Appendix B.2 for the detailed prompt):

$$\mathcal{R}^{\text{code}} = f_{\text{LLMcode}}\left(\mathcal{R}, \mathcal{G}^{\text{knowledge}}, \mathcal{G}^{\text{scene}}\right). \tag{4}$$

An example rule $r^{\text{code}}_i \in \mathcal{R}^{\text{code}}$ is shown on the right (all examples in Appendix D). Each rule returns: feedback (fd^{code}), success flag ($flag^{\text{code}}$), and suggestion (sug^{code}). The flag indicates transition success, while feedback and suggestion guide corrective re-planning. These compact functions distill symbolic knowledge to enhance planning.

```
def CodeRule_example(state, action, kg/sg):
  ...
  tool = action.get("args", {}).get("toolname")
  inventory = state.get("inventory", {})
  for res, amt in kg.get(tool, {}).items():
    if inventory.get(res, 0) < amt:
      return f"Action failed:Not enough {res}."
             , False, f"Collect more {res}."
  return "Success!", True, ""
```

3.1.4 Stage 4: Code Rules Pruning

To ensure that the code rules derived from neuro-symbolic learning are correct, executable, and compact, we employ a multi-stage validation and pruning pipeline that integrates empirical testing with optimization.

As described earlier, our world model predicts the success or failure of environment transitions. Accordingly, each code rule evaluates transition feasibility in the form: "if [state, action conditions] then [transition success/failure], otherwise [null]" (see code format in Section 3.1.3). A rule is triggered when its conditions are satisfied, returning a non-null output. By comparing these predictions against collected transitions, we can systematically pruning rules through the following steps.

Each candidate rule is first tested against ground-truth transitions sampled from the real trajectory τ^{real}. If a rule misfires, producing outputs inconsistent with the true environment dynamics, it is immediately discarded. This ensures that only rules faithfully aligned with actual behavior remain. Next, they are checked for executability, each rule is run in a controlled environment, and any rule that raises a runtime or logical error is eliminated. Together, these two verification steps guarantee that only correct and executable rules remain in the pool.

After verification, we optimize the retained rule set using a maximum coverage objective. The goal is to prioritize rules that best explain and correct incorrectly predicted transitions $\mathcal{D}^{\text{inc}}$. If a code rule r_i^{code} corrects a mispredicted transition δ_j^{inc}, we define that δ_j^{inc} is covered by r_i^{code}. Thus, we frame code rule selection as a maximum coverage problem, aiming to find a compact subset $\mathcal{R}^* \subseteq \mathcal{R}^{\text{code}}$ that maximally covers $\mathcal{D}^{\text{inc}}$,

$$\mathcal{R}^* = \arg\max_{\mathcal{R} \subseteq \mathcal{R}^{\text{code}}} \Big| \bigcup_{r^{\text{code}} \in \mathcal{R}} \mathcal{D}^{cov} \Big|, \text{ s.t., } |\mathcal{R}| \leq l, \tag{5}$$

where $\mathcal{D}^{cov}$ is the subset of transitions covered by code rules, i.e., $\mathcal{D}^{cov} \triangleq \{\delta^{\text{inc}} \in \mathcal{D}^{\text{inc}} : \delta^{\text{inc}} \text{ covered by } r^{\text{code}} \in \mathcal{R}^{\text{code}}\}$. The parameter $l > 0$ is the limit of selected code rules, and we find that a large l usually leads to better performance.

We solve this optimization problem using a greedy algorithm (see Appendix E). Through this process, we eliminate **code rules covering only correct transitions**, as they cannot address misalignments, and **redundant code rules** fully covered by more comprehensive ones (see Step 4 in Figure 3).

The NeuroSymbolic learning process consists of four stages: (1) comparing predicted and real trajectories, (2) extracting symbolic knowledge, (3) translating this knowledge into executable code rules, and (4) pruning the rules via verification and a maximum coverage objective. This procedure, defined as NSLEARNING(), yields a compact and effective rule set:

$$\mathcal{R}^{\text{code}} \leftarrow \text{NSLEARNING}(\tau^{\text{pred}}, \tau^{\text{real}}). \tag{6}$$

The resulting rules enhance the LLM world model's alignment and improve decision-making quality.

3.2 Model-Predictive Control for World Model-based LLM Agents

As shown in Figure 2, at each MPC step, the agent queries the LLM world model f_{LLMwm}, asking, "What happens if I take action a_t in observation o_t?", the model generates the next observation $\hat{o}_{t+1}$, textual feedback fb, detailed suggestion sug, and *flag* identifying action's success/failure, according to the agent's action a_t in o_t, as described below.

$$(\hat{o}_{t+1},\ \textit{fb},\ \textit{sug},\ \textit{flag}) = f_{\text{LLMwm}}(o_t, a_t). \tag{7}$$

To align predictions with the true environment dynamics, the code rules $\mathcal{R}^{\text{code}}$ derived from exploration are applied. Importantly, these code rules do not replace the LLM world model. Instead, they serve as corrective constraints that mitigate LLM's residual misalignments, while the LLM itself already captures most of the environment dynamics and contributes essential prior knowledge for prediction. Specifically, the LLM's outputs are verified against the rule set $\mathcal{R}^{\text{code}}$, which can generate corrective feedback ($\textit{fb}^{\text{code}}, \textit{sug}^{\text{code}}, \textit{flag}^{\text{code}}$) (as discussed in Section 3.1.3). The final aligned output is handled by

$$(\hat{o}_{t+1}, \textit{fb}, \textit{sug}, \textit{flag}) = \text{ALIGN}\big(\mathcal{R}^{\text{code}}, f_{\text{LLMwm}}, o_t, a_t\big), \tag{8}$$

which internally combines the outputs of f_{LLMwm} and $\mathcal{R}^{\text{code}}$ mentioned above. Concretely,

$$\text{ALIGN} = \begin{cases} f_{\text{LLMwm}}(o_t, a_t, \textit{fb}^{\text{code}}, \textit{sug}^{\text{code}}, \textit{flag}^{\text{code}}), & \text{if } \textit{flag}^{\text{code}} \neq \textit{flag}, \\ f_{\text{LLMwm}}(o_t, a_t), & \text{otherwise}, \end{cases} \tag{9}$$

In the conflicting case $\textit{flag}^{\text{code}} \neq \textit{flag}$, the world model is re-invoked with $\textit{fb}^{\text{code}}, \textit{sug}^{\text{code}}, \textit{flag}^{\text{code}}$ to revise its prediction in line with code rules. We avoid using $\mathcal{R}^{\text{code}}$'s outputs directly, as they lack the

full context needed for the LLM agent to follow reliably (as shown in Section 3.1.3). If *flag* = False, the agent uses the feedback and suggestions to revise its plan accordingly.

$$a_t \leftarrow \text{LLMAGENT}(o_t, \textit{fb}, \textit{sug}). \tag{10}$$

This refinement loop continues until the proposed action passes verification or the replanning limit is hit (as shown in Algorithm 1). The entire MPC process is defined as:

$$a_t, \hat{o}_{t+1} \leftarrow \text{MPC}(o_t, \mathcal{R}^{\text{code}}) \tag{11}$$

Crucially, unlike conventional MPC methods that rely on computationally expensive action search methods such as cross entropy method (CEM) [30] and derivative-free optimization (DFO) [31], WALL-E leverages the rich prior knowledge and reasoning capabilities of pretrained LLMs to implement a highly efficient **look-ahead optimizer** (i.e., $\text{LLMAGENT}(\cdot)$), substantially reducing the search space and computational complexity, especially in complex or high-dimensional environments.

WALL-E alternately executes **MPC** and **NeuroSymbolic learning**. In each MPC phase, actions proposed by the LLM agent are predicted and validated by the world model using the code rules. Meanwhile, NeuroSymbolic learning continuously extracts and updates code rules from recent interactions, which are then integrated into the world model. The full procedure is in Algorithm 2.

Algorithm 1 Model-Predictive Control (MPC)

Input: o_t, $\mathcal{R}^{\text{code}}$
Initialize: *fb* $\leftarrow []$, *sug* $\leftarrow []$, *count* $\leftarrow 0$
repeat
  $a_t \leftarrow \text{LLMAGENT}(o_t, \textit{fb}, \textit{sug})$ /*eq.(10)*/
  $\hat{o}_{t+1}, \textit{fb}, \textit{sug}, \textit{flag} \leftarrow \text{ALIGN}(\mathcal{R}^{\text{code}}, f_{\text{LLMwm}}, o_t, a_t)$ /*eq.(8)*/
  count $\leftarrow$ *count* $+ 1$
  if $flag$ **then**
    break /*Action accepted*/
  end if
until *count* $\geq \text{REPLANLIMIT}$
Output: a_t, $\hat{o}_{t+1}$

Algorithm 2 WALL-E

/* $\text{NSLEARNING}()$ in Sec. 3.1; $\text{MPC}()$ in Sec. 3.2 */
Initialize $\mathcal{R}^{\text{code}} \leftarrow \emptyset$, $\tau^{\text{real}}, \tau^{\text{pred}} \leftarrow []$, $t \leftarrow 0$, $o_0 \leftarrow \text{ENV}()$
while $\neg\text{TASKCOMPLETE}(o_t)$ **and** $\neg\text{AGENTDIED}(o_t)$ **do**
  $a_t, \hat{o}_{t+1} \leftarrow \text{MPC}(o_t, \mathcal{R}^{\text{code}})$ /*eq.(11)*/
  $o_{t+1} \leftarrow \text{ENV}(a_t)$
  $\tau^{\text{real}}.\text{APPEND}((o_t, a_t, o_{t+1}))$
  $\tau^{\text{pred}}.\text{APPEND}((o_t, a_t, \hat{o}_{t+1}))$
  $\mathcal{R}^{\text{code}} \leftarrow \text{NSLEARNING}(\tau^{\text{pred}}, \tau^{\text{real}})$ /*eq.(6)*/
  $t \leftarrow t + 1$
end while

4 Experiments

4.1 Benchmarks

- **Mars** [32] is an interactive benchmark for testing situated inductive reasoning—where agents must derive and apply new rules in unfamiliar contexts. Built on Crafter [33], Mars modifies terrain, survival settings, and task dependencies to generate diverse counter-commonsense worlds. Agents must interact with these worlds, adapting to novel rules rather than relying on pre-existing knowledge. This makes Mars a unique testbed for evaluating agent adaptability and reasoning in dynamic environments. See Appendix F for details.
- **ALFWorld** [34] is a virtual environment designed as a text-based simulation where agents perform tasks by interacting with a simulated household environment. This benchmark includes six distinct task types, each requiring the agent to accomplish a high-level objective, such as placing a cooled lettuce on a countertop. See Appendix F for details.

4.2 Evaluation Metrics

- **Mars:** (1) **Reward** (higher is better): the sum of sparse rewards given during an episode, including $+1$ for unlocking achievements, -0.1 for health points lost, and $+0.1$ for health points regained. The reward primarily reflects the number of achievements unlocked. (2) **Score** (higher is better): a single aggregated value derived from the geometric mean of success rates across achievements, weighing rare and difficult achievements more heavily to better reflect overall capability.
- **ALFWorld:** (1) **Success rate** (higher is better): percentage of tasks the agent completes.

Table 1: Comparison of WALL-E with RL-based (PPO [35] & DreamerV3 [36]) and LLM-based methods (ReAct [37], Reflexion [38], Skill Library [39], IfR [32] in Mars [32]. LLM-based methods and RL-based methods' results are averaged over 9 trials and 20 trials, respectively ($*$-reported in previous work). RL trains a policy separately for each world and is supposed to be better than LLM-based. Reward are accumulated and reflects unlocked achievements. Score (%) is the weighted geometric mean of success rates, emphasizing rare and challenging achievements. Both metrics report mean$\pm$standard deviation across trials. The "Average" row excludes the Default world to focus on counter-commonsense settings, following the Mars evaluation protocol. The best results within the training-free (LLM-based) methods are in **bold**. **WALL-E is the SOTA among training-free methods**, demonstrating strong adaptability and planning in both familiar and counter-commonsense settings that conflict with pretrained LLM priors. DreamerV3, while achieving higher absolute scores, is trained with 1 million environment steps and included for reference only, highlighting the contrast in sample efficiency and the rapid adaptation capabilities of WALL-E.

Metrics	Mod. Type[1]	RL-based methods		LLM-based methods				
		PPO*	DreamerV3*	ReAct*	Reflexion*	Skill Library*	IfR*	WALL-E
Reward ↑	Default	1.9 ± 1.4	11.5 ± 1.6	7.7 ± 1.6	6.0 ± 1.7	8.0 ± 2.1	9.0 ± 2.3	$\mathbf{9.5 \pm 2.1}$
	Terrain	-0.1 ± 0.6	9.3 ± 2.2	7.4 ± 2.7	6.4 ± 3.0	9.5 ± 2.9	8.0 ± 3.7	$\mathbf{10.7 \pm 2.6}$
	Survival	-0.6 ± 0.5	8.6 ± 4.1	6.4 ± 3.7	4.6 ± 3.9	7.9 ± 2.9	7.7 ± 3.7	$\mathbf{13.8 \pm 4.4}$
	Task. Dep.	2.1 ± 1.2	8.8 ± 2.8	5.0 ± 2.1	3.2 ± 1.6	1.5 ± 1.9	5.6 ± 2.9	$\mathbf{6.4 \pm 2.9}$
	Terr. Surv.	0.0 ± 0.7	7.1 ± 2.1	6.7 ± 2.5	4.9 ± 2.5	3.0 ± 2.5	$\mathbf{6.8 \pm 1.9}$	5.5 ± 2.7
	Terr. Task.	-0.7 ± 0.3	6.6 ± 0.7	4.8 ± 2.0	5.3 ± 2.5	5.5 ± 1.5	$\mathbf{6.9 \pm 1.8}$	5.8 ± 2.2
	Surv. Task.	-0.6 ± 0.4	9.6 ± 3.4	1.5 ± 1.3	1.0 ± 1.6	2.3 ± 1.5	$\mathbf{3.3 \pm 1.4}$	3.2 ± 1.4
	All three	0.1 ± 0.8	5.1 ± 1.8	0.7 ± 1.6	-0.4 ± 0.7	-0.5 ± 0.5	0.1 ± 0.5	$\mathbf{1.3 \pm 1.6}$
	Average	0.0	7.9	4.6	3.6	4.2	5.5	**6.7**
Score (%) ↑	Default	1.3 ± 1.7	14.2 ± 1.3	8.0 ± 1.5	5.3 ± 0.9	8.3 ± 1.3	13.0 ± 2.1	$\mathbf{20.3 \pm 1.8}$
	Terrain	0.3 ± 0.1	13.0 ± 1.6	7.6 ± 2.6	7.4 ± 1.6	11.9 ± 3.4	11.8 ± 2.9	$\mathbf{27.8 \pm 1.7}$
	Survival	0.2 ± 0.0	10.8 ± 2.8	8.0 ± 0.6	5.5 ± 1.7	9.7 ± 2.0	11.0 ± 3.7	$\mathbf{50.8 \pm 1.1}$
	Task. Dep.	1.7 ± 0.6	12.1 ± 1.9	4.6 ± 1.6	2.2 ± 0.8	1.5 ± 0.6	6.9 ± 2.5	$\mathbf{9.3 \pm 2.0}$
	Terr. Surv.	0.4 ± 0.1	7.9 ± 1.3	7.1 ± 3.0	4.7 ± 1.6	2.8 ± 0.6	6.7 ± 0.8	$\mathbf{8.6 \pm 1.9}$
	Terr. Task.	0.1 ± 0.1	4.2 ± 0.1	3.8 ± 0.3	5.5 ± 1.7	4.1 ± 0.7	$\mathbf{7.1 \pm 2.5}$	4.7 ± 2.0
	Surv. Task.	0.1 ± 0.1	15.9 ± 2.6	1.3 ± 0.2	1.1 ± 0.1	1.9 ± 0.1	2.1 ± 0.4	$\mathbf{3.3 \pm 1.9}$
	All three	0.6 ± 0.2	4.0 ± 0.3	1.0 ± 0.3	0.2 ± 0.1	0.2 ± 0.0	0.6 ± 0.0	$\mathbf{2.2 \pm 1.6}$
	Average	0.6	10.3	5.2	4.0	5.0	7.4	**15.3**

Table 2: Comparison of WALL-E and baselines on 134 testing tasks from the ALFWorld [34]. $*$-reported in previous work. The highest success rate (%) for each task is highlighted in **bold**. **WALL-E outperforms all other baselines.**

Method	Success Rate (%) ↑						
	Avg.	Pick	Clean	Heat	Cool	Examine	Picktwo
BUTLER* [15]	26	31	41	60	27	12	29
GPT-BUTLER* [15]	69	62	81	85	78	50	47
DEPS [41]	76	93	50	80	**100**	**100**	0
AutoGen* [14]	77	92	74	78	86	83	41
ReAct [11]	74	79	54	96	85	83	51
AdaPlanner [40]	91	**100**	**100**	89	**100**	97	47
Reflexion [12]	86	92	94	70	81	90	88
RAFA [29]	95	**100**	97	91	95	**100**	82
WALL-E (ours)	**98**	**100**	**100**	96	**100**	**100**	94
Human Performance* [42]	91	-	-	-	-	-	-

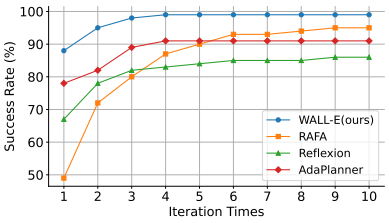

Figure 4: WALL-E vs. baselines on 134 ALFWorld tasks. **WALL-E achieves the highest success rate after only 4 iterations, surpassing all baselines.**

4.3 Main Results

WALL-E excels in planning and task-solving with rapid adaptation. As shown in Tables 1 and 2, WALL-E consistently outperforms baselines across diverse environments. In Mars, WALL-E achieves reward improvements of 16.1% to 51.6% and boosts task scores by at least 61.7%. While DreamerV3 reaches higher absolute rewards, it is trained for 1 million environment steps—far exceeding WALL-E 's budget of just 5,600 steps (0.56% of DreamerV3's budget). Our focus is not on absolute performance, but on enabling efficient few-shot adaptation and situated reasoning. DreamerV3 is included as a reference to emphasize this contrast in learning efficiency. In the "Survival" setting, our method shows high variance due to cow attacks: performance drops sharply when many cows spawn nearby, but remains strong when few appear, leading to a large standard deviation. In ALFWorld, WALL-E achieves the highest success rate after only four iterations, significantly surpassing strong baselines like RAFA [8] and AdaPlanner [40], as shown in Figure 4.

[1]World types include Default (original Crafter setting with no modifications) and counter-commonsense worlds, which feature individual modifications (Terrain, Survival, Task Dependency) or combinations of two or all three modifications (Terr. Surv., Terr. Task., Surv. Task., All Three).

WALL-E adapts effectively to environments that contradict the LLM's prior knowledge. As shown in Table 1, performance drops significantly (by 3.3%–97.1%) when the environment deviates from the LLM's priors. However, WALL-E overcomes this by inductively learning symbolic knowledge that realign the world model with actual environment dynamics. This leads to improved predictions and decision-making, yielding at least a 21.8% reward gain and 51.6% score gain over other methods in counter-commonsense settings. When environmental changes originate from a single source, WALL-E remains highly adaptable, improving reward and score by at least 31% and 66%, respectively. However, when multiple factors change, the improvement diminishes—likely because our rules are tailored to specific dynamics and cannot fully capture more complex mechanics.

WALL-E significantly outperforms the SOTA baseline IfR. Figure 5 shows that WALL-E improves from a reward of 3.6 to 6.7 and from a score of 4% to 15.3%, exceeding IfR by 17.9% and 51.6%, respectively. Two key innovations drive this gain: (1) NeuroSymbolic learning fuses action rules, knowledge graphs, and scene graphs for a richer environment model, whereas IfR uses only inductively learned rules; (2) We use executable code rules, ensuring strict, deterministic enforcement, while IfR's natural-language rules, applied via prompting, introduce variability when priors conflict with rules. This combination delivers more reliable, effective planning.

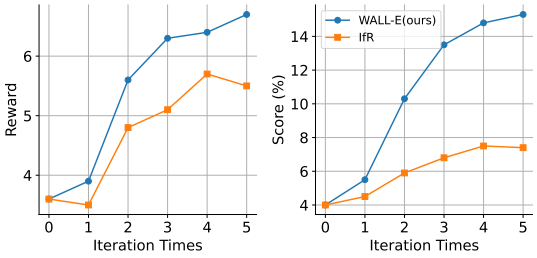

Figure 5: Comparison between WALL-E and IfR (the best baseline in Table 1) over learning iterations in Mars. **WALL-E achieves a clear advantage over IfR in both learning efficiency and overall performance**, due to the world alignment with diverse symbolic knowledge and code rules.

4.4 Effectiveness of NeuroSymbolic Learning

To evaluate the effectiveness of our proposed NeuroSymbolic Learning approach, we compare it with the skill library, a widely used agent learning method also employed by JARVIS-1 [39] and Voyager [43]. In the Mars benchmark, this approach is further simplified to better adapt to the environment, resulting in a framework consisting of a reasoning module, a reflector, and the skill library [32]. In contrast, WALL-E comprises a reasoning module, a reflector, and a neurosymbolic learning module. The skill library stores successful plans and utilizes them as in-context examples to align LLM agents with the environment's dynamics.

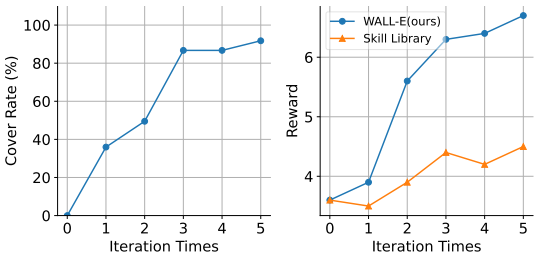

Figure 6: **Left**: WALL-E code rules' cover rate (higher the better) over neurosymsbolic learning iterations in Mars. The cover rate measures the percentage of LLM failed predictions that can be corrected by our world model. **Right**: Comparison between WALL-E's world model and skill library when applied to LLM agents with reasoning and reflection (base agent). WALL-E—base agent + world model; Skill Library—base agent + skill library. **WALL-E's neurosymbolic learning significantly improves world alignment and brings greater gains to LLM agents than skill library.**

NeuroSymbolic Learning enables efficient "world alignment". To assess how well the learned symbolic knowledge corrects the LLM world model, we collect the transitions that the world model mispredicts and compute the cover rate—the fraction of these errors fixed by newly acquired knowledge (details in Appendix F.3). By analyzing both the neurosymbolic learning process and the agent's performance across iterations (Figure 6), we observe that WALL-E consistently outperforms the skill library, and a significant performance improvement as the quality of the symbolic knowledge library increases (i.e., as the cover rate rises). This result highlights that WALL-E 's advancements primarily stem from acquiring and leveraging new symbolic knowledge.

Table 3: Ablation study of WALL-E with different symbolic knowledge (KNWL) types. KG—Knowledge Graph, SG—Scene Graph. The row highlighted in grey represents the configuration and performance of WALL-E. **Learning diverse types of symbolic knowledge is essential for effectively aligning the LLM world model with the environment.**

Symbolic KNWL		**Mars**		**ALFWorld**
Action Rules	KG/SG	**Reward** ↑	**Score (%)** ↑	**Success Rate (%)** ↑
✓		5.1	8.3	95
	✓	4.4	5.2	88
✓	✓	6.7	15.3	98

Table 4: Ablation study of WALL-E with different configurations on Mars tasks. Symbolic—apply code rules.The row highlighted in grey represents WALL-E. **Code rules translated from symbolic knowledge bring more improvement when applied to the world model, indicating the importance of world alignment.**

Agent	World Model	Reward ↑	Score (%) ↑
LLM	-	3.6	4.0
LLM	LLM	3.8	4.1
LLM+symbolic	-	5.5	7.4
LLM	LLM+symbolic	6.7	15.3
LLM+symbolic	LLM+symbolic	6.3	13.1

4.5 Ablation Study

Combining multiple types of symbolic knowledge enhances world alignment We evaluated world alignment using different symbolic knowledge combinations. When action rules are present, they run as executable code; otherwise, the system uses prompt learning alone. Table 3 shows that relying on a single type degrades performance: in the Mars environment, "rules only" cuts reward by 23.9% and score by 45.8%, while "KG+SG only" drops reward by 34.3% and score by 66.0%. These findings underscore the necessity of multiple symbolic knowledge types.

World models with symbolic knowledge is an effective structure. We examine the impact of learned symbolic knowledge and the world model by removing each component from WALL-E and observing changes in performance (Table 4). Adding symbolic knowledge, whether in the agent or the world model, consistently boosts the success rate. Specifically, applying symbolic knowledge in the world model yields about a 46.3% improvement, while applying them in the agent yields about a 30.9% gain. This difference likely arises because learned symbolic knowledge heavily depend on observation information (see Appendix D). Moreover, a standalone world model without symbolic knowledge offers little performance benefit, underscoring that symbolic knowledge-driven alignment with environment mechanism is key to WALL-E 's success.

Ablation on code rules pruning stage. We ablate the code rule pruning stage by evaluating performance both without pruning and under different code rule set limits l. Table 5 shows that pruning stage is essential—without it, performance drops sharply due to noisy or conflicting rules. As fewer rules are retained, performance degrades, confirming that pruning stage selects impactful, high-quality rules that align the LLM world model with environment dynamics.

Table 5: Ablation study on the code rule set pruning stage in Mars. The column in **bold** shows the configuration and performance of WALL-E. **Pruning is essential for selecting high-quality, impactful rules that align the LLM world model with environment dynamics.**

Metrics	**w/o Pruning**	**w/ Pruning - Rule Set Limit**				
		1	3	5	7	**no limit (9)**
Reward ↑	1.5	4.3	5.7	6.2	6.5	**6.7**
Score (%) ↑	1.6	8.1	9.5	12.8	14.5	**15.3**

5 Conclusion

We have demonstrated that LLMs can effectively serve as world models for agents when aligned with environment dynamics via neurosymbolic knowledge learning. Our neurosymbolic approach leverages code-based, gradient-free integrations of action rules, knowledge graphs, and scene graphs, thereby bridging the gap between the LLMs' prior knowledge and specific environments. By using a model-based framework, our agent, WALL-E, achieves substantial improvements in planning and task completion. In open-world environments such as Mars and ALFWorld, WALL-E outperforms Mars baselines by 16.1%–51.6% in success rate and achieves a new record 98% success rate in ALFWorld after only four iterations. These results underscore that additional symbolic knowledge is essential to align LLM predictions with environment dynamics, enabling high-performing model-based agents in complex settings.

References

[1] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv.2303.08774*, 2023.

[2] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, 2022.

[3] Y. Liu, W. Chen, Y. Bai, J. Luo, X. Song, K. Jiang, Z. Li, G. Zhao, J. Lin, G. Li, *et al.*, "Aligning cyber space with physical world: A comprehensive survey on embodied ai," *arXiv preprint arXiv:2407.06886*, 2024.

[4] Y. Mu, Q. Zhang, M. Hu, W. Wang, M. Ding, J. Jin, B. Wang, J. Dai, Y. Qiao, and P. Luo, "Embodiedgpt: Vision-language pre-training via embodied chain of thought," *arXiv preprint arXiv:2305.15021*, 2023.

[5] Y. Yang, T. Zhou, K. Li, D. Tao, L. Li, L. Shen, X. He, J. Jiang, and Y. Shi, "Embodied multi-modal agent trained by an llm from a parallel textworld," in *CVPR*, 2024.

[6] A. Das, S. Datta, G. Gkioxari, S. Lee, D. Parikh, and D. Batra, "Embodied question answering," in *CVPR*, 2018.

[7] X. Wu, T. Guan, D. Li, S. Huang, X. Liu, X. Wang, R. Xian, A. Shrivastava, F. Huang, J. L. Boyd-Graber, *et al.*, "Autohallusion: Automatic generation of hallucination benchmarks for vision-language models," *arXiv preprint arXiv:2406.10900*, 2024.

[8] 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," *arXiv preprint arXiv:2305.14992*, 2023.

[9] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *NeurIPS*, 2024.

[10] Z. Ge, H. Huang, M. Zhou, J. Li, G. Wang, S. Tang, and Y. Zhuang, "Worldgpt: Empowering llm as multimodal world model," *arXiv preprint arXiv:2404.18202*, 2024.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *ICLR*, 2023.

[12] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *NeurIPS*, 2024.

[13] B. Zitkovich, T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, *et al.*, "Rt-2: Vision-language-action models transfer web knowledge to robotic control," in *CoRL*, 2023.

[14] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.

[15] V. Micheli and F. Fleuret, "Language models are few-shot butlers," *arXiv preprint arXiv:2104.07972*, 2021.

[16] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, *et al.*, "Rt-1: Robotics transformer for real-world control at scale," *arXiv preprint arXiv:2212.06817*, 2022.

[17] Z. Hu and T. Shu, "Language models, agent models, and world models: The law for machine reasoning and planning," *arXiv preprint arXiv:2312.05230*, 2023.

[18] J. Gao, B. Sarkar, F. Xia, T. Xiao, J. Wu, B. Ichter, A. Majumdar, and D. Sadigh, "Physically grounded vision-language models for robotic manipulation," *arXiv preprint arXiv:2309.02561*, 2023.

[19] H. Tang, D. Key, and K. Ellis, "Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment," *arXiv preprint arXiv:2402.12275*, 2024.

[20] Z. Zhao, W. S. Lee, and D. Hsu, "Large language models as commonsense knowledge for large-scale task planning," *NeurIPS*, 2024.

[21] L. Wong, J. Mao, P. Sharma, Z. S. Siegel, J. Feng, N. Korneev, J. B. Tenenbaum, and J. Andreas, "Learning adaptive planning representations with natural language guidance," *arXiv preprint arXiv:2312.08566*, 2023.

[22] L. Guan, K. Valmeekam, S. Sreedharan, and S. Kambhampati, "Leveraging pre-trained large language models to construct and utilize world models for model-based task planning," *NeurIPS*, 2023.

[23] J. Xiang, T. Tao, Y. Gu, T. Shu, Z. Wang, Z. Yang, and Z. Hu, "Language models meet world models: Embodied experiences enhance language models," *NeurIPS*, 2024.

[24] K. Xie, I. Yang, J. Gunerli, and M. Riedl, "Making large language models into world models with precondition and effect knowledge," *arXiv preprint arXiv:2409.12278*, 2024.

[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *NeurIPS*, 2022.

[26] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, *et al.*, "Training a helpful and harmless assistant with reinforcement learning from human feedback," *arXiv preprint arXiv:2204.05862*, 2022.

[27] K. Yang, Z. Liu, Q. Xie, J. Huang, T. Zhang, and S. Ananiadou, "Metaaligner: Towards generalizable multi-objective alignment of language models," *arXiv preprint arXiv:2403.17141*, 2024.

[28] G. Seneque, L.-H. Ho, A. Kuperman, N. E. Saeedi, and J. Molendijk, "Abc align: Large language model alignment for safety & accuracy," *arXiv preprint arXiv:2408.00307*, 2024.

[29] Z. Liu, H. Hu, S. Zhang, H. Guo, S. Ke, B. Liu, and Z. Wang, "Reason for future, act for now: A principled framework for autonomous llm agents with provable sample efficiency," *arXiv preprint arXiv:2309.17382*, 2023.

[30] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, pp. 19–67, 2005.

[31] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to derivative-free optimization*. SIAM, 2009.

[32] X. Tang, J. Li, Y. Liang, S.-c. Zhu, M. Zhang, and Z. Zheng, "Mars: Situated inductive reasoning in an open-world environment," *arXiv preprint arXiv:2410.08126*, 2024.

[33] D. Hafner, "Benchmarking the spectrum of agent capabilities," *arXiv preprint arXiv:2109.06780*, 2021.

[34] M. Shridhar, X. Yuan, M.-A. Côté, Y. Bisk, A. Trischler, and M. Hausknecht, "Alfworld: Aligning text and embodied environments for interactive learning," *arXiv preprint arXiv:2010.03768*, 2020.

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[36] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, "Mastering diverse domains through world models," *arXiv preprint arXiv:2301.04104*, 2023.

[37] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[38] N. Shinn, F. Cassano, B. Labash, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning.(2023)," *arXiv preprint cs.AI/2303.11366*, 2023.

[39] Z. Wang, S. Cai, A. Liu, Y. Jin, J. Hou, B. Zhang, H. Lin, Z. He, Z. Zheng, Y. Yang, *et al.*, "Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models," *arXiv preprint arXiv:2311.05997*, 2023.

[40] H. Sun, Y. Zhuang, L. Kong, B. Dai, and C. Zhang, "Adaplanner: Adaptive planning from feedback with language models," *NeurIPS*, 2024.

[41] Z. Wang, S. Cai, G. Chen, A. Liu, X. Ma, Y. Liang, and T. CraftJarvis, "Describe, explain, plan and select: interactive planning with large language models enables open-world multi-task agents," in *NeurIPS*, 2023.

[42] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox, "ALFRED: A benchmark for interpreting grounded instructions for everyday tasks," in *CVPR*, 2020.

[43] 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," *arXiv preprint arXiv:2305.16291*, 2023.

[44] X. Zhu, Y. Chen, H. Tian, C. Tao, W. Su, C. Yang, G. Huang, B. Li, L. Lu, X. Wang, *et al.*, "Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory," *arXiv preprint arXiv:2305.17144*, 2023.

[45] Y. Qin, E. Zhou, Q. Liu, Z. Yin, L. Sheng, R. Zhang, Y. Qiao, and J. Shao, "Mp5: A multi-modal open-ended embodied system in minecraft via active perception," in *CVPR*, 2024.

[46] Z. Li, Y. Xie, R. Shao, G. Chen, D. Jiang, and L. Nie, "Optimus-1: Hybrid multimodal memory empowered agents excel in long-horizon tasks," *arXiv preprint arXiv:2408.03615*, 2024.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The Abstract and Introduction clearly state our main contributions, which are fully supported by the proposed method and experimental results.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in Section G.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: This paper does not include theoretical results, assumptions, or formal proofs.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Sections 4, F, and B provide sufficient details on setup, implementation, and prompts to reproduce the main results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: An anonymous GitHub repository with code and reproduction instructions is linked in the Abstract.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: All the training and test details are provided in Sections 4, B, and F.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Table 1 reports mean and standard deviation across 9 trials for LLM-based methods and 20 trials for RL-based methods.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: See Section F

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This research adheres fully to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: This work is foundational and focuses exclusively on improving the alignment of LLM-based world models and the planning efficiency of embodied agents. It does not directly address any societal applications or their consequences—positive or negative—since it is not tied to specific deployments or use cases. Therefore, there is no discussion of potential harms or benefits in the paper.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper does not introduce models or data with high risk of misuse; thus, safeguards are not applicable.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: See Appendix F.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide an anonymized URL in the abstract linking to the released assets, which include code, and documentation for reproducibility.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve any crowdsourcing or research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: This work does not involve human subjects or crowdsourcing.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are central to our method—they serve as the world model, induce symbolic knowledge, and generate executable rules. These roles are core to the originality and effectiveness of our approach.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

A Detailed Related Work

Using LLMs to Build World Models. Many studies have leveraged LLMs to construct world models for planning. For example, Ada [21] translates natural language instructions into adaptable planning representations via LLMs, enabling flexible and context-aware world modeling. Combining pre-trained LLMs with task-specific planning modules has also been shown to improve task success rates by providing a more detailed understanding of the environment [22]. Another approach, WorldCoder [19] exemplifies an LLM agent that constructs world models by generating and executing code to simulate various states and actions, refining its understanding iteratively.

These studies demonstrate the effectiveness of using LLMs to construct explicit world models from scratch to enhance planning and reasoning in complex environments. However, our key novelty lies in treating the LLM itself as the world model and aligning it with environment dynamics through training-free symbolic knowledge extraction. Rather than building a complete symbolic model from the ground up, we focus on identifying and correcting the misalignment between the base LLM's prior knowledge and the specific environment. This allows a small amount of complementary symbolic knowledge, combined with the LLM's existing priors, to form an accurate and generalizable world model. In contrast, prior methods rely on complex, environment-specific modeling procedures, which limits their scalability and adaptability to high-complexity environments like Mars.

Using LLMs as World Models. Several studies have explored using LLMs directly as world models by leveraging their implicit knowledge. Some methods rely on fine-tuning to align the LLM world model with the environment. For example, one approach fine-tunes LLMs using embodied experiences in simulation to improve reasoning and planning in embodied settings [23]. Another incorporates action preconditions and effects during fine-tuning to enable accurate outcome prediction and action reasoning [24].

Other approaches align LLMs as world models through prompting. LLM-MCTS [20] prompts LLMs to serve as both the policy and world model for large-scale task planning, integrating commonsense priors with guided search. RAP [8] uses prompting to guide LLMs in generating reasoning trees for structured decision making. RAFA [29] applies a Bayesian adaptive Markov Decision Process to prompt LLMs to predict future states for trajectory planning.

While these approaches demonstrate the potential of using LLMs as world models, they often require extensive fine-tuning or rely heavily on human-crafted prompts, making them labor-intensive and inflexible. Our approach leverages the LLM's inductive reasoning to automatically extract symbolic rules from exploration trajectories, drastically cutting human effort and computational overhead. This enables rapid, domain-agnostic adaptation across diverse environments.

A.1 Existing Agents in Minecraft

Although our method is evaluated in Mars, a 2D environment inspired by Minecraft, we also discuss prior agent methods developed in the Minecraft domain due to their conceptual relevance.

DEPS [41] uses an LLM to generate, explain, and revise plans, but it does NOT learn knowledge from previous tasks and generalize to new scenarios. Methods like GITM [44], MP5 [45], JARVIS-1 [39], and Voyager [43] use a memory buffer or skill library to directly store successful trajectories and reuse them later. However, no inductive reasoning is applied to these trajectories to learn condensed, generalized principles. So their agents have to reason based on many raw trajectories, which can suffer from poor generalization, overfitting, and inefficiency when adapted to new scenarios/tasks. Optimus-1 [46] partially addresses this issue by extracting knowledge graphs from historical trajectories, enabling more flexible knowledge transfer across tasks. However, it still relies heavily on logging trajectories and custom scripts tailored to a specific environment's data format for graph extraction, which cannot be easily generalized to new environments.

In contrast, our method does not store and reason directly based on raw trajectories. It overcomes these limitations by inductive reasoning using LLM on existing trajectories, condensing them into a compact, principal, generalizable, and comprehensive set of symbolic knowledge (action rules, knowledge graphs, and scene graphs) that can be reused efficiently and generalized across tasks/environments. This LLM-abstraction of environment is novel and critical to building an accurate world model. Another primary difference and novelty of WALL-E is to compile all symbolic knowledge

into executable code, which makes the planning verifiable with formal guarantees. This avoids possible hallucinations and mistakes of in-context learning by LLMs, which are common for planning in prior work.

A.2 Existing Agents in ALFWorld

ReAct [11], Reflexion [12], and AutoGen [14] leverage LLMs for reasoning, but they do NOT learn reusable knowledge from past tasks. AdaPlanner [40] introduces a skill library, but it merely stores trajectories from previous successful tasks. Reasoning solely based on these specific trajectories can mislead the agent and lead to suboptimal behavior in new scenarios. RAFA [29] relies on an LLM's prior knowledge for planning, which can be problematic when the environment dynamics and principles diverge from or conflict with LLM priors.

WALL-E differs from them in several key properties: Instead of simply recording past trajectories, WALL-E prompts the LLM to perform inductive reasoning over historical trajectories to extract compact and generalizable symbolic knowledge, as discussed above. Unlike RAFA, whose model LLM solely relies on LLM prior, WALL-E applies verifiable neurosymbolic learning to align the LLM's predictions with real environment dynamics. This "world alignment" greatly enhances both prediction accuracy and long-horizon planning capabilities.

B Detailed Prompt

B.1 Learn Action Rules from Real Trajectories

Prompt for Learning Action Rules from Real Trajectories

```
You are responsible for mining new rules from the given
    transitions, ensuring that these rules differ from the ones
    already provided. Focus on generating general and universal
    rules that are not tied to any specific item or tool. Your goal
     is to generalize across different objects, creating flexible
    rules that can be applied broadly to diverse contexts and
    situations.

I will give you an array of transitions:
[
    {
        'state_0': {
            "state feature 1": {"feature name": value, ...},
            ...
        },
        'action': {
            "name": "action name",
            "action feature 1": {"feature name": value, ...},
            ...
        },
        'action_result': {
        "feedback": "the environment feedback",
        "success": "Whether the action is executed successfully,",
        "suggestion": "..."
    }
    },
    ...
]
and an array of rules:
[
    "Rule 1: For action ..., if..., the action will fail; Checking
         Method: ...",
    ...
]

You should only respond in the format as described below:
RESPONSE FORMAT:
```

```
{
    "new_rules":[
        "Rule ...: For action ...,...; Checking Method: ...",
        "Rule ...: For action ...,...; Checking Method: ...",
        ...
    ]
}

Instructions:
- Ensure the response can be parsed by Python `json.loads`, e.g.:
    no trailing commas, **no single quotes**, etc.
- Please use you knowledge in <ENV>, do inductive reasoning. You
    need to dig up as many rules as possible that satisfy all
    transitions.
- Extract and utilize only the features that influence the outcome
     of the action.
- Please generate general and universal rules; the rules should
    not reference any specific item or tool! You need to generalize
     across various items or tools.
- Generate only the rules under what conditions the action will
    fail.
- While generating a rule, you also need to state how to check if
    a transition satisfies this rule. Please be specific as to
    which and how 'features' need to be checked
```

B.2 Translate Action Rules to Code

Prompt for Translating Action Rules to Code

```
You are responsible for generating code rules by implementing the
    learned rules in Python. Your task is to write a function that
    takes the current state and an action as inputs, optionally
    incorporating both the knowledge graph and the scene graph to
    provide additional contextual information (extra input),
    evaluating these conditions, and returns a Boolean value based
    on the specified rule. This function should effectively mirror
    the logic of the rules, enabling precise predictions for
    various state-action pairs.

The function should be defined as follows:

```python
def expected_rule_code(state, action, knowledge_graph/scene_graph)
    :
    # Your code here
    return feedback, success, suggestion
where
feedback: a string, give the action feedback based on success or
    not.
success: a bool, whether the action is executed successfully, give
     'True' or 'False'. If the action type is not the action type
    in the rule, count as success (e.g., success = True).
suggestion: a string, if the 'action' fails, 'suggestion' would be
     given based on 'rule', 'state' and 'action'.

Here are examples of the state and action format:
<Input Format>

For example:
    "Instead of obtaining [item] from [collecting_resource],
        players can acquire it from [alternative
        collecting_resource].",
    "To craft a [tool/object], players will need [
        crafting_material] and must use [crafting_platform].",
```
```

```
    "[crafting_material] needs to be gathered from [resource].",
    ...
Be sure to include such details to make the suggestions more
    engaging and relevant.

Knowledge Graph:
<Knowledge Graph>

Scene Graph:
<Scene Graph>

You should only respond in the format as described below, and do
    not give example usage or anything else:
RESPONSE FORMAT:
def expected_rule_code(state, action, knowledge_graph/scene_graph)
    :
    # Your code here
```

where "input format" please refer to Appendix C.

B.3 Learn Knowledge Graph from Real Trajectories

Prompt for Learning Knowledge Graph from Real Trajectories

```
You are a helpful assistant with inductive reasoning. Given the
    history trajectory, including action and observation, you need
    to reflect on the action execution results and identify and
    extract prerequisite or feasibility constraints, that is,
    discover when an action or item creation requires the presence
    of certain materials, resources, or other items.

We define the Knowledge Graph as:
{
  "V": "the set of entities (e.g., items, materials, location-
      specific objects, or abstract concepts)",
  "E": "the set of directed edges, each capturing a relationship
      or prerequisite among entities"
}

An edge takes the form:
(u, v, label),
where u and v are entities in V, and label indicates how u relates
     to v (for example, 'requires', 'consumes', 'collects', etc.).

I will give you an array of transitions:
[
    {
        'inital_state': '...',
        'action': '...',
        'action_result': "Whether the action is executed
            successfully, give 'True'
        or 'False' only"
    },
    {
        'inital_state': '...',
        'action': '...',
        'action_result': "Whether the action is executed
            successfully, give 'True'
        or 'False' only"
    },
    ...
]

You should ONLY respond in the following format:
```

```
{
    {'u':'entity_u', 'v':'entity_v', 'label':{'relation':'...', '
        quantity':'...'}},
    {'u':'entity_u', 'v':'entity_v', 'label':{'relation':'...', '
        quantity':'...'}},
    ...
}
example:
{'u':'wooden_sword', 'v':'table', 'label':{'relation':'requires',
    'quantity':None}},
{'u':'table', 'v':'wood', 'label':{'relation':'consumes', '
    quantity':'2'}}
```

C Environments' Observation Space and Action Space

The format of observation and action information is crucial for understanding the action rules we have extracted. In this section, we provide an description of the observation and action space used in different environments.

C.1 Mars

Observation Space. We collect observation information directly from the observation space provided by Mars [32]. The specific structure is illustrated in the following example.

Examples for Mars's Observation Space

```
obs = {
    "position": "grass",
    "in_front": "table",
    "visible_objects": [
        {
            "type": "plant",
            "x": -1,
            "y": 0
        },
        {
            "type": "grass",
            "x": 1,
            "y": 0
        },
        {
            "type": "table",
            "x": 0,
            "y": -1
        },...
    ],
    "near_objects": [
        "sand",
        "plant",
        "grass",
        "table"
    ],
    "status": {
        "health": 8,
        "food": 4,
        "drink": 5,
        "energy": 8
    },
    "inventory": {
        "sapling": 2,
        "wood_pickaxe": 2,
        "wood_sword": 1
```

```
    }
}
```

Action Space. We utilize the action space provided by the Mars directly, as demonstrated below.

Mars's Action Space

```
mine(block_name, amount) # mine amount blocks of the block_name.
attack(creature, amount) # attack the amount creature that can
    move. Creature include zombie, skeleton, cow, etc.
sleep(); # put the player to sleep.
place(block_name); # place the block. Note you need not craft
    table and furnace, you can place them directly.
make(tool_name); # craft a tool.
explore(direction, steps); # the player explore in the direction
    for steps.
```

C.2 ALFWorld

Observation Space. In the original ALFWorld setup, observation information is represented as natural language dialogue history. To facilitate the neurosymbolic learning process, we developed scripts to transform this dialogue history into a structured JSON format, as shown in the following example.

Examples for ALFWorld's Observation Space

```
obs = {
    "reachable_locations": [
        "cabinet 5",
        "cabinet 4",
        "cabinet 3",
        "cabinet 2",
        "cabinet 1",
        "coffeemachine 1",
        "countertop 2",
        "countertop 1",
        "diningtable 1",
        "drawer 2",
        "drawer 1",
        "fridge 1",
        "garbagecan 1",
        "microwave 1",
        "shelf 3",
        "shelf 2",
        "shelf 1",
        "sinkbasin 1",
        "stoveburner 4",
        "stoveburner 3",
        "stoveburner 2",
        "stoveburner 1",
        "toaster 1"
    ],
    "items_in_locations": {
        "fridge 1": [
            "lettuce 2",
            "mug 2",
            "potato 3"
        ],
        "microwave 1": []
    },
    "item_in_hand": {
        "item_name": "cup 1",
        "status": "normal"
```

```
    },
    "current_position": {
        "location_name": "microwave 1",
        "status": "open"
    }
}
```

Action Space. We utilize the action space provided by the ALFWorld directly, as demonstrated below.

Action Space for ALFWorld

```
go to [location/object]: Move to a specified location or object.
open [object]: Open a specified object like a cabinet or drawer.
close [object]: Close an opened object.
take [object] from [location]: Pick up an item from a specified
    location.
put [object] in/on [location]: Place an item in or on a specified
    location.
clean [object] with [location/tool]: Clean an object using a
    specific location or tool, like cleaning lettuce at the sink
    basin.
heat [object] with [tool]: Use an appliance, such as a microwave,
    to heat an item.
cool [object] with [tool]: Use a cooling tool or appliance, such
    as a fridge, to cool an item.
use [tool]: Activate or use a tool, such as a desklamp.
```

D Learned Rules

There are two points to note about the numbering of the action rules:

- The reason for duplicates is that the numbering is based on actions, and different actions have their own separate sequences. For example: Rules for Craft: [Rule 1, Rule 2, Rule 3, Rule 4, Rule 5...]; Rules for Mine: [Rule 1, Rule 2, Rule 3, Rule 4, Rule 5...].
- The reason the sequence may appear unordered is that some rules have been pruned (Section 3.1.3 Rule Set Pruning via Maximum Coverage). For instance, Rules for Craft where [Rule 1, Rule 2, Rule 4, Rule 5] has been removed, Rules for Mine where [Rule 1, Rule 3, Rule 4, Rule 5, Rule 6] has been removed, and the final rule set is Rules for Craft: [Rule 3, Rule 6] and Rules for Mine: [Rule 2, Rule 7].

Additionally, the feedback and suggestions returned by each code rule are automatically generated by prompting the LLM with the corresponding rule. The detailed prompts used to generate these code rules can be found in Appendix B.2. These feedback and suggestions play a crucial role in helping the agent refine and improve its planning process (Section 3.2).

D.1 Rules in Mars

Action Rules for Mars

```
Rule 6: For action 'make', if 'table' is not in 'near_objects',
    the action will fail; Checking Method: Check if 'table' is in
    the 'near_objects' list of the initial state.
Rule 7: For action 'make', if the player does not have the
    required materials in the inventory to craft the specified tool
    , the action will fail; Checking Method: Verify if the player's
     inventory contains the required materials for the tool
    specified in the 'make' action's 'tool_name' argument.
Rule 8: For action 'make', if the required resources for the tool
    are not present in the inventory or if a 'table' is not in '
    near_objects', the action will fail; Checking Method: Verify if
```

```
     the inventory contains the necessary resources for the tool
    being crafted and ensure that 'table' is present in '
    near_objects'.
Rule 1: For action 'place' with block_name 'sapling', if 'in_front
    ' is 'table', the action will fail; Checking Method: Check if '
    in_front' in the initial state is 'table'.
Rule 2: For action 'place', if the player does not have the
    required materials in the inventory to place the specified item
    , the action will fail; Checking Method: Verify if the player's
     inventory contains the required materials for the item
    specified in the 'place' action's 'block_name' argument.
Rule 3: For action 'place' with block_name 'table', if '
    visible_objects' or 'near_objects' already contain a 'table',
    the action should not be executed; Checking Method: Examine
    both the 'visible_objects' and 'near_objects' arrays in the
    state for an object with type 'table'. If found, do not proceed
     with placing a new table, and instead reuse the existing one.
Rule 2: For action 'mine', if the block_name is 'iron' and the
    player does not have 'stone_pickaxe' or better, the action will
     fail; Checking Method: Verify if the 'block_name' is 'iron'
    and ensure the inventory includes a 'stone_pickaxe' or better.
Rule 4: For action 'mine', if the player attempts to mine 'stone'
    and does not have a 'wood_pickaxe' or better in their inventory
    , the action will fail; Checking Method: Check if 'wood_pickaxe
    ' or a better tool is present in the inventory when attempting
    to mine 'stone'.
Rule 5: For action 'mine', if the block_name is 'plant', the
    action will fail; Checking Method: Check if 'block_name' in the
     action is 'plant'.
```

Code Rules for Mars

```
def CodeRule_6_make(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    args = action.get("args", {})

    # Check if the action is 'make'
    if action_name == "make":
        # Check if 'table' is in 'near_objects'
        if "table" not in state.get("near_objects", []):
            feedback = "Action failed: 'table' is not nearby."
            success = False
            suggestion = "Move closer to a 'table' to make the
                item."
            return feedback, success, suggestion

    # If the action is not 'make' or the condition is met
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_7_make(state, action, knowledge_graph):
    # Check if the action is 'make'
    if action['action_name'] == 'make':
        tool_name = action['args']['tool_name']

        # Use LLM_request to find out the required materials for
            the tool
        question = f"What are the required materials to craft a {
            tool_name}?"
        response_format = "Provide a list of materials and their
            quantities."
```

```
        required_materials = LLM_request(question +
            response_format)

        # Check if the player has the required materials in the
            inventory
        inventory = state['inventory']
        has_all_materials = True
        missing_materials = []

        for material, quantity in required_materials.items():
            if inventory.get(material, 0) < quantity:
                has_all_materials = False
                missing_materials.append(f"{material}: {quantity -
                inventory.get(material, 0)} more needed")

        if has_all_materials:
            feedback = "Action executed successfully."
            success = True
            suggestion = ""
        else:
            feedback = "Action failed: Not enough materials to
                craft the tool."
            success = False
            suggestion = f"To craft a {tool_name}, you need:
            {', '.join(missing_materials)}."
    else:
        # If the action is not 'make', consider it successful
        feedback = "Action executed successfully."
        success = True
        suggestion = ""

    return feedback, success, suggestion

def CodeRule_8_make(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    args = action.get("args", {})

    # Check if the action is 'make'
    if action_name != "make":
        return "Action executed successfully.", True, ""

    # Extract tool name
    tool_name = args.get("tool_name")

    # Check if 'table' is in 'near_objects'
    if 'table' not in state.get("near_objects", []):
        feedback = "Action failed: A 'table' is required nearby to
             make the tool."
        suggestion = "Move closer to a 'table' and try again."
        return feedback, False, suggestion

    # Check if the required resources are in the inventory
    required_resources = knowledge_graph.get(tool_name, {})
    inventory = state.get("inventory", {})

    for resource, amount in required_resources.items():
        if inventory.get(resource, 0) < amount:
            feedback = f"Action failed: Not enough {resource} to
                make {tool_name}."
            suggestion = f"Collect more {resource} to make {
                tool_name}."
            return feedback, False, suggestion

    # If all checks pass
```

```
    return "Action executed successfully.", True, ""

def CodeRule_1_place(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    block_name = action.get("args", {}).get("block_name")

    # Initialize feedback, success, and suggestion
    feedback = ""
    success = True
    suggestion = ""

    # Rule 1: For action 'place' with block_name 'sapling', if '
        in_front' is 'table', the action will fail
    if action_name == "place" and block_name == "sapling":
        if state.get("in_front") == "table":
            feedback = "Action failed: Cannot place a sapling in
                front of a table."
            success = False
            suggestion = "Try placing the sapling in front of a
                different object, such as grass or dirt."
        else:
            feedback = "Action succeeded: Sapling placed
                successfully."

    # If the action type is not 'place', count as success
    else:
        feedback = "Action succeeded: No rules apply to this
            action type."

    return feedback, success, suggestion

def CodeRule_2_place(state, action, knowledge_graph):
    # Check if the action is 'place'
    if action['action_name'] == 'place':
        block_name = action['args']['block_name']

        # Check if the required material is in the inventory
        if block_name in state['inventory'] and state['inventory
            ']['block_name] > 0:
            feedback = f"Successfully placed {block_name}."
            success = True
            suggestion = ""
        else:
            feedback = f"Failed to place {block_name}."
            success = False
            suggestion = f"You need to have {block_name} in your
                inventory to place it. Consider collecting or
                crafting it first."
    else:
        # If the action is not 'place', consider it successful
        feedback = "Action executed successfully."
        success = True
        suggestion = ""

    return feedback, success, suggestion

def CodeRule_3_place(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    block_name = action.get("args", {}).get("block_name")

    # Check if the action is 'place' and block_name is 'table'
    if action_name == "place" and block_name == "table":
```

```
        # Check for existing 'table' in 'visible_objects' or '
            near_objects'
        visible_objects = state.get("visible_objects", [])
        near_objects = state.get("near_objects", [])

        # Check if 'table' is in visible_objects
        table_in_visible = any(obj.get("type") == "table" for obj
            in visible_objects)
        # Check if 'table' is in near_objects
        table_in_near = "table" in near_objects

        # If a table is found in either, the action should not be
            executed
        if table_in_visible or table_in_near:
            feedback = "Action failed: A table is already present
                in the vicinity."
            success = False
            suggestion = "Reuse the existing table instead of
                placing a new one."
            return feedback, success, suggestion

    # If the action is not 'place' or block_name is not 'table',
        consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_2_mine(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    block_name = action.get("args", {}).get("block_name")

    # Check if the action is 'mine' and the block_name is 'iron'
    if action_name == "mine" and block_name == "iron":
        # Check if the player has a 'stone_pickaxe' or better in
            the inventory
        inventory = state.get("inventory", {})
        has_stone_pickaxe_or_better = any(
            tool in inventory for tool in ["stone_pickaxe", "
                iron_pickaxe",
            "diamond_pickaxe"]
        )

        if not has_stone_pickaxe_or_better:
            feedback = "Action failed: You need a stone_pickaxe or
                 better to mine iron."
            success = False
            suggestion = "Consider crafting or acquiring a
                stone_pickaxe or better to mine iron."
            return feedback, success, suggestion

    # If the action is not 'mine' or the block_name is not 'iron',
         consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_4_mine(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    block_name = action.get("args", {}).get("block_name")
    # Check if the action is 'mine' and the block is 'stone'
    if action_name == "mine" and block_name == "stone":
```

```
        # Check if 'wood_pickaxe' or a better tool is in the
            inventory
        inventory = state.get("inventory", {})
        has_wood_pickaxe = 'wood_pickaxe' in inventory
        better_tool_exists = False
        for tool in inventory:
            if tool != 'wood_pickaxe':
                response = LLM_request(f"is {tool} better than
                    wood_pickaxe?" +
                "only reply True or False")
                if response == "True":
                    better_tool_exists = True
                    break
        if has_wood_pickaxe or better_tool_exists:
            feedback = "Action executed successfully."
            success = True
            suggestion = ""
        else:
            feedback = "Action failed: You need a wood_pickaxe or
                a better tool to mine stone."
            success = False
            suggestion = "Consider crafting or acquiring a
                wood_pickaxe or a better tool to mine stone."
    else:
        feedback = "Action executed successfully."
        success = True
        suggestion = ""
    return feedback, success, suggestion

def CodeRule_5_mine(state, action, knowledge_graph):
    # Extract action details
    action_name = action.get("action_name")
    block_name = action.get("args", {}).get("block_name")

    # Initialize feedback, success, and suggestion
    feedback = ""
    success = True
    suggestion = ""

    # Check if the action is 'mine' and block_name is 'plant'
    if action_name == "mine" and block_name == "plant":
        feedback = "Action failed: You cannot mine a plant."
        success = False
        suggestion = "Consider mining other resources like 'tree'
            or 'stone'
        instead of 'plant'."

    else:
        feedback = "Action executed successfully."

    return feedback, success, suggestion
```

D.2 Rules in ALFWorld

Action Rules for ALFWorld

```
Rule 1: For action 'take', if the item is not present in the
    specified location, the action will fail; Checking Method:
    Check if the 'obj' in 'action' is not listed under '
    items_in_locations' for the 'source' location in 'inital_state
    '.
Rule 2: For action 'take', if the agent's hand is already holding
    an item, the action will fail; Checking Method: Check if '
    item_in_hand' in 'inital_state' has a non-null 'item_name'.
```

```
Rule 3: For action 'take', if the agent is not at the specified
    location, the action will fail; Checking Method: Check if '
    current_position' in 'inital_state' does not match the 'source'
     location in 'action'.
Rule 2: For action 'open', if the target location is not in the
    list of reachable locations, the action will fail; Checking
    Method: Check if 'action.args.target' is not in '
    reachable_locations'.
Rule 7: For action 'open', if the target location is a fridge and
    the current position is not the same as the target location,
    the action will fail; Checking Method: Check if 'action.args.
    target' is 'fridge' and 'current_position.location_name' is not
     equal to 'action.args.target'.
Rule 10: For action 'open', if the target location is a drawer and
     the current position is not the same as the target location,
    the action will fail; Checking Method: Check if 'action.args.
    target' starts with 'drawer' and 'current_position.
    location_name' is not equal to 'action.args.target'.
Rule 3: For action 'go to', if the item specified by 'item_name'
    in 'item_in_hand' is None and the action's 'target' in the
    Scene Graph is not 'Unexplored' or does not contain the state's
     'target_item' in the list, the action will fail; Checking
    Method: Check if 'item_in_hand.item_name' is None, and validate
     if the 'target' in 'action' is either not labeled 'Unexplored'
     or does not include the 'target_item' in the Scene Graph.
Rule 2: For action 'heat', if the object to be heated is not in
    the tool, the action will fail; Checking Method: Check if the '
    item_in_hand' matches the object to be heated and is not placed
     in the tool location.
Rule 2: For action 'put', if the item to be put is not in hand,
    the action will fail; Checking Method: Check if 'item_name' in
    'item_in_hand' of the 'inital_state' matches the 'obj' in
    action args.
Rule 4: For action 'put', if the current position is not the same
    as the target location, the action will fail; Checking Method:
    Check if 'location_name' in 'current_position' of the '
    inital_state' matches the 'target' in action args.
Rule 2: For action 'use', if the object is not in the current
    location, the action will fail; Checking Method: Check if the
    object specified in the action is not present in the '
    items_in_locations' of the 'current_position' in the initial
    state.
Rule 4: For action clean, if the object to be cleaned is not in
    hand, the action will fail; Checking Method: Check if the '
    item_in_hand' matches the object specified in the action.
```

Code Rules for ALFWorld

```
def CodeRule_1_take(state, action, scene_graph):
    # Extract action details
    action_name = action.get("name")
    action_args = action.get("args", {})
    # Check if the action is 'take'
    if action_name == "take":
        obj = action_args.get("obj")
        source = action_args.get("source")
        # Check if the object is present in the specified source
            location
        items_in_source = state.get("items_in_locations", {}).get(
            source, [])
        if obj not in items_in_source:
            feedback = f"Action failed: {obj} is not present in {
                source}."
            success = False
```

```
            suggestion = f"Check the items in {source} and try
                taking an available item."
            return feedback, success, suggestion
    # If the action is not 'take', consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_2_take(state, action, scene_graph):
    # Extract the action name
    action_name = action.get("name")
    # Check if the action is 'take'
    if action_name == "take":
        # Check if the agent's hand is already holding an item
        item_in_hand = state.get("item_in_hand", {}).get("
            item_name")
        if item_in_hand:
            feedback = "Action failed: The agent is already
                holding an item."
            success = False
            suggestion = "Put down the current item before taking
                another one."
            return feedback, success, suggestion
    # If the action is not 'take', consider it successful by
        default
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_3_take(state, action, scene_graph):
    if action['name'] == 'take':
        current_location = state['current_position']['
            location_name']
        source_location = action['args']['source']
        if current_location != source_location:
            feedback = "Action failed: You are not at the
                specified location to take the item."
            success = False
            suggestion = f"Move to {source_location} before
                attempting to take the item."
            return feedback, success, suggestion
    # If the action is not 'take', consider it successful by
        default
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_2_open(state, action, scene_graph):
    if action['name'] == 'open':
        target = action['args']['target']
        if target not in state['reachable_locations']:
            feedback = f"Action failed: The target location '{
                target}' is not reachable."
            success = False
            suggestion = f"Try moving closer to '{target}' before
                attempting to open it."
            return feedback, success, suggestion
    # If the action is not 'open', consider it successful by
        default
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
```

```
    return feedback, success, suggestion

def CodeRule_7_open(state, action, scene_graph):
    # Extract necessary information from state and action
    current_position = state["current_position"]["location_name"]
    action_name = action["name"]
    target = action["args"].get("target", "")

    # Check if the action is 'open' and the target is a fridge
    if action_name == "open" and "fridge" in target:
        # Check if the current position is not the same as the
            target location
        if current_position != target:
            feedback = "Action failed: You must be at the fridge
                to open it."
            success = False
            suggestion = f"Move to {target} before trying to open
                it."
            return feedback, success, suggestion

    # If the action is not 'open' or the rule does not apply,
        consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_10_open(state, action, scene_graph):
    # Extract necessary information from state and action
    current_position = state["current_position"]["location_name"]
    action_name = action["name"]
    target_location = action["args"].get("target", "")

    # Check if the action is 'open' and the target is a drawer
    if action_name == "open" and target_location.startswith("
        drawer"):
        # Check if the current position is not the same as the
            target location
        if current_position != target_location:
            feedback = "Action failed: You must be at the drawer
                to open it."
            success = False
            suggestion = f"Move to {target_location} before trying
                 to open it."
            return feedback, success, suggestion

    # If the action is not 'open' or the rule does not apply,
        consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_3_go_to(state, action, scene_graph):
    # Extract necessary information from state and action
    item_in_hand = state["item_in_hand"]["item_name"]
    target_item = state["target_item"]
    action_name = action["name"]

    # Check if the action is 'go_to'
    if action_name == "go to":
        target_location = action["args"]["target"]
        # Check if item in hand is None
        if item_in_hand is None:
```

```
            # Check if the target location is not 'Unexplored' and
                does not contain the target item
            if target_location in scene_graph["locations"]:
                location_items = scene_graph["locations"][
                    target_location]
                if "Unexplored" not in location_items and not any(
                    target_item in item for item in location_items)
                    :
                    # Search for a location containing the target
                        item
                    location_with_target_item = None
                    unexplored_locations = []

                    for location, items in scene_graph["locations
                        "].items():
                        if any(target_item in item for item in
                            items):
                            location_with_target_item = location
                            break  # Stop searching once a
                                location with the target item is
                                found
                        if "Unexplored" in items:
                            unexplored_locations.append(location)

                    # Prepare suggestion string
                    if location_with_target_item:
                        suggestion = f"According to scene graph.
                            Please go to the location containing
                            the target item: {
                            location_with_target_item}."
                    elif unexplored_locations:
                        suggestion = (
                            "According to scene graph. "
                            "Please explore the following
                                unexplored locations: "
                            + ", ".join(unexplored_locations) +
                                "."
                        )
                    else:
                        suggestion = "No valid locations found
                            that contain the target item or are
                            unexplored."

                    feedback = f"Action failed: There is no {
                        target_item} in {target_location} "
                    success = False
                    return feedback, success, suggestion

    # If the action is not 'go_to' or the conditions are not met,
        consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_2_heat(state, action, scene_graph):
    # Extract necessary information from the state and action
    action_name = action.get("name")
    action_args = action.get("args", {})
    obj_to_heat = action_args.get("obj")
    tool = action_args.get("tool")
    # Check if the action is 'heat'
    if action_name == "heat":
        # Check if the object to be heated is in hand and not in
            the tool location
```

```
        item_in_hand = state.get("item_in_hand", {}).get("
            item_name")
        current_position = state.get("current_position", {}).get("
            location_name")
        if item_in_hand == obj_to_heat and current_position !=
            tool:
            feedback = f"Failed to heat {obj_to_heat}. It must be
                placed in {tool} to be heated."
            success = False
            suggestion = f"Place {obj_to_heat} in {tool} before
                heating."
            return feedback, success, suggestion
    # If the action is not 'heat', consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_2_put(state, action, scene_graph):
    if action['name'] == 'put':
        obj_to_put = action['args']['obj']
        item_in_hand = state['item_in_hand']['item_name']
        if obj_to_put != item_in_hand:
            feedback = f"Action failed: {obj_to_put} is not in
                hand."
            success = False
            suggestion = f"Ensure you have {obj_to_put} in hand
                before attempting to put it."
            return feedback, success, suggestion
    # If the action is not 'put', consider it successful by
        default
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_4_put(state, action, scene_graph):
    if action['name'] == 'put':
        current_location = state['current_position']['
            location_name']
        target_location = action['args']['target']
        if current_location != target_location:
            feedback = "Action failed: You must be at the target
                location to put the item."
            success = False
            suggestion = f"Move to {target_location} before
                attempting to put the item."
            return feedback, success, suggestion
    # If the action is not 'put', consider it successful by
        default
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_2_use(state, action, scene_graph):
    if action['name'] == 'use':
        obj = action['args']['obj']
        current_location = state['current_position']['
            location_name']
        # Check if the object is in the current location
        if obj not in state['items_in_locations'].get(
            current_location, []):
            feedback = f"Action failed: {obj} is not in the
                current location {current_location}."
```

```
            success = False
            suggestion = f"Move to the location where {obj} is
                present or bring {obj} to the current location."
            return feedback, success, suggestion
    # If the action is not 'use', consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion

def CodeRule_4_clean(state, action, scene_graph):
    # Extract the action name and arguments
    action_name = action.get("name")
    action_args = action.get("args", {})

    # Check if the action is 'clean'
    if action_name == "clean":
        # Get the object to be cleaned from the action arguments
        obj_to_clean = action_args.get("obj")

        # Get the item currently in hand
        item_in_hand = state.get("item_in_hand", {}).get("
            item_name")

        # Check if the object to be cleaned is in hand
        if obj_to_clean != item_in_hand:
            feedback = f"Action failed: {obj_to_clean} is not in
                hand."
            success = False
            suggestion = f"Please ensure {obj_to_clean} is in hand
                 before cleaning."
            return feedback, success, suggestion

    # If the action is not 'clean', consider it successful
    feedback = "Action executed successfully."
    success = True
    suggestion = ""
    return feedback, success, suggestion
```

Algorithm 3 Greedy Algorithm for the Maximum Coverage Problem in Eq. (5)

Input: $\mathcal{D}^{\text{inc}} = \{\delta_1^{\text{inc}}, \delta_2^{\text{inc}}, .., \delta_N^{\text{inc}}\}$, $\mathcal{R}^{\text{code}} = \{r_1^{\text{code}}, r_2^{\text{code}}, .., r_M^{\text{code}}\}$, a_{ij}: Indicator matrix where $a_{ij} = 1$ if $\delta_j^{\text{inc}} \in r_i^{\text{code}}$, otherwise $a_{ij} = 0$
Initialize $\mathcal{R}^* \leftarrow \emptyset$, $\mathcal{D}^{cov} \leftarrow \emptyset$
while $\mathcal{D}^{cov} \neq \mathcal{D}^{\text{inc}}$ **do**
  For each rule $r_i^{\text{code}} \in \mathcal{R}^{\text{code}}$, compute:

$$\text{gain}(r_i^{\text{code}}) = \left|\mathcal{D}^{cov} \cup \{\delta_j^{\text{inc}} \in \mathcal{D}^{\text{inc}} : a_{ij} = 1\}\right| - |\mathcal{D}^{cov}|$$

  Get the index of r_i^{code} with the largest gain, i.e.,

$$i^* \leftarrow \arg\max \text{gain}(r_i^{\text{code}})$$

  if $\text{gain}(r_{i^*}^{\text{code}}) = 0$ **then**
    Break {Terminate if no r_i^{code} can cover any additional δ^{inc} }
  end if
  Add $r_{i^*}^{\text{code}}$ to the selected rules set:

$$\mathcal{R}^* \leftarrow \mathcal{R}^* \cup \{r_{i^*}^{\text{code}}\}$$

  Update the covered set:

$$\mathcal{D}^{cov} \leftarrow \mathcal{D}^{cov} \cup \{\delta_j^{\text{inc}} \in \mathcal{D}^{\text{inc}} : a_{i^*j} = 1\}$$

  if $|\mathcal{R}^*| = l$ **then**
    Break {Terminate if hit the limit l}
  end if
end while
Output: Set of selected rules $\mathcal{R}^*$

E Greedy Algorithm

We implement the following Algorithm 3 to solve the maximum coverage problem 5.

F Experiment Details

Environment Licenses. Both the Mars and ALFWorld environments are released under the MIT License, and our experiments fully comply with their respective licensing terms. All code, assets, and interaction logs used in this work adhere to the usage guidelines provided by the original environment authors.

Compute Resources. All experiments were conducted on a local Ubuntu 20.04.6 workstation with an Intel i5-13600KF CPU, 64GB RAM, and a single NVIDIA RTX 4060 Ti 16GB GPU. Each experimental run (e.g., one evaluation trial) typically completed within 5–10 minutes. The evaluation required less than 24 hours in total.

F.1 Mars

Task Details. Mars is an open-world environment designed for situated inductive reasoning, where agents must actively interact with their surroundings, induce generalizable rules, and apply them to achieve specific goals. Unlike traditional environments that rely on pre-existing commonsense knowledge, Mars introduces counter-commonsense mechanisms by modifying terrain distributions, survival settings, and task dependencies.

The agent's goal is to unlock various achievements, such as:

- Collecting Achievements: Collect Coal, Collect Diamond, Collect Drink, Collect Iron, Collect Sapling, Collect Stone, Collect Wood.

- Crafting Achievements: Make Wooden Pickaxe, Make Stone Pickaxe, Make Iron Pickaxe, Make Wooden Sword, Make Stone Sword, Make Iron Sword.
- Placing Achievements: Place Table, Place Furnace, Place Plant, Place Stone.
- Survival and Combat Achievements: Kill Skeleton, Kill Zombie, Kill Cow, Eat Plant, Wake Up.

However, achieving these goals requires adaptive reasoning since the default assumptions about item acquisition and crafting may no longer hold. For example: Mars introduces counter-commonsense modifications to challenge agents' reliance on prior knowledge. These modifications fall into three categories: Terrain, Survival Settings, and Task Dependencies, which can be combined to create diverse worlds.

- Terrain Modifications:
  - Distribution: Resources appear in unexpected locations (e.g., diamonds in sand, coal near grass).
  - Effects: Terrain properties change (e.g., lava is safe, grass is harmful, mining stone gives wood).
- Survival Setting Changes:
  - Entity Behavior: Cows may be aggressive, zombies passive, skeletons use melee instead of bows.
  - Food and Health: Eating cows might reduce health, drinking lava could restore it.
- Task Dependency Alterations:
  - Resource Collection: Mining may yield unexpected materials (e.g., trees drop iron, stone gives coal).
  - Crafting Changes: Items require different materials (e.g., tables need diamonds, pickaxes need iron).
  - Placement Rules: Placing objects may consume extra resources (e.g., tables require two diamonds).

By combining these modifications, Mars forces agents to learn dynamically, making pre-stored knowledge unreliable and requiring real-time adaptation.

Each episode in Mars generates a unique $64 * 64$ grid-based world where agents operate under partial observability ($7 * 9$ grid view). The environment ensures task achievability by maintaining resource balance, enforcing supply constraints, and allowing procedural rule induction.

Baselines. We compare our method against ReAct [37], which interleaves reasoning and actions; Reflexion [38], an extension of ReAct with self-reflection; Skill Library [32], an adaptation of JARVIS-1 [39] and Voyager [43] that stores successful plans for in-context learning; and IfR [32], which extends Skill Library with an induction-from-reflection module that derives and stores game rules for adaptive decision-making. These methods' planning framework and components are shown in Table 6.

Table 6: Comparison of baselines' planning framework with different components.

Method	Reasoning	Experience Reflector	Memory/Skill Library	Rules	Knowledge Graph	Scene Graph
ReAct [37]	✓					
Reflexion [38]	✓	✓				
Skill Library [39]	✓	✓	✓			
IfR [32]	✓	✓	✓	✓		
WALL-E (ours)	✓	✓		✓	✓	✓

Method Setup. We use GPT-4 as the backend for our method. Following the Mars setup, we run 5 learning episodes and report results for LLM-based methods and RL-based baselines averaged over 9 and 20 trials, respectively.

F.2 ALFWorld

Task Details. ALFWorld is a text-based household simulator aligned with the 3D ALFRED benchmark [42]. Each episode gives the agent a natural-language goal (e.g., "put the cooled lettuce on the countertop"). Agents issue high-level text commands such as "go to countertop 1," "take lettuce 1 from countertop 1," or "cool lettuce 1 with fridge 1," and receive textual observations describing their egocentric view, nearby objects, inventory contents, and action results. The agent's internal state is the running history of these observations.

ALFWorld includes six task families: Pick & Place, Pick & Examine, Cool & Place, Heat & Place, Clean & Place, Pick-Two & Place, mirroring the ALFRED dataset's diversity of object manipulations and state changes.

Method Setup. We use GPT-3.5-Instruct as the backbone model and perform neurosymbolic learning on the training set. Additionally, to enhance neurosymbolic learning, we developed scripts to convert the natural language dialogue history and action information into a structured JSON format, as illustrated in Appendix C.2. Evaluation is conducted on all 134 ALFWorld test tasks, following the evaluation protocol used by prior ALFWorld agents A.2.

F.3 Experiment Design for Effectiveness of NeuroSymbolic Learning

We set the number of learning episodes to 5. After each episode, the model, equipped with latest learned code rules or skill library, is tested on the testing set. The cover rate quantifies the extent to which the code rules derived from the neurosymbolic learning process address the LLM's failed predictions. Specifically, it represents the probability that mispredicted transitions by the LLM are correctly handled by the learned code rules.

To assess the alignment between the LLM-based world model and the actual environment, we first identify transitions where the LLM fails to make accurate predictions. This is achieved by utilizing an unaligned LLM world model to generate predictions for trajectories obtained from the test set. The discrepancies between the predicted observation $\hat{o}_{t+1}$ and the actual observation o_{t+1} are compiled into a set of mispredicted transitions. These mispredictions highlight areas where the LLM world model does not align with the environment's dynamics.

Subsequently, the learned code rules at each iteration are evaluated against the mispredicted transitions dataset to determine their effectiveness in correcting these mispredictions. If a code rule successfully predicts the outcome of a previously mispredicted transition, it demonstrates that the code rule effectively addresses the LLM's failure in that instance. The cover rate is then calculated as the ratio of correctly addressed mispredictions to the total number of mispredicted transitions:

$$\text{Cover Rate} = \frac{\text{Number of Mispredictions Addressed by Code Rules}}{\text{Total Number of Mispredicted Transitions}} \tag{12}$$

A higher cover rate indicates that the neurosymbolic learning process effectively enhances the alignment of the LLM world model with the environment, thereby improving the overall accuracy and reliability of the agent's planning.

G Limitation and Future Work

Our current neurosymbolic learning framework exhibits two limitations. First, it focuses on learning transition-level code rules that determine whether a transition succeeds or fails. While effective for aligning local dynamics, these code rules lack the expressiveness to capture more abstract, long-horizon planning constraints or temporally extended dependencies. Future work will explore inducing higher-level symbolic structures—such as hierarchical knowledge or temporal logic—that can support more complex reasoning and structured planning.

Second, the framework assumes deterministic dynamics, meaning that each action in a given state is expected to produce a single, predictable outcome. However, many embodied environments involve inherent stochasticity—for instance, actions in Mars may occasionally succeed or fail depending on contextual factors like time of day or agent health. Our current neurosymbolic learning framework does not account for such probabilistic variations, leading to potentially misleading failure classifications. Extending the framework to support probabilistic transition modeling is a key direction for improving robustness.