# CoEx – Co-evolving World-model and Exploration

**Minsoo Kim**
Seoul National University
minsoo9574@snu.ac.kr

**Seung-won Hwang**[*]
Seoul National University
seungwonh@snu.ac.kr

## Abstract

Planning in modern LLM agents relies on the utilization of LLM as an internal world model, acquired during pretraining. However, existing agent designs fail to effectively assimilate new observations into dynamic updates of the world model. This reliance on the LLM's static internal world model is progressively prone to misalignment with the underlying true state of the world, leading to the generation of divergent and erroneous plans. We introduce a hierarchical agent architecture, CoEx, in which hierarchical state abstraction allows LLM planning to co-evolve with a dynamically updated model of the world. CoEx plans and interacts with the world by using LLM reasoning to orchestrate dynamic plans consisting of subgoals, and its learning mechanism continuously incorporates these subgoal experiences into a persistent world model in the form of a neurosymbolic *belief state*, comprising textual inferences and code-based symbolic memory. We evaluate our agent across a diverse set of agent scenarios involving rich environments and complex tasks including ALFWorld, PDDL, and Jericho. Our experiments show that CoEx outperforms existing agent paradigms in planning and exploration.[1]

## 1 Introduction

While recent advances in large language model (LLM) capabilities have enabled significant performance gains in LLM-based agents, such agents fall short when tasked with achieving goals in novel environments with limited prior information. Meanwhile, humans particularly excel in such scenarios, actively **exploring** environments and **adapting** plans based on new observations (Lake et al., 2017; Ha and Schmidhuber, 2018). With exploration,

the understanding of the environment, or *world model* (Ha and Schmidhuber, 2018; Hafner et al., 2021), is continuously adapted, and planning is grounded on such understanding.

We attribute the limitations of existing LLM agents like ReAct (Yao et al., 2023) and Reflexion (Shinn et al., 2023) in novel environments, to their **monolithic** design, which entangles planning, reasoning, and action generation within a single LLM instance. Instantiated through in-context learning (ICL) with action-level few-shot exemplars, this design faces two fundamental limitations:

- Exploitation Bias: ICL, driven by demonstrations of successful trajectories, biases the agent toward repeating known successful actions (i.e., exploitation) over exploration. With this bias, action-level planning results in a myopic horizon, limiting generalization to longer horizon tasks.

- Limited Adaptation: The monolithic architecture complicates integration of new exploratory insights into a persistent world model.

In contrast, we argue that agents need both the ability to plan **exploratory goals**, and the ability to directly leverage the outcome of exploration to update an **adaptive world model**.

We introduce our proposed framework, **Co**-evolving World-model and **Ex**ploration (CoEx), which formulates LLM agent planning as a state machine over a subgoal-level belief state. Our first distinction is a Planner which conditions on a **subgoal-level** belief state, rather than low-level actions. This enables CoEx to leverage subgoals to plan deliberate exploration goals when faced with uncertainty. An additional flexibility of our approach is dynamic planning based on exploration,

---

[*]Corresponding author.
[1]https://github.com/mnskim/coex

where subgoals can be generated anew when the agent deems the current plan suboptimal.

Our second distinction is a robust mechanism for **world model co-evolution** through a process of verification and synthesis. Following each subgoal attempt, CoEx distills task-relevant insights from raw experience, using LLM-based filtering to generate targeted updates to an adaptive belief state.

Specifically, we propose a neurosymbolic belief state design, combining:

- Object-oriented symbolic memory for efficient low-level state tracking and,

- LLM-based verification and synthesis for integrating new discoveries into the adaptive world model.

Through this unified subgoal-level planning and exploration strategy, CoEx achieves co-evolving agent planning and adaptive world modeling, addressing the core limitations of monolithic LLM designs. To demonstrate the effectiveness of CoEx, we evaluate our proposed method on a diverse set of challenging agent planning scenarios including ALFWorld (Shridhar et al., 2021), Jericho (Hausknecht et al., 2020), and PDDL (McDermott et al., 1998) domains.

## 2    Related Work

In Fig. 1, we illustrate the landscape of existing LLM agent paradigms along the dimensions of planning granularity and world model adaptation. As overviewed in Section 1, existing LLM agents with static world model and myoptic action-level planning fall in the **lower-left** (Fig. 1), ReAct (Yao et al., 2023) and Reflexion (Shinn et al., 2023), relying on a monolithic agent design which conflates planning, reasoning, and action generation into a singular LLM agent. ExpeL (Zhao et al., 2024) implements an offline form of world model updates, but preclude real-time adaptation. WALL-E (Zhou et al., 2024) proposes offline rule learning for world model alignment, extracting symbolic rules from collected trajectories, but similarly does not support online adaptation.

Toward the **desired upper-right**, where ours is placed, existing efforts can be categorized by two directions. First, **world model update (upper left)**: The upper-left shows approaches leveraging world model for planning, repurposing LLMs as world models directly (Hao et al., 2023), or using LLMs
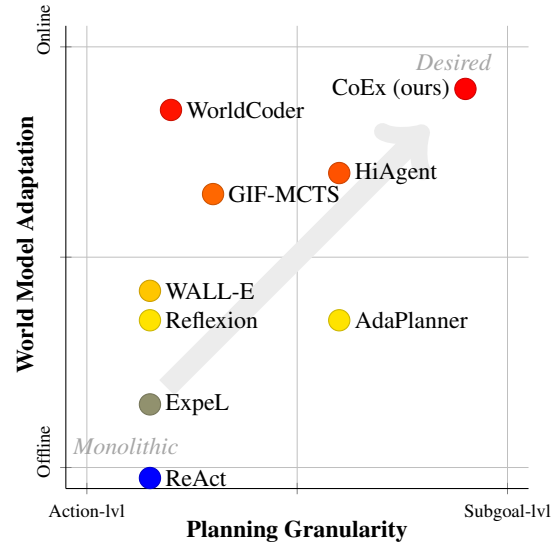


Figure 1: Landscape of LLM agent paradigms along the dimension of planning granularity (x-axis) ranging from action-level to subgoal-level, and frequency of world model adaptation (y-axis), ranging from offline to online.

to generate code-based world models (Tang et al., 2024; Dainese et al., 2024). In particular, the latter can represent adaptive knowledge as code (Tang et al., 2024; Dainese et al., 2024) in an online manner. However, by LLM strictly taking the role of code generator, LLM reasoning abilities cannot be leveraged for planning, instead delegating planning to external algorithms such as MCTS. In addition, they focus entirely on modeling action-level dynamics of the world, and in practice are limited to demonstrating basic competence in complex planning tasks such as AlfWorld (Tang et al., 2024).

Second, **subgoal-level planning (lower right)**: AdaPlanner (Sun et al., 2023) introduces dynamic plan improvement based on episodic and execution feedback, but rely on handcrafted examples to demonstrate plan adaptation in detail, and do not support learning with an explicit world model. HiAgent (Hu et al., 2025) employs subgoal-level planning, but similarly relies on implicit world models within monolithic agents, using subgoals as memory chunks and observation summarization to reduce context redundancy. Table 1 compares in CoEx with existing LLM agent frameworks in further detail.

## 3    Background

In this section, we discuss the notions of exploitation bias and limited adaptation within the context of LLM agents. We begin with a general reinforcement learning (RL) formulation of LLM agent's in-

Table 1: Comparison of CoEx with representative LLM agent architectures.

| Method | World Model | Planning levels | Exploration Mechanism |
|---|---|---|---|
| ReAct (Yao et al., 2023) | Implicit | Action-level | ✗ |
| ExpeL (Zhao et al., 2024) | Offline Insights | Action-level | ✗ |
| WALL-E (Zhou et al., 2024) | Offline Learned Rules | Action-level | ✗ |
| Reflexion (Shinn et al., 2023) | Episodic Self-reflection | Action-level | ✗ |
| AdaPlanner (Sun et al., 2023) | Implicit | Subgoal-level and Action-level (Monolithic LLM) | ✗ |
| HiAgent (Hu et al., 2025) | Implicit | Subgoal-level and Action-level (Monolithic LLM) | ✗ |
| CoEx (Ours) | Adaptive Belief State updated after subgoal | Subgoal-level (Planner) and Action-level (Actor) | Exploratory subgoals generated by Planner |

teraction with an environment, which can represent a wide range of interactive tasks with text-based observations (Côté et al., 2018; Hausknecht et al., 2020; Li et al., 2024).

An LLM agent task can be defined as a Partially Observable Markov Decision Process (POMDP), represented by the tuple $(S, A, O, \mathrm{T}, \Omega, \mathrm{R}, \gamma)$, where $S$ is the set of environment states $s$, $A$ is the set of actions $a$, and $O$ is the set of possible observations $o$. $T$ is the transition probability between states, $\Omega$ is the observation probability, $R$ the reward function, and $\gamma$ the discount factor, respectively.

### 3.1 Exploitation Bias: In-context Learning in LLM Agents

In standard LLM agent paradigms such as Re-Act (Yao et al., 2023) and Reflexion (Shinn et al., 2023) the agent is defined by a singular action-level policy $\pi(a_t|o_t, h_t)$ where $a_t$ is the next action generated by the agent, given the latest observation $o_t$ and the trajectory history $h_t$ at timestep $t$. This policy is instantiated through a process of in-context policy learning (Shinn et al., 2023; Monea et al., 2025) from action-level exemplars demonstrating a reward-maximizing trajectory $\tau_j = (a_{j,0}, o_{j,0}, a_{j,1}, o_{j,1}...a_{j,n}, o_{j,n})$ where $\tau_j$ is the $j$th exemplar.

A shortcoming of in-context policy learning is ineffectiveness in exploration, biasing the agent policy toward exploitation (Monea et al., 2025; Dai et al., 2024). A primary reason behind this deficiency is an inability to learn from complex in-context signals (Dai et al., 2024), which makes it difficult for the agent to acquire complex, long-horizon exploratory behaviors from action-level demonstrations.

In CoEx, we overcome the exploitation bias of action-level ICL, by instantiating exploratory behavior at the level of subgoals, decoupling explo-ration from being dependent on the action-level demonstrations. Instead of needing to plan exploratory action sequences, our approach allows exploration to be easily orchestrated at an abstract level, as a standard subgoal of the Planner.

### 3.2 Proposed World Model: Adaptive Belief State

In RL, the notion of *belief states* is introduced as a solution to partial observability. A key characteristic of POMDPs is the partial observability of the true $s_t$. That is, the observable state $o_t$ is an imperfect description of $s_t$, and the belief state $b$ acts as a sufficient estimate of the underlying $s_t$ (Hausknecht and Stone, 2015; Avalos et al., 2024), that can be modeled by the agent (Adhikari et al., 2020). Within the context of LLM agents in POMDPs, the notion of belief state is functionally equivalent to the definition of an LLM's world model, under the definition of world model as a *mechanism by which the LLM agent can estimate* some *task-sufficient world state* $s_t^*$, where $s_t^*$ encapsulates the minimal set of variables or properties of the world state that are necessary for optimal LLM planning towards the task goal (Li et al., 2024); That is, $s_t^*$ and the belief state $b_t$ are equivalent.

Given this, we can pinpoint a shortcoming of the monolithic LLM agent paradigm. As monolithic LLM agent's world model is implicit in the LLM agent's parametric encoding of the trajectory history $h_t$, $s_t^*$ cannot be reliably adapted in response to experiences, including exploratory ones. Given an implicit world model, controlled updates to it are difficult, and the implication of this is that, in POMDPs, the monolithic LLM agent cannot reliably leverage world model adaptation to resolve the uncertainty arising from partial observability.

In CoEx, we address this problem by defining an explicit belief state as an adaptable world model, whose formulation allows for controllable, targeted

updates to the LLM's world model. In order to facilitate efficient adaptation, we leverage a neurosymbolic design which integrates experiences at the subgoal level, which also neatly dovetails with subgoal-level organization of planning.

## 4 Method

### 4.1 Architecture

CoEx addresses these challenges through the co-evolution of world model and exploration, leveraging a hierarchical architecture consisting of the following components, each of which will be described in detail in subsequent sections. We highlight the distinction of ours in bold:

**Planner (4.2)** An LLM agent, implemented at the **subgoal-level**, which reasons and plans the task through the generation of abstract subgoals.

**Actor (4.3)** An LLM agent, whose policy designed to efficiently carry out subgoals in the environment, through a reasoning-and-acting loop.

**Adaptive Belief State (4.4)** An adaptable, explicit belief state representation comprised of a code-based, object-oriented symbolic memory, and a structured textual memory generated using LLM-based verification and synthesis module, as $b_k = (m_k, l_k)$.

Our approach integrates these components into a unified loop of planning, exploration, and learning from experience. The full operation of CoEx is presented in Alg.1, which we describe in detail below.

### 4.2 Subgoal Planner

CoEx's Planner $\pi$ functions as the primary driver of task as well as world exploration progress. Its implementation enables the application of LLM reasoning towards robust subgoal-level planning. Our planner is an LLM prompted as a state machine, where the states are defined by the neurosymbolic belief state $b_k$, and its action space consists of *subgoal execution commands*, which allows the planner to request any subgoal it can describe via natural language. The planner, given its context history $H_k$, which consists of past belief states, generated subgoals, and associated reasoning, generates a new subgoal by reasoning about this history and the latest belief state $b_k$ (Alg.1,*Ln.* 15).

In Table 2, we illustrate the subgoal planning procedure using an example from the Zork1 of the Jericho (Hausknecht et al., 2020) text-based game

| **Planner Step:** k=1 |
|---|
| **Goal**: You need to find your way into a secret passage where the entrance is in the living room of the house. |
| **Initial Observation**: You are standing in an open field west of a white house, with a boarded front door. There is a small mailbox here. |
| **Planner Reasoning**: To achieve the goal of finding the secret passage in the living room, I need to first enter the house. Since the front door is boarded, I should check the mailbox for any clues or items that might help me. |
| **Plan Subgoals:**<br>1. Examine the mailbox.<br>2. Look for an alternative way to enter the house (e.g., a window or back door).<br>3. Enter the house.<br>4. Navigate to the living room.<br>5. Find the secret passage. |
| **Execute Subgoal:** [Examine the mailbox.] |
| *(... Planner history $H_k$ and belief state $b_k$)* |
| **Planner Step:** k=5 |
| **Planner Reasoning**: I have successfully navigated to the living room. Now, I need to find the secret passage. In the living room, I have noticed a trap door, which could potentially lead to the secret passage. |
| **Plan Subgoals:**<br>1. Investigate the trap door in the living room.<br>2. Find the secret passage. |
| **Execute Subgoal:** [Investigate the trap door in the living room.] |

Table 2: Illustration of CoEx's subgoal-level planning in the classical Jericho game, *Zork1*.

suite. Given the goal *"You need to find your way into a secret passage where the entrance is in the living room of the house."*, and initial observation, $\pi$ reasons about the task and generates an initial plan consisting of subgoal steps. As it proceeds through the task, it adapts its plan to the evolving $b_k$, as demonstrated in the later planner step $k = 5$. The abstraction of subgoal-level planning as *reasoning over a belief state $b_k$*, distinct from action-level planning, allows the planner to exhibit a high degree of flexibility in planning, overcoming exploitation bias in action-level ICL.

We demonstrate two key properties of our planner in this example, 1) Dynamic Replanning: Our planner can not only generate entire plans for a task and drive the task by progressing through each subgoal, but can dynamically adjust the plan in response to the evolving belief state at each timestep $k$, as illustrated in its generation of an updated plan *1. Investigate the trap door in the living room., 2. Find the secret passage.*[2] 2) Exploration and adap-

---

[2] We instruct the planner to generate plans using a FULL PLAN tag, followed by enumerated subgoals of the plan. De-

tation: As the task goal never actually specifies that the secret passage is behind the trap door, solving the task requires exploring the world and adapting to new discoveries. In this example, CoEx adapts to the discovery of a trap door, and proceeds to explore the world further using the exploratory subgoal *Investigate the trap door in the living room*. As subgoals allow $\pi$ to easily organize exploration as discrete objectives, our approach enables strategic and systemic exploration. As a result, CoEx's subgoal planning, leveraging powerful LLM reasoning on belief states, facilitates both task progression and exploration within a unified framework.

### 4.3 Actor

The Actor $\alpha$ is responsible for carrying out the subgoals according to the specification generated by the planner. It generates the low-level actions to interact with the environment using a reason-and-act loop, with the explicit objective of completing the assigned subgoal (*Ln.* 16).

To design an effective subgoal-level Actor, we address two critical challenges: 1) The subgoal-level actor needs to handle potentially diverse subgoals requested by the planner. To address this, we prompt the actor using a shared skill library (Khot et al., 2023), which can be dynamically leveraged by the Actor depending on the subgoal. We generate a library of subgoal exemplars by decomposing existing task-level exemplars into smaller chunks, and annotating the smaller subtrajectories with an appropriate subgoal. For example, in the PDDL task `gripper`, we decompose a task-level trajectory into skills which include *pick up object*, *move to room*, *drop object*, etc.

2) To effectively collaborate with the Planner, the actor needs to generate a self-contained subgoal-execution episode, which efficiently attempts the subgoal and does not continue indefinitely if the subgoal cannot be completed. As a solution, we prompt the actor with instructions which enable it to self-judge the status of the subgoal-execution sub-episode, terminating if the subgoal is completed, is deemed unachievable, or if the sub-episode exceeds a set number of steps.[3] This design allows the actor to generate self-contained subgoal-execution trajectories, as well as flexibly pass control back to the planner. The actor's algorithm is shown in (Alg.2, *Ln.* 1), and example prompts are shown in Appendix A.4.2.

---
tailed prompts are provided in Appendix A.4.1.

[3]We set the number of steps to 35.

| Belief State ($b_k$) | |
|---|---|
| **Object-oriented Symbolic Memory** ($m_k$) | **Structured Text Memory** ($l_k$) |
| **[Agent]** Location: at Living room Inventory: - Obj: brown sack - Obj: clove garlic **[Visited Locations]** - Loc: Attic - Loc: Behind house - Loc: Clearing - Loc: Forest - Loc: Kitchen … **[Discovered Objects]** - Obj: clove garlic (at: inventory) - Obj: brown sack (at: inventory) - Obj: kitchen table (at: Kitchen) - Obj: ancient map (at: Living room) - Obj: brass lantern (at: Living room) … - Obj: trap door (at: Living room) | **[Current Plan]** Subgoals 1. Navigate to the living room. 2. Find the secret passage. Status: Successfully navigated to the living room, progressing towards finding the secret passage. **[Subgoal Verification]** - Description: Navigate to the living room - Outcome: Completed - Justification: The last subgoal of navigating to the living room was completed successfully, which is a direct step towards the next subgoal of finding the secret passage. **[Learned Facts]** - Error: Agent encountered a game error when attempting to go north, indicating a possible dead end in that direction. … |

Table 3: Illustration of a belief state in `Zork1`, consisting of symbolic and structured text memory.

### 4.4 Adaptive Belief State

The key to our method's world model adaptation is the explicit belief state $b_k$, which evolves as subgoals are planned and executed. The belief state is updated using the subgoal experience $\varepsilon_k$, through two complementary representations, $m_k$ and $l_k$. Our dual-representation design targets fast, symbolic updates for low-level state tracking, while leveraging more powerful LLM inference to augment the symbolic state with structured inferences.

**Symbolic Memory** ($m_k$) We draw from the notion of object-oriented representation in RL (Diuk et al., 2008), to focus the symbolic representation around objects and ego-centric agent information, generally applicable to wide array of agent scenarios. $m_k$ is implemented as a code-based, object-oriented representation storing concrete, factual information (e.g. agent location, object states, locations), derived from the raw trajectory $\varepsilon_k$. An example of the symbolic memory is shown in the left column of Table 3.

The symbolic memory is designed with an associated programmatic update mechanism, that sys-

tematically translates low-level actions $a_t$ and corresponding observations $o_t$ from interaction, into updates of the symbolic representation. Implementing symbolic memory in code allows for fast, efficient state tracking, deliberately offloading the burden of lower-level state tracking processing from the planner $\pi$. We update the symbolic memory at each timestep $t$ during subgoal execution.[4]

**Structured Textual Memory** ($l_k$)    Complementing the fast, factual tracking in $m_k$, the structured textual memory $l_k$ aims to capture the agent's higher-level understanding, manages uncertainty, tracks task progress, and synthesizes knowledge that may be less amenable to rigid symbolic representation, using natural language.

Distinctly from $m_k$, the structured textual memory is updated through a 2-stage verification and synthesis process, evaluating the subgoal execution outcome $m_k$ and $\varepsilon_k$ before incorporating information into the belief state, using an LLM-based Verification and Synthesis module $v$. An example of the resulting structured textual memory is shown in the right column of Table 3.

**Stage 1: Verification**    The first stage is verification, where the module analyzes the trajectory $\epsilon_k$ and the updated symbolic state $m_{k+1}$ through a series of queries, to assess consistency, detect execution failures or unexpected events, and determine the success or progress of the subgoal. To facilitate the incorporation unexpected discoveries, we also include a question about new facts learned, or surprising outcomes.

**Stage 2: Belief Synthesis**    Based on the verification QA results and $\epsilon_k$ and $m_{k+1}$ as well as previous belief state, the next stage synthesizes the structured updates to the belief state: We use an LLM to generate the `status line`, which reflects the last subgoal outcome, the `justification`, the rationale for the status line, and `learned facts`, which act as a storage for new facts and hypotheses formed. The generated output is is the new $l_k$.

Combined together, the updated $m_k$ and $l_k$ together constitute the new $b_k$. We show the prompts of the Verification and Synthesis modules in Appendix A.4.3.

# 5    Experiments

## 5.1    Experimental Setup

**Benchmarks**    We evaluate CoEx across 3 distinct agent benchmarks evaluating aspects of planning, exploration and world modeling. **ALF-World** (Shridhar et al., 2021) is a text-based embodied task, requiring grounding to household environments and multi-step execution. We follow baselines and evaluate on the standard unseen test split. **Jericho** (Hausknecht et al., 2020) is a suite of text-based adventure games requiring exploration, world modeling, and common-sense reasoning to achieve high scores. We adopt the setting from AgentBoard (Ma et al., 2024), which converts the games from open-ended exploration without a specified goal, to a task with a human-annotated goal (e.g. "Get out of the house. Then escape the city without getting caught via driving."). We also evaluate on **PDDL** (McDermott et al., 1998), a classical symbolic planning domain testing logical planning and state tracking, using the Gripper and Blocksworld domains involving complex multistep robotic planning.

**Baselines**    We compare CoEx against representative LLM agent architectures for planning. ReAct (Yao et al., 2023) is a canonical model-free baseline using a think-act loop. Reflexion (Shinn et al., 2023) extends ReAct with self-reflection for error correction based on past trials, implementing a low-frequency form of world model adaptation. AdaPlanner (Sun et al., 2023) is an agent that adapts its plan based on execution feedback, enabling dynamic replanning through detailed replanning exemplars. HiAgent (Hu et al., 2025) is an agent which employs subgoal-based planning by leveraging subgoal observation chunk summarization, to focus on subgoal-relevant contexts.

ExpeL (Zhao et al., 2024) implements a form of offline world model adaptation, learning static beliefs about the task through offline learning on success and failure trajectories. WALL-E (Zhou et al., 2024) also leverages offline rule learning from collected trajectories for world model alignment, but leverages code-based symbolic rules.

**Implementation Details**    We set a maximimum number of total steps taken in each environment at 100, 100 and 150 for ALFWorld, PDDL, and Jericho, respectively. We set the maximum number

---

[4]The implementation details and code of the symbolic memory can be found in the Appendix A.3.

Table 4: Results on the six tasks of the ALFWorld benchmark, using success rate (%) as evaluation metric. Best results are highlighted in bold.

| Method | Pick | Clean | Heat | Cool | Examine | Picktwo | Total |
|---|---|---|---|---|---|---|---|
| ExpeL | - | - | - | - | - | - | 64.20% |
| ReAct | 66.67% | 41.94% | 91.03% | 80.95% | 55.56% | 35.29% | 61.94% |
| Reflexion | 75.00% | 90.32% | 91.30% | 90.48% | 88.89% | 94.12% | 88.06% |
| AdaPlanner | **100.00%** | 96.77% | 95.65% | **100.00%** | **100.00%** | 47.06% | 91.79% |
| WALL-E | **100.00%** | **97.00%** | **100.00%** | 86.00% | 85.00% | **100.00%** | **95.00%** |
| CoEx (Ours) | **100.00%** | 83.87% | 91.3% | 90.48% | **100.00%** | 88.24% | 93.28% |

Table 5: Results on the PDDL task domains, using success rate (sr%) and progress rate (pr%) as evaluation metric. Progress rate measures subgoal completion progress, separately from task success.

| Method | Gripper (sr/pr %) | Blocksworld (sr/pr %) | Average (sr/pr %) |
|---|---|---|---|
| ReAct | 65.0%/89.5% | 50.0%/65.0% | 60.0%/81.3% |
| HiAgent | **75.0%**/89.9% | 50.0%/68.3% | 66.7%/82.7% |
| CoEx (Ours) | 70.0%/**94.2%** | **80.0%/90.0%** | **73.3%/92.8%** |

Table 6: Results on the Jericho text adventure games, using success rate (sr%) and progress rate (pr%) as evaluation metrics. Games are color-coded by difficulty: <span style="background-color:#d8f0d8">easy</span> and <span style="background-color:#f5d5d5">hard</span>.

| Game | ReAct pr% | CoEx (Ours) pr% |
|---|---|---|
| 905 | 0.0 | 85.7 |
| acorncourt | 9.1 | 45.5 |
| afflicted | 0.0 | 100.0 |
| balances | 42.9 | 42.9 |
| dragon | 100.0 | 100.0 |
| jewel | 66.7 | 0.0 |
| library | 25.0 | 25.0 |
| omniquest | 25.0 | 100.0 |
| reverb | 50.0 | 75.0 |
| snacktime | 100.0 | 100.0 |
| zenon | 16.7 | 50.0 |
| zork1 | 50.0 | 100.0 |
| zork2 | 50.0 | 50.0 |
| zork3 | 25.0 | 25.0 |
| detective | 0.0 | 0.0 |
| night | 0.0 | 0.0 |
| pentari | 83.3 | 60.0 |
| weapon | 66.7 | 50.0 |
| huntdark | 0.0 | 66.7 |
| loose | 16.7 | 33.3 |
| Avg. pr% (Easy) | 35.2 | **61.9** |
| Avg. pr% (Hard) | 37.4 | **50.7** |
| Avg. pr% (All) | 36.4 | **55.5** |
| Overall sr% | 10.0 | **25.0** |

of subgoal execution steps by the actor, at 35. We utilize GPT-4o-mini for all experiments, for all components of CoEx. Following conventions of previous works (Yao et al., 2023; Shinn et al., 2023; Ma et al., 2024), we report the results of single runs. For all prompting, we utilize a system prompt followed by an instance prompt. The details of

prompts can be found in Appendix A.4.

## 5.2 Results

**ALFworld** We report the results of experiments on ALFWorld in Table 4. CoEx demonstrates strong performance across all task types, achieving a total success rate of 93.28%, competitive with state-of-the-art baselines. ALFWorld tasks require a mixture of planning and exploration, as the tasks require locating objects whose locations are at first unknown, then carrying out further tasks involving them. Through its co-evolution of planning and world model adaptation, CoEx achieves superior performance compared to less dynamic methods for world model adaptation, such as ExpeL or Reflexion. It also achieves competitive performance with WALL-E, an offline world-model learning approach that requires a dedicated offline learning phase to generate rules across multiple episodes, demonstrating the efficiency of CoEx's online learning approach. Compared to AdaPlanner, which implements a strong dynamic planning approach, our method demonstrates better overall performance as well as significantly higher performance on the more challenging "picktwo" task type, involving locating two distinct objects and placing them in the correct target locations.

**PDDL** We compare CoEx against ReAct on PDDL domains, and report the results in Table 5. As PDDL tasks generally involve more numbers of subgoals than ALFWorld, we adopt the progress rate metric from Ma et al. (2024) which allows tracking subgoal progress rate independently from task success rate. CoEx outperforms ReAct by 13.3%/11.5% in both success rate and progress rate, demonstrate the strong capability of our subgoal-based approach to LLM agent planning in complex planning domains.

**Jericho** We report the results on the Jericho text-based games in Table 6. Jericho games re-

quire strong exploration capabilities, as significant amounts of information about the environment are only attainable through learning from exploration. The results demonstrate that CoEx excels over Re-Act in exploration, showing a 15%/19.1% gain in success rate and progress rate, respectively. Following Ma et al. (2024), we further divide the games into easy and hard difficulties based on the number of subgoals (hard games have more than 5 subgoals), where we observe that CoEx retains a 13.3% higher progress rate. These results lend strong support our core claim, that the co-evolution of exploration and world model adaptation in our approach enhances both exploration and planning in LLM agents.

## 5.3 RQ1: Ablation of Hierarchical Architecture

To evaluate the effectiveness of CoEx's hierarchical Planner-Actor architecture, we conduct an ablation study using HiAgent as our comparison baseline. This provides a principled ablation by comparing against a well-engineered monolithic alternative rather than artificially dismantling CoEx's integrated components. HiAgent effectively collapses our explicit Planner-Actor hierarchy into a single agent while maintaining subgoal-level planning. While both methods employ memory mechanisms (HiAgent uses observation summarization, CoEx uses neurosymbolic belief states), this comparison isolates the impact of our architectural separation of planning and execution. We evaluate HiAgent on the Gripper and Blocksworld tasks of PDDL with 100 steps, the same as CoEx. The results in Table 5 confirm the effectiveness of our hierarchical architecture.

## 5.4 RQ2: Is CoEx Computationally Efficient?

To understand the computational costs of CoEx, we perform analysis measuring LLM API usage by tokens. We find that CoEx's additional components incur only minimal overheads, with 70% of computational costs occurring in the Actor, whose costs are shared with baseline methods like ReAct. Other components, i.e. the main Planner, first stage of belief state update (verification and q&a), and belief synthesis, account for around 18%, 10%, 2% of computation cost respectively. CoEx achieves this efficiency by invoking its planning and belief update components at the subgoal level rather than at every low-level action step, substantially reducing the overhead of additional components.

## 5.5 RQ3: How does the World Model evolve in CoEx?

To understand how CoEx's world model co-evolves with the task subgoal progress, we perform a trajectory analysis of a 4-stage evolution of the world model for the task "put two soapbar in garbage-can" in the "picktwo" subset of ALFWorld[5]. In World Model Update 1, the executor's confusion between soapbottle/soapbar objects is reflected in the belief state. Consequently, in World Model Update 2, the planner issues the subgoal to find soapbar in remaining unsearched locations, resulting in the successful location discovery of both soapbars at toilet 1. In World Model Update 3, the planner issues the subgoal to take soapbar 1, but encounters repeated failures from the executor, leading to the world model learning the knowledge "Agent was unable to take soapbar 1 from toilet 1, indicating a potential restriction or condition not met for that action". Finally, in World Model Update 4, the planner strategically adapts by targeting soapbar 2 instead, which the executor succeeds by dropping the soapbottle first, and in subsequent steps, this leads to eventual success on the task. This example demonstrates how CoEx advances the world model by integrating execution feedback, and strategically adapts plans based on discovered information about the environment. We also note that, while LLM-based QA may introduce marginal inaccuracies during world model updates, as in existing monolithic agents, the distinction of CoEx is that it can repair such errors in subsequent updates to its explicit belief state.

## 6 Conclusion

We study the problem of overcoming the limitations of existing LLM agents, namely, exploitation bias and limited adaptation. We propose CoEx, enabling subgoal-driven exploration, allowing its internal world model to co-evolve with real-time observations. This is facilitated through a neurosymbolic belief state that integrates textual reasoning with symbolic memory, ensuring that the agent's understanding remains aligned with the true state of its environment. Our evaluations on benchmarks requiring planning and exploration, including ALF-World, Jericho, and PDDL, show that CoEx outperforms strong baselines across diverse tasks.

---

[5]The trajectory is provided in Appendix A.5.

## Limitations

While the framework adaptivity has been tested in the benchmark scales, more abrupt or unpredictable environmental shifts may require a new strategy for faster adaptation. Also, some real-life tasks of sufficiently larger scales may require memory pruning and abstraction mechanisms.

More sophisticated memory compression or enhanced real-time world-model synchronization are promising future directions.

## Acknowledgements

## References

Ashutosh Adhikari, Xingdi Yuan, Marc-Alexandre Côté, Mikuláš Zelinka, Marc-Antoine Rondeau, Romain Laroche, Pascal Poupart, Jian Tang, Adam Trischler, and Will Hamilton. 2020. Learning dynamic belief graphs to generalize on text-based games. In *Advances in Neural Information Processing Systems*, volume 33, pages 3045–3057. Curran Associates, Inc.

Raphaël Avalos, Florent Delgrange, Ann Nowe, Guillermo Perez, and Diederik M Roijers. 2024. The wasserstein believer: Learning belief updates for partially observable environments through reliable latent space models. In *The Twelfth International Conference on Learning Representations*.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. 2018. Textworld: A learning environment for text-based games. *CoRR*, abs/1806.11532.

Zhenwen Dai, Federico Tomasi, and Sina Ghiassian. 2024. In-context exploration-exploitation for reinforcement learning. In *The Twelfth International Conference on Learning Representations*.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating code world models with large language models guided by monte carlo tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Carlos Diuk, Andre Cohen, and Michael L. Littman. 2008. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 240–247, New York, NY, USA. Association for Computing Machinery.

David Ha and Jürgen Schmidhuber. 2018. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. 2021. Mastering atari with discrete world models. In *International Conference on Learning Representations*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173, Singapore. Association for Computational Linguistics.

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2020. Interactive fiction games: A colossal adventure. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7903–7910.

Matthew J. Hausknecht and Peter Stone. 2015. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. 2025. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32779–32798, Vienna, Austria. Association for Computational Linguistics.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. 2017. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40:e253.

Zichao Li, Yanshuai Cao, and Jackie CK Cheung. 2024. Do LLMs build world representations? probing through the lens of state abstraction. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. In *Advances in Neural Information Processing Systems*, volume 37, pages 74325–74362. Curran Associates, Inc.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. PDDL - The Planning Domain Definition Language.

Giovanni Monea, Antoine Bosselut, Kianté Brantley, and Yoav Artzi. 2025. Llms are in-context bandit reinforcement learners. *Preprint*, arXiv:2410.05362.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. {ALFW}orld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplanner: Adaptive planning from feedback with language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Hao Tang, Darren Yan Key, and Kevin Ellis. 2024. Worldcoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642.

Siyu Zhou, Tianyi Zhou, Yijun Yang, Guodong Long, Deheng Ye, Jing Jiang, and Chengqi Zhang. 2024. Wall-e: World alignment by rule learning improves world model-based llm agents. *Preprint*, arXiv:2410.07484.

# A Appendix

---
**Algorithm 1** CoEx Architecture
---
1: **Variables:**
2:      $l_k$: structured textual memory
3:      $m_k$: symbolic memory
4:      $H_k$: planner history
5:      $e_k$: subgoal description (text)
6:      $\varepsilon_k$: sub-episode trajectory (trace)
7:      $k$: main step counter
8: **Initialize:**
9:      $k \leftarrow 0$
10:      $l_0 \leftarrow []$
11:      $m_0 \leftarrow$ InitializeSymbolicMemory(Env.GetInitialObs())
12:      $H_0 \leftarrow []$
13: **Input:** Environment Env, Planner $\pi$, actor $\alpha$, VerificationAndSynthesisModule $v$
14: **while** task not complete and $k <$ max_total_steps **do**
15:      $e_k \leftarrow \pi(H_k)$                                      ▷ Planner generates next subgoal $e_k$ conditioned on belief state history
16:      $(\varepsilon_k, m_{k+1}) \leftarrow \alpha.$ExecuteSubgoalEpisode$(e_k)$                              ▷ Run subgoal-episode
17:      $l_{k+1} \leftarrow v.$BeliefUpdate$(b_k, m_{k+1}, \varepsilon_k, e_k)$                    ▷ Verification and synthesis for belief update
18:      $H_{k+1} \leftarrow H_k \cup \{(l_k, m_k)\}$                              ▷ Add new belief state to planner history
19:      $k \leftarrow k + 1$
20: **end while**
---

---
**Algorithm 2** CoEx Subroutines
---
1: **procedure** ExecuteSubgoalEpisode$(e_k)$
2:      **Initialize:** $t \leftarrow 0$, status $\leftarrow$ "running", $H_{\text{exec}} \leftarrow$ ExecutorInitHistory$(e_k)$
3:      **Param:** max_sub_steps
4:      **while** $t <$ max_sub_steps **do**
5:          $a_t \leftarrow \alpha(a|e_k, H_{\text{exec}})$
6:          **if** $a_t$ contains SUBGOAL COMPLETED **then**
7:              status $\leftarrow$ "completed"; **break**
8:          **end if**
9:          **if** $a_t$ contains REQUEST_REPLAN **then**
10:              status $\leftarrow$ "not completed"; **break**
11:          **end if**
12:          $o \leftarrow$ Env.Step$(a_t)$
13:          $H_{\text{exec}} \leftarrow H_{\text{exec}} \cup \{(a_t, o)\}$
14:          $t \leftarrow t + 1$
15:      **end while**
16:      **return** $H_{\text{exec}}$
17: **end procedure**

18: **procedure** BeliefUpdate$(b_{\text{prev}}, m_{\text{curr}}, \varepsilon, e,$ status$)$
19:      $qa\_results \leftarrow v.$Verification$(b_{\text{prev}}, m_{\text{curr}}, \varepsilon, e)$
20:      $l_{\text{new}} \leftarrow v.$Synthesis$(b_{\text{prev}}, m_{\text{curr}}, qa\_results, e)$
21:      **return** $l_{\text{new}}$
22: **end procedure**
---

## A.1   Algorithms

We present the algorithms detailing the operation of CoEx. Algorithm 1 describes the main loop that orchestrates the Planner, Actor, and belief state updates. Algorithm 2 describes the Actor's subgoal execution and the two-stage verification and synthesis process of the belief state update mechanism.

## A.2   Datasets

The statistics for the ALFWorld, PDDL, and Jericho benchmarks are shown in Table 7.

## A.3   Implementation Details of Symbolic Memory

We use use Gemini 2.5 pro preview to implement the symbolic memory in Python using example trajectories similar to Tang et al. (2024), with minor finetuning of the code through manual revision.

### A.3.1   Symbolic Memory Python Implementation

We provide examples of the Python implementations of symbolic memory for the PDDL blocksworld task.

Table 7: Test Set Distribution Across Benchmarks

| Benchmark | ALFWorld | | | | | | PDDL | | Jericho |
|---|---|---|---|---|---|---|---|---|---|
| Task/Domain | Pick | Examine | Clean | Heat | Cool | Pick Two | Gripper | Blocksworld | |
| Test Instances | 24 | 18 | 31 | 23 | 21 | 17 | 10 | 20 | 20 |

## Python Implementation of Symbolic Memory (PDDL - Base Class)

```python
import re

class SimpleSymbolicMemory:
    def __init__(self, domain_name):
        self.domain_name = domain_name
        self.predicates = set()
        self.holding = {}
        self.agent_location = None
        self.step = 0

    def update_memory(self, observation: str, last_action: str = None):
        """
        Updates the memory based on the observation.
        This base method does nothing; subclasses must override it.
        """
        self.step += 1
        print(f"\n--- Step {self.step} ---")
        print(f"Action: {last_action}")
        print(f"Observation: {observation.strip()}")
        print(f"Predicates BEFORE update: {sorted(list(self.predicates))}")
        print(f"Holding BEFORE update: {self.holding}")
        if self.agent_location is not None:
            print(f"Location BEFORE update: {self.agent_location}")

        self._parse_and_update(observation, last_action)

        print(f"Predicates AFTER update: {sorted(list(self.predicates))}")
        print(f"Holding AFTER update: {self.holding}")
        if self.agent_location is not None:
            print(f"Location AFTER update: {self.agent_location}")

    def _parse_and_update(self, observation: str, last_action: str = None):
        # Needs implementation in subclass
        raise NotImplementedError

    def _add_predicate(self, predicate_str):
        self.predicates.add(predicate_str)

    def _remove_predicate(self, predicate_str):
        self.predicates.discard(predicate_str)

    def _remove_predicates_about(self, *args):
        to_remove = set()
        for pred in self.predicates:
            for arg in args:
                if f"({arg}," in pred or f",{arg})" in pred or f"({arg})" ↩
                    in pred:
                    to_remove.add(pred)
        self.predicates -= to_remove

    def _clear_holding(self, manipulator):
        if manipulator in self.holding:
            held_item = self.holding[manipulator]
            if held_item:
                self._remove_predicates_about(
                f"holding({manipulator},{held_item})")
            self.holding[manipulator] = None
```

```
57
58    def _set_holding(self, manipulator, item):
59        self._clear_holding(manipulator) # Ensure manipulator wasn't ↩
              holding something else
60        self.holding[manipulator] = item
61
62    def get_planning_summary(self) -> str:
63        lines = [f"### {self.domain_name.upper()} Memory Summary (Step ↩
              {self.step}) ###"]
64        if self.agent_location:
65            lines.append(f"Agent Location: {self.agent_location}")
66        lines.append(f"Holding: {self.holding}")
67        lines.append("State:")
68        if self.predicates:
69            for pred in sorted(list(self.predicates)):
70                lines.append(f"  - {pred}")
71        else:
72            lines.append("  (None)")
73        lines.append("### END SUMMARY ###")
74        return "\n".join(lines)
75
76    def __str__(self):
77        return self.get_planning_summary()
```

Listing 1: SimpleSymbolicMemory Base Class in Python

## Python Implementation of Symbolic Memory (PDDL - BlocksWorld)

```
1  import re
2
3  from .base import SimpleSymbolicMemory
4
5
6  class BlocksWorldSymbolicMemory(SimpleSymbolicMemory):
7      def __init__(self):
8          super().__init__("BlocksWorld")
9          self.holding = {'arm': None}
10
11     def _parse_and_update(self, observation: str, last_action: str = None):
12         # 1. Determine Holding State
13         held_block = None
14         is_arm_empty = True
15         holding_match = re.search(r"You are holding (b\d+)", observation)
16         if holding_match:
17             held_block = holding_match.group(1).lower()
18             is_arm_empty = False
19         elif re.search(r"(Robot|Your|The)\s+arm\s+is\s+empty", observation, ↩
             re.IGNORECASE):
20             is_arm_empty = True
21         else:
22             held_block = self.holding.get('arm')
23             is_arm_empty = (held_block is None)
24
25
26         self.holding['arm'] = held_block
27
28         # 2. Gather Observed Relations
29         observed_on = {m.group(1).lower(): m.group(2).lower()
30                        for m in re.finditer(r"(b\d+) is on (b\d+)", ↩
                           observation, re.IGNORECASE)}
31         observed_on_table = {m.group(1).lower()
32                              for m in re.finditer(r"(b\d+) is on the ↩
                                 table", observation, re.IGNORECASE)}
33         observed_clear = {m.group(1).lower()
34                           for m in re.finditer(r"(b\d+) is clear", ↩
                              observation, re.IGNORECASE)}
```

```python
        observed_not_clear = {m.group(1).lower()
                            for m in re.finditer(r"(b\d+) is not clear", ↵
                            observation, re.IGNORECASE)}

        all_mentioned = (set(observed_on.keys()) | ↵
            set(observed_on.values()) |
                        observed_on_table | observed_clear | ↵
                        observed_not_clear)
        if held_block: all_mentioned.add(held_block)

        # 3. Rebuild State for Mentioned Blocks
        new_predicates = set()

        new_predicates.add("arm_empty" if is_arm_empty else "arm_not_empty")

        blocks_underneath = set(observed_on.values())

        for block in all_mentioned:
            pos_set = False
            if block == held_block:
                pos_set = True
            elif block in observed_on:
                new_predicates.add(f"on({block},{observed_on[block]})")
                pos_set = True
            elif block in observed_on_table:
                if block not in observed_on.values():
                    new_predicates.add(f"on_table({block})")
                    pos_set = True

            clear_set = False
            if block in observed_not_clear:
                new_predicates.add(f"not_clear({block})")
                clear_set = True
            elif block == held_block:
                new_predicates.add(f"clear({block})")
                clear_set = True
            elif block in observed_clear:
                new_predicates.add(f"clear({block})")
                clear_set = True
            elif block in blocks_underneath:
                new_predicates.add(f"not_clear({block})")
                clear_set = True
            else:
                if block != held_block:
                    new_predicates.add(f"clear({block})")
                    clear_set = True

        # 4. Update Memory
        predicates_to_remove = set()
        predicates_to_remove.add("arm_empty")
        predicates_to_remove.add("arm_not_empty")
        for block in all_mentioned:
            for pred in self.predicates:
                if re.search(rf'\b{re.escape(block)}\b', pred):
                    if pred.startswith(('on(', 'on_table(', 'clear(', ↵
                        'not_clear(')):
                        predicates_to_remove.add(pred)
        self.predicates -= predicates_to_remove

        self.predicates.update(new_predicates)
```

Listing 2: BlocksWorldSymbolicMemory in Python

## A.4 Prompts

### A.4.1 CoEx Planner Prompts

---

**System Prompt (ALFWorld)**

You are Alfred, an agent for the ALFWorld household environment. You will be given a task to complete, and you complete the task by breaking down the task into a task consisting of subgoals. You can use the `EXECUTE_SUBGOAL` action to execute a subgoal, which will be delegated to a subgoal executor agent, and you will receive feedback on the subgoal execution. When executing subgoals, please be detailed, and include the necessary relevant information about the status of the task along with the subgoal.

**Generating Plan**
Whenever you generate an entirely new plan, or you change the plan with different subgoals or new order of subgoals, you must include FULL PLAN in the response, followed by all of the steps in the new plan. Example:

```
...
FULL PLAN
Subgoals:
1. Go to ...
2. Take ...
3. Use ...
4. Go to ...
5. ...
```

**Processing Subgoal Feedback:**

- After a subgoal attempt, you will receive an 'analysis_feedback' message (role: assistant).

- **Parse the JSON content** of this message.

- Use the `new_belief` field to understand the current world state before planning the next subgoal.

**`EXECUTE_SUBGOAL` Action Format:**
You MUST issue subgoals using the following multi-line format precisely:

```
EXECUTE_SUBGOAL[
DESC: <Clear, natural language description of the specific subgoal>
SEARCH_LOCATIONS: [<loc1>, <loc2>, ...] # Include ONLY when we need to
search for an object, otherwise omit or null
```

**Generating `SEARCH_LOCATIONS` (Crucial for Find/Take Subgoals):**
When the `DESC` is `"Find and take [object]"`, you MUST:

1) Read the `new_belief` text from the most recent `analysis_feedback` message.

2) Identify **all receptacle IDs** mentioned as existing in the room within that belief text (e.g., `cabinet 1`, `fridge 1`, `countertop 1`).

3) Create a list of these known receptacle IDs. Prioritize likely locations if possible.

4) Populate the `SEARCH_LOCATIONS:` field with this **exact list**. Example: `SEARCH_LOCATIONS: [cabinet 1, fridge 1, countertop 1]`

Below are examples of a similar task:
`{{task exemplars}}`

---

You are an agent playing a text-based adventure game. You will be given a task in the game to complete, and you complete the task by breaking down the task into a task consisting of subgoals. You can use the EXECUTE_SUBGOAL action to execute a subgoal, which will be delegated to a subgoal executor agent, and you will receive feedback on the subgoal execution. When executing subgoals, please be detailed, and include the necessary relevant information about the status of the task along with the subgoal.

**Generating Plan**
Whenever you generate an entirely new plan, or you change the plan with different subgoals or new order of subgoals, you must include FULL PLAN in the response, followed by all of the steps in the new plan. When doing so you MUST include the FULL PLAN in the response, followed by all of the steps in the new plan. Example:

```
...
FULL PLAN
Subgoals:
1. Go to ...
2. Take ...
3. Use ...
4. Go to ...
5. ...
```

**Processing Subgoal Feedback:**

- After a subgoal attempt, you will receive an 'analysis_feedback' message (role: assistant).

- **Parse the JSON content** of this message.

- Use the new_belief field to understand the current world state before planning the next subgoal.

**EXECUTE_SUBGOAL Action Format:**
You MUST issue subgoals using the following multi-line format precisely:

```
EXECUTE_SUBGOAL[
DESC: <Clear, natural language description of the specific subgoal>
]
```

**Checking valid actions**

- If you are unsure about which actions can be taken, make sure to use the 'check valid actions' command.

Below are examples of a similar task:
{{task exemplars}}

```
Goal: {{ goal }}
{{ initial_observation }}
```

You are an agent carrying out a planning task. You will be given a task to complete, and you complete the task by breaking down the task into subgoals. You can use the EXECUTE_SUBGOAL action to execute a subgoal, which will be delegated to a subgoal executor agent, and you will receive feedback on the subgoal execution. When executing subgoals, please be detailed, and include the necessary relevant information about the status of the task along with the subgoal.

**Generating Plan**
Whenever you generate an entirely new plan, or you change the plan with different subgoals or new order of subgoals, you must include FULL PLAN in the response, followed by all of the steps in the new plan. When doing so you MUST include the FULL PLAN in the response, followed by all of the steps in the new plan. Example:

```
...
FULL PLAN
Subgoals:
1. Go to ...
2. Take ...
3. Use ...
4. Go to ...
5. ...
```

**Processing Subgoal Feedback:**

- After a subgoal attempt, you will receive an 'analysis_feedback' message (role: assistant).

- **Parse the JSON content** of this message.

- Use the new_belief field to understand the current world state before planning the next subgoal.

**EXECUTE_SUBGOAL Action Format:**
You MUST issue subgoals using the following multi-line format precisely:

```
EXECUTE_SUBGOAL[
DESC: <Clear, natural language description of the specific subgoal>
]
```

Below are examples of a similar task:
{{task exemplars}}

Goal:
{{ goal }}

Initial Observation:
{{ initial_observation }}

Begin by carefully summarizing the target goal state, and then generate the plan to achieve the target goal.

### A.4.2 CoEx Subgoal Prompts

The example below shows the actor prompt for PDDL. For PDDL, domain-specific instructions are provided for each domain, while for ALFWorld and Jericho, the instructions are shared for all tasks.

---

**System Prompt**

You are a master in planning. You will be given a subgoal to complete. Think step-by-step. Output your thought process followed by the command in markdown

When you execute a command, you will receive a response from the game.

- If the action is successful, you will receive the updated state of the world.

- If the action is unsuccessful, you will receive a message indicating the failure: `"The action is not valid and therefore takes no effect. Please check valid actions."`

- If you believe the subgoal is achieved based on the game's response, output the action `'SUBGOAL COMPLETED'` instead of a game command.

- If you get stuck, cannot proceed, or believe the subgoal is impossible, output `'REQUEST_REPLAN[<Reason for failure>]'`.

---

**Instance Prompt**

`{{domain_instructions}}`

Think step-by-step about your plan and the expected outcome before issuing a command. Format your response with your thought process, followed by the command in markdown backticks. Do not issue multiple commands at once. If you issue the first command, wait for the result before issuing the next command.

Example Format:
`{{example_format}}`

---

Your Assigned Subgoal: `{{ subgoal }}`

---

Your Current State:
`{{ location }}`

---

Execute the next command towards the subgoal, or output `SUBGOAL  COMPLETED` or `REQUEST_REPLAN[...]`

### A.4.3 CoEx Verification and Synthesis Prompts

The same verification and synthesis prompts are used for all tasks. The `context` variable is constructed by concatenating: the subgoal text $e_k$, the raw action and observation trace from the subgoal trajectory $\varepsilon_k$, and the most recent symbolic memory $m_k$. The LLM is prompted with the system prompt as well as an instance prompt per each question, generating the answers to the questions.

---

**System Prompt for Verification Stage**

You are an analytical assistant answering specific questions about an agent's execution trace. Provide a clear answer (Yes/No/Uncertain/Specific Value) and a brief justification based *only* on the provided context.

The assistant will be asked one of the following types of questions:

- "Did the subgoal '«subgoal»' contribute positively towards the main goal based on the trace?"
- "Did the agent successfully navigate to the intended location or interact with the intended object?"
- "Were there any errors (e.g., 'You can't do that', 'I don't understand') or loops?"
- "Did the agent's inventory change as expected?"
- "Based ONLY on the execution trace, what are the 1-3 most important new facts learned, errors encountered, or surprising outcomes observed during this subgoal attempt? List them concisely or state 'None'."

---

**Instance Prompt for Verification Stage**

```
Based ONLY on the provided context below, answer the following question.

CONTEXT:
{{context}}

QUESTION: {{question}}

ANSWER (e.g., Yes/No/Uncertain/Value): [Your Answer]
JUSTIFICATION: [Your Brief Reasoning]
```

The synthesis stage takes the outputs of the Verification QA stage, along with the previous belief state $b_{k-1}$, symbolic memory $m_k$, the subgoal text $e_k$, to generate the new belief state update.

---

**System Prompt for Synthesis Stage**

You are a high-level planner agent. Based on the previous belief, current memory state, latest plan, and analysis of the last subgoal execution (Q&A), decide the next course of action.

Generate ONLY the following:

1. A concise status line reflecting the current progress relative to the plan (starting with "`Status:`").
2. and the justification for the status line.
3. A list of concise new facts learned or hypotheses formed about the environment/task based *only* on the *last* subgoal's execution, especially failures or unexpected outcomes. Focus on actionable insights or constraints.

---

Respond ONLY with a valid JSON object containing the keys `"status_line"` (string), `"justification"` (string), and `"learned_facts"` (list of strings, can be empty).

---

**Instance Prompt for Synthesis Stage**

Previous Belief:
`{{ previous_belief }}`

Current Symbolic Memory Summary:
`{{ memory_summary }}`

Latest Overall Plan:
`{{ latest_plan }}`

Last Subgoal Attempted: `{{ subgoal }}`

Subgoal Execution Q&A Analysis:
`{{ qa_summary }}`

**Instructions:**

1. **Generate `status_line`:** Create a single sentence starting with `"Status: "` that reflects the 'Last Subgoal Outcome' and progress relative to the 'Latest Overall Plan'.

2. **Provide `justification`:** Briefly explain the reasoning for the status line.

3. **Generate `learned_facts`:** Based *only* on the Q&A and outcome of the *last* subgoal, list any new, concise, and actionable facts learned or hypotheses formed (e.g., `"Confirmed: Kettle is in Cabinet."`, `"Error: Agent tried incorrect action 'open fridge drawer.'"`). Output an empty list `[]` if nothing significant was learned.

Respond ONLY with a valid JSON object containing `"status_line"`, `"justification"`, and `"learned_facts"`.

---

## A.5 Example of World Model Evolution

Below we provide the trajectory showing a 4-stage evolution of the world model in CoEx, for the ALFWorld task "put two soapbar in garbagecan". This example demonstrates how CoEx integrates execution feedback into its belief state, and how the Planner strategically adapts its subgoals based on this newly discovered information about the environment.

---

**World Model Update 1**

**Subgoal Generation and Execution:**

**Planner Reasoning:** "Starting with the first subgoal. Likely locations for soapbar: countertop, cabinets, towelholders, sinkbasins"
**Subgoal:** Find and take the first soapbar (Search loc: [countertop 1, cabinet 1-4, handtowelholder 1-2, sinkbasin 1-2])
**Subgoal Result:** ✗ Took wrong object

**Updated Belief State (Abbreviated):**

- **Symbolic Memory:** Agent: "holding soapbottle 1"

- **Semantic:** "Error: Agent took soapbottle 1 instead of the target soapbar"

> **World Model Update 2**
>
> **Subgoal Generation and Execution:**
>
> **Planner Reasoning:** "Search the remaining unexplored locations: garbagecan, sinkbasins, toilet, toiletpaperhanger, and towelholder"
> **Subgoal:** Find and take the first soapbar (Search loc: [garbagecan 1, sinkbasin 1, sinkbasin 2, toilet 1, toiletpaperhanger 1, towelholder 1])
> **Subgoal Result:** ✓ Found soapbars
>
> **Updated Belief State (Abbreviated):**
>
> - **Symbolic Memory:** `toilet 1: contains=[soapbar 1, soapbar 2]`
> - **Semantic:** "Confirmed: Soapbar 1 and soapbar 2 are located at toilet 1"

> **World Model Update 3**
>
> **Subgoal Generation and Execution:**
>
> **Planner Reasoning:** "Since I have confirmed that soapbar 1 is currently in toilet 1, I will first take the soapbar from toilet 1"
> **Subgoal:** Take soapbar 1 from toilet 1 (Search loc: null)
> **Subgoal Result:** ✗ Repeatedly failed
>
> **Updated Belief State (Abbreviated):**
>
> - **Symbolic Memory:** Agent: `holding soapbottle 1`
> - **Semantic:** "Error: Agent was unable to take soapbar 1 from toilet 1, indicating a potential restriction or condition not met for that action"

> **World Model Update 4**
>
> **Subgoal Generation and Execution:**
>
> **Planner Reasoning:** "Since the soapbars are confirmed to be located at toilet 1, I will attempt to take soapbar 2 from toilet 1 directly, as the previous attempts to take soapbar 1 were unsuccessful"
> **Subgoal:** Take soapbar 2 from toilet 1 (Search loc: null)
> **Subgoal Result:** ✓ Success
>
> **Updated Belief State:**
>
> - **Symbolic Memory:** Agent: `holding soapbar 2`, toilet 1: `contains=[soapbar 1, soapbottle 1]`
> - **Semantic:** "Subgoal completed; soapbar 2 successfully taken from toilet 1"

## A.6 Compatibility of Symbolic Memory with Symbolic Solvers

Our symbolic memory maintains accurate object-oriented state tracking, and in PDDL the representation maps to formal predicates compatible with a symbolic solver. In ALFWorld and Jericho, a symbolic solver would be incompatible due to environment complexity. Our method handles such complexity by integrating an LLM-based hierarchical planner with symbolic memory and semantic belief state.

## B  Usage of AI Assistants

We utilized ChatGPT for minor language edits.

## C  Artifact Licenses

- ALFWorld dataset[6]: MIT license

- AgentBoard dataset[7]: GNU General Public License, version 2.

---

[6]https://github.com/alfworld/alfworld
[7]https://huggingface.co/datasets/hkust-nlp/agentboard