# TREE OF OPTIONS: TEMPORALLY EXTENDED WORLD MODELING, PLANNING, AND EXECUTION WITH LARGE LANGUAGE MODELS

**Xiaoling Zeng**[1] **Dingyang Chen**[2*] **Qi Zhang**[1]
[1]Worcester Polytechnic Institute, [2]Amazon
xzeng4@wpi.edu, dingyangchen118@gmail.com, qzhang9@wpi.edu

## ABSTRACT

With commonsense knowledge embedded, Large Language Models (LLMs) have been repurposed as world models that can be exploited by principled planning algorithms such as Monte Carlo Tree Search (MCTS). Prior works have been limited to exploiting LLMs for low-level world modeling, i.e., predicting immediate next world states and rewards upon primitive actions, which makes them unfit for long-horizon tasks where prediction errors compound quickly over time. This work develops an alternative framework where LLMs perform world modeling on *temporally extended actions* (i.e., options), to overcome their limitations in precise world modeling at small temporal scales. At this temporal abstraction level, LLMs will also be competent in suggesting reasonable options, enabling effective planning using MCTS. To execute the planned options with the primitive actions, we again turn to LLMs by prompting them to synthesize code implementing option-conditioned policies, which LLMs are known to excel at. Empirical results in Minecraft show that this approach substantially improves performance over prior LLM-based planners on long-horizon, compositional tasks for embodied agents.

## 1 INTRODUCTION

Pre-trained large language models (LLMs) have shown remarkable effectiveness for solving embodied decision-making tasks like games and robotics. The majority of prior works have applied LLMs 1) as high-level planners that decomposes the task of interest into a series of subgoals (Ahn et al., 2022; Huang et al., 2022b;a) and 2) for generating code that implements low-level controllers over primitive actions such as motor commands for robots (Liang et al., 2022; Singh et al., 2022). In a minority of prior works (Hao et al., 2023; Zhao et al., 2023; Zhou et al., 2024), LLMs have been repurposed as *world models* that predict and evaluate the long-term outcomes of a given sequence of actions on world states (e.g., environment configurations, agent capabilities), on top of which principled model-based methods such as tree search-based planning can be employed to identify promising actions. While LLM-based world models seem appealing, prior works have restrictively shown their effectiveness in short-horizon tasks, such as object rearrangement (e.g., Blocksworld (Valmeekam et al., 2023), VirtualHome (Puig et al., 2018)) and textual question answering (e.g., math reasoning (Cobbe et al., 2021), logical reasoning (Saparov & He, 2023)), which require a small number of actions or reasoning steps to complete. This is because these methods model the world upon primitive actions at every tick of the environment with prediction errors that are inevitable by LLMs compounding quickly over time, making them unreliable for long-horizon, complex tasks.

Addressing the challenge, we propose a novel paradigm where LLM-based world models are driven by temporally extended actions, firstly formalized and referred to as *options* by Sutton et al. (1999). As a crucial difference from classical methods on options, we represent options as natural language rather than abstract symbols, unlocking the capabilities of LLMs to propose, plan, and execute them. With the option-driven world model, we perform tree search that is guided by another LLM module prompted to generate and evaluate candidate options, outputting a sequence of options as subgoals for the task. This effectively shortens the task horizon and greatly alleviates the issue of LLM

*Work does not relate to position at Amazon.

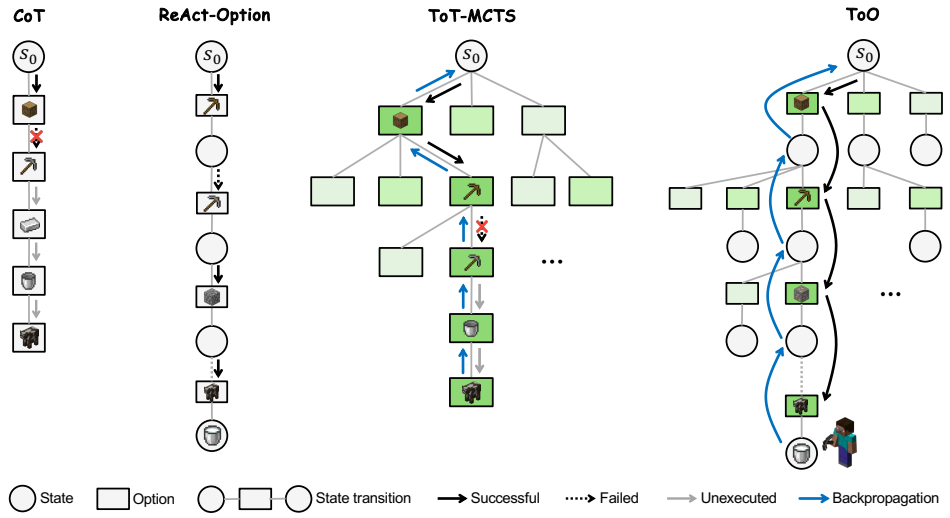


Figure 1: Illustrations of existing LLM prompting techniques compared with ToO for options planning execution. CoT generates a single linear sequence of options before executing them. ReAct-Option interleaves between proposing an option and executing it, using environment feedback and agent states to iteratively advance the execution sequence. ToT-MCTS explores multiple option branches via tree search and backpropagates reward evaluation (shades of green) through the tree to identify the most promising sequence of options, without state dynamics prediction. In addition to MCTS, ToO forwards proposed options to an option-level world model, which predicts the next state, reward, and termination. These transitions expand the search tree and guide the selection of subsequent options.

prediction errors. We refer to this paradigm as Tree of Options (ToO), incorporating two critical design choices, option-level world modeling and tree search based on LLMs, in its options planning. In our experiments, we repurpose three popular LLM prompting techniques, including Chain-of-Thought (CoT) (Wei et al., 2022), ReAct (Yao et al., 2023b), and Tree of Thoughts(ToT) (Yao et al., 2023a) for options planning. As illustrated in Figure 1, these existing techniques lack one of both ToO's design choices.

By design, ToO leverages LLMs' strengths in commonsense knowledge, high-level planning, and code generation while circumventing their weaknesses in low-level modeling at fine temporal scales. ToO treats LLMs just as black boxes to prompt and requires no finetuning. Even, ToO requires no gradient-based reinforcement learning (RL) as the only trial and error is performed by iterative code generation (Wang et al., 2023a) for option execution. This makes ToO easy to implement and efficient to deploy. On several intricate, long-horizon embodied tasks in MineDojo (Fan et al., 2022), an environment built for testing embodied agents in the game of Minecraft, ToO demonstrates superior performance over existing methods without its critical designs.

## 2 Background

This section sets up preliminaries for formalisms of decision making, options, and LLM prompting.
**MDPs.** The decision-making task faced by our agent can be formulated as a Markov decision process (MDP) that consists of a set of states $\mathcal{S}$, a set of action $\mathcal{A}$, a state transition function $p : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ where $\Delta(\mathcal{X})$ denotes the collection of probability measures on $\mathcal{X}$, and a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Starting in some initial state $s_0$ chosen by the task, at each discrete time step $t = 0, 1, \ldots$, the agent fully observes the state $s_t \in \mathcal{S}$, and chooses an action $a_t \in \mathcal{A}$ to take; the next state is sampled from the transition dynamics as $s_{t+1} \sim p(\cdot|s_t, a_t)$ while the agent receives the step reward $r_t := r(s_t, a_t)$; this yields trajectory $(s_0, a_0, r_0, s_1, \ldots)$. A (Markovian stationary) policy specifies an action-select rule as a probability distribution over the action space conditioned on the state, $\pi : \mathcal{S} \to \Delta(\mathcal{A})$. The agent aims to find a policy that maximizes the expected cumulative discounted reward, $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t]$, where $\gamma \in [0, 1)$ is the discount factor.

**Options.** Sutton et al. (1999) formalized the idea of temporally extended actions with the notion of options. An option $\omega \in \Omega$ is specified by a tuple $(\pi_\omega, \beta_\omega)$, where $\pi_\omega : \mathcal{S} \to \Delta(\mathcal{A})$ is its in-option policy, and $\beta_\omega : \mathcal{S} \to \{0, 1\}$ is its termination function. The options are executed in a *call-and-return* manner: the agent executes the current option $\omega$ by following its in-option policy $\pi_\omega$ until termination indicated by termination function $\beta_\omega$, at which point the agent picks the next option in $\Omega$ to execute.

**LLM prompting.** Let $\texttt{LLM}(y|x)$ denote the probability of getting output $y$ with input $x$ given to an LLM with $x, y$ being token sequences. We use $\texttt{LLM}_{\texttt{prompt}}(y|x) = \texttt{LLM}(y \mid \texttt{prompt}(x))$ to denote the practice of wrapping input $x$ with a certain prompt format (e.g., task descriptions, input-output examples) with wrapper $\texttt{prompt}(\cdot)$, which is the most common way to use LLMs. To improve multi-step reasoning capabilities for complex input-output relations, Chain of Thoughts (CoT) by Wei et al. (2022) introduces intermediate prompting steps between input $x$ and output $y$, enabling the model to generate a series of logically coherent intermediate steps. The thoughts, as token sequences, are sampled sequentially $z_i \sim \texttt{LLM}_{\texttt{CoT}}(\cdot \mid x, z_1, \ldots, z_{i-1})$. Then the final output $y$ is generated conditioned on the full thought sequence: $y \sim \texttt{LLM}_{\texttt{CoT}}(\cdot \mid x, z_1, \ldots, z_T)$. Building on CoT, Tree of Thoughts (ToT) by Yao et al. (2023a) introduces a tree-structured prompting paradigm by sampling multiple i.i.d. thoughts at each step: $\{z_i^{(j)}\}_j \overset{\text{iid}}{\sim} \texttt{LLM}_{\texttt{ToT}}(\cdot \mid \tau_{i-1})$ where $\tau_{i-1} := [x, z_1, \ldots, z_{i-1}]$. In ToT, the path from the root to each tree node represents $\tau_{i-1}$, a trace of intermediate steps, and an edge denotes a candidate next-step thought $z_i^{(j)}$ that expand the current prompting trace. By exploring multiple candidate paths, ToT demonstrates superior performance over CoT in complex tasks such as mathematical reasoning and creative generation.

## 3 Tree of Options

This section describes Tree of Options (ToO), our paradigm that applies LLM prompting techniques to implement the option framework. As an overview, our ToO solves a decision-making task in the following two phases:

- **Options planning** (Section 3.1). In the first phase, ToO prompts an LLM for the given task to generate a sequence of options represented by natural language. These options are planned out by Monte Carlo Tree Search (MCTS) that is guided by an option-level world model and an option generator, both of which are implemented via LLM prompting techniques.
- **Options execution** (Section 3.2). In the second phase, ToO executes the planned option sequence in the call-and-return manner, with the in-option policies and termination function implemented by LLM-generated code.

In our experiments, the planning in ToO is one-shot, i.e., by invoking the MCTS procedure from the initial state, the output sequence of options is used to solve the task without need of additional planning. A straightforward extension is option-level online planning, i.e., re-planning from current states during task execution for an updated sequence of future options. With ToO, we find one-shot planning already demonstrates superior performance on those complex and long-horizon tasks in MineDojo, and we leave online planning for future work.

### 3.1 Option World Modeling and Planning with LLMs

ToO builds a search tree of options where an edge $\hat{s}_{i-1} \xrightarrow{\omega_i} \hat{s}_i$ represents option $\omega_i$ executed from state $\hat{s}_{i-1}$ and terminated in state $\hat{s}_i$. Here, each tree node stores state $\hat{s}_i$ that is obtained from the *option-driven dynamics predictor* implemented by prompting an LLM:

$$\hat{s}_i \sim \texttt{LLM}_{\texttt{dynamics}}(\hat{s}_i \mid \hat{s}_{i-1}, \omega_i). \tag{1}$$

As the initial state $s_0$ is observable, we set $\hat{s}_0 = s_0$ as the root node. Therefore, a path from the root to a node of $\hat{s}_i$ yields a trajectory $\tau_i := [\hat{s}_i, \omega_1, \ldots, \omega_i]$, with a slight abuse of notation against Section 2. ToO adopts an MCTS to build the tree and search over candidate option sequences to execute, iterating over the following four procedures:

**i) Options selection:** A search iteration begins with a traversal from the root node to a leaf node. We select the edges/options during the traversal by applying the Upper Confidence Bound (UCB)

strategy to balance exploration and exploitation (Auer et al., 2002; Kocsis & Szepesvári, 2006):

$$\omega_{i+1} \leftarrow \arg\max_{\omega \in E(\tau_i)} \left[ Q(\tau_i, \omega) + c \cdot \sqrt{\frac{\ln N(\tau_i)}{N(\tau_i, \omega)+1}} \right]$$

where $E(\tau_i)$ is the set of outgoing edges, i.e., candidate next options from node $\tau_i$; $Q(\tau_i, \omega)$ is an estimate the expected total reward starting from executing $\omega$ from node $\tau_i$, as detailed later in the Backpropagation procedure; $N(\tau_i)$ and $N(\tau_i, \omega)$ denotes the number of times the node $\tau_i$ and the node-option pair $(\tau_i, \omega)$ have been visited, respectively; and the exploration weight $c$ quantifies the exploration and exploitation tradeoff.

**ii) Expansion:** Once a leaf node $\tau_{i-1}$ is reached, we expands the tree by proposing multiple candidate next options from the leaf node that are generated by prompting an LLM:

$$\tilde{\omega}_i^{(1)}, \cdots, \tilde{\omega}_i^{(k)} \sim \texttt{LLM}_{\texttt{propose}}\big(\cdot \mid \tau_{i-1}\big) \tag{2}$$

Here, to improve prompting efficiency, $\texttt{LLM}_{\texttt{propose}}$ generates multiple candidate options upon a single prompt.

We have found LLMs easily propose linguistic plausible yet infeasible options. As a remedy, we incorporate a feasibility test that immediately applies to the proposed options before expansion. Specifically, another prompting of LLM is invoked to assess the feasibility of the proposed options in the current state of the node, taking into account task-specific information. For options deemed infeasible, we prompt the LLM to proactively generate semantically aligned prerequisite steps that pursue similar high-level intentions while satisfying execution constraints, returning modified options with better feasibility as the final recommendation:

$$\omega_i^{(j)} \sim \texttt{LLM}_{\texttt{feasibility}}\big(\cdot \mid \tau_{i-1}, \tilde{\omega}_i^{(j)}\big), \quad j = 1, \ldots, k. \tag{3}$$

We call $\texttt{LLM}_{\texttt{propose}}$ in (2) and $\texttt{LLM}_{\texttt{feasibility}}$ in (3) together as our *option generator*. In contrast to post-execution feedback mechanisms such as self-validation and self-reflection in recent works (Shinn et al., 2023; Wang et al., 2023a; Liu et al., 2024b), our option generator ensures that the expanded branches are not only linguistically coherent but also environmentally executable.

Conditioned on the recommended options $\omega_i^{(j)}$, we use the dynamic predictor to simulate the state-option transition as in (1), i.e., $\hat{s}_i^{(j)} \sim \texttt{LLM}_{\texttt{dynamics}}\big(\hat{s}_i^{(j)} \mid \hat{s}_{i-1}, \omega_i^{(j)}\big)$, which expands the leaf node $\tau_{i-1} = [\hat{s}_{i-1}, \omega_1, \ldots, \omega_{i-1}]$ with children $\tau_i^{(j)} = \big[\hat{s}_i^{(j)}, \omega_1, \ldots, \omega_{i-1}, \omega_i^{(j)}\big]$.

**iii) Option-driven rollout:** If state $\hat{s}_l$ of the newly expanded leaf node $\tau_l$ is a terminal state , we skip this rollout procedure. We tell if a state is terminal by prompting an LLM as termination predictor for a binary indicator done:

$$\text{done} \sim \texttt{LLM}_{\texttt{termination}}(\cdot \mid \hat{s}). \tag{4}$$

Otherwise, to evaluate the non-terminal state, we simulate a state-action trajectory by repeatedly invoking the option-driven dynamics predictor (1) from it, yielding $(\hat{s}_l, \omega_{l+1}, \hat{s}_{l+1}, \omega_{l+2}, \ldots, \hat{s}_L)$ until reaching a terminal state, i.e., $\hat{s}_L$ is terminal, or a maximum rollout length $d$, i.e., $L = d$, where the options therein are chosen by some option rollout policy. For simplicity and similar to Hao et al. (2023), in our experiments, we choose the next option from the option generator that maximizes the immediate reward according to a *reward predictor*:

$$\omega_{l'+1} \leftarrow \arg\max_{\omega \in \Omega(\tau_{l'})} \hat{r}(\tau_{l'}, \omega) \tag{5}$$

where $\tau_{l'} = [\hat{s}_{l'}, \omega_1, \ldots, \omega_{l'}]$, $l' = l, l+1, \ldots, L-1$ and $\Omega(\tau_{l'})$ is a set of options proposed and modified by the option generator (2) (3) as described in the Expansion procedure. Inspired by heuristic state evaluation (Yao et al., 2023a) and the demonstrated advantages of reward composition (Hao et al., 2023; Ma et al., 2024), we implement a reward predictor $\hat{r}$ by prompting an LLM on a set of evaluation questions $\mathcal{Q} = \{q\}$, where each question assessing a unique aspect of interest, such as feasibility, task relevance, and logical coherence, to better guide search in long-horizon, compositional Minecraft tasks. The final reward is scalarized with weights $c_q$:

$$\begin{aligned} \hat{r}(\tau, \omega) &= \textstyle\sum_{q \in \mathcal{Q}} c_q r_q(\tau, \omega), \\ \text{with } r_q(\tau, \omega) &\sim \texttt{LLM}_{\texttt{reward}}(\cdot \mid \tau, \omega, q). \end{aligned} \tag{6}$$

**iv) Backpropagation:** The cumulative reward is then propagated back from the leaf to the root, updating the Q-value of each visited node-option pair as

$$Q(\tau_i, \omega_{i+1}) \leftarrow Q(\tau_i, \omega_{i+1}) + \frac{R_i - Q(\tau_i, \omega_{i+1})}{N(\tau_i, \omega_{i+1})}$$
$$\text{with } R_i = \sum_{j=i}^{L} \gamma^{j-i} \hat{r}(\tau_j, \omega_{j+1}), \quad i = 1, \ldots, l$$

which will guide the option selection in the next iteration.

By iterating over the four procedures, ToO progressively builds up a search tree over options with Q-value estimates. The sequence of options to recommend is obtained by greedily choosing the option maximizing the Q-value estimate.

### 3.2 Options Execution with LLM-Generated Code

In its second phase, ToO executes from the initial state the sequence of options $(\omega_i)_{i=1,2\ldots}$ as recommended from the MCTS in the first phase in the call-and-return manner, where the in-option policies $\pi_{\omega_i}$ over primitive actions are implemented by LLM-generated code (Austin et al., 2021; Liang et al., 2022). As code generated by LLMs is prone to errors, we adopt an iterative prompting mechanism similar to prior work (Ni et al., 2023; Shinn et al., 2023; Skreta et al., 2023; Wang et al., 2023a). Specifically, suppose we aim to execute option $\omega_i$ starting from state $s_{t_1}$ at timestep $t_1$, an LLM prompting generates the first in-option policy:

$$\pi_{\omega_i}^{(1)} \sim \texttt{LLM}_{\texttt{in-option}}(\cdot \mid s_{t_1}, \omega_i).$$

In iterations $m = 1, 2, \ldots$, in-option policy $\pi_{\omega_i}^{(m)}$ is executed from state $s_{t_m}$ at timestep $t_m$ to some $s_{t_{m+1}}$ at timestep $t_{m+1}$, so that either $s_{t_{m+1}}$ terminates the option, i.e., $\beta_{\omega_i}(s_{t_{m+1}}) = 1$, according the termination function implemented by another LLM prompting:

$$\beta_{\omega_i}(s_{t_{m+1}}) \sim \texttt{LLM}_{\beta}(\cdot \mid s_{t_{m+1}}, \omega_i)$$

or the execution was erroneous and we refine the policy

$$\pi_{\omega_i}^{(m+1)} \sim \texttt{LLM}_{\texttt{in-option}}(\cdot \mid s_{t_{m+1}}, \omega_i, \text{feedback}^{(m)})$$

where $\text{feedback}^{(m)}$ is the error messages from the code interpreter and/or the task environment when executing $\pi_{\omega_i}^{(m)}$. The refined in-option policy $\pi_{\omega_i}^{(m+1)}$ will be executed from state $s_{t_{m+1}}$ as the next iteration until the option is terminated or a maximum number of iterations is reached.

## 4 Experiments

### 4.1 Experimental Setup

**Environment and tasks.** We perform our experiments in Minecraft, an open-ended and interactive 3D environment with a compositional action space supporting diverse tasks, making it an ideal platform for assessing embodied decision-making tasks. Our task suite consists of eight representative objectives defined in MineDojo (Fan et al., 2022) as listed in the left most column of Table 1. These tasks are challenging and diverse, as all of them require long-horizon planning and execution in vast 3D terrains, ranging from relatively basics objectives like obtaining leather, where the agent needs to locate moving animals while preparing prerequisite tools, to complex tasks like crafting diamond tools that demand sequential resource gathering and progression through wood, stone, iron tiers, etc. All tasks have clearly verifiable success conditions used for the performance metric.

**LLMs, prompts, and control primitives.** To ensure apple-to-apple comparisons, all methods including ToO and baselines (detailed below) share the same implementation details on LLMs and their prompts whenever appropriate. Specifically, we utilize OpenAI's GPT-4 and GPT-3.5-turbo for options planning, with prompts minimally modified from those in Wang et al. (2023a), which we detail in Appendix A. We set LLM temperature to 0.2 for the option proposer to promote diversity and to 0 for all other modules to ensure consistent predictions. For options execution, we follow Voyager to match planned options to Voyager's skill library by encoding both options and skill descriptions, with the resulting in-option policies execute control primitives via the Mineflayer API for low-level interaction.

Table 1: Quality metrics on options execution. SR: success rate across three execution runs Time/Attempts: completion time in minutes/number of attempts averaged over successful runs. Bold indicate the best results among the compared methods. The complete set of raw per-run results is reported in Table 4 in Appendix C.1.

| Tasks | Model | SR | Time | Attempts | Tasks | Model | SR | Time | Attempts |
|---|---|---|---|---|---|---|---|---|---|
| Obtain leather | ReAct-action | - | - | - | Collect a lava bucket | ReAct-action | - | - | - |
| | CoT | 3 / 3 | 16.17 | 14 | | CoT | 2 / 3 | 19.3 | 22 |
| | ReAct-Option | 3 / 3 | **11.13** | 11.67 | | ReAct-Option | 3 / 3 | 17.27 | 21 |
| | ToT-MCTS | 2 / 3 | 12.41 | 15.5 | | ToT-MCTS | 2 / 3 | 21.68 | **17** |
| | ToO | **3 / 3** | 11.34 | **10** | | ToO | **3 / 3** | **15.7** | 17.3 |
| Cook meat | ReAct-action | - | - | - | Craft a golden sword | ReAct-action | - | - | - |
| | CoT | 2 / 3 | 17.65 | 23 | | CoT | 1 / 3 | 21.95 | 28 |
| | ReAct-Option | 3 / 3 | 12.87 | 16.67 | | ReAct-Option | 1 / 3 | 24.53 | 27 |
| | ToT-MCTS | 3 / 3 | 14.96 | 19.67 | | ToT-MCTS | 1 / 3 | 29.17 | 25 |
| | ToO | **3 / 3** | **9.29** | **10.33** | | ToO | **3 / 3** | **18.03** | **19.67** |
| Shear a sheep | ReAct-action | - | - | - | Craft a compass | ReAct-action | - | - | - |
| | CoT | 3 / 3 | 25.10 | 21 | | CoT | 1 / 3 | 24.1 | 29 |
| | ReAct-Option | 2 / 3 | 18.38 | 20 | | ReAct-Option | 2 / 3 | 19.45 | 24 |
| | ToT-MCTS | 2 / 3 | **17.03** | 17 | | ToT-MCTS | 2 / 3 | 24.6 | **22.5** |
| | ToO | **3 / 3** | 20.62 | **16.33** | | ToO | **3 / 3** | **16.75** | 23.3 |
| Milk a cow | ReAct-action | - | - | - | Craft a diamond pickaxe | ReAct-action | - | - | - |
| | CoT | 1 / 3 | 26.35 | 27 | | CoT | 0 / 3 | N/A | N/A |
| | ReAct-Option | 2 / 3 | 23.02 | 29 | | ReAct-Option | 0 / 3 | N/A | N/A |
| | ToT-MCTS | 2 / 3 | 29.34 | 28 | | ToT-MCTS | 1 / 3 | 25.95 | 27 |
| | ToO | **3 / 3** | **21.8** | **21.33** | | ToO | **3 / 3** | **10.49** | **15.33** |

**Baselines.** We compare our ToO with the baselines illustrated in Figure 1 that only differ in options planning:

- *CoT* directly decomposes the task goal into intermediate steps (interpreted as options) in a Chain-of-Thoughts manner with the initial state provided in the prompt.
- *ReAct-Option* adapts ReAct (Yao et al., 2023b) to operate over options. It interleaves reasoning and option execution via CoT prompting, and reacts to environment feedback and agent states to guide option selection.
- *ToT-MCTS* performs Tree-of-Thoughts prompting to generator candidate options. Compared to ToO, it lacks explicit state transition or validation, with each node formalized as $\tau_i^{\text{ToT}} := [s_0, \omega_1, \ldots, \omega_{i-1}]$. Further, similar to ToO, ToT-MCTS employs a reward module and performs MCTS to search for an option sequence.

In addition to these option-based methods, we consider *ReAct-Action* as a baseline without temporal abstraction by applying ReAct to generate and execute primitive actions.

**Hyperparameters.** Before presenting the results, we review the critical hyperparameters of the methods during their optionis planning and execution:

*Planning:* CoT and ReAct-Option does not involve tricky hyperparmeters for its options planning. For ToO and ToT-MCTS, both perform a similar MCTS procedure that is parameterized by $k$: the number of candidate options generated that determines the branching factor of the tree, $d$: the maximum rollout length that determines the depth of the tree, $iter$: the total number of search iterations, and $c_q$: the weights used for the reward module in (6). Here, $(k, d, iter)$ determines computation resources used for the MCTS. We first set $c_q$ to unit weight (i.e., $\pm 1.0$) for all $q$ and then tune it larger (to 1.2) for questions $q$ that directly reflect task progress (like reward and consistency, referring to Appendix A.6 for details).

*Execution:* All methods execute their planned sequence options with the same iterative prompting mechanism as described in Section 3.2. We refer each iteration here as an "attempt" and cap the maximum number of attempts to 30.

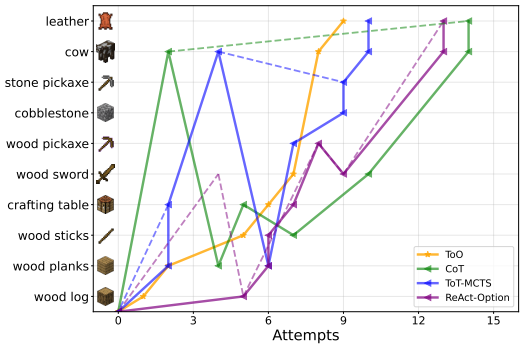


Figure 2: For `obtains leather`, an execution trajectorys of each method is visualized, where each node represents an option to execute. Solid lines denote the executed low-level action sequence, while dashed connections indicate the initial planned option sequence. Alignment between the two traces reflects how faithfully execution follows the planned sequence of options.

### 4.2 Main Results

As all methods except for ReAct-Action share the same options execution technique, the main goal of our experiments is to evaluate the quality and efficiency of their options planning. We here set the MCTS hyperparameters as $(k = 3, d = 12, iter = 10)$ for both ToT-MCTS and ToO; we will vary them for scalability analysis in Section 4.3.

**Quality of the planned options.** To evaluate the quality of planned options, we execute them sequentially for multiple runs and record the success rate for the task. We also record the total number of attempts and wall-clock time at completion. Table 1 presents these results, showing that our ToO achieves 100% success rate across all tasks, while other methods fail on several tasks. Moreover, for the tasks all methods complete successfully, ToO consistently consumes fewer attempts and less wall-clock time. These results indicate that ToO generates more coherent and reliable plans.

We present a detailed analysis of how the options planned by different methods are executed, with their progress chart illustrated in Figure 2 for the `obtain leather` task (refer to Figure 5 and Figure 6 in Appendix C.1 for other tasks). The x-axis denotes the number of attempts that have been used and the y-axis records the sequence of options actually executed (solid lines), which might be inconsistent with the planned options (dashed lines) due to iterative code refinement. The results first reveal that purely reactive primitive execution without abstraction prevents ReAct-Action from making meaningful progress on long-horizon tasks. Introducing option-level reasoning, as implemented in ReAct-Option, partially alleviates this issue but remains insufficient. CoT often generates options that are seemingly concise yet difficult to realize. As illustrated in the case study (refer to Figure 8 in Appendix C.2), CoT directly proposes *find a cow* followed by *kill a cow* in one run. However, the missing prerequisite step *craft a wooden sword* leads to repeated backtracking, resulting in 14 prompting attempts and the lowest execution efficiency. ToT-MCTS plans a more reasonable path but still overlooks essential crafting steps, causing unnecessary recovery. In contrast, ToO leverages its world model for option-wise state transition prediction during search, enabling it to identify an executable option sequence that avoids backtracking and completes the task efficiently. The results highlight ToO's unique design choices are critical for options planning.

**Token efficiency.** To evaluate efficiency, Table 2 compares the number of API calls and tokens during LLM prompting for both options planning and execution on a single run. As the four methods differ greatly in the LLM resources consumed per run, success rate alone (as reported in Table 1) ignores efficiency. To provide a fair comparison, we adopt the pass@B metric that assigns all methods the same token budget $B$ in total for both planning and execution and allows each method to execute as many complete runs as the budget permits. A task is considered "passed" if at least one of these runs succeeds, and pass@B in Table 2 reports the proportion of tasks passed out of the eight tasks, where ToO having passed all the tasks, CoT and ReAct-Option each achieve a pass@B of 0.75, while ToT-MCTS succeed on 62.5% of tasks. These results confirm ToO's efficiency with the best performance under the same LLM resource budget.

Table 2: LLM resources comparison. #APIs and #Tokens: number of LLM calls and the corresponding tokens used averaged over the runs on all tasks. Pass@B: fraction of tasks successfully solved up to a shared token budget $B$, set to the cost of a single ToO run.

| Method | Planning | | Execution | | pass@B |
|---|---|---|---|---|---|
| | #APIs | #Tokens | #APIs | #Tokens | |
| CoT | 0.001k | 0.5k | 0.06k | 194k | 0.75 |
| ReAct-Option | 0.03k | 9.2k | 0.06k | 171k | 0.75 |
| ToT-MCTS | 0.6k | 215k | 0.06k | 175k | 0.625 |
| ToO | 0.9k | 347k | 0.03k | 112k | 1.0 |

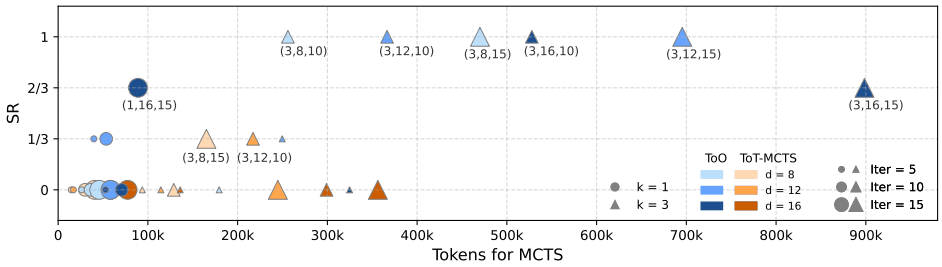


Figure 3: Scalability analysis of MCTS-based options planning with increasing branching factor $k$, rollout depth $d$, and number of iteration $iter$. Each point represents a configuration $(k, d, iter)$, plotted by its token consumption (x-axis) and success rate (SR) (y-axis) on the `craft a diamond pickaxe` task. $k$ is distinguished by marker shape (circle for $k$=1, triangle for $k$=3), $(d = 8, 12, 16)$ by increasing color intensity, $(iter = 5, 10, 15)$ by marker size. Results are shown for ToO in blue and ToT-MCTS in orange.

### 4.3 Scalability Analysis

The hyperparameters of $(k, d, iter)$ in ToT-MCTS and ToO determine the amount of resources consumed by their MCTS-based options planning, as they determine the branching factor, the rollout depth, and the number of search iterations, respectively. Here, we perform a scalability analysis by varying the hyperparameters and measuring LLM resources used against the success rate of the corresponding options planned for the `craft a diamond pickaxe` task, as shown in Figure 3. The lower-left cluster of circular markers indicates that a larger branching factor ($k = 3$) is necessary for this difficult task, as it enables broader exploration of candidate options. Given sufficient branching, the effect of rollout depth becomes more pronounced. A shallow depth of $d = 8$ provides insufficient lookahead and forces the actor to compensate for missing prerequisites during execution. In contrast, an excessive search with $d = 16$ expands the tree aggressively and leads to a substantial increase in token consumption without consistent performance gains. A moderate depth of $d = 12$ provides the best balance, achieving high success rates with manageable cost. The iteration budget also plays a critical role in planning stability. With $iter = 5$ iterations, the planner fails in most configurations, except when using $d = 12$. Increasing the budget to ten iterations yields stable success across multiple depths. However, even $iter = 15$ iterations remain insufficient at $d = 16$, indicating that very deep trees require significantly more simulations to be effective. Overall, both ToT-MCTS and ToO perform best with a larger branching factor, a moderate rollout depth, and an adequate number of iterations. Based on these observations, the comparative experiments in Section 4.2 set $(k = 3, d = 12, iter = 10)$.

### 4.4 Ablation Studies

We conduct ablation studies to assess the contribution of the LLM-based world model and feasibility check components in ToO. Specifically, **ToO w/o WM** removes the LLM-based world model and performs tree search using the same node representation as used by ToT, $\tau_i^{\text{ToT}} := [s_0, \omega_1, \ldots, \omega_{i-1}]$, without modeling state transitions; **ToO w/o FC** retains the same LLM-based world model for state transition simulation but disables feasibility checking during planning. As shown in Table 5, both components are critical to ToO's performance. Specifically, removing the world model forces plan-

ning to rely solely on the initial state and accumulated thought history, eliminating explicit conditioning over environmental dynamics. Consequently, the resulting plans lack state transition information, leading to degraded planning accuracy and frequent deviations, even when feasibility checks are retained. Conversely, removing the feasibility check preserves state transition imagination but allows the search to expand infeasible branches, resulting in unexecutable options, redundant attempts, and lower execution efficiency. When both components are enabled, ToO achieves higher success rates and more efficient execution. Additional qualitative comparisons are provided in Figure 9.

## 5 Related Work

**Temporal abstractions.** Building agents that solve decision-making tasks in a hierarchical way is a long standing topic. The options framework proposed by Sutton et al. (1999); Precup (2000) formalizes the notion of temporally extended actions in MDPs and for RL. Prior works on options mainly focus on option discovery, where the in-option policy and/or the termination function associated with an option is not given a priori but learned through trial-and-error RL, most commonly using gradient-based approach (Sorg & Singh, 2010; Comanici & Precup, 2010; Levy & Shimkin, 2011; Silver & Ciosek, 2012; Bacon et al., 2017). With minor differences, another line of works refer to such temporally extended actions as subgoals (Kulkarni et al., 2016; Tessler et al., 2017; Vezhnevets et al., 2017), which typically adopt a manager-work architecture where the manager prescribes subgoals for the work to achieve with primitive actions. For domains like board games, LLMs have been adopted but without involving temporal abstractions (Schultz et al., 2025), as it is still an open problem to design/discover useful temporal abstractions therein. In contrast, this work implements temporally extended actions with LLMs, where options are represented as natural language (instead of abstract symbols), so that their semantics such as in-option policies and termination conditions are generated by LLMs with their commonsense knowledge and coding capabilities. In this sense, this work follows recent ones on using LLMs to plan out options/subgoals for robots and video game agents (Ahn et al., 2022; Huang et al., 2022b;a; Wang et al., 2023a; Liu et al., 2024a). These prior works either need RL that requires a lot of samples or employs only CoT-like planning, while we are the first to plan options/subgoals with tree search and executes them using LLMs only without extensive trial-and-error RL.

**World modeling and tree search with LLMs.** Existing work is relatively limited on repurposing LLMs as world models or heuristics policies in a way that can be incorporated into tree search algorithms like MCTS, as most prior work uses LLMs for planning in a linear, reactive manner like CoT (Wei et al., 2022; Singh et al., 2022; Yao et al., 2023b; Shinn et al., 2023; Liu et al., 2024c). Zhang et al. (2023) use LLMs as a heuristics policy to guide MCTS for code generation. Hao et al. (2023); Zhao et al. (2023); Zhou et al. (2024) are the first to repurpose LLMs as a world model (i.e., transition and reward functions), which is incorporated by MCTS to solve reasoning and planning tasks. Zhou et al. (2023) propose LATS, which expands LLM-generated actions through environment interaction. As a key difference, these works operate with primitive actions with no abstractions, while our ToT leverages (only) LLMs to perform world modeling and planning over the temporal abstraction of options.

**Agents for Minecraft.** Minecraft is a popular open-world sandbox game that has been serving as a benchmark for building efficient and generalized agents. We refer to Liu et al. (2024a); Wang et al. (2023a) and references therein for a discussion on endeavors on Minecraft that do not leverage LLMs. This work follows recent ones that leverage LLMs as a high-level planner in Minecraft by decomposing the task into subgoals/options (Wang et al., 2023b; Nottingham et al., 2023; Yuan et al., 2023; Wang et al., 2023a). Unlike these works, we employs tree search as the planning algorithm over options that is enabled and enhanced via our option-driven world modeling, all implemented via LLMs only.

## 6 Conclusion

To conclude, we propose Tree of Options (ToO), a novel paradigm that leverage pretrained large language models to perform world modeling and tree search to plan over temporally extended actions, i.e., options, and then execute planned options via code generation. ToO is particularly designed

to tackle complex, long-horizon embodied decision-making tasks. Our experiments demonstrate its superior performance in such tasks in the game of Minecraft.

### ACKNOWLEDGMENTS

This work was supported by National Science Foundation (NSF) awards 2544947 and 2544949.

## REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, 2017.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Gheorghe Comanici and Doina Precup. Optimal policy switching algorithms for reinforcement learning. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pp. 709–714, 2010.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=rc8o_j8I8PX.

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

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pp. 9118–9147. PMLR, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022b.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

Kfir Y Levy and Nahum Shimkin. Unified inter and intra options learning using policy gradient methods. In *European Workshop on Reinforcement Learning*, pp. 153–164. Springer, 2011.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

Shaoteng Liu, Haoqi Yuan, Minda Hu, Yanwei Li, Yukang Chen, Shu Liu, Zongqing Lu, and Jiaya Jia. Rl-gpt: Integrating reinforcement learning and code-as-policy. *Advances in Neural Information Processing Systems*, 37:28430–28459, 2024a.

Shunyu Liu, Yaoru Li, Kongcheng Zhang, Zhenyu Cui, Wenkai Fang, Yuxuan Zheng, Tongya Zheng, and Mingli Song. Odyssey: Empowering minecraft agents with open-world skills. *arXiv preprint arXiv:2407.15325*, 2024b.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024c. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IEduRUO55F.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pp. 26106–26128. PMLR, 2023.

Kolby Nottingham, Prithviraj Ammanabrolu, Alane Suhr, Yejin Choi, Hanna Hajishirzi, Sameer Singh, and Roy Fox. Do embodied agents dream of pixelated sheep?: Embodied decision making using language guided world modelling. *ARXIV.ORG*, 2023. doi: 10.48550/arXiv.2301.12050.

Doina Precup. *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8494–8502, 2018.

Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=qFVVBzXxR2V.

John Schultz, Jakub Adamek, Matej Jusup, Marc Lanctot, Michael Kaisers, Sarah Perrin, Daniel Hennes, Jeremy Shar, Cannada A. Lewis, Anian Ruoss, Tom Zahavy, Petar Veličković, Laurel Prince, Satinder Singh, Eric Malmi, and Nenad Tomasev. Mastering board games by external and internal planning with language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=KKwBo3u3IW.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

David Silver and Kamil Ciosek. Compositional planning using optimal option models. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pp. 1267–1274, 2012.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv preprint arXiv:2303.14100*, 2023.

Jonathan Sorg and Satinder Singh. Linear options. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pp. 31–38, 2010.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Chen Tessler, Shahar Givony, Tom Zahavy, Daniel Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI conference on artificial intelligence*, 2017.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models - a critical investigation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=X6dEqXIsEW.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pp. 3540–3549. PMLR, 2017.

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

Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv: Arxiv-2302.01560*, 2023b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Haoqi Yuan, Chi Zhang, Hongcheng Wang, Feiyang Xie, Penglin Cai, Hao Dong, and Zongqing Lu. Plan4mc: Skill reinforcement learning and planning for open-world minecraft tasks. *arXiv preprint arXiv: 2303.16563*, 2023. URL https://arxiv.org/abs/2303.16563v1.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Lr8cOOtYbfL.

Zirui Zhao, Wee Sun Lee, and David Hsu. Large language models as commonsense knowledge for large-scale task planning. *Advances in neural information processing systems*, 36:31967–31987, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=njwv9BsGHF.

# A METHOD DETAILS

## A.1 STATE REPRESENTATION

We define the state $\hat{s}_i$ as a collection of attributes of the environment and the agent's state in Minecraft. It includes the inventory and equipment, the sets of nearby blocks and entities within a 32 block radius, and any recently seen blocks or visible chests, the current biome and time of day, the agent's health and hunger levels, and its position given by coordinates $(x, y, z)$.

**Agent state example**

Biome:
Time: night
Nearby blocks: water, sand, dirt, stone, grass_block, kelp, coal_ore, birch_log, seagrass, copper_ore, iron_ore
Nearby entities (nearest to farthest): cod
Health: 20.0/20
Hunger: 20.0/20
Position: x=198.7, y=64.0, z=116.5
Equipment: [None, None, None, None, 'crafting_table', None]
Inventory (4/36): {'stick': 2, 'wooden_pickaxe': 1, 'birch_planks': 3, 'crafting_table': 1}
Chests: None

## A.2 OPTION GENERATION

We formulate each option as a tuple by [verb] [quantity] [item], where the verb specifies the type of operation (e.g., mine, craft, smelt), the quantity denotes how many units involved, and the item identifies the target object. Compared to Voyager's task decomposition, which directly decomposes tasks into ["subgoal1", "subgoal2", ...] using Chain-of-Thought reasoning, our prompt conditions option generation on the predicted updated state and the history of previously selected options. Moreover, our prompt prevents abstract or redundant options and ensures that each generated option corresponds to a single executable skill which can be directly mapped to low-level skill library. The highlighted fields denote structured input entries that specify how information is provided to the procedure. All remaining prompt specification is identical across experiments.

**Option generation prompt**

You are an assistant that proposes {k} possible next subgoals that continue progress toward the final goal.

Minecraft situation:
Current state:{state}
Goal: {goal}
History actions: {option_sequence if option_sequence else "None"}

You must follow the these rules:
1. Each "task" must be a single concise actionable subgoal in the form: Mine [quantity] [block], Craft [quantity] [item], Smelt [quantity] [item], Cook [quantity] [food], Equip [item].
2. Do NOT generate abstract actions such as: explore, prepare, look for, search, go to, wait.
3. Do NOT repeat steps already completed in history.
4. Do NOT include unnecessary items (weapons, armor, torches) unless required by the goal.
5. Prioritize resource gathering, tool crafting, material processing, and crafting items for the final objective.
6. Hunger is irrelevant unless directly part of the goal.
7. A subgoal is always one concrete step; you do not need to break it into pre-steps (the

skill library handles that).

You should only respond in JSON format as described below:

Example format:
[{"reasoning": "To craft a bucket, you need iron.", "task": "Mine 3 iron ore"},
{"reasoning": "To progress toward goal, ...", "task": "..."},
{"reasoning": "...", "task": "..."}]

### A.3 State Transition

Following RAP, this module employs a fixed prompt that specifies Minecraft environment constraints and input output formats to define environment dynamics. The LLM-based world model predicts the next state $\hat{s}_{i+1}$ by given the current state $\hat{s}_i$ and the proposed option $\omega_i$, while adhering strictly to Minecraft mechanics.

**Dynamics prediction prompt**

You are simulating a Minecraft world state transition. Predict exact state changes from the option.

I will give you the following information:
Current state:{state}
Proposed option: {thought}

You must follow the these rules:
1. Exploration (find, search, explore) ONLY changes nearby_entities/blocks, NEVER inventory.
2. Crafting: consumes materials, adds products to inventory.
3. Mining adds items to inventory (if proper tool available).
4. Animal interaction (milk, shear, collect) requires specific items/animals, changes inventory.
5. If requirements not met, return original state unchanged.

Output: Pure JSON only. Start with {, end with }.

### A.4 Reward Evaluation

Motivated by prior heuristic state evaluation in Tree-of-Thought and the complexity of long-horizon tasks, we follow RAP in composing multiple evaluation dimensions rather than relying on a single scalar criterion. To evaluate how each intermediate step contributes to overall task progress, we define multiple questions to assess each option. The evaluation prompt takes the agent's current state, the sequence of previously generated options, and the ultimate task goal as input. This prompt enables a comprehensive in-context assessment of whether an option is logically coherent with the reasoning history and whether it effectively promotes the agent toward task completion.

**Reward evaluation prompt**

You are evaluating a predicted state in a Tree-of-Thought process for Minecraft.
I will give you the following information:

Action: {option}
Predicted State: {state}
Goal: {goal}

History actions: {option_sequence if option_sequence else "None"}

You should score the predicted state (0.0–1.0) on the following criteria:
[Success Probability]: Likelihood of eventually reaching the goal from this state.
[Reward Estimate]: Value of current progress toward the goal.
[Feasibility]: Whether the predicted state is realistic in Minecraft.
[Consistency]: Whether the action follows logically from history without skipping required steps.
[Relativity]: Relevance of the action to the goal (penalize unnecessary steps).
[Reachability]: Can this state be reached from the previous state using the action, given inventory/tool prerequisites?
[Task Specificity]: How concrete and actionable the action is (e.g., "Craft stone pickaxe" vs vague ideas like "find resources").
[Redundancy/Overkill]: Penalize redundant or unnecessarily high-tier upgrades for the goal.
[Reasoning]: Short explanation.

Strictly follow the format and provide all fields with numeric scores.

### A.5 Feasibility Check

To further guarantee that candidate options are executable within the Minecraft environment, we incorporate a feasibility check module. This component evaluates whether an intermediate option can be carried out given the agent's current state and available resources. By doing so, it prevents the agent from hallucinating unrealistic traces (e.g., *collect milk* by attempting *find a cow* and *milk a cow* without crafting the required bucket).

**Feasibility check prompt**

You are checking the feasibility of a Minecraft action.

I will give you the following information:
Current state:{state}
Proposed action: {option}

You must follow the following criteria:
1. Check if required materials exist in inventory.
2. Check if required tools/entities are available.
3. Determine if action is feasible.

Example check:
Action: craft bucket.
Check: Count iron ingots in inventory.
- If the count is less than 3, it is not feasible (insufficient materials).
- If the count is at least 3, it is feasible (sufficient materials available).

Return only the feasibility assessment in this exact format:
{
"action_valid": true/false,
"reason": "explain why the action is or is not feasible."
}

When an option is deemed infeasible, we leverage the explanation generated by the feasibility prompt to guide the LLM in producing the corresponding prerequisite steps, thereby enabling the agent to recover by explicitly addressing the missing conditions.

**Algorithm 1** ToO

**Input:** Initial root $\tau_0 = \hat{s}_0$, branching factor $k$, maximum rollout length $L$, iteration budget $iter$, discount factor $\gamma$, exploration weight $w$
**Output:** Reasoning trajectory $\tau_L := [\hat{s}_L, \omega_1, \ldots, \omega_L]$
▷ SEARCH$(\tau_0, iter)$
**for** $n = 1$ to $iter$ **do**
    SIMULATE$(\tau_0, \text{False}, 0)$
**end for**
**return** trajectory by greedily following $\arg\max_{\omega \in E(\tau_l)} Q(\tau_l, \omega)$ until terminal
▷ SIMULATE$(\tau_l, \text{done}, l)$
**while** done $=$ False $\wedge\ l \leq L$ **do**
▷ Expansion
    **if** $\tau_l \notin \mathcal{T}$ **then**
        Add $\tau_l$ to tree, initialize $N(\tau_l) \leftarrow 0$
        **return** ROLLOUT$(\tau_l, \text{done}, l)$
    **end if**
    $N(\tau_l) \leftarrow N(\tau_l) + 1$
    $\omega_{l+1} \leftarrow \arg\max_{\omega \in \Omega(\tau_l)} \left[ Q(\tau_l, \omega) + w\sqrt{\frac{\ln N(\tau_l)}{N(\tau_l, \omega)+1}} \right]$ ▷ Option Selection
    $\tau_{l+1} \leftarrow$ child node of $\tau_l$ via $\omega_{l+1}$
    $R \leftarrow$ SIMULATE$(\tau_{l+1}, \text{done}, l+1)$
    $Q(\tau_l, \omega_{l+1}) \leftarrow Q(\tau_l, \omega_{l+1}) + \frac{R - Q(\tau_l, \omega_{l+1})}{N(\tau_l, \omega_{l+1})}$ ▷ Backpropagation
    **return** $R$
**end while**
▷ ROLLOUT$(\tau_l, \text{done}, l)$
$R \leftarrow 0,\ \ i \leftarrow l$
**while** done $=$ False $\wedge\ i \leq L$ **do**
    $\tilde{\Omega}(\tau_i) \sim \texttt{LLM}_{\texttt{propose}}(\tau_i)$ ▷ Option Generation
    $\Omega(\tau_i) \leftarrow \{\omega \in \tilde{\Omega}(\tau_i) \mid \texttt{LLM}_{\texttt{feasibility}}(\tau_i, \omega)\}$
    $\omega_{i+1} \leftarrow \arg\max_{\omega \in \Omega(\tau_i)} \hat{r}(\tau_i, \omega)$ ▷ Greedy Rollout Policy
    $\hat{s}_{i+1} \sim \texttt{LLM}_{\texttt{dynamics}}(\hat{s}_{i+1} \mid \hat{s}_i, \omega_{i+1})$
    $\hat{r}(\tau_i, \omega_{i+1}) \leftarrow \sum_{q \in \mathcal{Q}} c_q\, r_q(\tau_i, \omega_{i+1})$
    $\text{done} \sim \texttt{LLM}_{\texttt{termination}}(\text{done} \mid \hat{s}_{i+1})$
    $\tau_{i+1} \leftarrow [\hat{s}_{i+1}, \omega_1, \ldots, \omega_{i+1}]$
    $R \leftarrow R + \gamma^{\,i-l} \hat{r}(\tau_i, \omega_{i+1})$
    $i \leftarrow i + 1$
**end while**
**return** $R$

### A.6 ToO Algorithm and MCTS Configuration

We present the pseudocode of ToO in Algorithm 1. The algorithm performs option-level planning via MCTS augmented with LLM-based world modeling. Starting from the initial root state $\tau_0 = \hat{s}_0$, ToO repeatedly invokes the SIMULATE procedure for a fixed number of iterations to incrementally expand and refine the search tree. At each simulation, the algorithm traverses the tree from the root using the UCB rule until a leaf node or a terminal condition is reached. When an unexpanded option-level state $\tau_l$ is encountered, it is added to the search tree, and a ROLLOUT is immediately triggered from that state to estimate its long-term return. During rollout, at each step, an option generator produces a candidate option set, which is subsequently filtered by a feasibility checking module. The greedily selected option is then applied to an LLM-based world model, which jointly predicts the next imagined state, provides an immediate reward estimate, and determines whether the rollout should terminate. The discounted return is accumulated until termination or a maximum rollout length $L$ is reached.

Once the rollout returns a cumulative reward, the algorithm backpropagates this return along the simulated path, updating the visit counts and $Q$ value estimates for all traversed nodes. This process is repeated across iterations, enabling the search tree to effectively balance exploration and exploita-

tion. After the search budget is reached, ToO extracts the final reasoning trajectory by greedily following the options with the highest $Q$ values from the root until a terminal state is reached.

Table 3: MCTS hyperparameters.

| **Setups** | **Value** |
|---|---|
| Branching factor $k$ | 3 |
| Rollout depth $d$ | 12 |
| Iteration budget $iter$ | 10 |
| Discount factor $\gamma$ | 0.95 |
| Exploration weight $c$ | 0.7 |
| Maximum rollout length $L$ | 12 |
| Primary reward weight | 1.2 |
| Moderate reward weight | 1.0 |
| Penalty term weight | $-1.0$ |
| World model | gpt-3.5-turbo-0125 |
| Option generation model | gpt-4 |

All configurations are fixed across tasks and shared by both ToT-MCTS and ToO to ensure a fair comparison. As mentioned in Section 3.1, each MCTS node corresponds to an option-level state $\tau_t$, which consists of the agent state and the accumulated option history. The search process is controlled by three key parameters: branching factor $k$, maximum rollout length $L$, and iteration budget $iter$. Based on the scalability analysis in Figure 4, we adopt a larger branching factor to encourage sufficient exploration over candidate options in long-horizon tasks. A moderate rollout depth is chosen to balance lookahead capability and computational cost, while the iteration budget is set to ensure stable planning performance without excessive token consumption.

In addition to the search parameters, we adopt a simple yet effective reward design to evaluate states during tree search. Instead of assigning separate weights to individual criteria, reward terms are grouped by their roles, with each group sharing the same weight. Specifically, primary reward weights are assigned a higher weight to capture factors that directly reflect long-horizon task progress, including reward, reachability, relativity, and consistency. Moderate reward weights are used for auxiliary assessments such as feasibility, which help filter implausible options. Penalty terms are introduced to discourage redundant or unnecessarily high-tier options that tend to increase execution cost without contributing to task completion. All reward weights are fixed across tasks and shared by both ToT-MCTS and ToO, avoiding task-specific tuning and ensuring a consistent evaluation criterion throughout the experiments. All experiments use unified configuration summarized in Table 3.

### A.7 MCTS Result

Based on the MCTS method described in Section 3.1, we perform 10 search iterations on `Shear a sheep` task. According to the final tree structure, we can greedily select the nodes with the highest Q-values to derive the optimal planning path.

Reasoning tree of shear a sheep

```
ROOT (N:10, Q:4.520)
1. Punch tree to obtain 4 wood logs (N:5, Q:5.093)
1.1. Mine 8 cobblestone (N:3, Q:4.983)
1.1.1. Craft 1 stone pickaxe (N:3, Q:4.953)
1.1.1.1. Chop 1 tree to obtain wood logs (N:2, Q:4.873)
1.1.1.1.1. Craft 1 furnace (N:2, Q:4.876)
1.1.1.1.1.1. Mine 3 iron ore (N:2, Q:4.802)
1.1.1.1.1.1.1. Mine 8 coal (N:2, Q:4.657)
1.1.1.1.1.1.1.1. Smelt 3 iron ore (N:1, Q:3.955)
```

1.1.1.1.1.1.1.1.1. Craft shears (N:1, Q:3.164)
1.1.1.1.1.1.1.1.2. Craft 4 wooden planks (N:0, Q:0.000)
1.1.1.1.1.1.1.1.3. Find 1 sheep (N:0, Q:0.000)
1.1.1.1.1.1.1.2. **Smelt 2 iron ore** (N:1, Q:4.953)
1.1.1.1.1.1.1.2.1. **Craft 1 shear** (N:1, Q:4.608)
1.1.1.1.1.1.1.2.1.1. **Find 1 sheep** (N:1, Q:4.117)
1.1.1.1.1.1.1.2.1.1.1. **Use shear on 1 sheep** (N:1, Q:3.256)
1.1.1.1.1.1.1.2.1.1.2. Equip shear (N:0, Q:0.000)
1.1.1.1.1.1.1.2.1.1.3. Shear 1 sheep (N:0, Q:0.000)
1.1.1.1.1.1.1.2.1.2. Smelt 1 iron ore (N:0, Q:0.000)
1.1.1.1.1.1.1.2.1.3. Craft 4 wooden planks (N:0, Q:0.000)
1.1.1.1.1.1.1.2.2. Smelt 3 iron ore (N:0, Q:0.000)
1.1.1.1.1.1.1.2.3. Find 1 more sheep (N:0, Q:0.000)
1.1.1.1.1.1.1.3. Look for a sheep (N:0, Q:0.000)
1.1.1.1.1.1.2. Chop 4 wood (N:0, Q:0.000)
1.1.1.1.1.1.3. Craft 5 wooden planks (N:0, Q:0.000)
1.1.1.1.1.2. Explore area (N:0, Q:0.000)
1.1.1.1.1.3. Craft 4 wooden planks (N:0, Q:0.000)
1.1.1.1.2. Mine 3 iron ore (N:0, Q:0.000)
1.1.1.1.3. Find iron ore (N:0, Q:0.000)
1.1.1.2. Craft 1 furnace (N:1, Q:4.766)
1.1.1.2.1. Explore area to find 1 sheep (N:1, Q:4.662)
1.1.1.2.1.1. Craft 1 shears (N:1, Q:4.069)
1.1.1.2.1.1.1. Use shears on sheep (N:1, Q:3.205)
1.1.1.2.1.1.2. Shear 1 sheep (N:0, Q:0.000)
1.1.1.2.1.1.3. Equip shears (N:0, Q:0.000)
1.1.1.2.1.2. Explore area (N:0, Q:0.000)
1.1.1.2.1.3. Mine 3 iron ore (N:0, Q:0.000)
1.1.1.2.2. Explore (N:0, Q:0.000)
1.1.1.2.3. Mine 3 iron ore (N:0, Q:0.000)
1.1.1.3. Find iron ore (N:0, Q:0.000)
1.1.2. Craft 4 wooden planks (N:0, Q:0.000)
1.1.3. Craft 16 wooden planks (N:0, Q:0.000)
1.2. Craft 4 wooden planks (N:1, Q:4.796)
1.2.1. Mine 3 iron ore (N:1, Q:4.767)
1.2.1.1. Smelt 3 iron ore (N:1, Q:4.715)
1.2.1.1.1. Craft 1 shears (N:1, Q:4.282)
1.2.1.1.1.1. Find sheep (N:1, Q:3.338)
1.2.1.1.1.2. Chop 2 wood logs (N:0, Q:0.000)
1.2.1.1.1.3. Explore landscape for a sheep (N:0, Q:0.000)
1.2.1.1.2. Craft 1 stone pickaxe (N:0, Q:0.000)
1.2.1.1.3. Craft 1 crafting table (N:0, Q:0.000)
1.2.1.2. Chop 4 wood (N:0, Q:0.000)
1.2.1.3. Craft 1 Crafting Table (N:0, Q:0.000)
1.2.2. Mine 8 cobblestone (N:0, Q:0.000)
1.2.3. Craft 1 wooden pickaxe (N:0, Q:0.000)
1.3. Craft 1 wooden pickaxe (N:1, Q:4.782)
1.3.1. Craft 4 wooden planks (N:1, Q:4.673)
1.3.1.1. Craft 4 sticks (N:1, Q:4.662)
1.3.1.1.1. Explore plains biome for sheep (N:1, Q:4.612)
1.3.1.1.1.1. Craft 1 shears (N:1, Q:4.144)
1.3.1.1.1.1.1. Equip shears (N:1, Q:3.077)
1.3.1.1.1.1.2. Mine 8 cobblestone (N:0, Q:0.000)
1.3.1.1.1.1.3. Shear 1 sheep (N:0, Q:0.000)
1.3.1.1.1.2. Mine 3 iron ore (N:0, Q:0.000)
1.3.1.1.1.3. Mine 8 cobblestone (N:0, Q:0.000)
1.3.1.1.2. Mine 8 cobblestone (N:0, Q:0.000)

```
1.3.1.1.3. Mine 3 iron ore (N:0, Q:0.000)
1.3.1.2. Chop 3 wood logs (N:0, Q:0.000)
1.3.1.3. Mine 8 cobblestone (N:0, Q:0.000)
1.3.2. Find gravel (N:0, Q:0.000)
1.3.3. Chop trees to obtain 2 wood logs (N:0, Q:0.000)
2. Chop 4 wood (N:5, Q:5.064)
2.1. Craft 1 wooden pickaxe (N:3, Q:5.102)
2.1.1. Mine 3 iron ore (N:3, Q:5.111)
2.1.1.1. Mine 8 cobblestone (N:3, Q:5.094)
2.1.1.1.1. Craft 1 furnace (N:2, Q:5.063)
2.1.1.1.1.1. Mine 8 coal (N:1, Q:4.865)
2.1.1.1.1.1.1. Explore grass patches for sheep (N:1, Q:4.642)
2.1.1.1.1.1.1.1. Smelt 2 iron ore (N:1, Q:4.167)
2.1.1.1.1.1.1.1.1. Craft 1 iron shears (N:1, Q:3.210)
2.1.1.1.1.1.1.1.2. Craft 4 wooden planks (N:0, Q:0.000)
2.1.1.1.1.1.1.1.3. Chop 7 wood (N:0, Q:0.000)
2.1.1.1.1.1.1.2. Craft 4 sticks (N:0, Q:0.000)
2.1.1.1.1.1.1.3. Smelt 3 iron ore (N:0, Q:0.000)
2.1.1.1.1.1.2. Smelt 2 iron ore (N:0, Q:0.000)
2.1.1.1.1.1.3. Smelt 3 iron ore (N:0, Q:0.000)
2.1.1.1.1.2. Smelt 3 iron ore (N:1, Q:4.892)
2.1.1.1.1.2.1. Craft 1 shears from iron ingot (N:1, Q:4.783)
2.1.1.1.1.2.1.1. Chop 8 wood (N:1, Q:4.177)
2.1.1.1.1.2.1.1.1. Craft 1 stone pickaxe (N:1, Q:3.843)
2.1.1.1.1.2.1.1.1.1. Find sheep (N:1, Q:2.836)
2.1.1.1.1.2.1.1.1.2. Explore around for sheep (N:0, Q:0.000)
2.1.1.1.1.2.1.1.1.3. Find 2 sheep (N:0, Q:0.000)
2.1.1.1.1.2.1.1.2. Chop 5 wood (N:0, Q:0.000)
2.1.1.1.1.2.1.1.3. Find 1 sheep (N:0, Q:0.000)
2.1.1.1.1.2.1.2. Explore the grass biome (N:0, Q:0.000)
2.1.1.1.1.2.1.3. Chop 3 wood (N:0, Q:0.000)
2.1.1.1.1.2.2. Craft 1 more shears from iron ingot(N:0,Q:0.000)
2.1.1.1.1.2.3. Explore to locate a sheep (N:0, Q:0.000)
2.1.1.1.1.3. Chop 7 wood (N:0, Q:0.000)
2.1.1.1.2. Mine 8 coal (N:1, Q:4.884)
2.1.1.1.2.1. Smelt 2 iron ore (N:1, Q:4.795)
2.1.1.1.2.1.1. Craft 4 stick (N:1, Q:4.592)
2.1.1.1.2.1.1.1. Smelt 1 iron ore (N:1, Q:4.430)
2.1.1.1.2.1.1.1.1. Find sheep (N:1, Q:3.984)
2.1.1.1.2.1.1.1.1.1. Craft 1 shears (N:1, Q:2.933)
2.1.1.1.2.1.1.1.1.2. Craft 1 stone pickaxe (N:0, Q:0.000)
2.1.1.1.2.1.1.1.1.3. Craft 4 wooden planks (N:0, Q:0.000)
2.1.1.1.2.1.1.1.2. Explore new area for sheep (N:0, Q:0.000)
2.1.1.1.2.1.1.1.3. Craft 1 shears (N:0, Q:0.000)
2.1.1.1.2.1.1.2. Craft 4 wooden planks (N:0, Q:0.000)
2.1.1.1.2.1.1.3. Craft 1 stone pickaxe (N:0, Q:0.000)
2.1.1.1.2.1.2. Craft 1 furnace (N:0, Q:0.000)
2.1.1.1.2.1.3. Chop 2 trees (N:0, Q:0.000)
2.1.1.1.2.2. Chop 3 wood (N:0, Q:0.000)
2.1.1.1.2.3. Find cobblestone (N:0, Q:0.000)
2.1.1.1.3. Chop 2 wood (N:0, Q:0.000)
2.1.1.2. Mine 8 coal (N:0, Q:0.000)
2.1.1.3. Chop 2 wood (N:0, Q:0.000)
2.1.2. Mine 8 cobblestone (N:0, Q:0.000)
2.1.3. Explore (N:0, Q:0.000)
2.2. Craft 4 wooden planks (N:2, Q:4.806)
2.2.1. Mine 8 cobblestone (N:2, Q:4.791)
```

```
2.2.1.1. Craft 1 furnace (N:1, Q:4.775)
2.2.1.1.1. Craft 1 stone pickaxe (N:1, Q:4.629)
2.2.1.1.1.1. Explore (N:1, Q:4.349)
2.2.1.1.1.1.1. Mine 3 iron ore (N:1, Q:4.207)
2.2.1.1.1.1.1.1. Smelt 3 iron ore (N:1, Q:3.856)
2.2.1.1.1.1.1.1.1. Craft 1 shears (N:1, Q:2.682)
2.2.1.1.1.1.1.1.2. Chop 2 wood (N:0, Q:0.000)
2.2.1.1.1.1.1.1.3. Chop 3 wood (N:0, Q:0.000)
2.2.1.1.1.1.1.2. Find 1 sheep (N:0, Q:0.000)
2.2.1.1.1.1.2. Chop 2 trees (N:0, Q:0.000)
2.2.1.1.1.1.3. Chop 2 wood (N:0, Q:0.000)
2.2.1.1.1.2. Chop 1 tree (N:0, Q:0.000)
2.2.1.1.1.3. Find a cave (N:0, Q:0.000)
2.2.1.1.2. Mine 3 iron ore (N:0, Q:0.000)
2.2.1.1.3. Place 1 furnace (N:0, Q:0.000)
2.2.1.2. Craft 1 stone pickaxe (N:1, Q:4.446)
2.2.1.2.1. Explore area (N:1, Q:4.286)
2.2.1.2.1.1. Mine 2 iron ore (N:1, Q:3.908)
2.2.1.2.1.1.1. Craft 1 shears (N:1, Q:3.205)
2.2.1.2.1.1.2. Mine 1 cobblestone (N:0, Q:0.000)
2.2.1.2.1.1.3. Smelt 2 iron ore (N:0, Q:0.000)
2.2.1.2.1.2. Mine 3 iron ore (N:0, Q:0.000)
2.2.1.2.2. Mine 1 cobblestone (N:0, Q:0.000)
2.2.1.2.3. Mine 3 iron ore (N:0, Q:0.000)
2.2.1.3. Mine 3 iron ore (N:0, Q:0.000)
2.2.2. Craft 1 wooden pickaxe (N:0, Q:0.000)
2.2.3. Craft 4 sticks (N:0, Q:0.000)
2.3. Mine 8 cobblestone (N:0, Q:0.000)
3. Craft 1 wooden pickaxe (N:0, Q:0.000)
```

## B VOYAGER SKILL LIBRARY INTEGRATION

**Algorithm 2** Option Execution

**Input:** Initial state $s_{t_1}$, option $\omega_i$, maximum refinement attempts $M$
**Output:** done
$\pi_{\omega_i}^{(1)} \sim \texttt{LLM}_{\texttt{in-option}}(\cdot \mid s_{t_1}, \omega_i)$ ▷ Initial In-option Policy Generation
$m \leftarrow 1,\ t_m \leftarrow t_1,\ s_{t_m} \leftarrow s_{t_1}$
**while** $m \leq M$ **do**
  $(s_{t_{m+1}}, \text{feedback}^{(m)}) \leftarrow \textsc{Execute}(\pi_{\omega_i}^{(m)}, s_{t_m})$ ▷ Option Execution in Environment
  $\text{done} \sim \texttt{LLM}_{\beta}(\text{done} \mid s_{t_{m+1}}, \omega_i)$ ▷ Option Completion Check
  **if** done = True **then**
    $\textsc{AddSkill}(\omega_i, \pi_{\omega_i}^{(m)})$ ▷ Skill Library Update
    **return** True
  **end if**
  $\pi_{\omega_i}^{(m+1)} \sim \texttt{LLM}_{\texttt{in-option}}(\cdot \mid s_{t_{m+1}}, \omega_i, \text{feedback}^{(m)})$ ▷ In-option Policy Refinement
  $m \leftarrow m+1,\ t_m \leftarrow t_{m+1},\ s_{t_m} \leftarrow s_{t_{m+1}}$
**end while**
**return** False

Algorithm 2 presents the option execution procedure in ToO, which grounds each planned option as an executable policy in the environment. Given an option $\omega_i$ selected by the planner and the current environment state $s_{t_1}$, the algorithm executes the option using an LLM-generated in-option policy with an iterative refinement process. The execution starts by prompting an LLM to generate an initial in-option policy $\pi_{\omega_i}^{(1)}$, which is represented as executable code that implements a structured control logic over primitive actions, conditioned on the current state and the option description. This

policy is then executed directly in the environment, producing a successor state and corresponding environment feedback.

After each execution attempt, an option completion check is performed by querying an LLM-based termination function. If the option is not completed, the environment feedback is used to refine the in-option policy, yielding an improved policy for the next execution attempt. This refinement loop continues until the option terminates successfully or a maximum number of refinement attempts is reached. Once the option is successfully completed, the resulting in-option policy is added to the skill library for future reuse. If the option cannot be completed within a fixed number of refinement attempts, the execution is terminated and the option is considered infeasible under the current environment conditions.

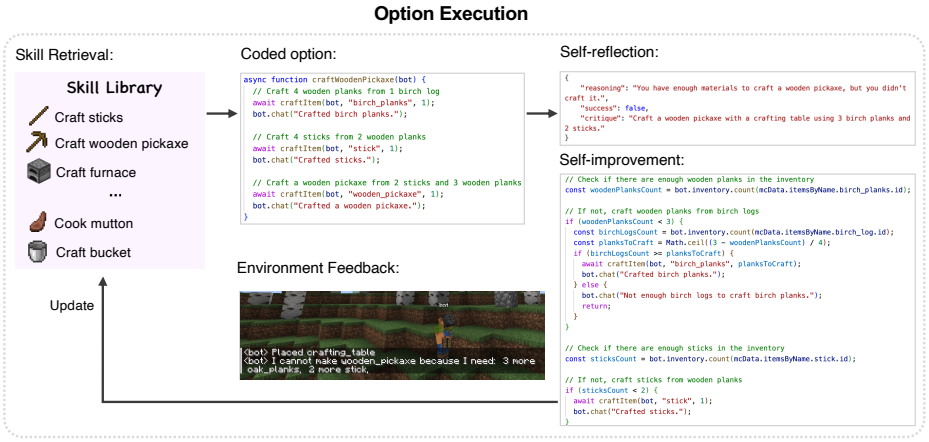


Figure 4: Option execution based on Voyager skill library.

Figure 4 provides an example of option execution process, illustrating how an option is grounded into executable code using the Voyager skill library and iteratively refined through environment feedback. Given an option selected by the planner (e.g., *craft a wooden pickaxe*), the execution starts by retrieving five most relevant skills from the skill library using semantic similarity matching based on text embeddings.

| Skill Manager retrieved skills |
| --- |
| craftStonePickaxe, craftIronPickaxe, craftWoodenPickaxe, craftStoneShovel, craftIronAxe |

Then, Voyager prompts GPT-4 with contexts including the agent's current observation, the question and answer about how to achieve the subgoal, execution feedback and generated code from last round, and the retrieved skills are included as in-context references to guide code generation.

**Example of Voyager state representation**

Health: 20.0/20
Hunger: 20.0/20
Position: x=209.5, y=82.0, z=141.5
Inventory: 'stick': 2, 'wooden_pickaxe': 1, 'birch_planks': 3, 'crafting_table': 1
Nearby blocks: birch_log, grass_block, stone
Task: Craft 1 stone pickaxe
Question: How to craft 1 stone pickaxe in Minecraft?
Answer: To craft 1 stone pickaxe in Minecraft, you will need 3 pieces of stone and 2 sticks. Place the 3 pieces of stone across the top row of the crafting table, and then place 2 sticks in a vertical line beneath the stones. This will create a stone pickaxe for you to use in the game.
Execution error: No error
Critique: None
Code from the last round: No code in the first round

Given this information, the LLM-based Voyager action agent first analyzes the task prerequisites to provide an explanation and plan, and then generates the following executable code:

**Voyager action agent output**

Explain: There is no error in the last round. The bot has 2 sticks, 1 wooden pickaxe, 3 birch planks, and 1 crafting table in its inventory. However, it doesn't have any stone, which is required to craft a stone pickaxe.

Plan:
1) The bot will mine 3 stone blocks using the wooden pickaxe.
2) The bot will place the crafting table near its current position.
3) The bot will craft a stone pickaxe using the crafting table.

```
async function craftStonePickaxe(bot) {
await mineBlock(bot, "stone", 3);
bot.chat("Collected stone.");
const craftingTablePosition = bot.entity.position.offset(1, 0, 0);
await placeItem(bot, "crafting_table", craftingTablePosition);
await craftItem(bot, "stone_pickaxe", 1);
bot.chat("Crafted a stone pickaxe.");
}
```

The generated code is executed via Voyager's control primitive APIs, which are specifically designed to be interpretable and reusable, and can be directly called by GPT-4. For example, `mineBlock(bot, "stone", 3)` instructs the agent to mine three stone blocks within range, while `craftItem(bot, "stone_pickaxe", 1)` triggers the crafting of a stone pickaxe using a nearby crafting table. Internally, the lower-level primitive Mineflayer APIs, which provide direct motor control over the Minecraft agent such as `await bot.pathfinder.goto(goal)` for navigating to a specific position and `bot.useOn(entity)` for using tools on entities.

After executing the generated code, the critic agent returns its assessment based on the execution log records combined with the state, and then outputs:

**Voyager critic agent output**

```
{
"reasoning": "You have a stone pickaxe in your inventory, which means you successfully crafted it.",
"success": true
}
```

# C Additional Results and Analyses

## C.1 Additional Results for Quality of the Planned Options.

Table 4: Per-run execution time (min) and number of attempts for all tasks and methods across three independent execution trials. "–" denotes failed runs that do not complete task within the maximum attempt budget (30).

| Task | Method | Time | Attempts | Task | Method | Time | Attempts |
|---|---|---|---|---|---|---|---|
| Obtain leather | CoT | 20.01 | 17 | Collect lava bucket | CoT | – | – |
| | | 13.37 | 11 | | | 17.62 | 18 |
| | | 15.13 | 14 | | | 20.98 | 26 |
| | ReAct-Option | 9.75 | 11 | | ReAct-Option | 15.63 | 19 |
| | | 11.12 | 11 | | | 17.43 | 21 |
| | | 12.52 | 13 | | | 18.75 | 23 |
| | ToT-MCTS | – | – | | ToT-MCTS | – | – |
| | | 9.48 | 10 | | | 20.63 | 20 |
| | | 15.34 | 21 | | | 22.73 | 24 |
| | ToO | 8.22 | 9 | | ToO | 17.32 | 19 |
| | | 14.1 | 11 | | | 14.08 | 16 |
| | | 11.7 | 10 | | | 15.7 | 17 |
| Cook meat | CoT | – | – | Craft golden sword | CoT | – | – |
| | | 17.07 | 21 | | | 21.95 | 28 |
| | | 18.23 | 25 | | | – | – |
| | ReAct-Option | 12.34 | 16 | | ReAct-Option | – | – |
| | | 13.11 | 17 | | | – | – |
| | | 13.16 | 17 | | | 24.53 | 27 |
| | ToT-MCTS | 16.54 | 20 | | ToT-MCTS | – | – |
| | | 13.22 | 18 | | | – | – |
| | | 15.12 | 21 | | | 29.17 | 25 |
| | ToO | 10.25 | 12 | | ToO | 17.41 | 20 |
| | | 8.6 | 9 | | | 17.03 | 19 |
| | | 9.02 | 10 | | | 19.65 | 20 |
| Shear a sheep | CoT | 24.38 | 20 | Craft a compass | CoT | 24.1 | 29 |
| | | 25.59 | 22 | | | – | – |
| | | 25.33 | 21 | | | – | – |
| | ReAct-Option | – | – | | ReAct-Option | – | – |
| | | 17.34 | 19 | | | 18.25 | 23 |
| | | 19.42 | 21 | | | 20.65 | 25 |
| | ToT-MCTS | – | – | | ToT-MCTS | 20.18 | 20 |
| | | 15.92 | 15 | | | 29.02 | 25 |
| | | 18.14 | 19 | | | – | – |
| | ToO | 19.68 | 15 | | ToO | 12.17 | 18 |
| | | 20.01 | 16 | | | 16.01 | 24 |
| | | 22.17 | 18 | | | 22.08 | 28 |
| Milk a cow | CoT | 26.35 | 27 | Craft a diamond pickaxe | CoT | – | – |
| | | – | – | | | – | – |
| | | – | – | | | – | – |
| | ReAct-Option | – | – | | ReAct-Option | – | – |
| | | 21.97 | 29 | | | – | – |
| | | 24.07 | 29 | | | – | – |
| | ToT-MCTS | – | – | | ToT-MCTS | – | – |
| | | 28.91 | 27 | | | 25.95 | 27 |
| | | 29.77 | 29 | | | – | – |
| | ToO | 19.72 | 20 | | ToO | 10.67 | 15 |
| | | 22.82 | 23 | | | 10.88 | 16 |
| | | 22.86 | 21 | | | 9.92 | 15 |

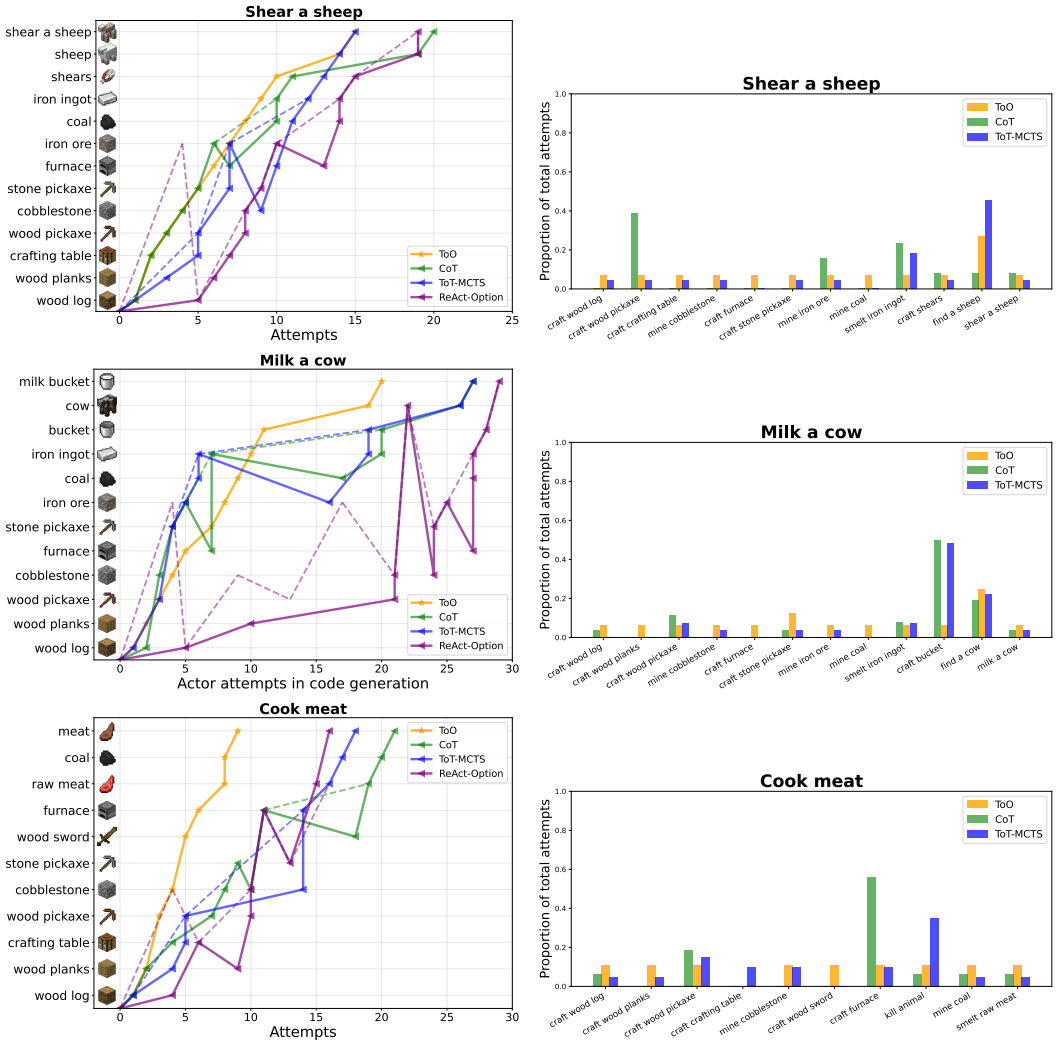


Figure 5: Comparison of execution dynamics across dynamic–immediate planning tasks.

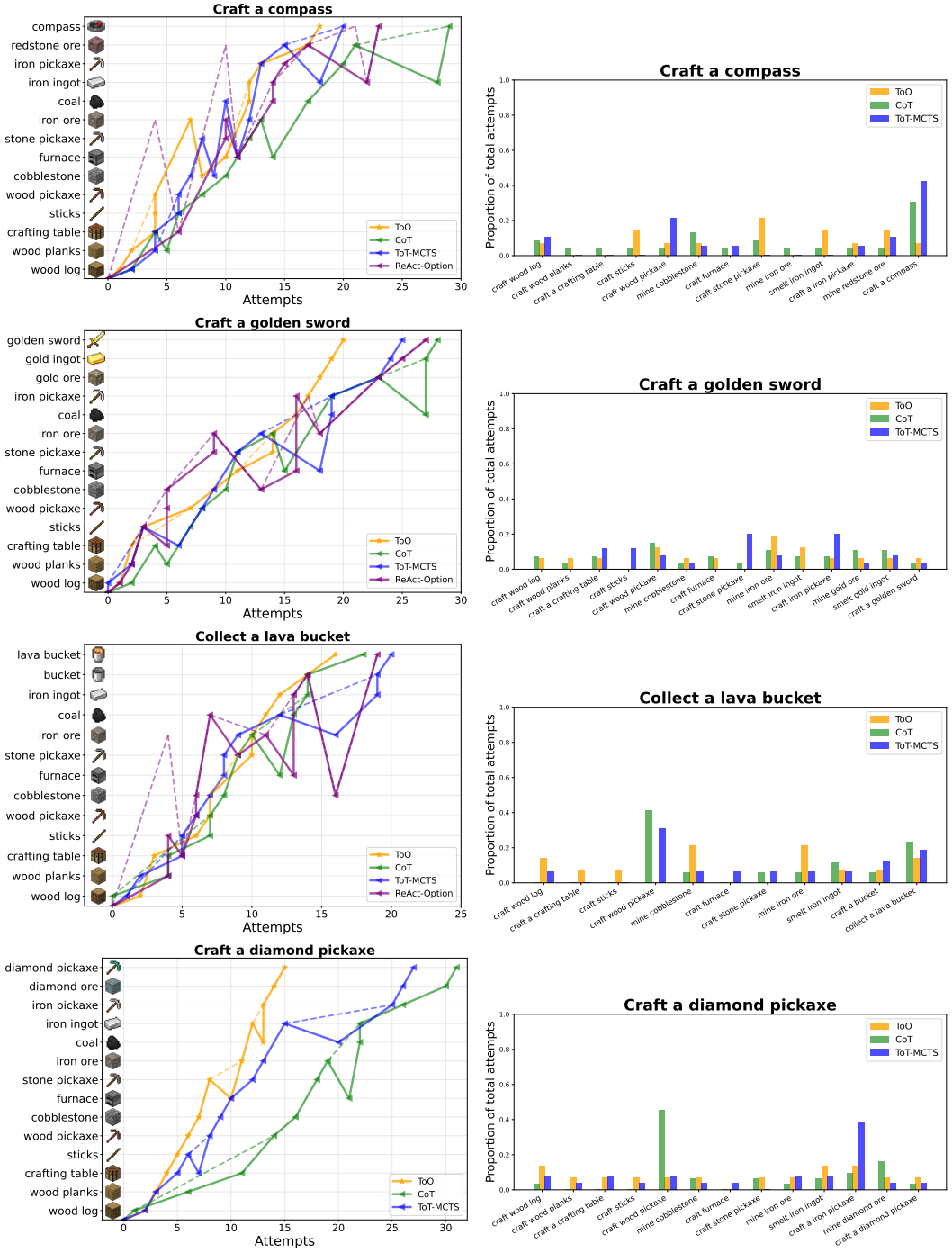


Figure 6: Comparison of execution dynamics across multi-step crafting tasks.

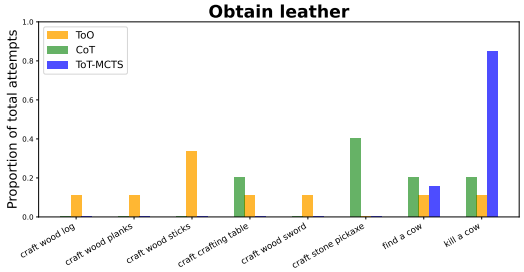


Figure 7: Comparison of execution attempts across planned options in dynamic–immediate planning task (`obtain leather`). Fewer attempts indicate the necessary prerequisites were met, while a larger proportion suggests inefficient proposals.

This appendix further reports how different methods allocate low-level execution effort across options to assess option dependency and planning stability. We include CoT, ToT-MCTS, and ToO only, as these methods rely on one-shot planning, while ReAct-Option interleaves planning and execution and is therefore not directly comparable in this analysis. Figure 7 visualizes the distribution of prompting attempts over intended options, corresponding to the execution dynamics illustrated in Figure 2. Each bar indicates the fraction of execution attempts devoted to a specific option, reflecting how heavily a method depends on specific options. The results show that CoT and ToT-MCTS exhibit highly uneven distributions, with a small number of options dominating the overall execution budget. This concentration arises when prerequisite steps are omitted, leading to error accumulation and repeated backtracking during execution, which results in disproportionately tall bars at later stages. In contrast, ToO produces a more balanced allocation of attempts across a deliberate sequence of options. By incorporating feasibility validation and dynamics prediction, ToO avoids wasting attempts on hallucinated or infeasible options and instead concentrates execution effort on inherently high-uncertainty steps in dynamic-immediate environments, such as *find a cow*.

### C.2 Case Study


Figure 8: How each method plans and executes options for the `obtains leather` task. All three methods finish this task, but their planned options are drastically different. CoT generates the shortest plan but neglects key intermediate steps. ReAct-Option completes the task via trial-and-error but suffers from redundant tool crafting. ToT-MCTS explores multiple branches and partially accounts for tool crafting. ToO yields a backtracking-free trajectory and the most efficient execution.

### C.3 Additional Results for Ablation Studies

Table 5: Ablation on ToO components.

| Task | Method | SR | Time | Attempts |
|---|---|---|---|---|
| Obtain leather | ToO | **3 / 3** | **11.34** | **10** |
| | ToO w/o WM | 2 / 3 | 15.28 | 18.5 |
| | ToO w/o FC | 3 / 3 | 14.97 | 13.33 |
| Cook meat | ToO | **3 / 3** | **9.29** | **10.33** |
| | ToO w/o WM | 3 / 3 | 13.81 | 18 |
| | ToO w/o FC | 3 / 3 | 13.90 | 18.33 |
| Shear a sheep | ToO | **3 / 3** | **20.62** | **16.33** |
| | ToO w/o WM | 3 / 3 | 19.17 | 19.67 |
| | ToO w/o FC | 2 / 3 | 18.73 | 17.5 |
| Milk a cow | ToO | **3 / 3** | **21.8** | **21.33** |
| | ToO w/o WM | 2 / 3 | 27.77 | 22.5 |
| | ToO w/o FC | 1 / 3 | 27.47 | 27 |

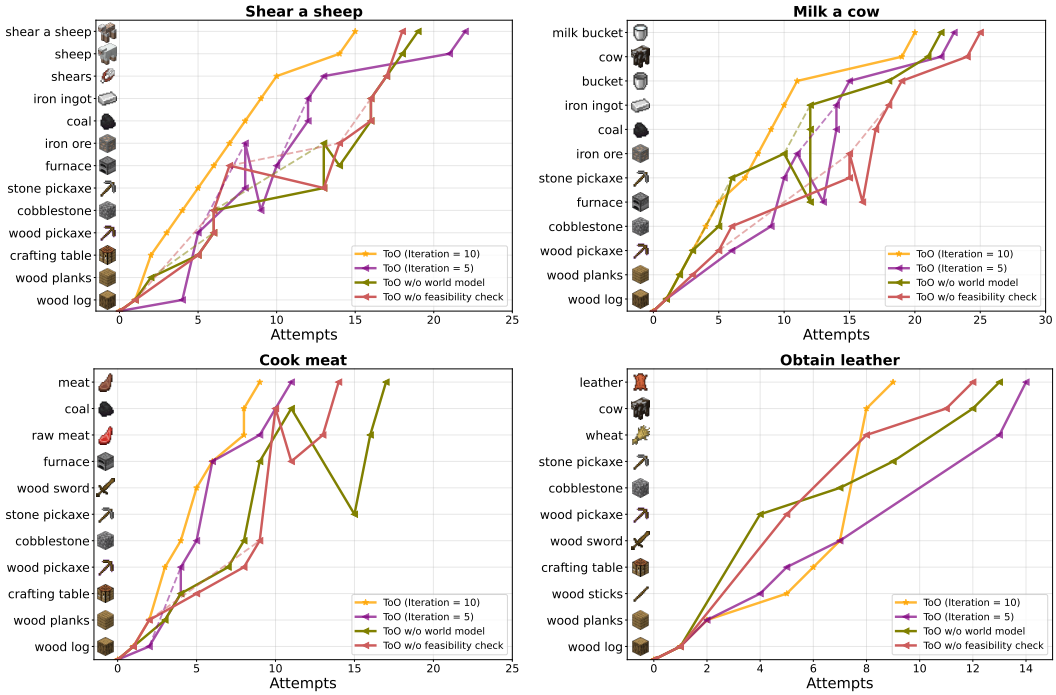


Figure 9: Ablation studies for LLM world model and feasibility check module.