Hierarchical Reward Design from Language: Enhancing Alignment of Agent Behavior with Human Specifications

Zhiqin Qian
Rice University
Houston, USA
bill.qian@rice.edu

Ryan Diaz
Rice University
Houston, USA
ryandiaz@rice.edu

Sangwon Seo
Rice University
Houston, USA
sangwon.seo@rice.edu

Vaibhav Unhelkar
Rice University
Houston, USA
vaibhav.unhelkar@rice.edu

ABSTRACT

When training AI agents to perform tasks, humans often care not only about *whether* a task is completed but also *how* it is performed. As agents tackle increasingly complex tasks, aligning their behavior with human-provided specifications becomes critical for responsible AI deployment. *Reward design* provides a direct channel for such alignment by translating human expectations into reward functions that guide reinforcement learning (RL). However, existing methods are often too limited to capture nuanced human preferences that arise in long-horizon tasks. Hence, we introduce **Hierarchical Reward Design for Language (HRDL)**: a problem formulation that extends classical reward design to encode richer behavioral specifications for Hierarchical RL agents. We further propose **Language to Hierarchical Rewards (L2HR)**, our proposed solution to HRDL. Human subject and numerical experiments show that Hierarchical RL agents trained with rewards designed via L2HR not only complete tasks effectively but also better adhere to human specifications. Together, HRD and L2HR advance the research on human-aligned AI agents.

KEYWORDS

Reward Design, Human-Centered AI, Hierarchical RL

ACM Reference Format:
Zhiqin Qian, Ryan Diaz, Sangwon Seo, and Vaibhav Unhelkar. 2026. Hierarchical Reward Design from Language: Enhancing Alignment of Agent Behavior with Human Specifications. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 35 pages.

1 Introduction

AI agents are being deployed in human-centric environments such as homes, hospitals, and disaster zones [19, 29, 49, 56, 66]. Their usefulness depends not only on accomplishing tasks, but on doing so in ways that respect human intentions, operational rules, and safety requirements (henceforth collectively referred to as *behavior specifications*). Aligning agent behavior with these specifications is central to safe and responsible AI deployment. Prior research has explored a range of approaches for conveying such specifications to agents [12]. In this work, we focus on the paradigm of *reward design*, which provides a direct way for humans to convey such specifications by translating them into reward signals that guide reinforcement learning (RL).

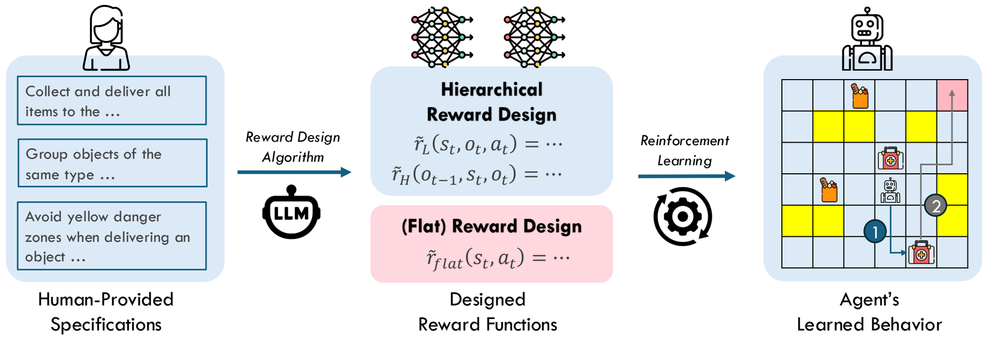
As AI agents take on increasingly complex, long-horizon tasks, more advanced reward design methods are needed to capture equally complex specifications. Humans rarely teach or think about tasks and associated specifications as monolithic goals [9, 10, 13, 30, 41, 53]. Instead, we naturally break them into subtasks: "first prepare the ingredients, then cook, then serve." Hierarchical frameworks in RL mirror this structure by decomposing tasks into subtasks and organizing them over long horizons [14, 15, 50, 64].

Research Gap. This *hierarchical approach to policy learning* has enabled agents to complete tasks of increasingly longer horizons. However, *the reward design for these hierarchical RL agents remains largely unexplored,* thereby limiting alignment of agent behavior with human specifications in long-horizon tasks. As illustrated in Fig. 1, specifications for long-horizon tasks often include details on *what* subtasks to perform, in *which* order, and *how* they are executed. Existing reward design methods encode these specifications via a flat reward function of the form $\tilde{r}_{flat}(s, a)$. We show *both theoretically and empirically that flat rewards are fundamentally limited in capturing specifications for long-horizon tasks.*

Summary of Contributions. To address this limitation,

- We introduce the *Hierarchical Reward Design (HRD)* problem, which enables designers to express behavioral specifications inspired by the same structured way people naturally think and teach. Unlike the classical (flat) reward design problem [57], HRD admits reward solutions that enables encoding of complex specifications for long-horizon tasks, capturing both what subtasks to perform and how to execute them. HRD is a general formulation that can be instantiated with multiple input modalities, analogous to how flat reward design has been realized via proxy signals or language [20, 34, 39]. Because natural language is an intuitive medium for specifying layered instructions, we then provide a language-based instantiation called *Hierarchical Reward Design from Language (HRDL, pronounced "hurdle")*.
- We prove that hierarchical rewards of HRDL are strictly more expressive than flat rewards used by prior works, while remaining compatible with standard decision-making frameworks (i.e.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus.



Collect and deliver all items to the ...
Group objects of the same type ...
Avoid yellow danger zones when delivering an object ...
Human-Provided Specifications
Reward Design Algorithm
LLM
Hierarchical Reward Design
$\tilde{r}_L(s_t, o_t, a_t) = \cdots$
$\tilde{r}_H(o_{t-1}, s_t, o_t) = \cdots$
(Flat) Reward Design
$\tilde{r}_{flat}(s_t, a_t) = \cdots$
Designed Reward Functions
Reinforcement Learning
1
2
Agent's Learned Behavior

Figure 1: This work introduces the Hierarchical Reward Design from Language (HRDL) problem. Unlike prior work on reward design, HRDL decomposes reward design into low- and high-level components ($\tilde{r}_L, \tilde{r}_H$). Language to Hierarchical Rewards (L2HR), our proposed solution to HRDL, leverages language models to guide the synthesis of these hierarchical rewards, enabling existing RL algorithms to train agents that are better aligned with human specifications.

Markov and semi-Markov Decision Processes) and reinforcement learning algorithms.

- We present **Language to Hierarchical Rewards (L2HR)**, an initial solution to HRDL that generates hierarchical rewards directly from natural language specifications, making reward design more accessible while leveraging the reasoning capabilities of large language models [2, 24, 37]. L2HR produces reward structures that guide both high-level subtask selection and low-level execution.

Through human subject and numerical experiments, we demonstrate the advantages of hierarchical over flat reward design. We show that hierarchical reward design (coupled with hierarchical RL) allows AI agents to not only successfully complete tasks but also better align their behavior with language specifications. We view this work as an initial but important step toward aligning AI systems with human expectations through the lens of HRD. Through theoretical analysis and empirical findings, this paper lays the groundwork for future research on designing human-aligned reward structures that employ hierarchies and human input.

2 Background and Related Work

We focus on AI agents tasked with problems that can be modeled as Markov Decision Processes (MDPs) [52], defined by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, r, \gamma, h)$. Here, $\mathcal{S}$ and $\mathcal{A}$ denote the state and action spaces, $T(s'|s, a)$ the transition dynamics, and $r(s, a)$ the immediate reward. The discount factor $\gamma \in [0, 1]$ trades off immediate and future rewards, and h is the horizon. The objective of reinforcement learning (RL) is to find a policy $\pi(a|s)$ that maximizes the expected discounted return $\mathbb{E}_{s_t \sim T, a_t \sim \pi}[\sum_{t=0}^{h} \gamma^t r(s_t, a_t)]$. MDPs provide a flexible framework for modeling a wide range of sequential tasks, including those tackled by virtual agents and robotic assistants. MDPs can be solved using reinforcement learning, which provides an approach to computing the optimal policy that maximizes the expected cumulative discounted reward [63].

2.1 Hierarchical Reinforcement Learning

While capable, RL algorithms find it challenging to solve long-horizon tasks. Hierarchical RL (HRL) seeks to solve MDPs with long horizons by decomposing them into simpler subtasks [14, 15, 50, 64]. A widely adopted HRL paradigm is the *options framework* [5, 64, 71], where the agent has access to a discrete set of temporally-extended behaviors called *options*, denoted as O. Each option $o \in O$ corresponds to an intra-option policy given by $\pi_L(a|s, o)$ and a termination condition $\beta(s, o^-)$. A high-level policy $\pi_H(o|o^-, s)$ selects which option to execute, and the corresponding low-level policy π_L generates primitive actions until the selected option terminates. The reward model for options computes the expected cumulative reward until an option terminates. Following [64], we let $\mathcal{E}(o, s, t)$ denote the event where option o is initiated in state s at timestep t, and define the option-level reward as:

$$r_{opt}(s, o) \doteq \mathbb{E}[\Sigma_{i=1}^{k} \gamma^{i-1} r_{t+i} \mid \mathcal{E}(o, s, t)] \tag{1}$$

where k is the random variable denoting number of steps after which the initiated option o terminates, determined by its termination condition β.

Another line of HRL research follows the *feudal/goal-conditioned framework* [14, 27, 33, 43, 65], which also decomposes a task into subtasks but differs in how the hierarchical policies are trained and how reward signals are assigned. In this framework, the high-level manager selects subgoals and receives a *task* reward, as in the options framework. However, unlike the options framework, the low-level worker receives a separate *pseudo-reward* $r_p(s, o, a)$ that measures progress toward achieving the current subgoal.

2.2 Reward Design

A core challenge in using MDPs and RL is reward design [57, 63]. Given a well-designed reward function, agents can use RL/HRL algorithms to solve the MDP. However, in practice, the design of a reward function is non-trivial and can lead to a host of problems, including poor alignment between humans and agents [4].

2.2.1 Reward Design Problem. To formally study reward design, Singh et al. introduce the (flat) Reward Design Problem (RDP) [57]. RDP is formalized as a tuple $P = (\mathcal{M}_p, \mathcal{R}, \mathcal{A}_{\mathcal{M}_p}, F)$, where

- $\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma, h)$ is the *world model*;
- $\mathcal{R}$ is the space of reward functions;
- $\mathcal{A}_{\mathcal{M}_p}(r) : \mathcal{R} \rightarrow \Pi$ is an algorithm to compute policy $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ that optimizes reward $r \in \mathcal{R}$ in the MDP $(\mathcal{M}_p, r)$;
- $F : \Pi \rightarrow \mathbb{R}$ is the fitness function that produces a scalar evaluation of a policy, only accessible via policy queries.

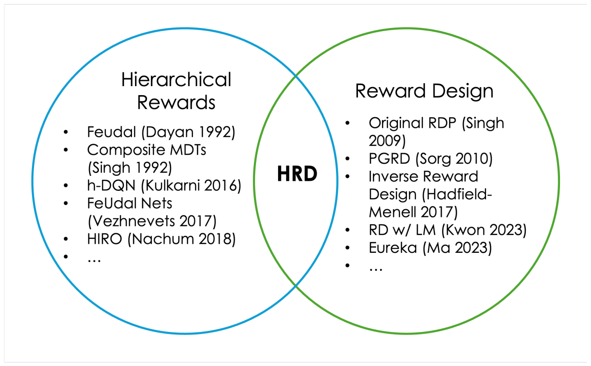


Figure 2: Although prior works have utilized multi-level rewards to train hierarchical agents, the *design* of such rewards remains underexplored and lacks a concrete problem formulation. This work seeks to bridge this gap.

In an RDP, the goal is to output a reward function $r \in \mathcal{R}$ such that *the policy* $\pi := \mathcal{A}_{\mathcal{M}_p}(r)$ *that optimizes* r achieves the highest Fitness score $F(\pi)$.

Rather than treating the reward as fixed and exogenous, RDP reframes reward design as a *search problem*, a perspective that has profoundly shaped subsequent research. It inspired methods that optimize or evolve reward functions directly [46, 60] as well as formulations that infer or generate them from indirect signals, such as Inverse Reward Design [20], which recovers true rewards from proxy rewards, and Eureka [39], which synthesizes executable rewards from natural language. Many other paradigms can be viewed as RDP instantiations: inverse reinforcement learning [1, 17, 22, 72] treats expert behavior as evidence for the reward search, while preference-based learning [7, 54] and RLHF [6, 48] extend this to human feedback.

2.2.2 Rewards Design for Hierarchical RL. Collectively, RDP provides a unifying framework that has catalyzed advances in both reinforcement learning and human-AI alignment; however, it does not explicitly consider hierarchical RL. Many prior works have explored the use of hierarchical rewards, from early studies in feudal reinforcement learning [14] and precursors to the options framework [59] to more recent advances in deep HRL [33, 43, 65]. While these works often *assume access* to hierarchical rewards for training agents to complete tasks, **the problem of *designing* such hierarchical rewards has received little attention and, to our knowledge, has yet to be formally defined as a concrete research problem**.

A literature search using the keywords "hierarchical reward design" primarily returns domain-specific studies that discuss how *using* hierarchical rewards enables solving application-level problems, such as fleet management [11, 25, 44, 45]. Other works either employ the term "hierarchy" in different contexts (for example, to express the relative importance of multiple reward signals [36] or sequential action constraints without explicitly modeling subtasks [28]) or focus on narrower scopes, such as learning hierarchical rewards specifically from demonstrations [62].

Since no previous work formally defines hierarchical reward design as a general problem, there is a lack of consistent language and theoretical foundation for studying it, unlike the well-established Reward Design Problem [57]. This work addresses this gap by formalizing the Hierarchical Reward Design (HRD) problem and introducing a reward structure that is hierarchical, compact, and capable of capturing nuanced behavioral specifications for both *what* subtasks to select and *how* to execute them.

Just as RDP lays the groundwork for studying algorithmic reward design in flat settings, we posit that HRD will provide a principled foundation for reasoning about hierarchical reward structures. In line with the research that originated from RDP, we anticipate that HRD will enable a broad range of problem instantiations (of reward design with different types of human inputs) and solution methods for designing hierarchical rewards. A Venn diagram illustrating how this work relates to prior research on hierarchical rewards and reward design is shown in Figure 2.

2.2.3 Reward Design from Language. Early work addressing the RDP focused on designing rewards for intrinsic motivation and reward shaping [58, 60, 61]. More recently, research in this area has explored aligning agent behavior more closely with human-provided specifications, using learning or large language models (LLMs) to infer and generate reward functions [20, 34]. In these cases, the human or an oracle, either implicitly or explicitly, serves as the fitness function by evaluating the policy. However, most existing work on reward design or generation focuses exclusively on non-hierarchical (flat) RL settings, producing reward functions of the form $r_{flat}(s, a)$ or $r_{flat}(s)$ [8, 16, 20, 21, 34, 35, 39, 58, 60, 61, 68, 69]. While sufficient for certain behaviors, *flat reward functions are fundamentally limited when specifying complex preferences, such as desired subtask sequences or option-conditioned execution strategies, that naturally arise in long-horizon tasks.*

To our knowledge, the only prior work that explicitly considers a hierarchical setting is [40], though its focus differs substantially from ours. Their approach does not formalize the hierarchical reward design problem or analyze the expressivity gap between flat and hierarchical formulations. Moreover, the rewards generated by their LLMs are limited to task completion objectives and do not capture behavioral specifications. In contrast, L2HR generates both high- and low-level rewards that encode natural language behavioral preferences while preserving task feasibility. [1]

3 Hierarchical Reward Design

This section formally introduces the Hierarchical Reward Design (HRD) problem in the context of HRL, drawing insights from both the options framework and the feudal framework. We begin by defining the low- and high-level reward functions in HRD and proceed to show that they naturally induce a family of MDPs at the low-level and a semi-MDP (SMDP) at the high-level. Using these insights, we formally define the general HRD problem and introduce a specific instantiation, the *Hierarchical Reward Generation from Language (HRDL)* problem, which we address in this paper. Proofs and additional details for all propositions are provided in Appendix Sec. B.1.

[1]Please see the Sec. A in the Appendix for a further discussion of related works.

3.1 Low-level and High-level Reward Models

Definition 1 (Low-level Reward). *The **low-level reward** is a function $r_L : \mathcal{S} \times \mathcal{O} \times \mathcal{A} \to \mathbb{R}$, which provides feedback for selecting a low-level action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ while pursuing option $o \in \mathcal{O}$.*

Intuitively, $r_L(s, o, a)$ encodes specifications for *how* the agent should execute the subtask associated with option o in state s.

Proposition 1 (Low-level MDP Models). *Let $\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma)$ be a world model, $\mathcal{O}$ a set of options, and $r_L : \mathcal{S} \times \mathcal{O} \times \mathcal{A} \to \mathbb{R}$ a low-level reward. For a fixed option $o \in \mathcal{O}$, the tuple $\mathcal{M}_{L,o} = (\mathcal{S}, \mathcal{A}, T, r_L(\cdot, o, \cdot), \gamma, h_o)$ defines an MDP, where h_o is the horizon determined by the option's termination condition $\beta(\cdot, o)$.*

Definition 2 (High-level Reward). *The **high-level reward** is a function $r_H : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \mathbb{R}$, which specifies the expected reward for executing option $o \in \mathcal{O}$ until termination, given that o is initiated in state $s \in \mathcal{S}$ and the previous option was $o^- \in \mathcal{O}$.*

The high-level reward $r_H(o^-, s, o)$ encodes specifications for *what* subtask should be executed, possibly conditioned on both the current state and prior option. This allows for expressing preferences over subtask *ordering* and dependencies between subtasks.

Proposition 2 (High-level SMDP Model).
Let $\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma, h)$ be a world model, $\mathcal{O}$ a set of options, and $r_H : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \mathbb{R}$ the high-level reward. Then, $\mathcal{M}_H = (\mathcal{O} \times \mathcal{S}, \mathcal{O}, T_H, r_H, \gamma, h)$ forms a semi-MDP, where $T_H : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \Delta(\mathcal{O} \times \mathcal{S} \times \mathbb{N})$ defines the joint distribution over the next augmented state and transit time, where $\mathbb{N}$ is the set of natural numbers.

Alternatively, the high-level process can be modeled as a standard MDP when *single-step* high-level rewards are used. This flexibility highlights that the HRD framework is compatible with both semi-MDP and MDP formulations, allowing the use of a wide range of existing RL algorithms. We provide the formal MDP definition and corresponding proof in Sec. B.1 in the Appendix.

3.2 The HRD Problem

Definition 3 (Hierarchical Reward Design (HRD)). *The **Hierarchical Reward Design (HRD)** problem is defined by the tuple $P = (\mathcal{M}_p, \mathcal{O}, \mathcal{R}, \mathcal{A}_{\mathcal{M}_p}, F)$, where*

- *$\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma, h)$ is the world model;*
- *$\mathcal{O}$ is a finite option set;*
- *$\mathcal{R} = \mathcal{R}_H \times \mathcal{R}_L$ is the space of candidate reward structures, where $\mathcal{R}_H = \{r_H : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \mathbb{R}\}$ and $\mathcal{R}_L = \{r_L : \mathcal{S} \times \mathcal{O} \times \mathcal{A} \to \mathbb{R}\}$;*
- *the learning routine $\mathcal{A}_{\mathcal{M}_p}(\cdot) : \mathcal{R} \to \Pi_H \times \Pi_L$ maps each reward pair (r_H, r_L) to a hierarchical policy (π_H, π_L), where $\pi_H : \mathcal{O} \times \mathcal{S} \to \Delta(\mathcal{O})$ optimizes r_H in the high-level decision making model $\mathcal{M}_H$ and $\pi_L : \mathcal{S} \times \mathcal{O} \to \Delta(\mathcal{A})$ optimizes r_L in each underlying MDP $\mathcal{M}_{L,o}$; and*
- *the fitness function $F : \Pi_H \times \Pi_L \to \mathbb{R}$ evaluates the quality of hierarchical policies.*

The goal of HRD is to find $(r_H^*, r_L^*) = \arg\max_{(r_H, r_L) \in \mathcal{R}} F(\mathcal{A}_{\mathcal{M}_p}(r_H, r_L))$.

Connections to Existing Algorithms. We show in Sec. B.2 that $\mathcal{A}_{\mathcal{M}_p}$ can be instantiated with existing RL algorithms. In our implementation, the low-level policy $\pi_L(a \mid s, o)$ is trained with PPO [55] due to its robustness in control, while the high-level policy $\pi_H(o \mid o^-, s)$ uses DQN-style methods [42], following common practice in SMDP formulations [5, 64]. Stronger structural assumptions on (r_H, r_L) can enable the use of more specialized HRL algorithms. For instance, if the low-level reward depends only on state and action, $r_L(s, o, a) = r_{flat}(s, a)$, and the high-level reward r_H is a single-step reward constructed as $\sum_a r_{flat}(s, a)\pi_L(a|s, o)$, the problem reduces to the two augmented MDPs formulation introduced in [71]. In these cases, algorithms such as double actor-critic [71] and option-critic [5] can be applied to learn hierarchical policies. A detailed exploration of the connections between structural reward assumptions and the applicability of existing HRL algorithms for instantiating $\mathcal{A}_{\mathcal{M}_p}$ is left for future work.

3.3 Hierarchical Reward Design from Language

As discussed in Sec. 1, real-world deployments often require agents to satisfy additional behavioral specifications beyond task completion. In these cases, the task reward can typically be defined once and reused across different behavioral contexts. In contrast, additional rewards must be redesigned for each new behavior specification. While the cost of task reward design is amortized, the cost of designing rewards that match human specifications grows linearly with the number of distinct behaviors desired. This motivates the need for an automated approach to generate rewards to encode behavioral specifications while reusing the existing domain dynamics and task objectives. The challenge of this problem is twofold: (1) The generated rewards should have distinct functional forms – one guiding high-level option selection, and another governing low-level action execution. (2) The generated rewards must remain compatible with existing task rewards, ensuring that agents continue to achieve the original task objectives. We formally define this as a specific instantiation of the HRD problem.

Definition 4 (Hierarchical Reward Design from Language (HRDL)). *The **HRDL** problem is an instance of the HRD problem with additional inputs: (1) a task reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, (2) a subtask completion reward (pseudo-reward) $r_p : \mathcal{S} \times \mathcal{O} \times \mathcal{A} \to \mathbb{R}$, and (3) behavior specifications $l \in \Sigma^*$, provided as a natural language description. l guides the reward generation during training, and the fitness function F is accessible **only** during evaluation. The objective of HRDL is to generate high- and low-level designed rewards, $R^* = (\tilde{r}_H^*, \tilde{r}_L^*) \in \mathcal{R}$, such that the resulting hierarchical policy (π_H^*, π_L^*), trained under the composite rewards $(r_{opt} + \tilde{r}_H^*, r_p + \tilde{r}_L^*)$ using $\mathcal{A}_{\mathcal{M}_p}$, maximizes the fitness score: $(\tilde{r}_H^*, \tilde{r}_L^*) = \arg\max_{(\tilde{r}_H, \tilde{r}_L) \in \mathcal{R}} F(\mathcal{A}_{\mathcal{M}_p}(r_{opt} + \tilde{r}_H, r_p + \tilde{r}_L))$.*

If a *non-hierarchical* reward design method is used, the designed reward has the flat form $\tilde{r}_{flat}(s, a)$. To integrate this flat reward into the hierarchical setting, we must decompose it into high- and low-level rewards:

$$r_L(s, o, a) = r_p(s, o, a) + \tilde{r}_{flat}(s, a) \tag{2}$$

$$r_H(s, o) = r_{opt}(s, o) + \tilde{r}_{flat,H}(s, o) \tag{3}$$

where $\tilde{r}_{flat,H}(s, o)$ aggregates $\tilde{r}_{flat}(s, a)$ using the same expression as Eq. 1. While flat designed rewards $\tilde{r}_{flat}(s, a)$ can encode some behavior specifications, the definitions of high- and low-level rewards in HRD provide a significantly more expressive mechanism for specifying agent behavior:

$$r_L(s, o, a) = r_p(s, o, a) + \tilde{r}_L(s, o, a) \tag{4}$$

$$r_H(o^-, s, o) = r_{opt}(s, o) + \tilde{r}_H(o^-, s, o) \tag{5}$$

In fact, the flat reward is a special case of hierarchically designed rewards, where $\tilde{r}_L(s, o, a) = \tilde{r}_{flat}(s, a)$ and $\tilde{r}_H(o^-, s, o) = \tilde{r}_{flat,H}(s, o)$. The hierarchical formulation is strictly more general than the flat formulation, offering greater expressiveness in the following ways:

Property 1. *Certain specifications on sub-task selection can be expressed through $\tilde{r}_H(s, o^-, o)$, but they cannot be expressed by flat function: $\tilde{r}_{flat}(s, a)$.*

Property 2. *Certain specifications on sub-task execution can be expressed through $\tilde{r}_L(s, o, a)$, but they cannot be expressed by flat function: $\tilde{r}_{flat}(s, a)$.*

Proofs of these properties are provided in Appendix Sec. B.3. In the following sections, we introduce an algorithm for generating hierarchical rewards $(\tilde{r}_H, \tilde{r}_L)$ from natural language specifications and empirically compare its performance against flat reward design that generates alignment rewards of the form $\tilde{r}_{flat}(s, a)$.

4 Language to Hierarchical Rewards (L2HR)

We now present L2HR, an algorithm to solve HRDL that uses large language models (LLMs) to generate reward functions directly from natural language specifications. An illustration of L2HR's input and output is provided in Fig. 3.

4.1 LLM Prompting Strategy

To generate executable reward functions from natural language specifications, we design a prompting strategy inspired by recent works on code generation for reward design [39]. Unlike [39], our prompting design is tailored for generating feasible reward functions without relying on using fitness F to evaluate and iteratively improve polices during training. More specifically, the LLM is provided with a structured prompt consisting of the following components:

(1) *Task Description*: A natural language summary of the overall task objective, including the approximate scale of the task reward. The actual task reward code is intentionally withheld – both to reflect realistic scenarios in which only the reward signal (but not the code) is accessible and to prevent the LLM from overfitting to specific implementation details.
(2) *Environment Code Context*: Extracted snippets of environment source code that expose the state and action spaces without leaking simulation internals. This follows the methodology proposed in [39].
(3) *Relevant Action-Related Spaces*: Descriptions of the option space O and action space $\mathcal{A}$, including the semantic role of each. We include these descriptions to help the LLM correctly distinguish between high-level and low-level decision spaces.
(4) *Behavior Specification*: A natural language string that describes the desired agent behaviors beyond task completion.
(5) *Formatting and Reward Design Tips*: Brief coding guidelines, such as avoiding defining new global variables, and recommendations for balancing designed rewards with the underlying task reward.

4.2 Training Procedure

While LLMs can generate plausible reward code in a zero-shot manner, code generation is inherently noisy: syntax errors, invalid variable references, and runtime failures may occur. To address this variability, we sample k reward candidates independently from the LLM and apply a lightweight filtering process to ensure validity. During filtering, we verify whether the code compiles without syntax errors, and whether it references only permitted state, option, and action variables exposed in the environment prompt. We find that in practice, at least one sample in the batch passes these checks and preserves task feasibility. As a result, we forego more complex iterative refinement strategies, such as evolutionary search or "reward reflection" [39], which would be challenging without fitness F during training. However, we recognize this as an important future direction; methods that incorporate feedback could further improve the robustness of generated reward functions.

The full two-stage training algorithm **L2HR** is provided in Sec. C of the Appendix. In the first stage, we use the LLM to generate k candidate low-level alignment functions $\tilde{r}_L^{(1)}, \ldots, \tilde{r}_L^{(k)}$ from the specification l. The low-level prompt differs from the high-level one only in its description of the action space and level-specific behavior specifications. Each $\tilde{r}_L^{(i)}$ is then used to train a corresponding low-level policy $\pi_L^{(i)}$ with the combined objective of pseudo-rewards plus the LLM-generated $\tilde{r}_L^{(i)}$. Only the policies $\pi_L^{(i)}$ that surpass a predefined threshold of subgoal completion, based on cumulative pseudo-rewards, are considered as *valid*.

In the second stage, we generate k candidate high-level alignment functions $\tilde{r}_H^{(1)}, \ldots, \tilde{r}_H^{(k)}$ and use them to train high-level policies $\pi_H^{(j)}$, each conditioned on a *valid* $\pi_L^{(i)}$ from the first stage. These policies are optimized using with the combined objective of option-level task reward plus the LLM-generated $\tilde{r}_H^{(j)}$. Then, we return all designed rewards $(\tilde{r}_H^{(j)}, \tilde{r}_L^{(i)})$ and corresponding trained policy pairs $(\pi_H^{(j)}, \pi_L^{(i)})$ that achieve cumulative task rewards above a threshold. This two-stage structure promotes modularity and allows for selective reuse of subtask policies.

5 Experiments

We empirically evaluate whether framing reward generation as a hierarchical problem offers advantages over the traditional flat (non-hierarchical) formulation. Specifically, we aim to evaluate:

Q1. Are the generated alignment rewards syntactically correct?
Q2. Do they preserve task feasibility?
Q3. Do they successfully induce behaviors that match the provided specifications?

To investigate **Q3**, we additionally conduct user studies. Further experimental details are provided in Sec. D of the Appenfix.

5.1 Baselines

To evaluate L2HR (referred to as *Hier* in the tables), we consider the following baselines. For all LLM-based experiments, we generate $k = 8$ reward function candidates per trial and repeat this process 3 times, resulting in a total of 24 reward candidates per configuration.

5.1.1 Language to Flat Reward (Flat). We generate flat alignment reward $\tilde{r}_{flat}(s, a)$ from language and incorporate this function into the hierarchical setting using Eq. 4 and 5 to maintain a consistent two-level training framework. Prompts are identical to those used for L2HR, except that the prompts for flat reward generation

Task description: The objective is to pick up all apples and eggs on the dining table and place them in the sink...
Environment context:

```
'''
Background: PnP_LL_Actions = [...], PnP_HL_Actions = [...] ...
'''
class ThorPickPlaceEnv(gym.Env):
    def __init__(...): ...
```

Relevant task spaces: The agent's option/subtask space consists of picking up and placing the two types of objects...
Low-level user preference: The agent should avoid the stool while on its way to pick up or drop an egg...
High-level user preference: The agent should pick up an object type that is different from what was previously picked...
Additional info: Do not use function attributes or global variables...

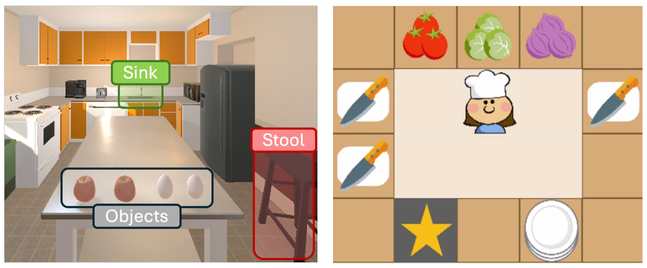
(a) Natural Language Specifications

```
# High-level preference reward
def get_high_level_pref_gpt(state: Dict, prev_option: int, option: int) ->
    Tuple[float, Dict[str, float]]:
    ...
    return reward, reward_components

# Low-level preference reward
def get_low_level_pref_gpt(state: Dict, option: int, action: int) ->
    Tuple[float, Dict[str, float]]:
    ...
    return reward, reward_components
```

(b) Reward Functions designed by L2HR

Figure 3: Illustration of L2HR input and output. See Appendix Sec. D.5 for more details.



(a) iTHOR Domain **(b) Kitchen Domain**

Figure 4: Rendered scenes from the iTHOR and Kitchen domains. Fig. 1 (right) illustrates the Rescue World domain.

exclude option-related instructions (since flat rewards are independent of options) and express user preferences as a single combined description rather than separate high- and low-level specifications.

5.1.2 No Reward Design (Task). This baseline does not utilize LLMs and uses only the task reward r and pseudo-reward r_p without incorporating any behavioral specifications (i.e., $\tilde{r}_H = \tilde{r}_L = 0$). As this setting is less noisy, we run it for 5 trials.

5.2 Domains

We evaluate the approaches on three long-horizon, multi-subtask domains: Rescue World, iTHOR, and Kitchen. Each domain poses distinct challenges in subtask sequencing and low-level execution. Furthermore, as we see later, flat rewards cannot in principle capture all specifications in Rescue World and iTHOR, whereas they can in Kitchen. This contrast allows us to investigate, *in practice*, how effectively language specifications can be translated into flat and hierarchical rewards across both types of scenarios. Additional details are provided in Sec. D.1, Sec. D.4, and Fig. 8.

5.2.1 Rescue World. A variant of the Rescue World for Teams (RW4T) domain [47], where the agent must collect and deliver all supplies in the environment. This domain features a large state space represented by a 407-dimensional vector and poses a long-horizon challenge, requiring the agent to complete 8 subtasks, each lasting up to around 10 steps. Behavioral specifications include: (1) a high-level *persistence* specification for delivering all supplies of one type before switching to another and (2) a low-level *safety* specification for avoiding hazardous zones while carrying supplies.

5.2.2 iTHOR. Built upon the Unity game engine, iTHOR is an environment within the AI2-THOR [32] framework that features several realistic household scenes in which an agent can navigate and interact with everyday objects. Here, we focus on a long-horizon pick and place task within a kitchen setting consisting of 8 subtasks, each requiring up to approximately 30 steps to complete. The agent must deliver a set of apples and eggs located on the dining table to the sink on the other side of the room. The state space is represented by a 30-dimensional vector that contains object and agent positions and object states. Behavioral specifications include: (1) a high-level *diversity* specification that requires delivering a different item from the one previously delivered and (2) a low-level *avoidance* specification that prevents the agent from going near a stool placed in the environment while picking up or delivering an egg.

5.2.3 Kitchen. A single-agent variant of Overcooked, a benchmark environment originally developed for studying human-AI collaboration in kitchen tasks [38, 67]. In our setting, the agent needs to prepare a salad with lettuce, tomatoes, and onions. This domain features an even larger state space, represented by a 699-dimensional continuous vector that captures various ingredient states. It also involves a long-horizon task requiring the completion of 5 subtasks in a strict sequence, with the final 2 subtasks dependent on the successful completion of all preceding ones. The high-level behavioral specification is a preferred *chopping* sequence (e.g., tomatoes → onions → lettuce). Since the environment uses fixed low-level policies, we skip low-level reward design in this domain.

5.3 Numerical Experiments

As a precursor to evaluting solutions to HRDL, we also conducted experiments where hand-crafted flat and hierarchical rewards were used directly to train policies *without requiring reward generation from language specifications*. These experiments serve as a proof-of-concept and demonstrate that:

- given expert-specified hierarchical rewards $(\tilde{r}_H^*, \tilde{r}_L^*)$, existing RL algorithms can effectively learn hierarchical policies (π_H^*, π_L^*) that achieve high task performance and strong alignment with designer specifications; and
- while expert-specified flat rewards $\tilde{r}_{flat}^*$ can capture some behavioral specifications, they fail to express ones that require knowledge of the previous subtask (e.g., the *persistence* specification in Rescue World and the *diversity* specification in iTHOR).

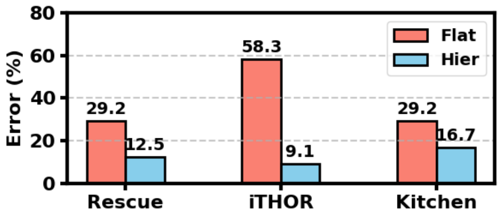


Figure 5: Syntax error rates for LLM-based reward generation, computed over 24 candidates per configuration.

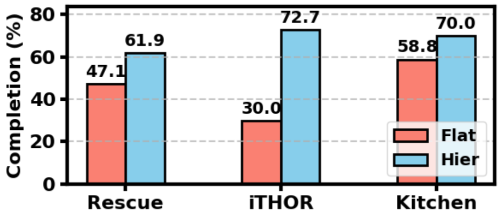


Figure 6: Task completion rates for LLM-generated rewards, calculated as the proportion of designed rewards that preserve task feasibility among syntactically valid candidates.

We report these preliminary results in Sec. D.2 in the Appendix. Now, we return to the core HRDL setting, where designed rewards must be synthesized from natural language inputs.

5.3.1 ***Q1****. Are the designed rewards syntactically correct?* Figure 5 shows that hierarchical reward generation achieves substantially lower code generation error rates than flat reward generation in all three domains, *suggesting that formulating the HRL reward design problem hierarchically can simplify reward synthesis for LLMs.* In Rescue World, the main source of errors for flat rewards is the infeasibility of expressing the *persistence* specification using only state features – this requires access to the prior option, which flat rewards cannot capture. Despite this, the LLM attempts to enforce the specification, hallucinating variables like `last_delivered_type` that are not available in the state or prompt. Flat reward generation similarly struggles in iTHOR when trying to express the *diversity* specification as this also requires access to the prior option, causing the LLM to attempt to access internal environment state variables that are not available to it. In Kitchen, higher error rates stem from the complexity of reasoning over low-level actions. For example, correctly checking if the agent is chopping an onion on the low-level requires inspecting coordinate-level state variables, which frequently leads to errors such as `'int' object is not subscriptable`. In contrast, having access to the options space in HRD enables direct reasoning over high-level behaviors (e.g., `chop onion`), greatly simplifying reward generation.

5.3.2 ***Q2****. Do the designed rewards preserve task feasibility? Hier* consistently better preserves task feasibility than *Flat* also across all domains (Figure 6), which is essential for real-world deployment. In Rescue World, attempts to encode the *persistence* specification with a flat reward often rely on spurious assumptions (e.g., inferring the last delivered item type from location), leading to unintended behaviors. In iTHOR, the agent's task completion rate is notably low. Similar to Rescue World, the agent often relies on spurious heuristics to determine the type of object previously delivered, resulting in unintended reward accumulation. Moreover, when executing the "PickNearestTarget" primitive, the agent fails to correctly identify which object it is picking up: it checks object states (e.g., whether an item is in the sink or on the dining table) rather than computing distances to determine the nearest object. This behavior further contributes to incorrect reward accumulation. In Kitchen, flat rewards often struggle to reason over low-level actions; for instance, it can be error-prone to infer which cutting board or ingredient the agent is interacting with based solely on directional actions, leading to alignment rewards that interfere with task completion.

5.3.3 ***Q3****. Do the designed rewards actually lead to agent behavior that match the behavioral specifications?* Table 1 summarizes how well each method aligns with the behavioral specifications, as measured by the handcrafted ground-truth rewards ($\tilde{r}_H, \tilde{r}_L$). The *Total* metric combines task and alignment rewards, serving as an empirical proxy for the overall fitness F. In Rescue World, *Hier* substantially outperforms both *Task* and *Flat* baselines on high-level alignment, achieving an average return of 16.65 and fully matching the high-level specification in 76.92% of successful runs. This highlights *Hier*'s ability to encode the *persistence* specification, which flat rewards fundamentally cannot represent. While *Flat* occasionally (12.50%) attains expert-level high-level alignment, these instances are coincidental. Both methods perform comparably on low-level alignment, as the agent's carrying status can be inferred directly from observable states without explicit option conditioning. Overall, *Hier* achieves the highest total return, with 69.23% of policy pairs reaching expert-level alignment on both levels, demonstrating that it captures designer intent while maintaining task success.

In iTHOR, *Hier* outperforms *Flat* on high-level alignment (14.19 vs. 7.67) in a similar manner as in Rescue World, as the *diversity* specification cannot be fully captured by the flat reward without access to the agent's current option. In this case, *Hier* also significantly outperforms *Flat* on low-level alignment (-3.75 vs. -35.20) as well. We notice that although the LLM is explicitly asked to penalize the agent's proximity to the stool when *on its way* to picking up or dropping an egg, the generated flat rewards only apply the penalty in the timestep that the agent is specifically performing the pick or place action, leading the agent to not avoid the stool. This makes sense, as it is difficult to discern the agent's intent in picking up or dropping an egg from just the state without option information.

In Kitchen, *Hier* again surpasses *Flat*, achieving higher high-level alignment returns (0.39 vs. 0.06) and a substantially greater success rate in capturing the *chopping* specification (92.86% vs. 10.00%). Importantly, while the flat reward formulation is *theoretically capable* of capturing the desired chopping behavior, doing so requires complex and error-prone logic: only 1 flat reward candidate successfully implemented the intended specification. This demonstrates a key advantage of designing rewards for HRL with HRD: even when flat rewards are theoretically sufficient, hierarchical rewards can simplify reward design through high-level abstractions and lead to better alignment with behavioral specifications. Example videos of policies for all domains are included in supplementary material.

Table 1: Table showing the performance of policies trained with the task reward either alone or combined with LLM-generated flat or hierarchical rewards. For each metric, we report both the cumulative reward returns and the percentage of policies at expert-level alignment (attaining the maximum possible cumulative return for that metric). "High-Level" and "Low-Level" rewards for evaluation are computed using handcrafted alignment rewards. "Total" represents the sum of the task reward and both the high and low-level alignment rewards. Means and standard deviations are computed over all runs for the *Task* baseline, and only over the LLM-generated reward candidates that successfully complete the task for *Flat* and *Hier*.

Domain	Method	High-Level		Low-Level		Total	
		Rewards ↑	Expert % ↑	Rewards ↑	Expert % ↑	Rewards ↑	Expert % ↑
Rescue	Task	11.22 ± 5.57	20.00	-16.46 ± 5.49	0.00	73.80 ± 5.70	0.00
	Flat	9.38 ± 7.02	12.50	-2.62 ± 5.19	62.50	85.13 ± 9.33	12.50
	Hier	**16.65 ± 6.93**	**76.92**	**-0.69 ± 1.58**	**76.92**	**93.98 ± 9.01**	**69.23**
iTHOR	Task	4.10 ± 1.34	0.00	-23.38 ± 1.86	0.00	12.31 ± 0.66	0.00
	Flat	7.67 ± 4.48	0.00	-35.20 ± 12.42	0.00	3.27 ± 9.31	0.00
	Hier	**14.19 ± 2.23**	**87.50**	**-3.75 ± 8.14**	**75.00**	**37.68 ± 6.68**	**62.50**
Kitchen	Task	0.00 ± 0.00	0.00	–	–	0.75 ± 0.00	0.00
	Flat	0.06 ± 0.13	10.00	–	–	0.80 ± 0.12	10.00
	Hier	**0.39 ± 0.05**	**92.86**	–	–	**1.08 ± 0.05**	**92.86**

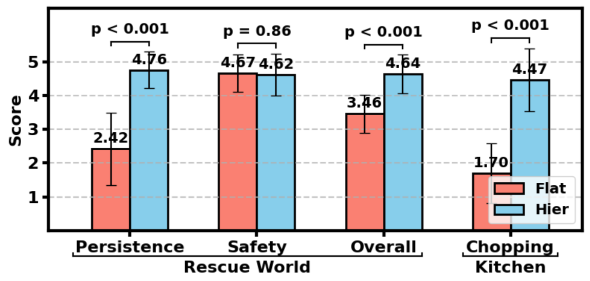


Figure 7: Human-provided ratings for agent alignment.

5.4 Evaluations with Human Participants

In real-world applications, manually designed ground-truth rewards are rarely available. To better reflect practical deployment scenarios, we conducted an IRB-approved user study on Rescue World and Kitchen using human participants recruited via Prolific. The goal was to assess whether *Hier* agents are perceived as better aligned with behavioral specifications than *Flat* agents.

In this study, non-expert participants effectively served the role of fitness function F, providing human-centered evaluations. Participants viewed videos of agent behaviors produced by both methods and rated their alignment with textual specifications on a scale from 1 (least aligned) to 5 (most aligned), similar to the scale employed in the evaluation methodology of [34]. Participants were not aware of the underlying reward design methods of the policies. We collected usable responses (e.g., those that passed attention checks) from 30 participants, evenly split across the two domains. Further details of the study design are provided in Sec. E in the Appendix.

Fig. 7 shows that participants rated *Hier* agents substantially higher than *Flat* agents for aligning with high-level behavioral specifications. In Rescue World, *Hier* significantly outperforms *Flat* on the *persistence* preference (4.76 vs. 2.42), a specification that flat rewards struggle to capture due to the lack of previous option information. While both methods achieved similar ratings for the low-level *safety* specification, *Hier* achieves significantly higher *overall* alignment scores (4.64 vs. 3.46). These statistically significant differences suggest that HRD can be better suited for capturing complex behavioral specifications.

Table 2: Candidate agent policies (%) that received a perfect alignment score from all human participants.

Method	Rescue World			Kitchen
	Persistence ↑	Safety ↑	Overall ↑	Chopping ↑
Flat	12.50%	50.00%	12.50%	10.00%
Hier	**76.92%**	**61.54%**	**53.85%**	**71.43%**

In Kitchen, *Hier* sees an even more pronounced improvement for the *chopping* specification (4.47 vs. 1.70, $p < 0.001$). This larger gap arises because, unlike in Rescue World, the preferred behavior in Kitchen rarely occurs accidentally, as aligning with the *chopping* specifications requires taking additional steps in the environment. Notably, across both domains, *Hier* policies consistently receives average ratings above 4, indicating a strong perception of alignment. These findings suggest that, in practice, when task completion is used to filter out unsuccessful policies, the remaining *Hier* candidates are consistently well-aligned with behavioral specifications.

Table 2 shows that over half of the policies produced by *Hier* achieve perfect human ratings across all behavioral specifications, substantially outperforming those generated by *Flat*. Overall, *Hier* consistently outperforms *Flat* in capturing language-based behavioral specifications across domains in both simulated evaluations and user studies.

6 Conclusion

This paper introduces the **Hierarchical Reward Design (HRD) problem**, which (1) formulates a more expressive reward structure than flat rewards, (2) integrates seamlessly with existing decision-making frameworks and RL algorithms, and (3) better encodes behavioral specifications for long-horizon tasks, with an initial solution to this problem (**L2HR**) achieving considerably better or comparable results in both numerical and human evaluations.

While our results provide strong motivation for HRD, several limitations and interesting areas for future investigation remain. First, our experiments focus on complex but simulated domains; future work should evaluate the effectiveness of HRD in real-world applications, such as robotics and interactive AI systems. Additionally, exploring more sophisticated reward generation techniques (potentially incorporating evolutionary optimization or human feedback) remains a promising direction for future research.

Last but not least, *we emphasize that human-agent alignment is inherently challenging and HRD should not be viewed in isolation.* Rather, it serves as a complementary approach within a broader ecosystem of methods, including learning from demonstrations, rankings, user corrections, and more. Understanding how HRDL and L2HR interact with these approaches is an important avenue for future work, bringing us closer to AI agents that are reliably aligned with human needs, values and objectives.

REFERENCES

[1] Pieter Abbeel and Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*. 1.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

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

[4] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* (2016).

[5] Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[6] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862* (2022).

[7] Chandrayee Basu, Mukesh Singhal, and Anca D Dragan. 2018. Learning from richer human guidance: Augmenting comparison-based learning with feature queries. In *Proceedings of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*. 132–140.

[8] Siddhant Bhambri, Amrita Bhattacharjee, Durgesh Kalwar, Lin Guan, Huan Liu, and Subbarao Kambhampati. 2024. Extracting Heuristics from Large Language Models for Reward Shaping in Reinforcement Learning. *arXiv preprint arXiv:2405.15194* (2024).

[9] Matthew M Botvinick, Yael Niv, and Andew G Barto. 2009. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *cognition* 113, 3 (2009), 262–280.

[10] Richard Catrambone. 1998. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of experimental psychology: General* 127, 4 (1998), 355.

[11] Ruihai Chen, Hao Li, Guanwei Yan, Haojie Peng, and Qian Zhang. 2023. Hierarchical reinforcement learning framework in geographic coordination for air combat tactical pursuit. *Entropy* 25, 10 (2023), 1409.

[12] Sonia Chernova and Andrea L Thomaz. 2022. *Robot learning from human teachers*. Springer Nature.

[13] Carlos G Correa, Mark K Ho, Frederick Callaway, Nathaniel D Daw, and Thomas L Griffiths. 2023. Humans decompose tasks by trading off utility and computational cost. *PLoS computational biology* 19, 6 (2023), e1011087.

[14] Peter Dayan and Geoffrey E Hinton. 1992. Feudal reinforcement learning. *Advances in neural information processing systems* 5 (1992).

[15] Thomas G Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of artificial intelligence research* 13 (2000), 227–303.

[16] Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. 2023. Guiding pretraining in reinforcement learning with large language models. In *International Conference on Machine Learning*. PMLR, 8657–8677.

[17] Justin Fu, Katie Luo, and Sergey Levine. 2017. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248* (2017).

[18] Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo. 2023. Hierarchies of reward machines. In *International Conference on Machine Learning*. PMLR, 10494–10541.

[19] Juan Angel Gonzalez-Aguirre, Ricardo Osorio-Oliveros, Karen L Rodriguez-Hernandez, Javier Lizárraga-Iturralde, Ruben Morales Menendez, Ricardo A Ramirez-Mendoza, Mauricio Adolfo Ramirez-Moreno, and Jorge de Jesus Lozoya-Santos. 2021. Service robots: Trends and technology. *Applied Sciences* 11, 22 (2021), 10702.

[20] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. 2017. Inverse reward design. *Advances in neural information processing systems* 30 (2017).

[21] Xu Han, Qiannan Yang, Xianda Chen, Zhenghan Cai, Xiaowen Chu, and Meixin Zhu. 2024. Autoreward: Closed-loop reward design with large language models for autonomous driving. *IEEE Transactions on Intelligent Vehicles* (2024).

[22] Jonathan Ho and Stefano Ermon. 2016. Generative adversarial imitation learning. *Advances in neural information processing systems* 29 (2016).

[23] Guy Hoffman and Xuan Zhao. 2020. A primer for conducting experiments in human–robot interaction. *ACM Transactions on Human-Robot Interaction (THRI)* 10, 1 (2020), 1–31.

[24] Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403* (2022).

[25] Xiaohui Huang, Jiahao Ling, Xiaofei Yang, Xiong Zhang, and Kaiming Yang. 2023. Multi-agent mix hierarchical deep reinforcement learning for large-scale

fleet management. *IEEE Transactions on Intelligent Transportation Systems* 24, 12 (2023), 14294–14305.

[26] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. 2022. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research* 73 (2022), 173–208.

[27] Yiding Jiang, Shixiang Shane Gu, Kevin P Murphy, and Chelsea Finn. 2019. Language as an abstraction for hierarchical deep reinforcement learning. *Advances in Neural Information Processing Systems* 32 (2019).

[28] Kexin Jin, Guohui Tian, Bin Huang, Yongcheng Cui, and Xiaoyu Zheng. 2024. Reward Design Framework Based on Reward Components and Large Language Models. In *2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*. IEEE, 278–282.

[29] Cory D Kidd and Cynthia Breazeal. 2008. Robots at home: Understanding long-term human-robot interaction. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 3230–3235.

[30] Juho Kim, Robert C Miller, and Krzysztof Z Gajos. 2013. Learnersourcing subgoal labeling to support learning from how-to videos. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*. 685–690.

[31] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, et al. 2017. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474* (2017).

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

[34] Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. 2023. Reward design with language models. *arXiv preprint arXiv:2303.00001* (2023).

[35] Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. 2024. Auto mc-reward: Automated dense reward design with large language models for minecraft. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16426–16435.

[36] Meng Li, Zhibin Li, Bingtong Wang, and Shunchao Wang. 2024. A Bounded Rationality-Aware Car-Following Strategy for Alleviating Cut-In Events and Traffic Disturbances in Traffic Oscillations. *IEEE Transactions on Intelligent Transportation Systems* (2024).

[37] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[38] Jijia Liu, Chao Yu, Jiaxuan Gao, Yuqing Xie, Qingmin Liao, Yi Wu, and Yu Wang. 2023. Llm-powered hierarchical language agent for real-time human-ai coordination. *arXiv preprint arXiv:2312.15224* (2023).

[39] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931* (2023).

[40] Shinya Masadome and Taku Harada. 2025. Reward Design Using Large Language Models for Natural Language Explanation of Reinforcement Learning Agent Actions. *IEEJ Transactions on Electrical and Electronic Engineering* (2025).

[41] George A Miller, Galanter Eugene, and Karl H Pribram. 2017. Plans and the Structure of Behaviour. In *Systems research for behavioral science*. Routledge, 369–382.

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[43] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. 2018. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems* 31 (2018).

[44] Yusei Naito, Tomohiko Jimbo, Tadashi Odashima, and Takamitsu Matsubara. 2025. Task-priority Intermediated Hierarchical Distributed Policies: Reinforcement Learning of Adaptive Multi-robot Cooperative Transport. In *2025 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 1556–1562.

[45] Charles Newton, Christopher Ballinger, Michael Sloma, and Keith Brawner. 2022. Hierarchical, Discontinuous Agent Reinforcement Learning Rewards in Complex Military-Oriented Environments. In *The International FLAIRS Conference Proceedings*, Vol. 35.

[46] Scott Niekum, Andrew G Barto, and Lee Spector. 2010. Genetic programming for reward function search. *IEEE Transactions on Autonomous Mental Development* 2, 2 (2010), 83–90.

[47] Liubove Orlov Savko, Zhiqin Qian, Gregory Gremillion, Catherine Neubauer, Jonroy Canady, Vaibhav Unhelkar, and Catherine Neubauer. 2024. RW4T Dataset: Data of Human-Robot Behavior and Cognitive States in Simulated Disaster Response Tasks. In *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*. 924–928.

[48] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[49] Caroline Pantofaru, Leila Takayama, Tully Foote, and Bianca Soto. 2012. Exploring the role of robots in home organization. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*. 327–334.

[50] Ronald Parr and Stuart Russell. 1997. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems* 10 (1997).

[51] Boris T Polyak and Anatoli B Juditsky. 1992. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization* 30, 4 (1992), 838–855.

[52] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[53] Preeti Ramaraj. 2023. *Analysis of Situated Interactive Non-Expert Instruction of A Hierarchical Task to a Learning Robot*. Ph.D. Dissertation.

[54] Dorsa Sadigh, Anca Dragan, Shankar Sastry, and Sanjit Seshia. 2017. *Active preference-based learning of reward functions*.

[55] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[56] Massimiliano Scopelliti, Maria Vittoria Giuliani, Alexandra M D'amico, and Ferdinando Fornara. 2004. If I had a robot at home... Peoples' representation of domestic robots. In *Designing a more inclusive world*. Springer, 257–266.

[57] Satinder Singh, Richard L Lewis, and Andrew G Barto. 2009. Where do rewards come from. In *Proceedings of the annual conference of the cognitive science society*. Cognitive Science Society, 2601–2606.

[58] Satinder Singh, Richard L Lewis, Andrew G Barto, and Jonathan Sorg. 2010. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development* 2, 2 (2010), 70–82.

[59] Satinder Pal Singh. 1992. Transfer of learning by composing solutions of elemental sequential tasks. *Machine learning* 8 (1992), 323–339.

[60] Jonathan Sorg, Richard L Lewis, and Satinder Singh. 2010. Reward design via online gradient ascent. *Advances in Neural Information Processing Systems* 23 (2010).

[61] Jonathan Sorg, Satinder P Singh, and Richard L Lewis. 2010. Internal rewards mitigate agent boundedness. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 1007–1014.

[62] Liting Sun. 2019. *Intelligent and High-performance Behavior Design of Autonomous Systems via Learning, Optimization and Control*. University of California, Berkeley.

[63] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

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

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

[66] Huaxiaoyue Wang, Kushal Kedia, Juntao Ren, Rahma Abdullah, Atiksh Bhardwaj, Angela Chao, Kelly Y Chen, Nathaniel Chin, Prithwish Dan, Xinyi Fan, et al. 2024. MOSAIC: A Modular System for Assistive and Interactive Cooking. *arXiv preprint arXiv:2402.18796* (2024).

[67] Xihuai Wang, Shao Zhang, Wenhao Zhang, Wentao Dong, Jingxiao Chen, Ying Wen, and Weinan Zhang. 2025. Zsc-eval: An evaluation toolkit and benchmark for multi-agent zero-shot coordination. *Advances in Neural Information Processing Systems* 37 (2025), 47344–47377.

[68] Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. 2023. Text2reward: Reward shaping with language models for reinforcement learning. *arXiv preprint arXiv:2309.11489* (2023).

[69] Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. 2023. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647* (2023).

[70] Jesse Zhang, Jiahui Zhang, Karl Pertsch, Ziyi Liu, Xiang Ren, Minsuk Chang, Shao-Hua Sun, and Joseph J Lim. 2023. Bootstrap your own skills: Learning to solve new tasks with large language model guidance. *arXiv preprint arXiv:2310.10021* (2023).

[71] Shangtong Zhang and Shimon Whiteson. 2019. DAC: The double actor-critic architecture for learning options. *Advances in Neural Information Processing Systems* 32 (2019).

[72] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, Anind K Dey, et al. 2008. Maximum entropy inverse reinforcement learning.. In *Aaai*, Vol. 8. Chicago, IL, USA, 1433–1438.

APPENDIX TABLE OF CONTENTS

A Further Discussion of Related Work

Reward Machines (RMs) Reward Machines (RMs) [26] provide a structured representation of reward functions. However, their primary objective differs fundamentally from that of Reward Design and, consequently, HRD. RMs aim to help RL agents exploit its reward structure to improve sample efficiency during learning. In contrast, HRD focuses on producing hierarchical reward structures that maximize policy fitness, which in our case is measured by the policy's alignment with behavioral specifications. In addition to the different goals, the underlying representations are also distinct. RMs rely on temporal logic formulas to define structured rewards, while HRD extends the original Reward Design Problem [57] and defines hierarchical reward functions inspired by established HRL frameworks (e.g., options and feudal hierarchies). Building on RMs, [18] introduced Hierarchies of Reward Machines (HRMs) to increase the expressivity of RMs and further accelerate policy convergence in some cases. Again, while HRMs aim to improve learning efficiency, HRD focuses on enabling the capture of complex behavioral specifications through a hierarchical reward structure, particularly for long-horizon tasks. Because RMs or HRMs do not focus specifically on reward design from human input, the ease of specifying RMs or HRMs from human input remains to be investigated. In contrast, for the HRD problem using the L2HR algorithm and human evaluations, we have empirically shown that language models can leverage the structure of HRD to generate hierarchical rewards that induce policies well-aligned with complex behavioral specifications. An interesting future direction is to explore novel approaches that leverage complementary strengths of RMs and HRD to facilitate both sample-efficient and human-aligned AI.

LLM-RL Hybrid Agents Although our work focuses on hierarchical RL paradigms, there is a growing body of research on hybrid agents that combine large language models (LLMs) for subtask selection with reinforcement learning for subtask execution [3, 70]. HRD can help contribute to this line of work in two key ways. First, it can provide a formalism for analyzing reward design in hybrid LLM-RL systems. Second, it can inform the development of advanced methods that leverage LLMs with hierarchical reward structures, combining the *expressivity* of hierarchical rewards with a *user-friendly* reward design process.

B Theoretical Analysis

B.1 Low- and High-Level Decision Models

B.1.1 Low-Level MDP Models

Proposition 1 (Low-level MDP Models). *Let $\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma)$ be a world model, O a set of options, and $r_L : \mathcal{S} \times O \times \mathcal{A} \to \mathbb{R}$ a low-level reward. For a fixed option $o \in O$, the tuple $\mathcal{M}_{L,o} = (\mathcal{S}, \mathcal{A}, T, r_L(\cdot, o, \cdot), \gamma, h_o)$ defines an MDP, where h_o is the horizon determined by the option's termination condition $\beta(\cdot, o)$.*

Proof. The state space is $\mathcal{S}$ and the action space is $\mathcal{A}$. The transition function T is the transition function of the world model and is Markovian. Defining $r_{L,o}(s, a) = r_L(s, o, a)$ produces a Markov reward function that depends only on the state-action pair. Thus, the tuple satisfies the four standard components of an MDP. □

B.1.2 High-Level SMDP Model

Proposition 2 (High-level SMDP Model). *Let $\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma, h)$ be a world model, $\mathcal{O}$ a set of options, and $r_H : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \mathbb{R}$ the high-level reward. Then, $\mathcal{M}_H = (\mathcal{O} \times \mathcal{S}, \mathcal{O}, T_H, r_H, \gamma, h)$ forms a semi-MDP, where $T_H : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \Delta(\mathcal{O} \times \mathcal{S} \times \mathbb{N})$ defines the joint distribution over the next augmented state and transit time, where $\mathbb{N}$ is the set of natural numbers.*

Proof. A semi-MDP (SMDP) requires (1) a set of states, (2) a set of actions, (3) for each state-action pair, an expected discounted reward, and (4) a well-defined joint distribution over the next state and transit time [64].

We define the state space as the Cartesian product $\mathcal{O} \times \mathcal{S}$, combining the previous option $o^- \in \mathcal{O}$ and the current environment state $s \in S$. The action space is the set of options $\mathcal{O}$. The reward function for each state-action pair $\big((o^-, s), o\big)$ is given by the provided high-level reward $r_H(o^-, s, o)$.

To define the transition dynamics, consider the transition probability from a given augmented state (o^-, s) upon selecting an option o. Let s' denote the state upon option termination and k the number of timesteps to reach s'. The transition function T_H is defined as:

$$T_H(o, s', k | o^-, s, o) = \sum_{\tau:\ (o^-, s) \to (o, s')} \Pr(\tau; \pi_{L,o}) \cdot \mathbb{I}_{\{k = |\tau|\}} \cdot \beta(s', o) \tag{6}$$

where τ is any trajectory that starts in state s after option o^- and reaches state s' via option o, $\Pr(\tau; \pi_{L,o})$ is the probability of trajectory τ under policy $\pi_{L,o}$, and $|\tau|$ is the number of timesteps taken by the trajectory. Given a specific $\tau = (o_{t-1}, s_t, o_t, s_{t+1}, \ \dots, o_{t+\eta-1}, s_{t+\eta})$ and knowing that $o_{t-1} = o^-$ and $o_t, \ \dots, o_{t+\eta-1} = o$, the probability of the trajectory is given by:

$$\Pr(\tau; \pi_{L,o}) = \prod_{i=0}^{\eta-1} \Big(\sum_a \pi_{L,o}(a | s_{t+i}) T(s_{t+i+1} | s_{t+i}, a) \Big) \cdot \prod_{i=0}^{\eta-2} \Big(1 - \beta(s_{t+i+1}, o) \Big) \tag{7}$$

The first product accounts for the probabilities of transitioning through the intermediate states under $\pi_{L,o}$ and the environment's underlying dynamics T. The second product ensures that the option does not terminate at intermediate states prior to reaching s'. As all four conditions are satisfied, the tuple $\mathcal{M}_H$ forms a valid SMDP.

□

B.1.3 High-Level MDP Model Alternatively, the high-level process can be modeled as an MDP if *single-step* high-level rewards $r_H^{step}(o_{t-1}, s_t, o_t)$ are defined. [71] was the first to model high-level decision-making as an MDP within the options framework. While our formulation differs from theirs, we draw inspiration from their use of *single-step* high-level rewards and demonstrate that the high-level process in our setting can also be modeled as an MDP.

Proposition 3 (High-level MDP Model). *Let $\mathcal{M}_p = (\mathcal{S}, \mathcal{A}, T, \gamma, h)$ be a world model, $\mathcal{O}$ a set of options, and $r_H^{step} : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \mathbb{R}$ a "single-step" high-level reward. Then, $\mathcal{M}_H^{step} = (\mathcal{O} \times \mathcal{S}, \mathcal{O}, T_H^{step}, r_H^{step}, \gamma, h)$ forms an MDP, where $T_H^{step} : \mathcal{O} \times \mathcal{S} \times \mathcal{O} \to \Delta(\mathcal{O} \times \mathcal{S})$.*

Proof. We again define the state space as $\mathcal{O} \times \mathcal{S}$ and the action space as $\mathcal{O}$. The transition function is defined as: $T_H^{step}(o, s' | o^-, s, o) = \sum_a \pi_L(a | s, o) \cdot T(s' | s, a) \cdot \mathbb{I}_{\{o' = o\}}$. Termination condition is not modeled in the transition function, since option selection occurs at every step. The reward function is r_H^{step}, which satisfies the MDP requirements. □

As a side note, we can derive the expected SMDP high-level reward from the single-step rewards as follows:

$$r_H(o^-, s, o) \doteq \mathbb{E}[\sum_{i=1}^{k} \gamma^{i-1} {r_H^{step}}_{t+i} | \mathcal{E}(o^-, o, s, t)] \tag{8}$$

where $\mathcal{E}(o^-, o, s, t)$ is the event of initiating option o in state s at timestep t following option o^-, and k is the random variable denoting the number of steps after which the initiated option o terminates, as determined by its termination condition β.

B.2 Policy Learning with Hierarchical Rewards

Given hierarchical reward functions in Definition 1 and Definition 2, agents can learn low- and high-level policies through interactions with the environment under their respective decision-making models.

B.2.1 Low-Level Policy Learning For each option $o_i \in \mathcal{O}$, the low-level decision-making process can be formulated as an MDP $\mathcal{M}_{L,o_i}$ (see Proposition 1). Let $r_{L,o_i}(s, a) = r_L(s, o_i, a)$ represent the reward function for option o_i and $\pi_{o_i}(a|s)$ a policy in the MDP. The objective for low-level policy learning is to maximize the cumulative discounted rewards until the termination of option o_i:

$$\pi_L(a | s, o{=}o_i) = \arg\max_{\pi_{o_i}} \mathbb{E}_{s_t \sim T, a_t \sim \pi_{o_i}} [\sum_{t=0}^{h_o} \gamma^t r_{L,o_i}(s_t, a_t) | \mathcal{M}_{L,o_i}] \tag{9}$$

where h_o is the horizon determinied by the option's termination condition $\beta(\cdot, o_i)$. Standard RL algorithms for MDPs can be directly used to obtain low-level policies.

B.2.2 High-Level Policy Learning (SMDP) When the high-level decision-making is modeled by an SMDP, the high-level policy $\pi_H(o|o^-, s)$ selects options only at the termination of the previous option, operating on a coarser temporal scale compared to the low-level policy $\pi_L(a|s, o)$. Let $u = 0, \ldots, h$ index the *high-level* decision points, and define η_{o_u} as the number of primitive timesteps taken to execute option o_u. Then, the high-level policy learning objective is:

$$\pi_H(o|o^-, s) = \arg\max_{\pi} \mathbb{E}_{(o_{u-1}, s_u, \eta_{o_{u-1}}) \sim T_H,\, o_u \sim \pi}[\sum_{u=0}^{h} \gamma^{T_u} r_H(o_{u-1}, s_u, o_u)|\mathcal{M}_H], \tag{10}$$

$$\text{where } T_u = \sum_{j=-1}^{u-1} \eta_{o_j}$$

Here, we need to extend the SMDP by introducing a dummy initial option $o_\#$, with zero duration, and let $o_{-1} = o_\#$. Any algorithm for learning option-level policies in an SMDP can be used here.

B.2.3 High-Level Policy Learning (MDP) The learning objective when the high-level decision-making is modeled by an MDP is:

$$\pi_H^{step}(o|o^-, s) = \arg\max_{\pi} \mathbb{E}_{(o_{t-1}, s_t) \sim T_H^{step},\, o_t \sim \pi}[\sum_{t=0}^{h} \gamma^t r_H^{step}(o_{t-1}, s_t, o_t))|\mathcal{M}_H^{step}] \tag{11}$$

Similarly, we let $o_{-1} = o_\#$. When the termination condition is known, as in our setting, the high-level policy can be expressed as: $\pi_H^{step}(o|o^-, s) \doteq \beta(s, o^-)\pi_H(o|o^-, s) + (1 - \beta(s, o^-))\mathbb{I}_{(o=o^-)}$, where $\pi_H(o|o^-, s)$ specifies the policy at decision points. This allows leveraging known termination conditions to constrain policy rollouts.

B.3 Expressivity of HRD

Property 1. *Certain specifications on sub-task selection can be expressed through $\tilde{r}_H(o^-, s, o)$, but they cannot be expressed by a flat reward function: $\tilde{r}_{flat}(s, a)$.*

Proof. Consider two distinct subtask sequences, $\{o_1 \longrightarrow o_2\}$ and $\{o_1' \longrightarrow o_2'\}$ such that after executing either o_1 or o_1', the agent arrives at the same state s^*. The designer's specification is for the agent to follow the corresponding subtask sequences: execute o_2 after o_1 and o_2' after o_1'. A flat reward function $\tilde{r}_{flat}(s, a)$ cannot represent this specification, as the reward signal depends only on the current state and action, and cannot distinguish whether the agent reached s^* via o_1 or o_1'. In contrast, with HRD's high-level reward function, we can specify the ordering even in s^* by defining $\tilde{r}_H(o^-{=}o_1, s{=}s^*, o{=}o_2) > \tilde{r}_H(o^-{=}o_1, s{=}s^*, o \neq o_2)$ and $\tilde{r}_H(o^-{=}o_1', s{=}s^*, o{=}o_2') > \tilde{r}_H(o^-{=}o_1', s{=}s^*, o \neq o_2')$. □

Property 2. *Certain specifications on sub-task execution can be expressed through $\tilde{r}_L(s, o, a)$, but they cannot be expressed by a flat reward function: $\tilde{r}_{flat}(s, a)$.*

Proof. Consider a setting where the behavioral specification explicitly depends on the current option o. Such specifications cannot be represented using a flat reward function, as the reward signal $\tilde{r}_{flat}(s, a)$ is identical for all option values. □

C Language to Hierarchical Rewards (L2HR) Pseudocode

Algorithm 1 Language to Hierarchical Rewards (L2HR)

```
1:  Input: World model $\mathcal{M}_p$, set of options $O$, learning routine $\mathcal{A}_{\mathcal{M}_p}$, task reward $r$, pseudo-reward $r_p$, specifications $l$, thresholds for
    successful subtask and task completion $t_L$ and $t_H$, and an LLM.
2:  $\tilde{r}_L^{(1)}, \ldots, \tilde{r}_L^{(k)} \leftarrow$ LLM(LowLevelPrompt($l$))            ▷ Generate low-level alignment rewards
3:  for $i = 1$ to $k$ do
4:      if Static_Check($\tilde{r}_L^{(i)}$) then
            // $\mathcal{A}_{\mathcal{M}_{p:L,\cdot}}$ denotes the learning subroutine for low-level MDPs $\mathcal{M}_{L,\cdot}$.
5:          $\pi_L^{(i)} \leftarrow \mathcal{A}_{\mathcal{M}_{p:L,\cdot}}(r_p + \tilde{r}_L^{(i)})$
6:      end if
7:  end for
8:  $\mathcal{I} \leftarrow$ Indices of $\pi_L^{(i)}$ achieving cumulative pseudo-rewards above threshold $t_L$
9:  $\tilde{r}_H^{(1)}, \ldots, \tilde{r}_H^{(k)} \leftarrow$ LLM(HighLevelPrompt($l$))           ▷ Generate high-level alignment rewards
10: for $j = 1$ to $k$ do
11:     if Static_Check($\tilde{r}_H^{(j)}$) then
12:         Select low-level policy $\pi_L^{(i)}$, where $i \in \mathcal{I}$           ▷ Done via hashing in our implementation
            // $\mathcal{A}_{\mathcal{M}_{p:H}}$ denotes the learning subroutine for the high-level model $\mathcal{M}_H$
13:         $\pi_H^{(j)} \leftarrow \mathcal{A}_{\mathcal{M}_{p:H}}(r + \tilde{r}_H^{(j)}; \pi_L^{(i)})$
14:     end if
15: end for
16: Output: Return all alignment rewards $(\tilde{r}_H^{(j)}, \tilde{r}_L^{(i)})$ and corresponding trained policy pairs $(\pi_H^{(j)}, \pi_L^{(i)})$ that achieve cumulative task rewards
    above threshold $t_H$
```

D Experimental Details

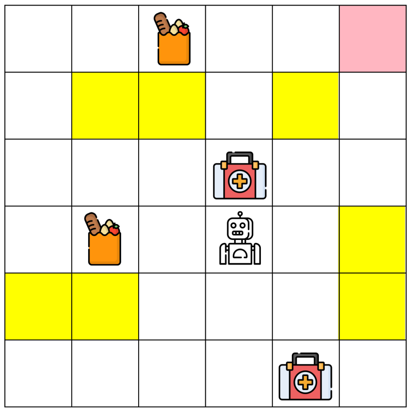
D.1 Further Domain Details

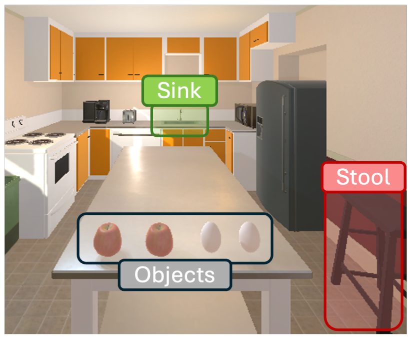
Rescue World is a variant of the RW4T domain [47], a configurable testbed for simulating disaster response scenarios in which a first responder deploys robots to collect scattered supplies and deliver them to designated areas. For our experiments, we configure the environment with a single robot and represent the world using a discrete grid-based layout (Fig. 8a). The robot must determine both *the delivery order* of supplies and *the optimal path* for completing deliveries. This setting naturally motivates a hierarchical action representation due to the need for subtask sequencing and execution. The primitive action space $\mathcal{A}$ includes six actions: pick, drop, and four directional movements (up, down, left, right). The option space O consists of multi-step pick-up and delivery macro-actions for two supply types: *food* and *medical supplies*.

iTHOR is a simulator built within the AI2-THOR framework [32], featuring realistic household environments where an agent can navigate and interact with everyday objects. We use *FloorPlan 20* as the environment for our experiments. In our setup, apples and eggs are spawned on one side of a long kitchen table, the sink is located on the opposite side, and a stool is positioned to the right of the table, as shown in Fig. 8b. The option space O consists of pick-up and place operations for each object type, while the action space $\mathcal{A}$ includes navigation actions (move forward, turn left, turn right) and pick/place primitives. While conducting experiments with expert-provided rewards (Sec. D.2), we observe that inducing the agent to follow the *diversity* preference is highly sensitive to the reward scale. To address this, we include additional contextual information in the LLM prompt specifically about the typical length of the options to guide more consistent reward generation.

Kitchen is a single-agent variant of Overcooked, a benchmark environment originally designed for studying human-AI collaboration in kitchen tasks [38, 67]. In our setting, the agent must prepare a salad using lettuce, tomatoes, and onions. We adopt the structured option space from [38], which includes high-level options such as `chop onion` and `combine chopped onion and chopped tomatoes`. The environment also provides hard-coded low-level controllers for executing these options. The primitive action space $\mathcal{A}$ includes directional movement actions that enable interactions with countertops and other kitchen objects.

The visualizations of the three task domains used in our study are shown in Fig. 8. The Rescue World visualization was generated using Pygame based on its Gym environment, the iTHOR visualization was created by visually rendering the environment within the Unity engine, and the Kitchen domain visualization was adopted from [38].


(a) Rescue World Domain



(b) iTHOR Domain

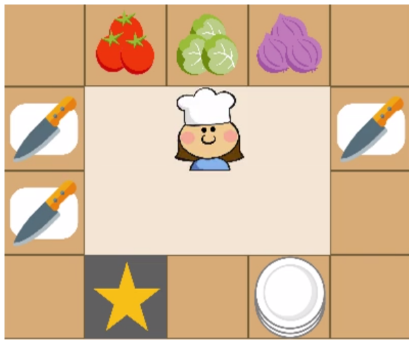

(c) Kitchen Domain

Figure 8: Screenshots of renderings of the three task domains used in our study.

Table 3: Table showing the performance of policies trained with the task reward alone and with task reward combined with expert-provided flat or hierarchical rewards. For each metric ("High-Level," "Low-Level," and "Total"), we report both the cumulative returns and the percentage of policies achieving expert-level alignment. A policy is considered expert-aligned at the high or low level if it attains the maximum possible cumulative return for that metric. "Total" represents the sum of the task reward, high-level alignment reward, and low-level alignment reward, and a policy is deemed expert-level overall if it aligns at both levels.

Domain	Method	High-Level		Low-Level		Total	
		Rewards ↑	Expert % ↑	Rewards ↑	Expert % ↑	Rewards ↑	Expert % ↑
Rescue	Task	11.22 ± 5.57	20.00	-16.46 ± 5.49	0.00	73.80 ± 5.70	0.00
	Flat*	6.84 ± 5.24	0.00	**-0.32 ± 0.72**	**80.00**	85.88 ± 4.66	0.00
	Hier*	**20.00 ± 0.00**	**100.00**	-1.00 ± 2.24	80.00	**97.15 ± 2.94**	**80.00**
iTHOR	Task	4.10 ± 1.34	0.00	-23.38 ± 1.86	0.00	12.31 ± 0.66	0.00
	Flat*	7.80 ± 2.41	0.00	**0.00 ± 0.00**	**100.00***	35.41 ± 2.98	0.00
	Hier*	**15.00 ± 0.00**	**100.00**	**0.00 ± 0.00**	**100.00**	**42.61 ± 0.35**	**100.00**
Kitchen	Task	0.00 ± 0.00	0.00	–	–	0.75 ± 0.00	0.00
	Flat*	**0.40 ± 0.00**	**100.00**	–	–	**1.10 ± 0.00**	**100.00**
	Hier*	**0.40 ± 0.00**	**100.00**	–	–	**1.10 ± 0.00**	**100.00**

D.2 Results with Expert-Provided Rewards

Table 3 reports the results of policies trained with expert-provided flat $\tilde{r}^*_{flat}$ and hierarchical rewards $(\tilde{r}^*_H, \tilde{r}^*_L)$ (shown in the table as *Flat** and *Hier** respectively). In other words, no LLMs were used for reward generation in these experiments. Across all three domains (Rescue World, iTHOR, and Kitchen), *Hier** consistently achieves the highest total returns.

In Rescue World, *Hier** matches *Flat** in low-level alignment return but significantly outperforms it in high-level alignment, achieving an average total return of 97.15 with 80.00% policies reaching expert alignment, compared to 85.88 and 0.00% for *Flat**. A similar pattern is observed in iTHOR, where *Hier** achieves an average total return of 42.61 with all policies reaching expert alignment, while *Flat** achieves 35.41 with none reaching expert alignment. We also note that although *Flat** matched *Hier** in low-level alignment for iTHOR, it did so by following the low-level *avoidance* preference regardless of the option. In contrast, *Hier** selectively applies this preference to options involving eggs, demonstrating finer-grained alignment with low-level human specifications. These results highlight that *Hier** better captures behavioral dependencies related to the current and previous options, enabling it to represent high-level *persistence* in Rescue World and *diversity* in iTHOR, which are beyond the representational capacity of flat rewards.

In Kitchen, *Hier** and *Flat** perform equally well on both task and high-level returns. While *Flat** is theoretically capable of capturing dependencies on the previous subtask in Kitchen by inferring it from environment state, doing so is difficult and error-prone without expert-designed rewards, as shown in Sec. 5. Overall, these results demonstrate that expert-designed hierarchical rewards can be easily integrated with task-related rewards to train policies that (1) achieve strong task performance comparable to baselines without reward design (2) align well with behavioral specifications, including those that flat rewards cannot effectively represent.

D.3 Training Setup

In our implementation, we used LLMs to generate *single-step* high-level rewards, allowing the high-level decision process to be modeled as either an MDP or an SMDP. When using an SMDP, we computed the corresponding SMDP rewards using Eq. 8.

D.3.1 Rescue World For Rescue World, we modeled the high-level decision-making as an SMDP and trained the high-level policy π_H using DQN [42]. The hyperparameters for training π_H are as follows:

- Network: 2 layers with 64 units each and ReLU non-linearities
- Optimizer: Adam [31]
- Learning rate: $1 \cdot 10^{-4}$
- Batch size: 256
- Discount: 1.0
- Total timesteps: $3 \cdot 10^6$
- Buffer size: $1 \cdot 10^6$
- Exploration fraction: 0.2
- Initial exploration probability: 0.1
- Final exploration probability: 0.05
- Model update frequency: 4
- Number of gradient steps per rollout: 1
- Target update interval: $1 \cdot 10^4$
- Polyak-averaging [51]: 1.0

We trained the low-level policy π_L using PPO [55]. The hyperparameters for training π_L are as follows:

- Network: 2 layers with 64 units each and ReLU non-linearities
- Optimizer: Adam [31]
- Learning rate: $3 \cdot 10^{-4}$
- Batch size: 64
- Discount: 1.0
- Total timesteps: $2 \cdot 10^6$
- Initial entropy coefficient: 1
- Final entropy coefficient: 0.01
- Entropy decay fraction: 0.5
- Number of environment steps per update: 2048

D.3.2 iTHOR For iTHOR, we modeled the high-level decision-making as an SMDP and trained the high-level policy π_H using DQN [42]. The hyperparameters for training π_H are as follows:

- Network: 2 layers with 128 units each and ReLU non-linearities
- Optimizer: Adam [31]
- Learning rate: $1 \cdot 10^{-4}$
- Batch size: 32
- Discount: 0.99
- Total timesteps: $5.0 \cdot 10^5$
- Buffer size: $5 \cdot 10^5$
- Exploration fraction: 0.25
- Initial exploration probability: 1.0
- Final exploration probability: 0.05
- Model update frequency: 4
- Number of gradient steps per rollout: 1
- Target update interval: $1 \cdot 10^4$
- Polyak-averaging [51]: 1.0

We trained the low-level policy π_L using PPO [55]. The hyperparameters for training π_L are as follows:

- Network: 2 layers with 64 units each and ReLU non-linearities
- Optimizer: Adam [31]
- Learning rate: $3 \cdot 10^{-4}$

- Batch size: 64
- Discount: 1.0
- Total timesteps: $1.5 \cdot 10^6$
- Initial entropy coefficient: 1
- Final entropy coefficient: 0.01
- Entropy decay fraction: 0.5
- Number of environment steps per update: 2048

D.3.3 Kitchen For Kitchen, we modeled the high-level decision-making as an MDP and trained the high-level policy π_H using DQN [42], implemented so that termination conditions β were enforced during rollouts. We adopted the MDP formulation because it outperformed the SMDP setting with DQN in this domain. Given the highly delayed rewards and the importance of subtask sequencing in Kitchen, we also incorporated specification-agnostic demonstrations to bootstrap policy learning. Hyperparameters for learning the high-level policy π_H are as follows:

- Network: 2 layers with 256 units each and ReLU non-linearities
- Optimizer: Adam [31]
- Learning rate: $1 \cdot 10^{-6}$ with linear scheduling
- Batch size: 256
- Discount: 0.99
- Total timesteps: $3 \cdot 10^6$
- Buffer size: $1 \cdot 10^6$
- Exploration fraction: 0.33
- Initial exploration probability: 0.5
- Final exploration probability: 0.1
- Model update frequency: 4
- Number of gradient steps per rollout: 1
- Target update interval: $1 \cdot 10^4$
- Polyak-averaging [51]: 1.0

For Rescue World and Kitchen, we used a server with 30 vCPUs and an NVIDIA A10 GPU (24GB PCIe) to train k policies in parallel, each corresponding to one of the k reward candidates generated by the LLM. For iTHOR, we used a server with an NVIDIA GeForce RTX 5090 GPU to train our policies.

D.4 LLM Prompts

The prompts used in our work are adapted from [39], but differ in important ways to realize hierarchical rewards. Specifically, our prompts are designed to: (1) reflect a hierarchical reward structure; and (2) generate rewards that align with behavioral specifications while preserving task feasibility.

D.4.1 System Prompt As in [39], our system prompt provides a concise, domain-agnostic description of the reward design task and defines the function signature that the LLM should use in its output. The full system prompt is provided in Prompt 1 on Page 18.

D.4.2 User Prompt The user prompts follow a similar methodology to that of [39], using code snippets as contextual input and an accompanying task-specific natural language description. To support hierarchical reward generation, we extend the accompanying task description by providing the following: (1) a description of relevant action spaces (i.e., the option space O and/or action space $\mathcal{A}$) to help the LLM distinguish between temporally extended behaviors and primitive actions; (2) a behavioral specification describing preferences beyond task completion; (3) additional code formatting guidelines to emphasize that the LLM should capture behavioral logic without making the reward function stateful (e.g., storing variables across calls). Moreover, the complexity of our domains introduces two additional challenges, which we address by augmenting the code snippets with further contextual information:

Cross-file Dependencies. In our environments, key components of the task logic often depend on constants and definitions from separate supporting files (e.g., `utils.py`). To address this, we manually copied the necessary definitions from these files and included them as background comments at the top of the environment code provided to the LLM. This ensures all relevant constants and definitions are explicitly exposed during reward generation.

Complex Observation Representations. Our environments feature structured observations, such as spatial maps, whose semantics are not fully captured by the observation's shape or naming alone. For example, it can be difficult to infer what each value in the observation (e.g., map cell) represents from the environment code. To mitigate this, we also provided an example observation input as part of the background comments in cases where the LLM might find it challenging to correctly interpret the structure and meaning of the observation space.

To ensure a fair comparison across conditions, the same code snippets of each domain were used for all reward generation tasks, whether generating low-level, high-level, or flat rewards. The full environment contexts and corresponding prompts for both Rescue World and Kitchen are shown in Prompt 2 and Prompt 3 on Page 18 and Page 26 respectively.

Prompt 1: System Prompt

You are a reward engineer trying to write reward functions to solve reinforcement learning tasks as effective as possible. A programmer has already specified the task reward, and your job is to specify additional rewards according to the user's preference. More specifically, your goal is to write an additional reward function for the environment to help the agent complete the task according to user preference. Your reward function should use useful variables from the environment as inputs. As an example, the reward function signature can be:

The LLM is presented with one of the following function signatures, selected based on the desired reward function to be designed.

```
def get_high_level_pref_gpt(state: Dict, prev_option: int, option: int) ->
    Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    prev_option: the last option (subtask) executed by the agent to reach the
    current state.
    option: the option (subtask) the agent is about to perform in the current
    state.
    '''
    ...
    return reward, reward_components
```

```
def get_low_level_pref_gpt(state: Dict, option: int, action: int) ->
    Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    option: the option (subtask) selected by the agent in the current state.
    action: the action that the agent is about to perform in the current state.
    '''
    ...
    return reward, reward_components
```

```
def get_flat_sa_pref_gpt(state: Dict, action: int) -> Tuple[float,
    Dict[str, float]]:
    '''
    state: the current state of the environment.
    action: the (low-level) action that the agent is about to perform in the
    current state.
    '''
    ...
    return reward, reward_components
```

The output of the reward function should consist of two items:
(1) the user preference reward,
(2) a dictionary of each individual reward component in the user preference reward.
The code output should be formatted as a python code string: "python ... ".
Some helpful tips for writing the reward function code:
(1) Most importantly, the reward code's input variables must contain only attributes of the provided environment class definition (namely, variables that have prefix self.). Under no circumstance can you introduce new input variables.

Prompt 2: User Prompt for Rescue World

The Python environment is

```
'''
Background:

1) Initial game map example
init_map = np.array(
    [[
        rw4t_utils.RW4T_State.empty.value,
        rw4t_utils.RW4T_State.empty.value,
        rw4t_utils.RW4T_State.circle.value,
        rw4t_utils.RW4T_State.empty.value,
        rw4t_utils.RW4T_State.yellow_zone.value,
        rw4t_utils.RW4T_State.school.value
    ],
     [
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.empty.value
     ],
     [
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.square.value,
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.empty.value
     ],
     [
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.circle.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value
     ],
     [
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.yellow_zone.value,
         rw4t_utils.RW4T_State.yellow_zone.value
     ],
     [
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.empty.value,
         rw4t_utils.RW4T_State.square.value,
         rw4t_utils.RW4T_State.empty.value
     ]])

2) rw4t.utils:
class RW4T_LL_Actions(Enum):
  go_left = 0
  go_down = 1
  go_right = 2
  go_up = 3
  pick = 4
  drop = 5
  idle = 6


class RW4T_HL_Actions_EZ(Enum):
  go_to_circle = 0
  deliver_circle = 1
  go_to_square = 2
  deliver_square = 3
```

```
class RW4T_HL_Actions_With_Dummy_EZ(Enum):
  go_to_circle = 0
  deliver_circle = 1
  go_to_square = 2
  deliver_square = 3
  dummy = 4


class RW4T_State(Enum):
  empty = 0
  circle = 1
  square = 2
  triangle = 3
  obstacle = 4
  yellow_zone = 5
  orange_zone = 6
  red_zone = 7
  school = 8
  hospital = 9
  park = 10


class Holding_Obj(Enum):
  empty = 0
  circle = 1
  square = 2
  triangle = 3
'''

import numpy as np
import gymnasium as gym

import rw4t.utils as rw4t_utils


class RW4T_GameState:

  def __init__(self, obs: np.ndarray, pos: np.ndarray, holding: int,
               option_mask: np.ndarray):
    '''
    :param obs: a 2D numpy of the current environment
    :param pos: a 1D numpy array of the agent's (x, y) position in the
                environment
    :param holding: an integer indicating what object the agent is currently
                    holding if any.
                    This parameter only has a non-empty value AFTER the agent
                    performs a 'pick up ...' option and BEFORE it performs a
                    'deliver ...' option.
    :param option_mask: a 1D array indicating the valid options to select next
                        (should not be used when computing rewards, this is only
                        used in some downstream algorithms)
    '''
    # Y pos in bound
    assert pos[1] >= 0 and pos[1] < len(obs)
    # X pos in bound
    assert pos[0] >= 0 and pos[0] < len(obs[0])
    # holding should be a value in the Holding_Obj Enum
    assert holding < len(rw4t_utils.Holding_Obj)
    self.obs = obs
    self.pos = pos
    self.holding = holding
    self.option_mask = option_mask
  def state_to_dict(self):
    return {
        'map': np.array(self.obs, dtype=np.int32),
        'pos': np.array(self.pos, dtype=np.int32),
        'holding': self.holding,
        'option_mask': self.option_mask
    }


class RW4TEnv(gym.Env):

  def get_state(self):
    state = RW4T_GameState(self.map, self.agent_pos, self.agent_holding,
                           self.option_mask)
    state_dict = state.state_to_dict()
    return state_dict
```

Write a reward function for the following task:

The LLM receives one of the prompts below, chosen according to the reward function we want it to design. The prompt for each setting is identical except for (1) descriptions of relevant task spaces (e.g., the flat-reward prompt omits the options space description) and (2) the behavioral specification ("user preference") string.

High-Level

Task description:
The task objective is to deliver all objects on the map. In the task reward, the agent gets a reward of +30 when it successfully delivers an object, and a step cost of -1 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's option/subtask (referred to as HL_Action in the code) space consists of going to and delivering the two types of objects. Each option takes multiple action steps to complete. Taking a 'go to' option means that the agent will navigate to a supply and pick it up. Taking a 'deliver' option means that the agent will navigate to the delivery location and drop the object. Note that the agent has to first go to the object to pick it up before delivering the object.
User preference:
The agent should pick up an object type that's the same as the previously delivered object type, if there are still objects of that type remaining in the environment. Otherwise, the agent should pick up an object of a different type.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables.) You should also not write any other helper functions.

Low-Level

Task description:
The task objective is to deliver all objects on the map. In the task reward, the agent gets a reward of +30 when it successfully delivers an object, and a step cost of -1 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's option/subtask (referred to as HL_Action in the code) space consists of going to and delivering the two types of objects. Each option takes multiple action steps to complete. Taking a 'go to' option means that the agent will navigate to a supply and pick it up. Taking a 'deliver' option means that the agent will navigate to the delivery location and drop the object. Note that the agent has to first go to the object to pick it up before delivering the object. The agent's action (referred to as LL_Action in the code) space consists of moving in the four cardinal directions, as well as atomic actions pick and drop. The agent can only perform LL_Action "pick" if it is at the same location as the object.
User preference:
The agent should avoid yellow danger zones when it is delivering an object. However, the agent does not need to avoid danger zones when it is going to an object.

Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables.) You should also not write any other helper functions.

Flat

Task description:
The task objective is to deliver all objects on the map. In the task reward, the agent gets a reward of +30 when it successfully delivers an object, and a step cost of -1 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's action (referred to as LL_Action in the code) space consists of moving in the four cardinal directions, as well as atomic actions pick and drop. The agent can only perform LL_Action "pick" if it is at the same location as the object.
User preference:
The agent should pick up an object type that's the same as the previously delivered object type, if there are still objects of that type remaining in the environment. Otherwise, the agent should pick up an object of a different type. In addition, the agent should avoid yellow danger zones when it is delivering an object. However, the agent does not need to avoid danger zones when it is going to an object.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables.) You should also not write any other helper functions.

Prompt 3: User Prompt for iTHOR

The Python environment is
```
'''
Background:

1) utils:
PnP_LL_Actions = [
    "MoveAhead",
    "RotateLeft",
    "RotateRight",
    "PickupNearestTarget",
    "PutHeldOnReceptacle",
]

class PnP_HL_Actions(Enum):
    pick_apple = 0
    pick_egg = 1
    drop_apple = 2
    drop_egg = 3

class PnP_HL_Actions_With_Dummy(Enum):
    pick_apple = 0
    pick_egg = 1
    drop_apple = 2
    drop_egg = 3
    dummy = 4
'''

import random
import gymnasium as gym
import numpy as np
from gymnasium import spaces
from ai2thor.controller import Controller
from ai2thor.platform import CloudRendering
from typing import Dict, List, Optional
from HierRL.envs.ai2thor.pnp_training_utils import (PnP_HL_Actions,
                                                    PnP_HL_Actions_With_Dummy,
                                                    PnP_LL_Actions)
from HierRL.envs.ai2thor.pnp_config import avoid_stool


class ThorPickPlaceEnv(gym.Env):
  """
  Pick-and-place environment on top of AI2-THOR, using the Gymnasium API.

  Episode structure:
    - reset() loads a kitchen scene, curates it (move/disable/spawn a few
      items), and returns an observation.
    - step(a) applies either low-level nav (Move/Rotate/Look) or a simple HL
      manipulation (Pickup nearest target / Put on nearest receptacle / Drop).
    - reward is currently 0/1 placeholder (see _compute_reward_and_done).
  """
  metadata = {"render_modes": ["rgb_array"]}

  def __init__(
      self,
      scene: str = "FloorPlan20",  # scene id
      pref_dict: Dict[str, List[int]] = avoid_stool,  # preference dictionary
      visibilityDistance: float = 1,  # meters for "visible" flag (not reach)
      grid_size: float = 0.25,  # movement step in meters
      snap_to_grid: bool = True,  # keep motion aligned to grid
      rotate_step_degrees: int = 90,  # degree per rotate action
      render_depth: bool = False,
      render_instance_masks: bool = False,
      target_types=('Apple',
                    'Egg'),  # categories of objects that the agent can pick
      receptacle_types=("SinkBasin", ),  # categories we allow "PutObject" on
      max_steps: int = None,
      low_level: bool = False,  # whether we are working with low-level only
      hl_pref_r=None,
      option: PnP_HL_Actions = None,
      seed: Optional[int] = None,
      render: bool = True):
    super().__init__()
    # Save config
    self.scene = scene
    self.max_steps = max_steps
    self.target_types = set(target_types)
    self.receptacle_types = set(receptacle_types)
    self._rng = random.Random(seed)

    h, w = 600, 600
    platform = None if render else CloudRendering
    self.need_render = render
    self.controller = Controller(
        width=w,
        height=h,
        scene=self.scene,
        gridSize=grid_size,
        snapToGrid=snap_to_grid,
        rotateStepDegrees=rotate_step_degrees,
        renderDepthImage=render_depth,
        renderInstanceSegmentation=render_instance_masks,
        visibilityDistance=visibilityDistance,
        platform=platform)
    self.controller.step(action="Initialize", gridSize=grid_size)
```

```
# Observation: dictionary-based state space.
self.observation_space = spaces.Dict({
    "apple_1_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),
    "apple_2_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),
    "egg_1_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),
    "egg_2_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),
    "stool_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),
    "sink_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),
    "agent_pos":
    spaces.Box(-3.0, 3.0, (2, ), dtype=np.float32),  # x and z pos
    "agent_rot":
    spaces.Box(0.0, 1.0, (4, ),
               dtype=np.float32),  # y rot (one-hot encoded)
    "apple_1_state":
    spaces.Discrete(3),  # 0 = on table, 1 = held, 2 = in sink
    "apple_2_state":
    spaces.Discrete(3),  # 0 = on table, 1 = held, 2 = in sink
    "egg_1_state":
    spaces.Discrete(3),  # 0 = on table, 1 = held, 2 = in sink
    "egg_2_state":
    spaces.Discrete(3),  # 0 = on table, 1 = held, 2 = in sink
})

# Whether we are working with the low-level only
self.low_level = low_level

# Adjust task/subtask horizons
if max_steps is not None:
  self.max_steps = max_steps
else:
  if self.low_level:
    self.max_steps = 100
  else:
    self.max_steps = 500

# Define action spaces
self.pnp_ll_actions = PnP_LL_Actions
self.pnp_hl_actions = PnP_HL_Actions
self.pnp_hl_actions_with_dummy = PnP_HL_Actions_With_Dummy

# Low level action space: iThor environment commands
self.ll_action_space = spaces.Discrete(len(self.pnp_ll_actions))
self.hl_action_space = spaces.Discrete(len(self.pnp_hl_actions))

# High level action space: Options (pick up/drop specific items)
# Option values
self.option = option

# Initialize environment
self._setup_env()
if self.low_level:
  if self.option is None:
    self.option = random.choice(list(self.pnp_hl_actions)).value
  self.action_space = self.ll_action_space
  self.reset(options={'option': self.option})
else:
```

```
            # Replace option with dummy value for high level training
            self.option = self.pnp_hl_actions_with_dummy.dummy.value
            self.action_space = self.hl_action_space
            self.reset()

        self.steps = 0

        # Set preferences
        self.pref_dict = pref_dict

        # Rewards initialization
        self.hl_pref_r = hl_pref_r

        self._per_step_reward = -0.01
        self._obj_drop_reward = 10.0
        self._obj_pick_reward = 10.0
        self._wrong_obj_pick_reward = -5.0
        self._dist_shaping_factor = -0.05
        self._ll_penalty = -1
        self._ll_radius = 1.5
        self._hl_diversity_reward = 5.0

        self.prev_option = self.pnp_hl_actions_with_dummy.dummy.value
        self.c_task_reward = 0
        self.c_pseudo_reward = 0
        self.c_gt_hl_pref = 0
        self.c_gt_ll_pref = 0

        # Used for determining successful placement into receptacle
        self._drop_success = False
        self._pick_apple_success = False
        self._pick_egg_success = False
```

Write a reward function for the following task:

The LLM receives one of the prompts below, chosen according to the reward function we want it to design. The prompt for each setting is identical except for (1) descriptions of relevant task spaces (e.g., the flat-reward prompt omits the options space description) and (2) the behavioral specification ("user preference") string.

High-Level

Task description:
The task objective is to pick up all apples and eggs on the dining table and place them in the sink. In the task reward, the agent gets a reward of +10 after it successfully picks up an object and places it in the sink, and a step cost of -0.1 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's option/subtask (referred to as self.pnp_hl_actions in the code) space consists of picking up and placing the two types of objects. Each option takes multiple action steps to complete. Taking a 'pick' option means that the agent will navigate to an object and pick it up. Taking a 'place' option means that the agent will navigate to the delivery location and place the object there. Note that the agent has to first go to the object to pick it up before placing the object.
User preference:
The agent should pick up an object type that's different from the previously placed object type, as long as there are objects of the other type on the table need to be picked.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. It can take up to 30 steps to reach an object and pick it up, or to reach the sink and drop it off. Make sure your reward scaling gives the preference for alternating objects much more weight than the negative step rewards, but still lower than the positive task reward. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables). You should also not write any other helper functions.

Low-Level

Task description:
The task objective is to pick up all apples and eggs on the dining table and place them in the sink. In the task reward, the agent gets a reward of +10 after it successfully picks up an object and places it in the sink, and a step cost of -0.1 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's option/subtask (referred to as self.pnp_hl_actions in the code) space consists of picking up and placing the two types of objects. Each option takes multiple action steps to complete. Taking a 'pick' option means that the agent will navigate to an object and pick it up. Taking a 'place' option means that the agent will navigate to the delivery location and place the object there. Note that the agent has to first go to the object to pick it up before placing the object. The agent's action (referred to as self.pnp_ll_actions in the code) space consists of the primitives for: moving forward, rotating left, rotating right, picking up the closest object, and placing a held object in receptacle. The agent can only perform the low level pick or place primitive only if the agent is close enough to an object or a receptacle.
User preference:
The agent should avoid the stool in the environment both when it is on its way to pick up an egg and place an egg down. More specifically, the agent should be penalized when it is within 1.5 meters of the stool. However, the agent does not need to avoid the stool when it is on its way to pick up an apple or place an apple down.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables). You should also not write any other helper functions.

Flat

Task description:
The task objective is to pick up all apples and eggs on the dining table and place them in the sink. In the task reward, the agent gets a reward of +10 after it successfully picks up an object and places it in the sink, and a step cost of -0.1 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's action (referred to as self.pnp_ll_actions in the code) space consists of the primitives for: moving forward, rotating left, rotating right, picking up the closest object, and placing a held object in receptacle. The agent can only perform the low level pick or place primitive only if the agent is close enough to an object or a receptacle.
User preference:
The agent should avoid the stool in the environment both when it is on its way to pick up an egg and place an egg down. More specifically, the agent should be penalized when it is within 1.5 meters of the stool. However, the agent does not need to avoid the stool when it is on its way to pick up an apple or place an apple down. In addition, the agent should pick up an object type that's different from the previously placed object type, as long as there are objects of the other type on the table need to be picked.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. It can take up to 30 steps to reach an object and pick it up, or to reach the sink and drop it off. Make sure your reward scaling gives the preference for alternating objects much more weight than the negative step rewards, but still lower than the positive task reward. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables.) You should also not write any other helper functions.

Prompt 4: User Prompt for Kitchen

The Python environment is

```
'''
Background:
1) Ingredients:
class Ingredients(Enum):
  empty = 0
  tomato = 1
  onion = 2
  lettuce = 3
```

```
2) Salad types:
class SoupType(Enum):
  no_soup = 0
  alice = 1
  bob = 2
  cathy = 3
  david = 4

3) All available options:
{'Chop Tomato': 0, 'Chop Lettuce': 1, 'Chop Onion': 2,
'Prepare David Ingredients': 3, 'Plate David Salad': 4}

4) All available actions:
{0: (0, -1),
 1: (0, 1),
 2: (1, 0),
 3: (-1, 0),
 4: (0, 0)}
If the agent is standing next to a counter, performing an action in the
direction of the counter interacts with the counter.
For example, if the agent is standing under a counter, performing action 0
(goes up) interacts with the counter above the agent.
'''

import gymnasium as gym


class OvercookedSimple(gym.Env):

  def get_plain_state(self, raw_info):
    '''
    The output of this function will be the input state in the generated reward
    function.

    The state is a dictionary that maps object names to their locations on the
    map.

    If the object 'obj' is at location (x, y), then state['obj'][y, x] == 1.
    Otherwise, state['obj'][y, x] == 0.
    '''
    num_rows = self.world_size[1]
    num_cols = self.world_size[0]
    state_dict = {}

    # Process Grid Squares Map
    GRIDSQUARES = [
        "Floor", "Counter", "Cutboard", "Bin", "Pot", "FreshTomatoTile",
        "FreshOnionTile", "FreshLettuceTile", "PlateTile"
    ]
    gridsquares_map = raw_info['gridsquare']
    for gridsquare_type in GRIDSQUARES:
      grid_map = gridsquares_map[gridsquare_type].T
      assert grid_map.shape == (num_rows, num_cols)
      state_dict[gridsquare_type] = grid_map

    # Process Object Map
    OBJECTS = ['FreshTomato', 'FreshLettuce', 'FreshOnion'] + [
        'ChoppingTomato', 'ChoppingOnion', 'ChoppingLettuce'
    ] + ['ChoppedTomato', 'ChoppedOnion', 'ChoppedLettuce'] + ['Plate']
    objects_map = raw_info['objects']
    for obj_type in OBJECTS:
      obj_map = objects_map[obj_type].T
```

```
        assert obj_map.shape == (num_rows, num_cols)
        state_dict[obj_type] = obj_map

    # Process Agent Map
    agent_map = raw_info['agent_map']['agent-1'].T
    assert agent_map.shape == (num_rows, num_cols)
    state_dict['agent'] = agent_map

    return state_dict
```

Write a reward function for the following task:

The LLM receives one of the prompts below, chosen according to the reward function we want it to design. The prompt for each setting is identical except for (1) descriptions of relevant task spaces (e.g., the flat-reward prompt omits the options space description) and (2) the behavioral specification ("user preference") string.

High-Level

Task description:
The task objective is to prepare one David's salad with three ingredients: onion, lettuce, and tomatoes. To make this salad, the agent needs to: a. Chop ingredients (onion, lettuce, and tomatoes). Only one ingredient of each type is needed to complete the salad. b. Combined chopped ingredients. c. Plate the salad. The task reward already encodes the task objective. In the task rewrd, the agent receives a reward of +1 when it completes a salad, and a step cost of -0.01 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's option (subtask) space consists of macro cooking actions, such as 'Chop Onion' and 'Plate David Salad'. Each option takes multiple action steps to complete.
User preference:
The agent should chop an onion after it chops a tomato, and the agent should chop a lettuce after it chops an onion. If the ingredients are already chopped or a combined salad already exists, the agent should not chop more ingredients. The mixing order of the ingredients does not matter.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables.) You should also not write any other helper functions.

Flat

Task description:
The task objective is to prepare one David's salad with three ingredients: onion, lettuce, and tomatoes. To make this salad, the agent needs to: a. Chop ingredients (onion, lettuce, and tomatoes). Only one ingredient of each type is needed to complete the salad. b. Combined chopped ingredients. c. Plate the salad. The task reward already encodes the task objective. In the task reward, the agent receives a reward of +1 when it completes a salad, and a step cost of -0.01 for each time step taken. The reward function you write does not need to encode the task objective.
Relevant task spaces:
The agent's action (low-level action) space consists of moving in the four cardinal directions (up, down, left, right) and idle. When an agent is standing next to a counter, performing an action in the direction of the counter interacts with the counter.
User preference:
The agent should chop an onion after it chops a tomato, and the agent should chop a lettuce after it chops an onion. If the ingredients are already chopped or a combined salad already exists, the agent should not chop more ingredients. The mixing order of the ingredients does not matter.
Additional info:
You need to write a reward function to encode this user preference. The preference function you write will be used together with the task reward to train the agent. Please make sure NOT to make the reward function stateful (i.e. you should not use function attributes or global variables.) You should also not write any other helper functions.

D.5 Example LLM-Generated Rewards

Here, we provide examples of high-level, low-level, and flat rewards generated using the prompts from the previous section.

D.5.1 Rescue World

Example Flat Preference Reward for Rescue World

```python
def get_flat_sa_pref_gpt(state: Dict, action: int) -> Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    action: the (low-level) action that the agent is about to perform in the current state.
    '''
    current_pos = state['pos']
    current_map = state['map']
    current_holding = state['holding']
    recently_delivered = -1  # To be inferred from context; assuming function is aware of last delivery
    reward = 0.0
    reward_components = {
        'consistency_bonus': 0.0,
        'danger_zone_penalty': 0.0
    }

    # Reward consistency in type picking
    if action == rw4t_utils.RW4T_LL_Actions.pick.value:
        # Get the object type at current position
        object_type = current_map[current_pos[1], current_pos[0]]

        # Check if the object type is the same as the recently delivered one
        if object_type == recently_delivered:
            reward += 1.0  # Encourage same type pick
            reward_components['consistency_bonus'] += 1.0

    # Penalize entering yellow zones when holding an object
    if current_holding != rw4t_utils.Holding_Obj.empty.value:
        # Calculate the next position based on the action taken
        next_pos = current_pos.copy()
        if action == rw4t_utils.RW4T_LL_Actions.go_up.value:
            next_pos[1] -= 1
        elif action == rw4t_utils.RW4T_LL_Actions.go_down.value:
            next_pos[1] += 1
        elif action == rw4t_utils.RW4T_LL_Actions.go_left.value:
            next_pos[0] -= 1
        elif action == rw4t_utils.RW4T_LL_Actions.go_right.value:
            next_pos[0] += 1

        # Ensure the next position is within bounds
        if (0 <= next_pos[0] < current_map.shape[1]) and (0 <= next_pos[1] < current_map.shape[0]):
            next_area_type = current_map[next_pos[1], next_pos[0]]
            if next_area_type == rw4t_utils.RW4T_State.yellow_zone.value:
                reward -= 1.0  # Penalize for entering a danger zone
                reward_components['danger_zone_penalty'] -= 1.0

    return reward, reward_components
```

Example Low-Level Preference Reward for Rescue World

```python
def get_low_level_pref_gpt(state: Dict, option: int, action: int) -> Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    option: the option (subtask) selected by the agent in the current state.
    action: the action that the agent is about to perform in the current state.
    '''

    reward = 0.0
    reward_components = {}

    # Extract necessary information from the state
    agent_pos = state['pos']
    map_state = state['map']
    current_cell = map_state[agent_pos[1], agent_pos[0]]

    # Check if the agent is in a danger zone (yellow zone)
    is_in_yellow_zone = current_cell == rw4t_utils.RW4T_State.yellow_zone.value

    # Determine if the current option is a delivery option
    is_delivery_option = option in [
        rw4t_utils.RW4T_HL_Actions_EZ.deliver_circle.value,
        rw4t_utils.RW4T_HL_Actions_EZ.deliver_square.value
    ]

    # Apply a penalty if the agent is delivering and currently in a yellow zone
    if is_delivery_option and is_in_yellow_zone:
        danger_zone_penalty = -5.0  # User defined penalty
        reward += danger_zone_penalty
        reward_components['danger_zone_penalty'] = danger_zone_penalty
    else:
        reward_components['danger_zone_penalty'] = 0.0

    # No additional reward for being outside danger zones
    return reward, reward_components
```

Example High-level Preference Reward for Rescue World

```
def get_high_level_pref_gpt(state: Dict, prev_option: int, option: int) -> Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    prev_option: the last option (subtask) executed by the agent to reach the current state.
    option: the option (subtask) the agent is about to perform in the current state.
    '''

    reward = 0.0
    reward_components = {}

    # Determine the type of the previous and current option
    prev_pick_type = None
    curr_pick_type = None

    if prev_option == rw4t_utils.RW4T_HL_Actions_EZ.deliver_circle.value:
        prev_pick_type = rw4t_utils.RW4T_State.circle.value
    elif prev_option == rw4t_utils.RW4T_HL_Actions_EZ.deliver_square.value:
        prev_pick_type = rw4t_utils.RW4T_State.square.value

    if option == rw4t_utils.RW4T_HL_Actions_EZ.go_to_circle.value:
        curr_pick_type = rw4t_utils.RW4T_State.circle.value
    elif option == rw4t_utils.RW4T_HL_Actions_EZ.go_to_square.value:
        curr_pick_type = rw4t_utils.RW4T_State.square.value

    # Count remaining objects of each type on the map
    circle_count = (state['map'] == rw4t_utils.RW4T_State.circle.value).sum()
    square_count = (state['map'] == rw4t_utils.RW4T_State.square.value).sum()
    # Add preference reward based on the user preference
    if prev_pick_type is not None:
        if curr_pick_type == prev_pick_type:
            if (curr_pick_type == rw4t_utils.RW4T_State.circle.value and circle_count > 0) or \
               (curr_pick_type == rw4t_utils.RW4T_State.square.value and square_count > 0):
                reward += 5.0  # Reward for picking the same type if available
                reward_components['same_type_pick_bonus'] = 5.0
            else:
                reward_components['same_type_pick_bonus'] = 0.0
        else:
            reward_components['same_type_pick_bonus'] = 0.0

    return reward, reward_components
```

D.5.2 iTHOR

Example Flat Preference Reward for iTHOR

```
def get_flat_sa_pref_gpt(state: Dict, action: int) -> Tuple[float, Dict[str, float]]:
    """
    Computes the user preference reward based on agent's proximity to the stool
    and preferences for alternating object types during the pickup/drop process.

    state: the current state of the environment.
    action: the current action the agent is performing.
    """
    reward = 0.0
    reward_components = {
        'stool_penalty': 0.0,
        'alternating_bonus': 0.0
    }

    # Agent and stool position
    agent_pos = state['agent_pos']
    stool_pos = state['stool_pos']

    # Calculate distance to stool
    distance_to_stool = np.linalg.norm(np.array(agent_pos) - np.array(stool_pos))

    # If the agent is close to the stool while dealing with eggs, apply penalty
    dealing_with_eggs = (state['egg_1_state'] in [0, 1] or state['egg_2_state'] in [0, 1])
    if dealing_with_eggs and distance_to_stool < 1.5:
        reward -= 2.0  # Penalize for being too close to the stool
        reward_components['stool_penalty'] = -2.0

    # Check if the action is a pickup or drop intention
    if action in [PnP_LL_Actions.index("PickupNearestTarget"), PnP_LL_Actions.index("PutHeldOnReceptacle")]:
        # Determine last placed object type to encourage alternating pick-up
        last_picked = 'apple' if (state['apple_1_state'] == 2 or state['apple_2_state'] == 2) else 'egg'
        available_apples = (state['apple_1_state'] == 0 or state['apple_2_state'] == 0)
        available_eggs = (state['egg_1_state'] == 0 or state['egg_2_state'] == 0)

        if last_picked == 'apple' and available_eggs:
            reward += 5.0  # Encourage picking eggs if last placed was apple
            reward_components['alternating_bonus'] = 5.0
        elif last_picked == 'egg' and available_apples:
            reward += 5.0  # Encourage picking apples if last placed was egg
            reward_components['alternating_bonus'] = 5.0

    return reward, reward_components
```

Example Low-Level Preference Reward for iTHOR

```python
def get_low_level_pref_gpt(state: Dict, option: int, action: int) -> Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    option: the option (subtask) selected by the agent in the current state.
    action: the action that the agent is about to perform in the current state.
    '''

    # Define rewards and thresholds
    stool_penalty = -2.0  # Penalty for being too close to the stool
    stool_avoidance_radius = 1.5  # Distance within which to penalize for being too close to the stool

    # Initialize preference reward and its components
    reward = 0.0
    reward_components = {"stool_avoidance_penalty": 0.0}

    # Get stool and agent positions
    stool_pos = np.array(state["stool_pos"])
    agent_pos = np.array(state["agent_pos"])

    # Calculate distance between agent and stool
    dist_to_stool = np.linalg.norm(agent_pos - stool_pos)

    # Determine if the current option involves interacting with eggs
    interacting_with_egg = option in {PnP_HL_Actions.pick_egg.value, PnP_HL_Actions.drop_egg.value}

    # Apply penalty if the agent is too close to the stool and dealing with eggs
    if interacting_with_egg and dist_to_stool < stool_avoidance_radius:
        reward += stool_penalty
        reward_components["stool_avoidance_penalty"] = stool_penalty

    return reward, reward_components
```

Example High-level Preference Reward for iTHOR

```python
def get_high_level_pref_gpt(state: Dict, prev_option: int, option: int) -> Tuple[float, Dict[str, float]]:
    # Initialize reward and components
    preference_reward = 0.0
    reward_components = {
        "alternation_bonus": 0.0,
        "not_alternate_penalty": 0.0
    }

    # Determine the type of object just placed and the type to pick
    if prev_option == PnP_HL_Actions.drop_apple.value:
        last_placed_type = "apple"
    elif prev_option == PnP_HL_Actions.drop_egg.value:
        last_placed_type = "egg"
    else:
        last_placed_type = None

    if option == PnP_HL_Actions.pick_apple.value:
        current_pick_type = "apple"
    elif option == PnP_HL_Actions.pick_egg.value:
        current_pick_type = "egg"
    else:
        current_pick_type = None

    # Check items state to determine if alternation is possible
    remaining_apples = state["apple_1_state"] == 0 or state["apple_2_state"] == 0
    remaining_eggs = state["egg_1_state"] == 0 or state["egg_2_state"] == 0
    # Encourage alternating picking
    if last_placed_type and current_pick_type:
        if last_placed_type != current_pick_type:
            # Give a bonus for alternating types
            preference_reward += 8.0  # Scaled to be significant but less than the task completion reward
            reward_components["alternation_bonus"] = 8.0
        elif (current_pick_type == "apple" and remaining_eggs) or (current_pick_type == "egg" and remaining_apples):
            # Apply a penalty if not alternating but possible
            preference_reward += -2.0
            reward_components["not_alternate_penalty"] = -2.0

    return preference_reward, reward_components
```

D.5.3 Kitchen

Example Flat Preference Reward for Kitchen

```python
def get_flat_sa_pref_gpt(state: Dict, action: int) -> Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    action: the (low-level) action that the agent is about to perform in the current state.
    '''
    # Define a map for action directions
    action_effect_map = {
        0: (0, -1),  # move up
        1: (0, 1),   # move down
        2: (1, 0),   # move right
        3: (-1, 0),  # move left
        4: (0, 0)    # idle
    }

    reward = 0.0
    reward_components = {
        'tomato_chopped': 0.0,
        'onion_chopped': 0.0,
        'lettuce_chopped': 0.0
    }

    # Check positions of chopped ingredients
    chopped_tomato_exists = state['ChoppedTomato'].any()
    chopped_onion_exists = state['ChoppedOnion'].any()
    chopped_lettuce_exists = state['ChoppedLettuce'].any()

    # Calculate agent's new position based on the action
    agent_pos = state['agent'].argmax()
    agent_y, agent_x = divmod(agent_pos, state['agent'].shape[1])
    dy, dx = action_effect_map[action]
    new_pos = (agent_y + dy, agent_x + dx)

    # Check if the agent is chopping at the new position
    is_near_cutboard = state['Cutboard'][new_pos] == 1
    chopping_tomato_active = is_near_cutboard and state['ChoppingTomato'][new_pos] == 1
    chopping_onion_active = is_near_cutboard and state['ChoppingOnion'][new_pos] == 1
    chopping_lettuce_active = is_near_cutboard and state['ChoppingLettuce'][new_pos] == 1

    # Add preference rewards
    if chopping_tomato_active and not chopped_tomato_exists:
        reward_components['tomato_chopped'] += 0.1  # Encourage chopping tomato
    if chopped_tomato_exists and chopping_onion_active and not chopped_onion_exists:
        reward_components['onion_chopped'] += 0.2  # Encourage chopping onion after chopping tomato
    if chopped_onion_exists and chopping_lettuce_active and not chopped_lettuce_exists:
        reward_components['lettuce_chopped'] += 0.3  # Encourage chopping lettuce after chopping onion

    reward = sum(reward_components.values())

    return reward, reward_components
```

Example High-level Preference Reward for Kitchen

```python
def get_high_level_pref_gpt(state: Dict, prev_option: int, option: int) -> Tuple[float, Dict[str, float]]:
    '''
    state: the current state of the environment.
    prev_option: the last option (subtask) executed by the agent to reach the current state.
    option: the option (subtask) the agent is about to perform in the current state.
    '''

    reward = 0.0
    reward_components = {
        "onion_after_tomato": 0.0,
        "lettuce_after_onion": 0.0,
        "avoid_extra_chop": 0.0,
    }

    # Define option indices for ease of reference
    CHOP_TOMATO = 0
    CHOP_LETTUCE = 1
    CHOP_ONION = 2

    # Check for chopped states
    tomato_chopped = state['ChoppedTomato'].any()
    onion_chopped = state['ChoppedOnion'].any()
    lettuce_chopped = state['ChoppedLettuce'].any()

    # User preferences
    if prev_option == CHOP_TOMATO and option == CHOP_ONION:
        reward += 0.5
        reward_components["onion_after_tomato"] = 0.5

    if prev_option == CHOP_ONION and option == CHOP_LETTUCE:
        reward += 0.5
        reward_components["lettuce_after_onion"] = 0.5

    # Avoid chopping ingredients again if they are already chopped
    if (option == CHOP_TOMATO and tomato_chopped) or \
       (option == CHOP_ONION and onion_chopped) or \
       (option == CHOP_LETTUCE and lettuce_chopped):
        reward -= 1.0
        reward_components["avoid_extra_chop"] = -1.0

    return reward, reward_components
```

E User Study Details

E.1 Experiment Protocol

We conducted a user study to evaluate how well agent policies trained with hierarchical rewards (experimental group) generated by language models are perceived to align with given behavioral specifications compared to those trained with flat rewards (control group), also generated using language models. Each participant was randomly assigned to either the Rescue World or Kitchen domain.

(1) **Consent and Study Overview.** Participants were first presented with a detailed overview of the study, including its purpose, procedures, and any potential risks. An IRB-approved consent form was provided, and participants were required to give informed consent before proceeding.

(2) **Demographic Questionnaire.** After providing consent, participants completed a brief demographic questionnaire, where we asked for their age and sex.
(3) **Domain Introduction.** Participants were introduced to the assigned domain through textual descriptions accompanied by screenshots. This step was designed to ensure that they had sufficient context to understand the environment and the tasks performed by the agent.
(4) **Presentation of Behavioral Specifications and Attention Checks.** Next, participants were shown the behavioral specifications the agent was expected to follow. To ensure they carefully read these specifications, we included attention check questions. For example, in the Rescue World domain, where the agent must handle two object types (food and medical kits) and may encounter avoid danger zones (marked by yellow grids), part of the *safety* specification states that the robot should "avoid yellow danger zones when it is delivering an object". We asked participants, whether according to the specifications, it would be considered safe for the robot to "go through danger zones while delivering food". While participants were not required to answer these questions correctly to proceed, we filtered out all responses with incorrect answers during data analysis to ensure data quality.
(5) **Video Evaluation.** To ensure consistent evaluation, only policies that successfully completed the task were shown. Additionally, to control for variability, all videos shown to a participant were drawn from policies trained with the same random seed, although they could originate from different reward candidates. In the *Rescue World* domain, each participant viewed 6 videos: 3 showing policies trained with flat rewards $\tilde{r}_{flat}$ and 3 showing policies trained with hierarchical rewards $(\tilde{r}_H, \tilde{r}_L)$. In the *Kitchen* domain, participants viewed 4 videos: 2 per reward method. Fewer videos were shown in this domain because, in some cases, only 2 out of 8 flat reward candidates produced policies capable of successfully completing the task. Questions for each video appeared on the same page, and participants were allowed to replay the videos as many times as they wished. After each video, participants first answered a multiple-choice question to verify they had watched the video (e.g., "What order did the robot deliver the objects in?"). They were then asked to rate how well the agent's behavior aligned with the specified behaviors using a 5-point scale, with 1 indicating "least aligned" and 5 indicating "most aligned". Participants could optionally provide any comments explaining their ratings.
(6) **Final Feedback and Compensation.** At the end of the study, participants were invited to leave any additional feedback about their experience. On average, participants took 14.2 minutes to complete the study (SD = 8.7 minutes). All participants who completed the full study were compensated $3 for their time.

E.2 Participants

We conducted the user study on Prolific, recruiting a total of 40 participants from the United States. After applying attention check filters, we obtained usable data from 30 participants, with 15 participants assigned to each domain. Among these 30 participants, the youngest was 20, the oldest was 67, and the median age was 35. The sex distribution was fairly balanced, with 14 female participants, 15 male participants, and 1 participant identifying as "non-binary / third gender."

E.3 Data Analysis

We filtered out data from participants who incorrectly answered the attention check questions about the behavioral specifications. For participants who passed the attention checks, if they answered a video content question incorrectly, we excluded their ratings for that specific video but retained their responses for other videos.

To generate Fig. 7, we computed the average rating each participant assigned to *Flat* and *Hier* policies, and then aggregated these per-participant means to report group-level comparisons. Since each participant evaluated both *Flat* and *Hier* policies trained with the same random seed, we used the Wilcoxon Signed-Rank test to assess the statistical significance of the observed rating differences [23]. All reported results are based on the filtered dataset, with the final sample sizes specified in the Participants section.

F Code Availability and Release

An anonymized folder containing the source code developed for this project is available at https://anonymous.4open.science/r/hierarchical_reward_design-88B3/.