Compositional Policy Optimization with Language Models

Wensen Mao
Rutgers University
wm300@cs.rutgers.edu

Yuanlin Duan
Rutgers University
yuanlin.duan@rutgers.edu

He Zhu
Rutgers University
hz375@cs.rutgers.edu

Abstract

Large language models (LLMs) possess remarkable ability to understand natural language descriptions of complex robotics environments. Earlier studies have shown that LLM agents can use a predefined set of skills for robot planning in long-horizon tasks. However, the requirement of prior knowledge of the skill set required for a given task constrains its applicability and flexibility. We present a novel approach L2S (short for Language2Subtasks) to leverage the generalization capabilities of LLMs to decompose the description of a complex task in natural language into definitions of reusable subtasks. Each subtask is defined by an LLM-generated dense reward function and a termination condition, which in turn lead to effective subtask training and chaining. However, LLMs lack detailed insight into the specific low-level control intricacies of the environment, such as threshold parameters within the generated reward and termination functions. To address this uncertainty, L2S (1) enables LLM reflection feedback loop to improve task decomposition and code generation, and (2) trains parameter-conditioned subtask policies that perform well in a broad spectrum of parameter values. As the impact of these parameters for one subtask on the overall task becomes apparent only when its following subtasks are trained, L2S selects the most suitable parameter value during the training of the subsequent subtasks to effectively mitigate the risk associated with incorrect parameter choices. During training, L2S autonomously accumulates a subtask library from continuously presented tasks and their descriptions, using guidance from the LLM agent to effectively apply this subtask library in tackling novel tasks. Our experimental results show that L2S is capable of generating reusable subtasks to solve a wide range of robot manipulation tasks.

1 Introduction

In recent years, the integration of language models with robotics has opened up new avenues for advancing autonomous learning in robotic systems. Large Language models (LLMs) possess the remarkable ability to understand complex tasks and environments. Leveraging this capability, researchers have explored the use of language models in various aspects of robotics, ranging from task planning and navigation to manipulation and control. Previous work in this domain has primarily focused on leveraging language models for robot planning, where a predefined set of skills is provided to the model. However, this approach has limitations, as it assumes prior knowledge of the skill set required for the given task, thus constraining its applicability and flexibility.

Automatic skill acquisition has long been studied in the context of hierarchical reinforcement learning [Barto and Mahadevan, 2003] in the form of temporally extended actions [Sutton et al., 1999]. Despite the proven effectiveness of skills in expediting learning [McGovern and Sutton, 1998], a fundamental question remains: how can agents autonomously develop valuable skills through interaction with their environment? There has been a significant body of work aimed at discovering skills. For example, Option-Critic [Bacon et al., 2017] learns skills by optimizing the skill policies as well

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

as their termination functions in a gradient-based manner, assuming all the skills can be applied everywhere. However, it is known to be prone to inefficient task decomposition, such as learning a sub-policy that terminates at every time step or discovering one efficient sub-policy that executes throughout the entire episode. Vezhnevets et al. [2017], Nachum et al. [2018], Levy et al. [2019] address this issue by automatically decomposing a complex task into subtasks and solving them by optimizing the subtask objectives. These methods excel in learning multiple levels of policies in sparse reward tasks. However, the low-level skills learned are tied to a specific environment and it is unclear whether they are adaptable to new tasks. Skill chaining (DSC)[Konidaris and Barto, 2009, Bagaria and Konidaris, 2020] involves a sequential discovery and chaining of skills, starting from the end goal state and progressing backward to the initial state. However, as the agent generates new skills using the initial states of the preceding skill on the skill chain as their goal states, this poses challenges in robot manipulation tasks.

We present a novel approach L2S (short for "Language to Subtasks") for subtask discovery in robot learning by leveraging large language models to overcome the limitations of prior methods. We aim to empower robotic systems to autonomously discover and adapt subtasks to a wide range of tasks. We assume that robotic tasks can generally be decomposed and solved through a sequence of subtasks executed in chronological order. L2S harnesses the generalization capability of large language models (LLMs) to decompose the natural language task description of a complex task to definitions of reusable subtasks. Each subtask is defined by an LLM-generated *dense* reward function and a termination condition, which in turn lead to effective subtask policy training and chaining for task execution. For example, consider the "turn faucet left" task depicted in Fig. 1. The LLM agent can break down this task into two subtasks: (1) positioning the robot's end effector near the right side of the faucet π_{o_1} and (2) rotating the faucet handle to the left π_{o_2}. Chaining these two subtasks together successfully solves the task.

L2S excels in sequential task learning by autonomously building a library of parameterized subtasks (explained below) as it encounters tasks during training. This accumulated subtask library can then be reused to tackle new tasks, guided by the LLM agent. For example, consider a scenario where the agent is presented with the task of "turn faucet right" after it has already been trained on "turn faucet left". The LLM agent identifies that the first subtask π_{o_1} in "turn faucet left" can be tuned to position the end effector on the left side of the faucet handle (by adjusting its parameters). Thus, L2S only needs to train a new subtask π'_{o_2} to rotate the faucet handle right. By reusing existing subtask in this manner, L2S significantly reduces the computational burden associated with learning new tasks from scratch, enabling more efficient task solving over time.

The main challenge faced by L2S is that while LLMs can outline the overall structure of subtasks necessary to tackle a task, they lack detailed insight into the specific low-level control intricacies of the environment. To address the uncertainty surrounding the parameters used by the LLM agent, L2S trains parameter-conditioned subtask policies, denoted as $\pi_o(a|s;\ \texttt{params})$, where the parameters `params` are akin to "goals" in goal-conditioned reinforcement learning. As the impact of these parameters on the overall task becomes apparent only when subsequent subtasks are trained, L2S adopts a strategy of training a subtask policy that performs well across a broad spectrum of parameter values and selects the most suitable parameter value during the training of subsequent subtasks. In this way, L2S effectively mitigates the risk associated with potentially incorrect parameter choices. The parameter-conditioned subtask policies in L2S also facilitate seamless subtask reuse. For example, the parameter in the subtask for putting robot's end effector on the right side of the faucet in "Turn faucet left" task can be adjusted to position the robot's end effector on the other side of the faucet for the "Turn faucet right" task.

The core novelty of L2S is converting an LLM-sketched subtask chain into an executable controller by jointly learning its reusable parameter-conditioned subtask policies and their handoffs, i.e., the reward and termination parameters that determine the states from which the next subtask starts. Each handoff is optimized according to the performance of its successor subtask. Compared to state-of-the-art LLM-guided reward generation methods such as Text2Reward [Xie et al., 2023] and Eureka [Ma et al., 2023], which generate dense reward functions to train single, monolithic policies for each robotic task, L2S instead creates reusable, parameterized subtasks for *sequential task learning*. These subtasks effectively generalize to new tasks through parameterization. While previous work, such as Ahn et al. [2022], has explored decomposing complex tasks into skills using LLMs' semantic knowledge, it relies on manually engineered skill libraries, whereas L2S autonomously learns such a library with parameterization to enable efficient generalization. Our experimental evaluations on

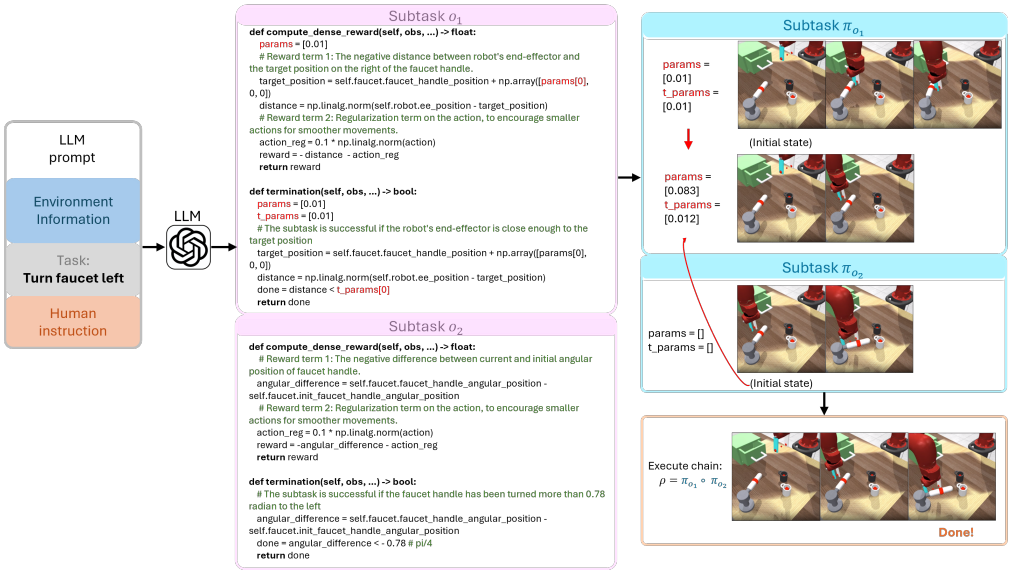

Figure 1: An example “Turn faucet left" to explain the workflow of L2S. The training of subtask π_{o_2} optimizes the policy and termination parameter of subtask π_{o_1}.

a suite of robotics manipulation tasks show that L2S not only solves continuously presented tasks much faster but also achieves higher success rates compared to state-of-the-art methods.

2 Problem Definition

Sequential decision-making problems can be formalized as Markov Decision Processes (MDPs). An MDP is defined by a tuple $e = \langle S, A, T, R, \gamma, \eta \rangle$, where S represents the state space, A represents the action space, $T : S \times A \times S \to [0, 1]$ denotes the transition function, $R : S \times A \to \mathbb{R}$ denotes the reward function, $0 < \gamma < 1$ is the discount factor, and $s_0 \sim \eta(\cdot)$ defines the initial states. At each time step t, the agent selects an action $a_t \in A$ in state $s_t \in S$, receives a reward $r_t = R(s_t, a_t) \in \mathbb{R}$, and transitions to another state s_{t+1} with a probability determined by T. We assume **sparse reward** functions that provide signals only upon task success (1.0) or failure (0.0). The primary objective is to learn a policy $\pi : S \to A$ for e that maximizes the expected return, defined as the discounted sum of rewards: $\max_{\pi \in \Pi} \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right]$, where $a_t = \pi(s_t)$.

Subtasks. A significant challenge for reinforcement learning (RL) algorithms lies in learning and planning over long horizons, particularly in scenarios where rewards are sparse. The options framework, proposed by Sutton et al. [1999], offers a formalism for temporal abstraction, which aids in both exploration and credit assignment. The central concept is to decompose the overarching problem that the agent seeks to solve into subtasks, each typically characterized by its own reward function and capable of being accomplished by a distinct subtask. Our method L2S is inspired by the options framework and we define subtasks similar to options in the options framework. A subtask o consists of (a) its termination condition, $\beta_o(s)$, which determines whether subtask execution must terminate in state s and (b) its closed-loop subtask policy, $\pi_o(s)$, which maps state s to a low level action $a \in A$.

Subtask Chaining for Single-Task Learning. Given a *single* task MDP e, and its task description $\mathcal{L}_e$ in natural language, L2S constructs a chain of subtasks [Konidaris and Barto, 2009, Bagaria and Konidaris, 2020] such that successful execution of each subtask in the chain allows the agent to execute another subtask. A task description $\mathcal{L}_e$ refers to a language command describing the desired goal for the agent, like “turn faucet left". The inductive bias of creating sequentially executable subtasks guarantees that as long as the agent successfully executes each subtask in its chain, it can solve the original task in e. Intuitively, subtask chaining amounts to learning subtasks such that the termination condition β_{o_i} of a subtask o_i induces an initiation condition of the subtask that follows it in the subtask chain. We formally define subtask chaining as follows. A subtask chain

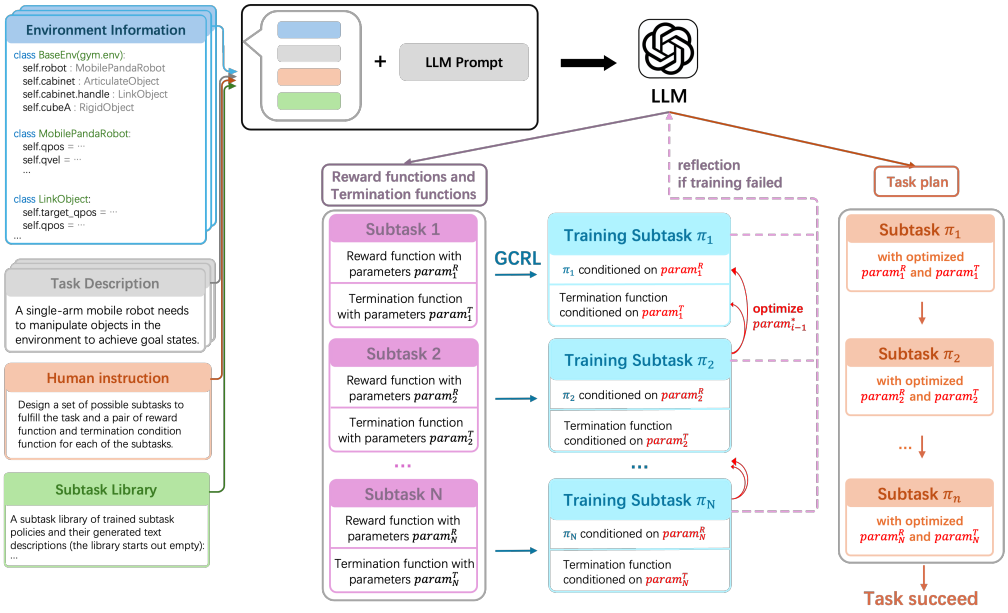

Figure 2: The L2S overall framework.

$\rho = o_0 \circ o_1 \circ \ldots \circ o_k$ defines a *controller* $\pi_\rho = (\pi_{o_0}, \beta_{o_0}) \circ (\pi_{o_1}, \beta_{o_1}) \circ \ldots \circ (\pi_{o_k}, \beta_{o_k})$ that navigates from an environment initial state of an MDP e to a state where β_{o_k} (the termination condition of o_k) holds. In particular, π_ρ executes π_{o_j} (starting from $j = 0$) until reaching β_{o_j}, after which it increments $j \leftarrow j + 1$ (unless $j = k$). Note that π_ρ is stateful since it internally keeps track of the index j of the current subtask policy. We denote by $\rho_i = o_0 \circ o_1 \circ \ldots \circ o_{i-1}$ the prefix of ρ up to and including subtask o_{i-1}.

Subtask Library Construction in Sequential Task Learning. The main objective of L2S is to efficiently tackle a sequence of related tasks by autonomously building a subtask library from ongoing tasks and reusing it for future tasks. Formally, given a *sequence* of tasks, where each task is represented as an MDP e and accompanied by a description $\mathcal{L}_e$, L2S builds a subtask library $\mathcal{O} \equiv \{o, L_o\}$ tailored for solving these tasks. Each subtask o within $\mathcal{O}$ is associated with a descriptive text L_o. ***The subtask library $\mathcal{O}$ starts out empty***. As L2S encounters new tasks $(e, \mathcal{L}_e)$ in the sequence, it progressively adds new subtasks in the subtask chains for solving these tasks to $\mathcal{O}$, while also developing plans that make use of the the existing subtasks in $\mathcal{O}$ whenever possible.

3 Language to Subtasks

The primary goal of L2S is to utilize LLMs to automatically build subtask libraries $\mathcal{O}$ for sequential task learning. Given the textual description of a new task, L2S uses LLMs to generate code that defines both the reward function and termination condition for each new subtask in the subtask chain for solving the task (Sec 3.1), facilitating the learning of the subtask's policy. An LLM reflection feedback loop is introduced to refine improper task decomposition and reward function generation (Sec 3.4). These learned subtasks, along with LLM-generated descriptions, are subsequently added to $\mathcal{O}$, enabling the LLMs to effectively reuse them when building subtask chains for future tasks.

3.1 Prompt Construction

For a task MDP e and its natural language task description $\mathcal{L}_e$, L2S prompts an LLM agent with an abstraction of e and $\mathcal{L}_e$ to generate a subtask chain for solving the task. The environment abstraction is needed by the LLM agent to ground reward generation for understanding how object states are represented, including robot and object configurations.We adopt a compact Pythonic representation, similar to Xie et al. [2023], as illustrated in Fig. 2. This approach offers a higher level of abstraction compared to listing all environment-specific information in a table or list format. The LLM agent is instructed to generate a skill chain for e as $\rho = o_0 \circ o_1 \circ \ldots \circ o_k$, and the reward function $\mathcal{R}_{o_i}[\phi_i](s, a)$ and termination condition $\beta_{o_i}[\varphi_i](s)$ (as python programs) for each skill o_i in ρ, where ϕ_i and φ_i are

the parameters within the subtask reward and termination functions for o_i respectively. Additionally, L2S asks the LLM agent to generate a description $\mathcal{L}_o$ for each subtask o.

For continuously presented tasks, as discussed in Sec. 2, L2S maintains a subtask library $\mathcal{O} \equiv \{(o, \mathcal{L}_o)\}$ where each subtask o is accompanied with its text description $\mathcal{L}_o$ (generated by the LLM agent). L2S starts with no predefined subtasks, meaning **the subtask library $\mathcal{O}$ is initially empty**. Subtasks within the subtask chain devised for one task are added to $\mathcal{O}$ for reusing them when building subtask chains for future tasks. Thus, it is possible part of a generated subtask chain $\rho = o_0 \circ o_1 \circ \ldots$ reuses subtasks in $\mathcal{O}$. To achieve this, L2S prompts the LLM agent with the text description $\mathcal{L}_o$ of each subtask o in $\mathcal{O}$, along with a command instructing the LLM agent to select and reuse existing subtasks whenever possible.

Although LLMs have good understanding of high-level task structures, we have found that they are not yet reliable enough to generate correct rewards and termination conditions in a zero-shot manner for complex tasks. To handle this, we also prompt LLMs with few-shot examples as Xie et al. [2023]. Detailed prompt examples can be found in the Appendix H.

3.2 Subtask Chain Training

During training, L2S iteratively processes (learning and optimizing, if necessary) subtasks in a subtask chain ρ starting from the initial subtask o_0, continuing until it processes the final subtask on ρ. It maintains the property that for every subtask o_i it processes, it has already trained policies for all subtasks preceding o_i. The key challenge with this approach is that, although LLMs can define the overarching structure of a subtask chain for a given task, they often lack precise knowledge of the low-level control details within the environment. As a result, the parameters used in the generated reward and termination functions tend to be inaccurate, reflecting inherent uncertainties (as illustrated in the turn faucet example in Sec. 1 and Fig. 1).

An important aspect of L2S is that each subtask policy is parameter-conditioned (similar in concept to goal-conditioned reinforcement learning), denoted as $\pi_{o_i}(a|s;\ \phi_i)$, where ϕ_i represents the parameters in the reward function $\mathcal{R}_{o_i}$ for o_i. The training objective is for the policy $\pi_{o_i}(a|s;\phi_i)$ to maximize the expected rewards over a broad range of parameter values $\phi_i \sim \tilde{q}_{\phi_i}$, ensuring robust performance across varying conditions. The parameter distribution $\tilde{q}_{\phi_i}$ for ϕ_i is configured by the user. For example, one can set $\tilde{q}_{\phi_i}$ as a Gaussian distribution $N(v_{\phi_i}, \sigma)$, where v_{ϕ_i} is the mean centered at the LLM agent's inferred parameter values for ϕ_i, and σ is the user-defined variance. The execution of π_{o_i} is also influenced by the initial states of o_i, which are determined by both the preceding subtask policies $\pi_{o_{0:i-1}}(\phi_{0:i-1}) = \{\pi_{o_k}(a|s;\ \phi_k)$, for o_k in $\rho_{i-1}\}$ and the termination condition $\beta_{o_{0:i-1}}(\varphi_{0:i-1}) = \{\beta_{o_k}(\varphi_k)$, for o_k in $\rho_{i-1}\}$ of its preceding subtask chain ρ_{i-1} . Thus, the training for $\pi_{o_i}(a|s\ \phi_i)$ also needs to optimize the parameters associated with ρ_{i-1}, which involves finding the correct policy parameters for $\phi_{0:i-1}$ and properly setting its termination condition parameters $\varphi_{0:i-1}$, where η_{o_i} is the initial state distribution of o_i:

$$\max_{\phi_{0:i-1}, \varphi_{0:i-1}, \pi_{o_i}} \mathbb{E}_{s_0 \sim \eta_{o_i}[\phi_{0:i-1}, \varphi_{0:i-1}], \phi_i \sim \tilde{q}_{\phi_i}, \tau \sim \pi_{o_i}(a_t|s_t;\phi_i)} \left[\sum_{t=0}^{T} \gamma^t \mathcal{R}_{o_i}[\phi_i](s_t, a_t) \right] \tag{1}$$

A key choice L2S makes is what initial state distribution η_{o_i} to choose to train the subtask policy π_{o_i}. Consider a prefix of a subtask chain $\rho_k = o_0 \circ o_1 \circ \ldots \circ o_{k-1}$, where all policies for the subtasks π_{o_0} through $\pi_{o_{k-1}}$ along the chain have been trained. L2S chooses the initial state distribution $\eta_{o_k} = \eta_{\rho_k}$ for training π_{o_k} to be the distribution of states reached by the controller π_{ρ_k} (Sec. 2) from a random environment initial state $s_0 \sim \eta$. The induced distribution η_{ρ_k} is defined inductively on the length of ρ_k. Formally, for the zero-length path ρ_k (so $\pi_{o_k} = \pi_{o_0}$), we define $\eta_{\rho_k} = \eta$ to be the initial state distribution of the MDP e. Otherwise, we have $\rho_k = \rho_{k-1} \circ \pi_{o_{k-1}}$. Then, we define η_{ρ_k} to be the state distribution over $\beta_{o_{k-1}}$ (the termination condition of o_{k-1}) induced by any trajectory τ generated using $\pi_{o_{k-1}}$ from $s_0 \sim \eta_{\rho_{k-1}}$. Given an infinite trajectory $\tau = s_0 \to s_1 \to \ldots$ if there exists i such that $\beta_o(s_i)$ holds, we denote the smallest such i by $i(\tau, \beta_o)$. Formally, η_{ρ_k} is the probability distribution over $\beta_{o_{k-1}}$ such that for any set of states $S' \subseteq \beta_{o_{k-1}}$, the probability of S' according to η_{ρ_k} is

$$\Pr_{s \sim \eta_{\rho_k}}[s \in S'] = \Pr_{s_0 \sim \eta_{\rho_{k-1}}, \tau \sim \pi_{o_{k-1}}(a_t|s_t;\ \phi_{k-1})} \left[s_{i(\tau,\ \beta_{o_{k-1}}[\varphi_{k-1}])} \in S' \right]$$

Main Algorithm can be found in Appendix C

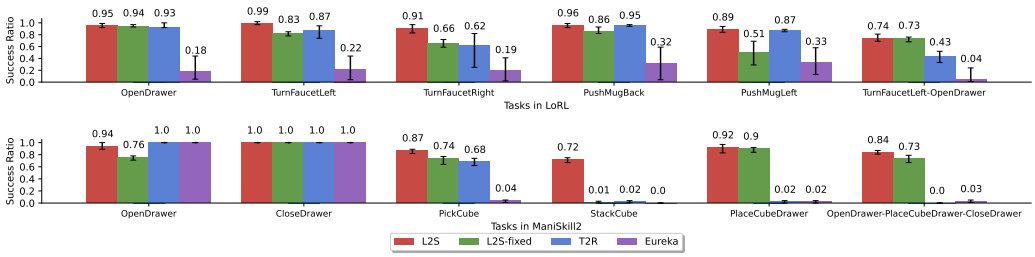

Figure 3: We report the average evaluation success rates at convergence across all tasks on LORL-Meta-World (top) and ManiSkill2 (bottom), averaged over 5 random seeds.

3.3 Reinforcement Learning for Single Subtask

We now describe how L2S learns a policy π_{o_k} for a single subtask o_k based on Equation 1 once the initial state distribution $\eta_{o_k} = \eta_{\rho_k}$ is known (Line 5 of Algorithm 1). At a high level, it trains π_{o_k} based on the reward function $\mathcal{R}_{o_k}(\phi_k)$ with the parameters $\phi_k \sim \tilde{q}_{\phi_k}$ sampled from a distribution $\tilde{q}_{\phi_k}$ (akin to "goals" in goal-conditioned reinforcement learning). Specifically, it uses Equation 2 to optimize the parameters of the preceding skill based on (frozen) π_{o_k}, which can be solved using any black-box optimization algorithm such as CEM.

$$\max_{\phi_{0:k-1},\varphi_{0:k-1}} \mathbb{E}_{s_0\sim\eta_{\rho_k}[\phi_{0:k-1},\varphi_{0:k-1}],\phi_k\sim\tilde{q}_{\phi_k},\tau\sim\pi_{o_k}(a|s;\phi_k)}\left[\sum_{t=0}^{T}\gamma^t\mathcal{R}_{o_k}[\phi_k](s_t,a_t)\right] \tag{2}$$

CEM searches only a small set of scalar interface parameters (about ten in our experiments), which are introduced on demand when downstream training fails (Appendix D.2). It uses Equation 3 to learn π_{o_k} based on the parameters of its preceding skills, which can be solved using a standard RL algorithm such as SAC [Haarnoja et al., 2018].

$$\max_{\pi_{o_k}}\mathbb{E}_{s_0\sim\eta_{\rho_k}[\phi_{0:k-1},\varphi_{0:k-1}],\phi_k\sim\tilde{q}_{\phi_k},\tau\sim\pi_{o_k}(a|s;\phi_k)}\left[\sum_{t=0}^{T}\gamma^t\mathcal{R}_{o_k}[\phi_k](s_t,a_t)\right] \tag{3}$$

Our subtask training algorithm iteratively optimizes both Equation 2 and Equation 3 until convergence.

3.4 Subtask Generation Improvement via Feedback

Because LLMs are prone to hallucination, the proposed subtask decompositions may be underspecified, internally inconsistent, or simply incorrect. To address this, we introduce a feedback loop that uses downstream learning outcomes to iteratively refine the LLM-generated subtasks.

The feedback procedure has two stages. First, when learning fails on a given subtask, the LLM audits whether the generated reward and termination functions fully implement the subtask specification by examining the reward terms and success conditions encoded in the implementation, rather than overlooking success-critical details in the textual description. At this stage, reward terms that provide little useful learning signal are removed, and missing terms are added as needed. We consider a reward term to be stagnant when its value changes minimally over subtask training, making it ineffective for guiding policy improvement. A term is flagged as stagnant when the deviation of its rolling-averaged value from its initial value, measured relative to the average magnitude of its initial and final values, stays below a small fixed threshold; a bootstrapped confidence check guards against RL noise, and all hyperparameters are shared across tasks and environments (Appendix D.1). The example can be found in Figure 15 in the appendix. Second, if the implementation is judged consistent with the specification, the LLM then revisits the subtask definition itself, since a vague, incomplete, or internally inconsistent specification can make correct implementation unattainable from the outset. The example can be found in Figure 12 in the appendix.

4 Experiments

Table 1: Descriptions of tasks in the benchmarks. The left table outlines the sequence of tasks in the LORL-Meta-World, while the right table details the task sequence for ManiSkill2.

LORL-Meta-World Task Sequence	**ManiSkill2 Task Sequence**
Task1: Open drawer	Task1: OpenDrawer
Task2: Turn faucet left	Task2: CloseDrawer
Task3: Turn faucet right	Task3: PickCube
Task4: Push white mug back	Task4: StackCube
Task5: Push white mug left	Task5: PlaceCubeDrawer
Task6: Turn faucet left and Open drawer	Task6: OpenDrawer, PlaceCubeDrawer and CloseDrawer

Benchmarks. We demonstrate the capability of L2S across various environments and tasks within the Meta-World [Yu et al., 2019] and ManiSkill2 [Gu et al., 2023] benchmarks. We conducted tasks within the **LORL-Meta-World** environment [Nair et al., 2021] (Fig. 4 left), a simulated domain built atop Meta-World. This environment features a Sawyer robot interacting with a tabletop setup that includes a drawer, a faucet, and two mugs. **ManiSkill2** offers a diverse range of simulated object manipulation tasks. We integrate the cube and cabinet environments into ManiSkill2 (Fig. 4). Sawyer robots in this environment can interact with two cubes and a cabinet with drawers and doors.



(a) LORL-Meta-World (b) ManiSkill2

Figure 4: Benchmarks

As detailed in Table 1, we evaluated five basic tasks in each environment. In both environments, we also introduce tasks that require the robot to achieve multiple goals (Task 6). The full list of evaluated tasks and their corresponding instructions can be found in Task list section in Appendix. G.

Approaches. We conducted a comparative analysis between our method (L2S) and three state-of-the-art baseline methods, along with an ablation study:

- **Text2Reward (T2R)** [Xie et al., 2023] uses an LLM to generate dense reward functions for training a single monolithic policy per robotic task. It relies on the same Python-based environment abstraction and task description that we provide to the LLM.
- **Eureka** [Ma et al., 2023] adopts an evolutionary approach. It feeds the environment script to the LLM to generate multiple reward functions, enabling policies to be trained in parallel. Batch success rates are then used as feedback to refine the reward functions in subsequent iterations, forming an iterative improvement loop. For our benchmarks, we ran Eureka for three rounds with eight samples per round. We performed multiple runs and report results from runs that produced at least one successful sample in each round.
- **League++** [Li et al., 2024] generates reward functions by having the LLM select and weight metric functions predefined by human experts (Table 11). We compare with it on a representative subset of tasks, averaged over five random seeds (Table 12).
- **L2S-fixed** is a variant of L2S that uses fixed LLM-generated parameters in the reward and termination functions, rather than optimizing them as in L2S . This allows us to isolate and evaluate the impact of suboptimal parameter choices made by the LLM.
- **HIRO** [Nachum et al., 2018] and **HAC** [Levy et al., 2019] are hierarchical RL baselines that discover their own subgoals, and **Summed sub-rewards** trains a single policy on the sum of the subtask rewards generated by L2S ; both are evaluated in the controlled ablation (Appendix E).

Experiment setup. We use GPT-4 as our `LLMAgent`. For reinforcement learning of subtask policies, we employ SAC [Haarnoja et al., 2018], maintaining consistent hyper-parameters across all tasks and experiments within these benchmarks. To evaluate the robustness of L2S , each task was conducted using 5 different random seeds. The hyperparameters for SAC are detailed in Appendix D. Under the same interaction budget, L2S uses 2.35× the LLM tokens and 1.45× the GPU-hours of T2R, far less than an Eureka-style search (Appendix F.4); its subtask policies are small MLPs with negligible inference cost (Appendix D). GPT-5 and GPT-5-mini backbones also produce correct reward and termination programs (Appendix F.3).

Overall Results.

- Fig. 3 summarizes per-task success at convergence across **LoRL** and **ManiSkill2**, which suggests L2S matches strong baselines on easy tasks while delivering substantial robustness gains on complex, multi-stage tasks where reward specification and termination sensitivity are critical. **In**

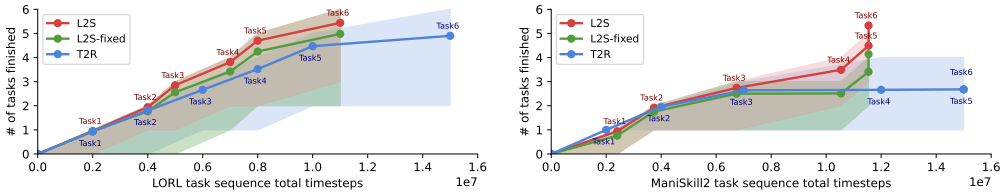

Figure 5: Given the sequence of tasks in Table 1, we report the average number of tasks trained to convergence on LORL-Meta-World (left side) and ManiSkill2 (right side), averaged over 5 random seeds. A policy is considered converged when its evaluation success rate stabilizes—changing by no more than 2% over 10 evaluation epochs—and reaches a value significantly above zero. Eureka is omitted because its evolutionary search demands considerably more training steps than the other methods.

LoRL, L2S is consistently strong on the single-goal tasks (e.g., LORL Tasks 1–5 in Table 1) and remains the top-performing method on the composed task TurnFaucetLeft–OpenDrawer(LORL Tasks 6 in Table 1), where the baseline methods degrade substantially and **Eureka** often fails. **In ManiSkill2**, the separation is even clearer on harder and compositional manipulation: while most methods can solve the simplest drawer tasks, L2S is markedly more reliable on single-goal tasks and the long-horizon OpenDrawer–PlaceCubeDrawer–CloseDrawer sequence(MS2 Tasks 6 in Table 1), where baselines frequently collapse.

- **L2S achieves a higher execution success rate for each generated subtask** compared to whole-task reward functions generated in **Text2Reward**. The results highlight the effectiveness of L2S in breaking down complex tasks into manageable components and improving the reliability of code generation. Since each subtask represents only a portion of the overall task which reduces complexity, generating free-form function code for individual subtasks is inherently simpler than generating a single function for the entire task. **League++**'s reliance on predefined Metric Functions may limit its ability to capture the full complexity of tasks or appropriately penalize suboptimal behavior, potentially reducing overall performance. Detailed results can be found in Appendix F.9 . This result reinforces the advantage of using free-form reward functions to improve the adaptability of subtask training.
- **L2S is more sample-efficient and scalable than Text2Reward, especially as tasks become longer-horizon and more compositional.** Fig. 5 tracks how many tasks converge as training progresses. Across both LORL and ManiSkill2, L2S consistently converges on more tasks and does so with fewer training steps than Text2Reward. The advantage is modest in LORL but much larger in ManiSkill2, where Text2Reward solves substantially fewer tasks. In general, L2S achieves higher task coverage while reducing training cost relative to Text2Reward in both environments.
- **Executability of LLM-generated reward functions** can be found in Appendix F.2.

Subtask improvement via feedback. Overall, most tasks require little or no refinement, suggesting that the initial LLM-generated decomposition is often already adequate and that the feedback mechanism mainly serves as a targeted correction procedure rather than a heavy iterative redesign process. These results support the usefulness of the proposed feedback loop for repairing subtasks when the initial decomposition is imperfect. As shown on the left side of Fig.6, PickCube and Pick&Place require no feedback iterations, indicating that for these simpler tasks the LLM can generate usable subtasks and implementations in a single pass. In contrast, tasks with stronger sequential or relational structure, especially StackCube, benefit more from iterative refinement. A representative example is shown on the right side of Fig.6, which illustrates the result of iteratively refining the skill "Move the cube above the other cube" in StackCube. More detailed examples can be found in Appendix F.7

Subtask reuse. Parameter conditioning allows learned subtasks to be reused across tasks by adjusting their parameters. For example, the first subtask of "Turn faucet left" positions the end-effector to the right of the faucet handle with an optimized offset `params[0]` = 0.083 m; for "Turn faucet right", L2S reuses the same policy with `params[0]` = −0.095 m to approach the opposite side (Fig. 9). Similarly, the cube-grasping subtask learned in PickCube is reused in StackCube and PlaceCubeDrawer, expediting sequential task learning.

Long-horizon tasks. We report evaluation results on additional complex, long-horizon tasks in both LORL and ManiSkill2 in Table 2. Notably, we prompted GPT-4 in both L2S and Text2Reward to

ManiSkill2 Task	Avg Iterations
PickCube	1.0
PlaceCubeDrawer	1.0
StackCube	1.8 (0.75)
OpenDrawer	1.4 (0.49)
CloseDrawer	1.2 (0.4)

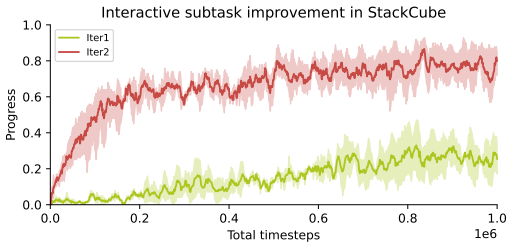

Figure 6: **Left**: Average number of feedback iterations required for each ManiSkill2 task over 5 seeds. **Right**: A representative example of iterative feedback-based refinement for the StackCube skill "Move the cube above the other cube". The values in parentheses denote the standard deviation across seeds.

reuse policies learned from prior single tasks whenever possible, ensuring a fair comparison between the two approaches. Every task listed in Table 2 is a combination of single-goal task from their corresponding environment (Table 1). Specifically, the ManiSkill2 tasks highlight the advantages of reusing learned skills for long-horizon task execution. In Place Two/Three Cubes, the agent must repeatedly place multiple cubes into a cabinet drawer. In Stack Three Cubes, the agent must stack three cubes into a single pile. Both tasks therefore require repeated reuse of learned skills, demonstrating the value of skill composition in more complex sequential settings.

Table 2: Performance of L2S on more long-horizon tasks.

Benchmark	Task	Text2Reward(Std)	L2S(Std)
LORL	PushMugBack-OpenDrawer	0.91(0.043)	0.93(0.030)
	OpenDrawer-TurnFaucetRight	0.89(0.096)	0.93(0.062)
	MugBack-OpenDrawer-TurnFaucetRight	0.76(0.071)	0.90(0.044)
ManiSkill2	OpenDrawer-Place**TwoCubes**Drawer-CloseDrawer	0.01(0.002)	0.77(0.056)
	OpenDrawer-Place**ThreeCubes**Drawer-CloseDrawer	0.01(0.001)	0.70(0.085)
	Stack**Three**Cubes	0.0	0.77(0.033)

Ablation Study. Comparison of L2S with L2S-fixed further indicates that optimizing the parameters proposed by the LLM matters most in tasks that require tighter geometric or temporal precision (e.g., stacking and lateral pushing), whereas simpler tasks show smaller differences. In Fig. 5, although the ablation L2S-fixed demonstrates a similar convergence rate to L2S for tasks in LORL environment, Fig. 3 reveals that it converges to sub-optimal policies. In ManiSkill2, L2S-fixed struggles to solve more challenging tasks, such as StackCube, underscoring the necessity of optimizing LLM-generated parameters in the reward and termination functions to address the inherent uncertainty of LLM agents when dealing with low-level control intricacies. A controlled ablation over all ten tasks with hierarchical RL baselines (Appendix E) shows that, on multi-stage tasks, training one policy on the summed subtask rewards performs far worse than chaining sub-policies with the same rewards (e.g., PlaceCubeDrawer 0.28 vs. 0.90), that optimizing handoff parameters adds further and sometimes decisive gains (StackCube 0.01 $\rightarrow$ 0.72), and that HIRO and HAC achieve little or no success on most tasks while DSC fails to initialize its skill chain. The gains therefore come from combining decomposition, chained sub-policies, and handoff optimization, not from reward information or hierarchy alone.

Additional experimental results are provided in Appendix F, including: (1) analysis of training progress, (2)an example of subtask reusing, (3) parameter optimization under varying levels of user-defined variance, and (4) the performance of L2S on additional long-horizon tasks that benefit from composing learned subtasks.

5 Related Work

LLM reasoning. Prompting techniques such as Chain-of-Thought [Wei et al., 2023], Zero-shot-CoT [Kojima et al., 2023], and ReAct [Yao et al., 2023] elicit multi-step reasoning from LLMs such

as GPT-3 [Brown et al., 2020] and GPT-4 [OpenAI et al., 2024]. L2S leverages this reasoning ability to interpret the semantics of robotic tasks and decompose them into subtasks.

LLM planning for robotics. Recent work integrates LLMs into robotic task and motion planning [Firoozi et al., 2023]. Huang et al. [2022a] investigate LLMs for direct trajectory planning, Inner Monologue [Huang et al., 2022b] incorporates real-time environment feedback into LLM-generated plans, and Text2Motion [Lin et al., 2023] verifies that skill sequences are geometrically feasible before execution. Another direction translates natural language into intermediate formal task representations, as in NL2TL [Chen et al., 2024b] and AutoTAMP [Chen et al., 2024a], which correct syntactic and semantic errors autoregressively. SayCan [Ahn et al., 2022] selects skills by combining task-grounding and world-grounding probabilities, and BOSS [Zhang et al., 2023] uses an LLM to compose primitive skills into longer skill chains. These methods plan over a given set of skills, whereas L2S learns the skill library itself from language-generated subtask definitions.

LLM-based reward and code generation. Text2Reward [Xie et al., 2023] and Eureka [Ma et al., 2023] generate free-form dense reward programs and refine them with human or automated feedback, and L2R [Yu et al., 2023] uses LLMs to define optimizable reward parameters. Follow-up work improves reward quality through self-alignment ranking [Zeng et al., 2024], reward-structure evolution and parameter alignment [Li et al., 2025], language-agent search over reward candidates [Gao et al., 2026], coder–evaluator feedback [Sun et al., 2024], iterative reward-weight refinement [Afonso et al., 2025], and reward machines [Castanyer et al., 2026], while League++ [Li et al., 2024] composes rewards from expert-defined metric functions. Another line of work, including ProgPrompt [Singh et al., 2022] and Code as Policies [Liang et al., 2023], generates executable control programs directly from language instructions. These methods ultimately produce one monolithic policy or program per task. Eureka in particular relies on repeated LLM queries and full policy training for every reward update, and League++ requires human experts to predefine metric functions. L2S instead uses language to define reusable subtasks with free-form rewards whose policies are learned with RL, and optimizes the parameters of their reward and termination functions according to downstream subtask performance.

Hierarchical RL and skill chaining. Option-Critic [Bacon et al., 2017] learns options and their termination functions end-to-end but can degenerate into options that terminate at every step or span the whole episode, while FuN [Vezhnevets et al., 2017], HIRO [Nachum et al., 2018], and HAC [Levy et al., 2019] learn multiple levels of policies that must discover subgoals from sparse task-level feedback. Skill chaining [Konidaris and Barto, 2009, Bagaria and Konidaris, 2020] constructs skills backward from the goal and therefore requires goal-reaching trajectories before learning can begin. L2S shares their aim of learning temporally extended sub-policies with compatible handoffs, but obtains semantically meaningful subtasks with dense local rewards from language and chains them forward from initial states (Appendix E). A broader discussion of related work is provided in Appendix A.

6 Conclusion

We present L2S that leverages Large Language Models (LLMs) to autonomously construct a subtask library for sequential task learning. L2S progressively builds a subtask library guided by LLMs and efficiently reuses them across new tasks, enabling the learning algorithm to effectively handle increasingly challenging environments. To handle the uncertainty in LLM-generated reward and termination functions, L2S trains a parameter-conditioned policy that perform well across a broad range of parameter values for each subtask and selects the most suitable parameter values during the training of its subsequent subtasks, mitigating the risk of incorrect parameter choices by LLMs.

Experimental results demonstrate that L2S outperforms baselines in solving complex, multi-step tasks, largely due to its ability to automatically construct a subtask library for sequential task learning. L2S is currently evaluated in state-based robotic manipulation, where the generated termination functions read privileged simulator state; extending it to visual observations, real hardware, and other domains such as navigation, as well as to higher-dimensional parameter spaces that may require a more sample-efficient optimizer than CEM, is left for future work (Appendix B).

References

António Afonso, Iolanda Leite, Alessandro Sestini, Florian Fuchs, Konrad Tollmar, and Linus Gisslén. Self-correcting reward shaping via language models for reinforcement learning agents in games, 2025. URL https://arxiv.org/abs/2506.23626.

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

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In Satinder Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 1726–1734. AAAI Press, 2017. doi: 10.1609/AAAI.V31I1.10916. URL https://doi.org/10.1609/aaai.v31i1.10916.

Akhil Bagaria and George Konidaris. Option discovery using deep skill chaining. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=B1gqipNYwH.

Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discret. Event Dyn. Syst.*, 13(1-2):41–77, 2003. doi: 10.1023/A:1022140919877. URL https://doi.org/10.1023/A:1022140919877.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, and et al. Girish Sastry. Language models are few-shot learners, 2020.

Roger Creus Castanyer, Faisal Mohamed, Pablo Samuel Castro, Cyrus Neary, and Glen Berseth. Arm-fm: Automated reward machines via foundation models for compositional reinforcement learning, 2026. URL https://arxiv.org/abs/2510.14176.

Yongchao Chen, Jacob Arkin, Charles Dawson, Yang Zhang, Nicholas Roy, and Chuchu Fan. Autotamp: Autoregressive task and motion planning with llms as translators and checkers, 2024a.

Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. Nl2tl: Transforming natural languages to temporal logics using large language models, 2024b.

Roya Firoozi, Johnathan Tucker, Stephen Tian, Anirudha Majumdar, Jiankai Sun, Weiyu Liu, Yuke Zhu, Shuran Song, Ashish Kapoor, Karol Hausman, Brian Ichter, Danny Driess, Jiajun Wu, Cewu Lu, and Mac Schwager. Foundation models in robotics: Applications, challenges, and the future, 2023.

Ning Gao, Xiuhui Zhang, Xingyu Jiang, Mukang You, Mohan Zhang, and Yue Deng. Rf-agent: Automated reward function design via language agent tree search, 2026. URL https://arxiv.org/abs/2602.23876.

Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, Xiaodi Yuan, Pengwei Xie, Zhiao Huang, Rui Chen, and Hao Su. Maniskill2: A unified benchmark for generalizable manipulation skills, 2023. URL https://arxiv.org/abs/2302.04659.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, 2022a.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models, 2022b.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

George Dimitri Konidaris and Andrew G. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, pages 1015–1023. Curran Associates, Inc., 2009. URL https://proceedings.neurips.cc/paper/2009/hash/e0cf1f47118daebc5b16269099ad7347-Abstract.html.

Andrew Levy, George Dimitri Konidaris, Robert Platt Jr., and Kate Saenko. Learning multi-level hierarchies with hindsight. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=ryzECoAcY7.

Pengyi Li, Jianye HAO, Hongyao Tang, Yifu Yuan, Jinbin Qiao, Zibin Dong, and YAN ZHENG. R*: Efficient reward design via reward structure evolution and parameter alignment optimization with large language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=qZMLrURRr9.

Zhaoyi Li, Kelin Yu, Shuo Cheng, and Danfei Xu. League++: Empowering continual robot learning through guided skill acquisition with large language models. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. URL https://openreview.net/forum?id=xXo4JL8FvV.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control, 2023.

Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2motion: from natural language instructions to feasible plans. *Autonomous Robots*, 47(8):1345–1365, November 2023. ISSN 1573-7527. doi: 10.1007/s10514-023-10131-7. URL http://dx.doi.org/10.1007/s10514-023-10131-7.

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

Amy McGovern and Richard S. Sutton. Macro-actions in reinforcement learning: An empirical analysis. 1998. URL https://api.semanticscholar.org/CorpusID:5821100.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3307–3317, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/e6384711491713d29bc63fc5eeb5ba4f-Abstract.html.

Suraj Nair, Eric Mitchell, Kevin Chen, Brian Ichter, Silvio Savarese, and Chelsea Finn. Learning language-conditioned robot behavior from offline data and crowd-sourced annotation, 2021.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, and et al. Sam Altman. Gpt-4 technical report, 2024.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

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

Shengjie Sun, Runze Liu, Jiafei Lyu, Jing-Wen Yang, Liangpeng Zhang, and Xiu Li. A large language model-driven reward design framework via dynamic feedback for reinforcement learning, 2024. URL https://arxiv.org/abs/2410.14660.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999. doi: 10.1016/S0004-3702(99)00052-1. URL https://doi.org/10.1016/S0004-3702(99)00052-1.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3540–3549. PMLR, 2017. URL http://proceedings.mlr.press/v70/vezhnevets17a.html.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Automated dense reward function generation for reinforcement learning, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. *CoRR*, abs/1910.10897, 2019. URL http://arxiv.org/abs/1910.10897.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis, 2023.

Yuwei Zeng, Yao Mu, and Lin Shao. Learning reward for robot skills using large language models via self-alignment, 2024. URL https://arxiv.org/abs/2405.07162.

Jesse Zhang, Jiahui Zhang, Karl Pertsch, Ziyi Liu, Xiang Ren, Minsuk Chang, Shao-Hua Sun, and Joseph J. Lim. Bootstrap your own skills: Learning to solve new tasks with large language model guidance, 2023. URL https://arxiv.org/abs/2310.10021.

A More Related Work

LLMs for Reasoning. Recent studies on large language models (LLMs), e.g. **GPT-3** [Brown et al., 2020], **GPT-4** [OpenAI et al., 2024], have demonstrated significant advancements of its reasoning capabilities. Prompting proposed by **Chain-of-Thought (CoT)** [Wei et al., 2023], has shown efficacy in improving reasoning by eliciting detailed reasoning paths in LLMs, which helps in tasks involving multi-step reasoning. Similarly, **ReAct** [Yao et al., 2023] combines reasoning with actions, enhancing performance on tasks by enabling dynamic reasoning and interactions with external information, demonstrating significant improvements. Additionally, **Zero-shot-CoT** [Kojima et al., 2023] has been proved effective in enhancing the zero-shot reasoning abilities of models across various tasks, enhancing the potential of LLMs in tasks requiring complex multi-hop thinking without the need for task-specific fine-tuning. These advancements suggests a promising direction for further enhancing the reasoning powers of LLMs through advanced prompting techniques and integrated reasoning-action paradigms. We leverage such reasoning ability to make LLMs understand the semantics of robotic tasks and follow the instructions from human correctly.

LLM Planning for Robotics. Recent research has highlighted the integration of Large Language Models (LLMs) in robotic task and motion planning (TAMP) [Firoozi et al., 2023]. Huang et al. [2022a] investigated LLMs for direct trajectory planning, revealing limitations in spatial and numerical reasoning that necessitate frequent re-prompting to align with task constraints. Following works aim to mitigate the gap on feasibility and correctness when applying LLM-generated plans to simulated or real-world robotic environments. Inspired by the in-context learning ability of LLMs, **Inner Monologue** [Huang et al., 2022b] allows robotic systems to integrate real-time environmental feedback into LLM-generated plans. This strategy significantly enhances the adaptability and effectiveness of robotic agents by using continuous feedback to adjust planning strategies. **Text2Motion** [Lin et al., 2023] goes a step further by not only generating feasible task plans (a sequence of skills) but also ensuring these plans are geometrically executable before initiation. Another direction is to utilize LLMs for translating natural language into intermediate formal task representations, **NL2TL** [Chen et al., 2024b] and **AutoTAMP** [Chen et al., 2024a] significantly enhancing task completion through auto-regressive error correction of both syntax and semantics. The planning ability of LLMs plays a great role in L2S for generating subtask chains and make plan on it to complete robotic tasks. To finish wider range of tasks, **BOSS**[Zhang et al., 2023] leverages LLM to build skill library with large amount of complex and useful skill chains generated from a set of primitive skills. Also, **SayCan** [Ahn et al., 2022] ranks all the possible skills by the task-grounding probability (usefulness) and world-grounding probability (feasibility) and select the one with highest probability at each step for LLM decision making within a given embodiment.

LLM-Based Reward Generation and Refinement. **L2R** [Yu et al., 2023] introduces a new paradigm that leverages the flexibility of reward function representations by using LLMs to define reward parameters that can be optimized to accomplish a variety of robotic tasks. To generate RL reward functions for robotic tasks, Zeng et al. [2024] incorporates self-alignment ranking to improve the quality of generated reward functions using samples ranked by both LLMs and reward functions. **Text2Reward** [Xie et al., 2023] generates interpretable, free-form dense reward functions as executable programs grounded in a compact representation of the environment in either zero-shot or few-shot settings. **Eureka** [Ma et al., 2023] generates dense reward functions without task-specific prompting or pre-defined reward templates. Both **Text2Reward** and **Eureka** leverage the in-context capabilities of LLMs to improve reward functions using human-provided or automated feedback, respectively. Follow-up works further extend this direction through more structured reward optimization, including reward structure evolution and parameter alignment in **R*** [Li et al., 2025], language-agent search over reward candidates in **RF-Agent** [Gao et al., 2026], coder–evaluator feedback in **CARD** [Sun et al., 2024], iterative reward-weight refinement in **Self-correcting Reward Shaping** [Afonso et al., 2025], and structured reward specification via reward machines in **ARM-FM** [Castanyer et al., 2026]. In **League++** [Li et al., 2024], reward functions are generated by selecting and weighting pre-defined metric functions provided by human experts. Our method differs from these reward-generation approaches in its ability to generate reusable subtasks for *sequential task learning* while iteratively refining the parameterization of reward and termination functions based on subtask-chain training. Compared with **League++**, our method additionally enables free-form reward generation and requires less human expert effort. Compared with **Eureka**, although its evolutionary approach can improve reward functions, it relies heavily on costly LLM interactions for environment

feedback and expensive policy training for each parameter update, and it does not support subtask learning.

LLM-Based Policy and Code Generation. Another line of work explores the use of LLMs for robot-centric policy generation by directly producing executable code or structured plans from language instructions, such as **ProgPrompt** [Singh et al., 2022] and **Code as Policies** [Liang et al., 2023]. Unlike these methods, which focus on generating control programs or policy-level behaviors, L2S focuses on decomposing tasks into chains of subtasks, learning subtasks as reusable primitives, building a subtask library from subtask chains, optimizing parameters in generated reward and termination functions to improve task performance, and reusing subtasks and subtask chains to improve learning efficiency.

B Discussion and limitations

Task domains. L2S has been evaluated solely in robotic manipulation domains. Our tool demonstrates that decomposing natural language tasks into subtask chains significantly enhances performance across a broad range of robotic tasks while reducing the cost of neural policy learning. Applying LLM-based subtask discovery to other task types, such as navigation, would necessitate more advanced reasoning about environmental structures, which we leave as an avenue for future research.

High-dimensional parameter spaces. L2S keeps the CEM search space small through demand-driven parameterization (Appendix D.2). Genuinely high-dimensional, coupled parameter spaces may require a more sample-efficient black-box optimizer, which can replace CEM without changing the rest of L2S . **Robustness of generated code.** LLM hallucinations present challenges in generating robust free-form reward and termination function code. Constraining code generation within a structured intermediate representation, possibly defined by a domain-specific language, might offer a balance between generation stability and the exploration of the reward space.

B.1 Generalization Across Observation and Action Spaces

Our experiments make two interface assumptions: sub-policies observe privileged simulator state, and all subtasks share the same state and action spaces. We discuss how L2S extends when each assumption is relaxed.

Visual observations and sim-to-real transfer. L2S currently operates within state-based environments, as the LLM-generated termination functions require explicit state information to assess whether termination thresholds are met. This is a deliberate experimental scope: L2S studies how language-generated subtasks, rewards, and termination conditions are grounded into compositional policies while isolating policy learning from perception. Under a standard sim-to-real pipeline, each subtask policy can instead receive RGB-D observations, while its reward is computed from privileged simulator state during training. Similarly, the generated termination functions can automatically label simulated observations for training image-based termination predictors. Visual and dynamics randomization can then be used to transfer both the policies and termination predictors to real-world images. All parameter search and successor-guided interface optimization occur in simulation and deployment uses the selected parameters together with the learned visual policies and termination predictors. Thus, L2S 's decomposition, downstream-guided optimization, and policy-composition mechanism remain unchanged, while the structured programs serve as privileged training supervision. We identify image-based and real-hardware validation as future work.

Varying state and action spaces. The current implementation assumes a shared observation and action interface for direct subtask-policy reuse. If these spaces change, L2S 's decomposition and downstream interface optimization remain applicable, but the learned policies cannot be transferred directly. They would require either retraining in the new space or task/embodiment-specific state encoders and action decoders that map to a shared latent skill space; the reward and termination programs must likewise be grounded to the new state API. We identify embodiment-conditioned policies or learned cross-domain adapters as a natural extension.

C Main Algorithm

Algorithm 1 L2S LearningAlgorithm

Require: A sequence of tasks $\mathcal{T} = \{(e, \mathcal{L}_e)\}$ each with task MDP e and text description $\mathcal{L}_e$
Require: Code generating LLM LLMAgent
Ensure: Subtask Library $\mathcal{O}$, Task Controllers $\mathcal{C}$
1: $\mathcal{O} \leftarrow \varnothing, \mathcal{C} \leftarrow \varnothing$
2: **for each** task $(e, \mathcal{L}_e) \in \mathcal{T}$ **do**
3: $(\rho \equiv o_0 \circ o_1 \circ \ldots),\ \mathcal{R}_o,\ \beta_o,\ \mathcal{L}_o \leftarrow \mathsf{LLMAgent}(\mathsf{prompt}(\mathsf{encode}(e), \mathcal{L}_e, \mathcal{O}))$
4: **for** $k = 0, 1, \ldots, \text{LEN}(\rho) - 1$ **do**
5: Train π_{o_k} and update $\phi_{0:k-1}, \varphi_{0:k-1}$ via Eq. 1
6: $\mathcal{O} \leftarrow \mathcal{O} \cup \{o_k, \mathcal{L}_{o_k}\}$
7: **end for**
8: $\pi_\rho \leftarrow (\pi_{o_0}[\phi_0], \beta_{o_0}[\varphi_0]) \circ (\pi_{o_1}[\phi_1], \beta_{o_1}[\varphi_1]) \circ \cdots$
9: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\pi_\rho\}$
10: **end for**

We depict the overall subtask training algorithm of L2S in Algorithm 1. It handles a sequence of tasks $\mathcal{T} = \{(e, \mathcal{L}_e)\}$ each with task MDP e and text description $\mathcal{L}_e$. At line 3, it prompts the LLM agent with the pythonic representation of e, $\mathcal{L}_e$ and the subtask library $\mathcal{O}$ (initialized to empty) to generate the subtask chain ρ for e (Sec. 3.1). For each task, at line 5, it iteratively trains the subtasks in ρ (Sec. 3.2). When the LLM agent selects a subtask o_k from the subtask library $\mathcal{O}$, the algorithm trains the subtask controller π_{o_k}, beginning with the existing policy and value functions (and the replay buffer if using an offline RL algorithm), which often leads to policy reuse or results in fast convergence. At line 6, the algorithm incorporates the trained subtask into the subtask library $\mathcal{O}$ for reuse in future tasks[1]. The final subtask chaining controller π_ρ, constructed for ρ (line 8), is added to $\mathcal{C}$, which holds the controllers for all the tasks in the input sequence $\mathcal{T}$ (line 9).

D Implementation Details and Hyper-parameters

Table 3: Hyper-parameter of SAC algorithm applied to tasks in two benchmarks

SAC Hyper-parameters	LORL(Meta-World)	ManiSkill2
Discount factor γ	0.99	0.95
Target update frequency	2	1
Learning rate	$3e^{-4}$	$3e^{-4}$
Train frequency	1	8
Soft update τ	$5e^{-3}$	$5e^{-3}$
Gradient steps	1	4
Learning starts	4000	4000
Hidden units per layer	256	256
# of layers	3	2
Batch Size	512	1024
Initial temperature	0.1	0.2
Rollout steps per episode	500	100/200
Replaybuffer size	$5e5$	$5e5$

We document the hyper-parameters used for LLM code generation and RL learning algorithms in this section. For generating dense reward function code and termination condition function code, we utilized GPT-4 as the LLM agent with the sampling temperature set to 0.2 and the top_p (the cumulative probability of next token candidates) set to 0.1 for each experiment in L2S . The baseline (T2R) maintained the default values for temperature and top_p at 0.7 and 1, respectively. Results with GPT-5 and GPT-5-mini as alternative backbones are reported in Appendix F.3. For the reinforcement learning algorithm, we employed the implementation from Stable-Baselines3 (Raffin et al. [2021]) with the hyper-parameters listed in Table 3.

Subtask-policy architecture. Each sub-policy is an MLP (two hidden layers of 256 units on ManiSkill2, three on LORL) trained with SAC (twin-Q, tanh-squashed Gaussian policy). Its input is

[1] In our implementation, we add a subtask o_k to the library $\mathcal{O}$ only if its controller is well-trained and achieves a high return.

the environment state s concatenated with the subtask's conditioning parameters ϕ (analogous to the goal in goal-conditioned RL); its output is the low-level action a.

D.1 Stagnant Reward-Term Criterion

The feedback loop in Sec. 3.4 removes stagnant reward terms, i.e., terms whose value changes minimally over subtask training. We define this criterion operationally as follows. At each checkpoint t during subtask training, we record $v_{j,t}$, the mean value of reward term j across the training trajectories collected around that checkpoint, and compute its K-checkpoint rolling average $\tilde{v}_{j,t}$. Let $\bar{v}_j^{\text{initial}}$ and $\bar{v}_j^{\text{final}}$ denote its averages over the first and last K checkpoints. We measure the largest sustained deviation from its initial value:

$$m_j = \max_t \frac{\left|\tilde{v}_{j,t} - \bar{v}_j^{\text{initial}}\right|}{\frac{1}{2}\left(\left|\bar{v}_j^{\text{final}}\right| + \left|\bar{v}_j^{\text{initial}}\right|\right) + \epsilon}. \tag{4}$$

A term is flagged as stagnant when it never deviates meaningfully from its initial value, i.e., $m_j < 0.05$. We use $K = 5$ and $\epsilon = 10^{-6}$, fixed across all tasks. Anchoring the comparison at the beginning of training distinguishes stagnation from convergence: a term that improves and then stabilizes has a large deviation from its initial value, while a term that remains unchanged throughout training has a small m_j. If a term changes temporarily and later returns to its initial value, it is still recognized as non-stagnant.

Robustness is provided at two levels: every checkpoint value is averaged across multiple trajectories, and the rolling average suppresses isolated checkpoint fluctuations. We additionally bootstrap the training trajectories and flag a term only when the upper bound of the 95% confidence interval for m_j is below 0.05. This is conservative under noise: greater uncertainty makes the pipeline less likely to declare a term stagnant.

D.2 Parameter Search Space for CEM

CEM is not generally immune to high dimensionality. In L2S , it searches over the set of scalar interface parameters (offsets, thresholds, and tolerances) exposed by the generated subtask programs, rather than over program structure, policy parameters, or raw state. Although Eq. 2 may jointly optimize parameters from several preceding subtasks, these interfaces remain small (~10) in our experiments. L2S also avoids constructing a large parameter space speculatively. It begins with the parameters required by the current generated program, and additional parameters are introduced only when downstream failure reveals that the existing parameterization is insufficient. Fig. 12 illustrates this demand-driven expansion. The original StackCube "move-above" reward was parameter-free and used only `norm(cubeA - cubeB)`. When training revealed that this objective did not maintain the required vertical separation, feedback revised the reward and introduced exactly one missing degree of freedom: a vertical offset between `cubeA` and `cubeB`. CEM then optimized that offset. Thus, L2S does not ask the LLM to enumerate and jointly optimize every potentially relevant geometric quantity in advance; it expands the parameterization selectively in response to observed learning failures. This failure-triggered refinement limits unnecessary parameter proliferation and keeps the search space small in our experiments. Genuinely high-dimensional, coupled parameter spaces may require a more sample-efficient black-box optimizer, which can replace CEM without changing the rest of L2S (Appendix B).

Task	Task Description
Task1	Push mug backward and open drawer
Task2	Open drawer and turn faucet left
Task3	Open drawer and push white mug backward
Task4	Turn faucet left and open drawer
Task5	Push mug backward and turn faucet left
Task6	Push mug backward and open drawer and turn faucet left

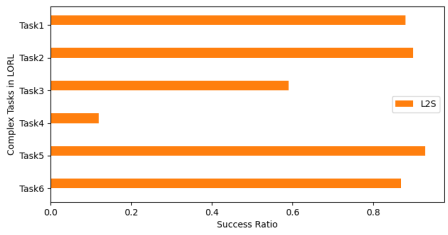

Figure 7: Complex task instruction(left) and success ratio on complex tasks in LORL environment(right).

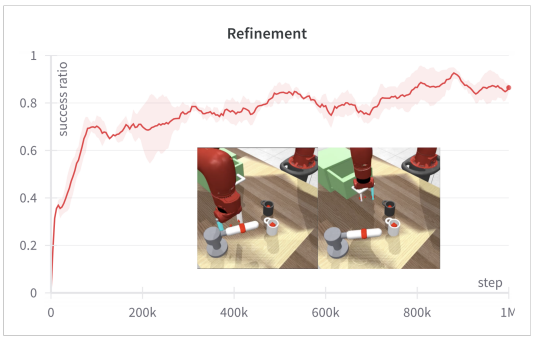

Figure 8: Success ratio on refining reused subtask in task "Turn faucet left and open drawer".

E Relation to Hierarchical RL and Skill Chaining

Positioning. The primary contribution of L2S is neither reward generation nor task decomposition alone. The core novelty of L2S is converting an LLM-sketched subtask chain into an executable controller by jointly learning its reusable parameter-conditioned subtask policies and their handoffs, i.e., the reward and termination parameters that determine the states from which the next subtask starts. Each handoff is optimized according to the performance of its successor subtask. Concretely, the LLM provides an initial subtask chain together with parameterized reward and termination programs. Rather than accepting the LLM's numerical choices or training each subtask independently, L2S trains policies *conditioned* on these parameters, and while learning a successor subtask it optimizes the reward and termination parameters (the handoff) of the preceding subtasks according to the successor's achieved return (Eqs. 2 and 3 implement this alternating optimization between successor-policy learning and predecessor-handoff optimization). A preceding subtask is therefore optimized not only for local completion, but to produce states from which the next subtask can be learned and executed.

L2S shares with hierarchical RL [Sutton et al., 1999, Vezhnevets et al., 2017, Nachum et al., 2018, Levy et al., 2019, Konidaris and Barto, 2009, Bagaria and Konidaris, 2020] the goal of learning temporally extended sub-policies with compatible handoffs. Its distinguishing mechanism is how those handoffs are represented and optimized: as explicit numerical parameters inside LLM-generated, free-form reward and termination programs, supported by reusable parameter-conditioned policies and selected by downstream subtask performance. Language-based decomposition and reward generation supply the initial compositional structure; L2S 's technical contribution is the learning mechanism that grounds this structure into an executable policy chain. This also separates L2S from LLM reward-generation methods (Text2Reward [Xie et al., 2023], Eureka [Ma et al., 2023]), which refine a whole-task reward and ultimately train a monolithic policy.

The design has concrete advantages over standard HRL for language-described, sparse-reward manipulation. Standard HRL must discover useful subgoals from sparse task-level feedback or rely on hand-specified subgoal structures; L2S instead uses language to construct semantically meaningful subtasks with dense local learning signals, which shortens the learning horizon and improves credit assignment, while downstream-aware handoff optimization reduces failures at subtask boundaries and parameter conditioning allows learned subtasks to be reused without retraining. These properties explain L2S 's stronger sample efficiency and success on long-horizon compositional tasks. GPT-4 and SAC are deliberately standard components; the novelty is this mechanism, not the model or optimizer. The experiments below test this distinction empirically.

Deep Skill Chaining (DSC). DSC [Konidaris and Barto, 2009, Bagaria and Konidaris, 2020] is conceptually relevant, so we implemented and evaluated it. DSC requires successful goal-reaching trajectories before backward skill construction can begin. These trajectories need not represent a reliable solution; they serve as seeds for learning the final option's initiation set, after which preceding options are constructed backward to expand the set of states from which the task can be solved. In our contact-rich manipulation tasks, however, obtaining even these seed trajectories requires an exploratory policy to accidentally execute the full sequence of reaching, gripper alignment, stable grasping, lifting, transportation, and placement. Under the matched interaction budget, DSC produced no successful trajectories (0 goal-reaching episodes out of 40 exploration rollouts, on both simulators).

It therefore could not initialize its final option, and no skill chain was constructed (`skills_built` = 0 in every probe). We report this as an empirical bootstrapping failure rather than claiming that DSC is theoretically inapplicable. L2S avoids this because it chains forward from initial states, so it never needs a full-task success to begin learning. We additionally compare with HIRO [Nachum et al., 2018] and HAC [Levy et al., 2019], whose hierarchical learning mechanisms do not require successful full-task trajectories before learning can begin.

Controlled factorial ablation. To determine whether the improvement comes from reward design, decomposition, or simply multiple sub-policies, we run one factorial over all 10 evaluated tasks (5 ManiSkill2 + 5 LORL) that also adds the forward hierarchical RL baselines (HIRO, HAC). The controlled variants (summed sub-rewards, L2S-fixed, and L2S) hold L2S 's subtask boundaries fixed, and all methods share the same SAC settings, interaction budget, and seeds; HIRO and HAC are the deliberate exception, since they invent their own subgoals rather than using a decomposition (that is precisely the factor they test). Table 4 reports success at convergence, mean ± std over 3 seeds.

Table 4: Success at convergence (mean ± std over 3 seeds) of hierarchical RL baselines and controlled variants of L2S . Summed sub-rewards, L2S-fixed, and L2S share L2S's subtask boundaries; HIRO and HAC discover their own subgoals. MS2 denotes ManiSkill2. "/" marks a task on which the Summed sub-rewards experiment was omitted.

Task	Text2Reward	HIRO	HAC	Summed sub-rewards	L2S-fixed	L2S
MS2 PickCube	0.68 ± 0.06	0.00 ± 0.00	0.00 ± 0.00	0.28 ± 0.06	0.74 ± 0.10	0.87 ± 0.05
MS2 StackCube	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.01 ± 0.01	0.72 ± 0.05
MS2 PlaceCubeDrawer	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.28 ± 0.12	0.90 ± 0.06	0.92 ± 0.08
MS2 OpenDrawer	1.00 ± 0.00	0.76 ± 0.15	0.55 ± 0.17	0.92 ± 0.04	0.76 ± 0.05	0.94 ± 0.06
MS2 CloseDrawer	1.00 ± 0.00	0.23 ± 0.16	0.74 ± 0.15	/	1.00 ± 0.00	1.00 ± 0.00
LORL OpenDrawer	0.93 ± 0.07	0.02 ± 0.02	0.00 ± 0.00	1.00 ± 0.00	0.94 ± 0.03	0.95 ± 0.04
LORL TurnFaucetLeft	0.87 ± 0.13	0.02 ± 0.01	0.19 ± 0.07	1.00 ± 0.00	0.83 ± 0.05	0.99 ± 0.01
LORL TurnFaucetRight	0.62 ± 0.37	0.00 ± 0.00	0.01 ± 0.00	/	0.66 ± 0.07	0.91 ± 0.08
LORL PushMugBack	0.95 ± 0.02	0.02 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.86 ± 0.07	0.96 ± 0.04
LORL PushMugLeft	0.87 ± 0.02	0.01 ± 0.01	0.00 ± 0.00	/	0.51 ± 0.22	0.89 ± 0.05

These comparisons separate the contributions as follows. HIRO and HAC are hierarchical agents (a high-level policy proposes subgoals, a low-level policy reaches them), so they share the multi-policy structure of L2S , but have no task decomposition and no LLM reward. Summed sub-rewards trains one policy on the sum of L2S 's generated subtask rewards, preserving the available dense reward information but removing temporal decomposition and policy chaining. L2S-fixed executes the same subtasks with chained sub-policies while retaining the LLM's initial, unoptimized reward and termination parameters. Full L2S additionally optimizes those parameters according to successor-subtask performance. Therefore, Summed sub-rewards → L2S-fixed measures the benefit of operationalizing the decomposition through chained policies and explicit handoffs, while L2S-fixed → L2S isolates the contribution of downstream-guided parameter optimization.

The results support three conclusions. First, despite multiple policy levels, HIRO and HAC achieve little or no success on most of the contact-rich, multistep tasks. Their stronger results on the simpler drawer tasks suggest that generic HRL can succeed when exploration and credit assignment are less demanding, but it does not reliably discover the required manipulation structure. Second, dense subtask rewards alone (Summed sub-rewards) are insufficient: converting the same reward information into chained sub-policies substantially improves PickCube (0.28 → 0.74), PlaceCubeDrawer (0.28 → 0.90), and PushMugBack (0.00 → 0.86). Third, optimizing the subtask interfaces provides an additional and sometimes decisive gain: most notably, StackCube improves from 0.01 with fixed LLM parameters to 0.72 with full L2S . Overall, the gains cannot be attributed to reward information alone (Summed sub-rewards), multiple policies alone (L2S-fixed), or generic hierarchy (HIRO and HAC). They arise from combining language-generated decomposition with chained sub-policy learning and downstream-guided optimization of the interfaces between those policies.

F Additional Experiment Results

F.1 Subtask Reusing and Refinement on complex task.

In this section, we show the result of reusing subtasks or subtask chain from basic single-goal task to complete complex tasks in Fig. 7. We showcases the effectiveness of subtasks refinement in L2S

when necessary. For example, the Task4 "Turn faucet left and open drawer" performs only 12% success ratio with directly reusing subtasks from subtask library. However, with refinement by L2S , the performance can be greatly improved to close to perfect, with evaluation curve shown in Fig. 8.

In the LORL environment, L2S leverages skills learned from “Turn faucet left" and “Push white mug backward" to expedite training for “Turn faucet right" and “Push white mug left" respectively. Take Fig. 9 as an example. The first subtask of the “Turn faucet left" task, $\pi_{o_1}(a|s;\texttt{params})$, guides the robot’s end-effector to the right side of the faucet handle at a target location with $\texttt{params[0]} = 0.083$m away from the handle. This parameter-conditioned subtask was reused with $\texttt{params[0]} = -0.095$m to guide the end-effector to the opposite side of the faucet in the “Turn faucet right" task. In Task 6 of the LORL environment, although the LLM agent recognizes that this combination of tasks can be addressed by reusing existing skills, as shown in Fig. 5, L2S still requires several training steps to fine-tune these skills for adaptation to the environment due to shifts in the initial state distributions. Similarly, as one example in ManiSkill2, L2S leverages the skill for grasping the cube in the “PickCube" task for the “StackCube" and “PlaceCubeDrawer" tasks, thereby expediting sequential task learning.

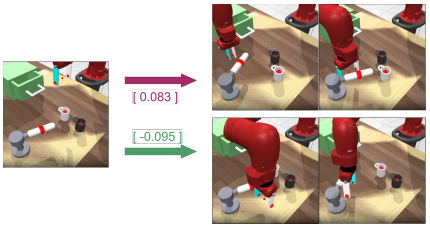

Figure 9: Subtask reusing example. The first subtask policy $\pi_{o_1}(a|s;\texttt{params})$ in “Turn faucet left" is reused for “Turn faucet right" with the optimized $\texttt{params[0]} = -0.095$m.

F.2 Error Analysis on generated functions.

For the reward generation experiment in the LORL and ManiSkill2 environments, we selected 5 simple tasks from each environment(LORL: "OpenDrawer, TurnFaucetLeft, TurnFaucetRight, PushMugBack, PushMugLeft"; ManiSkill2: "OpenDrawer, CloseDrawer, PickCube, StackCube, PlaceCubeDrawer"), as shown in Figure 5 in the paper, and queried the LLM for 20 samples per task. Across these 10 tasks, the number of subtasks generated ranged from 2 to 5. The reported results reflect the success rate for completing the entire tasks.

As shown in Table 5, L2S achieves a higher execution success rate for each generated subtask compared to the whole-task reward function generated in Text2Reward. This is because generating free-form function code for individual subtasks is inherently simpler than generating a single function for the entire task. Each subtask represents only a portion of the overall task, reducing complexity. The results highlight the effectiveness of L2S in breaking down complex tasks into manageable components and improving the reliability of code generation.

Table 5: Error Analysis on generated functions. LLM’s performance in generating correct function code for both LORL and ManiSkill2 environments more than 100 samples each.

LLM(GPT-4)	LORL	ManiSkill2
L2S-Correct/Error	**92%** / 8%	**87%** / 13%
T2R-Correct/Error	90% / 10%	87% / 13%

F.3 Robustness to the LLM Backbone

To examine whether the results depend on the choice of LLM, we swapped the LLM backbone (GPT-4, GPT-5, GPT-5-mini) on five ManiSkill2 tasks, holding the RL training, optimizer, and prompts fixed. Every backbone eventually produced valid, rule-compliant, correct reward and termination functions on all five tasks. The policies trained from these programs achieved comparable downstream success. Table 6 and Fig. 10 report per-task tokens, iteration counts, and generation wall-clock, averaged over 5 seeds. *Iterations* count generation attempts (L2S introduces a feedback loop that uses downstream learning outcomes to iteratively refine the reward and termination functions of LLM-generated subtasks, Sec. 3.4; 1.0 means correct on the first generation with no feedback repair).

The average numbers of generation attempts to reach a confirmed-correct program were 1.28 for GPT-4, 1.16 for GPT-5, and 1.24 for GPT-5-mini, indicating that most task-seed runs succeeded on

Table 6: LLM-backbone ablation on five ManiSkill2 tasks (mean ± std over 5 seeds): LLM tokens spent, generation iterations (1.0 means correct on the first generation), and generation wall-clock per task. The GPT-4 iteration counts are those reported in Fig. 6.

Metric	Task	GPT-4	GPT-5	GPT-5-mini
Tokens	PickCube	4,624 ± 177	9,528 ± 342	8,401 ± 208
	PlaceCubeDrawer	4,963 ± 110	10,247 ± 517	10,078 ± 316
	StackCube	8,831 ± 196	16,628 ± 435	12,116 ± 591
	OpenDrawer	8,163 ± 157	11,998 ± 227	14,769 ± 472
	CloseDrawer	7,087 ± 87	11,490 ± 483	11,030 ± 138
Iterations	PickCube	1.0 ± 0.00	1.0 ± 0.00	1.0 ± 0.00
	PlaceCubeDrawer	1.0 ± 0.00	1.0 ± 0.00	1.0 ± 0.00
	StackCube	1.8 ± 0.75	1.6 ± 0.80	1.6 ± 1.20
	OpenDrawer	1.4 ± 0.49	1.2 ± 0.40	1.4 ± 0.80
	CloseDrawer	1.2 ± 0.40	1.0 ± 0.00	1.2 ± 0.40
Wall-clock (s)	PickCube	24 ± 3	56 ± 7	46 ± 5
	PlaceCubeDrawer	44 ± 4	68 ± 9	57 ± 6
	StackCube	57 ± 3	94 ± 6	75 ± 3
	OpenDrawer	73 ± 5	46 ± 4	65 ± 8
	CloseDrawer	45 ± 5	48 ± 6	38 ± 4

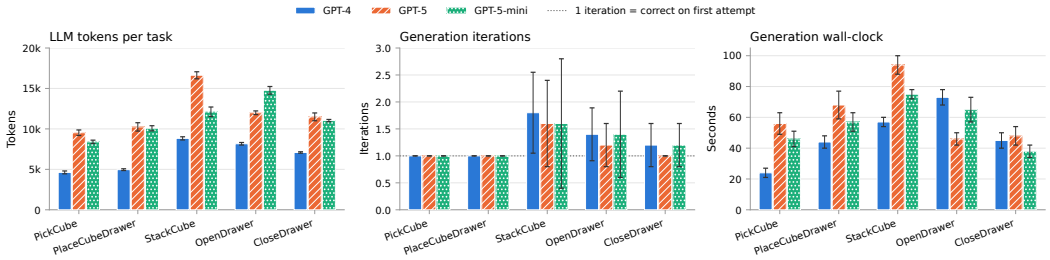

Figure 10: LLM-backbone ablation on five ManiSkill2 tasks (mean ± std over 5 seeds): LLM tokens spent, generation iterations (1 means correct on the first generation), and generation wall-clock per task. The GPT-4 iteration counts are those reported in Fig. 6.

the first attempt and that feedback repair was infrequent across all three models. Backbone choice primarily affects the one-time generation cost: GPT-5 and GPT-5-mini use approximately 1.78× and 1.67× as many tokens as GPT-4, respectively, while mean generation latency stays below 95 seconds per task for every backbone.

F.4 Computational Cost

We report the full per-task cost of L2S and Text2Reward (mean ± std over seeds) in Table 7. The columns are: **Env steps**, the environment transitions consumed (one shared interaction budget, identical for both methods by construction since budgets are totals split across subtasks); **L2S / T2R tokens**, the LLM tokens spent generating the reward and termination programs; and **L2S / T2R (h)**, the total training wall-clock in GPU-hours.

Across the ten tasks, L2S and Text2Reward receive the same total environment-interaction budget of 23.0 million transitions. For L2S , this total includes both SAC data collection and the environment steps taken in the CEM candidate-evaluation rollouts. The per-task budget is divided across subtask-policy training and parameter optimization, whereas Text2Reward uses the same total budget to train a single policy. L2S consumes 55.1k LLM tokens, compared with 23.4k for Text2Reward, a 2.35× increase. This additional 31.7k tokens across the suite (approximately 3.2k per task) is the cost of generating a decomposition and separate reward and termination functions for each subtask; Text2Reward generates only one task-level reward function. Under this matched interaction budget, L2S requires 50.39 ± 3.90 GPU-hours, compared with 34.81 ± 1.96 for Text2Reward, a 1.45× increase. This training-time overhead reflects parameter-conditioned subtask learning, policy composition, and CEM-based interface optimization.

Table 7: Per-task cost of L2S and Text2Reward (T2R), mean ± std over seeds. Env steps: environment transitions consumed (a shared budget, identical for both methods). Tokens: LLM tokens spent generating the reward and termination programs. (h): total training wall-clock in GPU-hours. MS2 denotes ManiSkill2.

Task	Env steps	L2S tokens	T2R tokens	L2S (h)	T2R (h)
MS2 OpenDrawer	2.0×10^6	8,163 ± 157	2,066 ± 21	4.29 ± 0.89	3.83 ± 0.69
MS2 CloseDrawer	2.0×10^6	7,087 ± 87	2,045 ± 67	3.26 ± 0.44	1.35 ± 0.35
MS2 PickCube	2.5×10^6	4,624 ± 177	2,185 ± 26	8.79 ± 1.88	4.06 ± 0.50
MS2 StackCube	4.0×10^6	8,831 ± 196	2,230 ± 16	15.35 ± 2.77	6.26 ± 0.76
MS2 PlaceCubeDrawer	2.5×10^6	4,963 ± 110	2,578 ± 57	6.16 ± 0.60	7.94 ± 0.72
LORL OpenDrawer	2.0×10^6	4,502 ± 176	2,441 ± 11	4.00 ± 0.76	3.24 ± 0.79
LORL TurnFaucetLeft	2.0×10^6	4,213 ± 36	2,477 ± 42	2.01 ± 0.89	1.86 ± 0.82
LORL TurnFaucetRight	2.0×10^6	4,284 ± 22	2,415 ± 15	2.28 ± 0.32	1.89 ± 0.54
LORL PushMugBack	2.0×10^6	4,186 ± 13	2,471 ± 51	2.16 ± 0.40	2.02 ± 0.49
LORL PushMugLeft	2.0×10^6	4,236 ± 68	2,516 ± 39	2.09 ± 1.01	2.36 ± 0.23
Total	$\mathbf{23.0 \times 10^6}$	**55,089 ± 389**	**23,424 ± 124**	**50.39 ± 3.90**	**34.81 ± 1.96**

For context, Eureka's procedure [Ma et al., 2023] trains eight reward candidates in each of three sequential rounds, corresponding to 24 policy-training runs per task. Applying this procedure to the same ten-task suite and per-policy interaction budgets requires approximately 552 million environment transitions and 1,209 GPU-hours. Even though the eight candidates within each round were trained in parallel on eight GPUs, the three sequential rounds require approximately 151 hours. On LORL, our League++ runs (Appendix F.9) require 1.9–3.0 GPU-hours per task, which is of the same order as L2S 's 2.0–4.0 GPU-hours (for example, 2.81 versus 2.01 GPU-hours on TurnFaucetLeft).

F.5 Optimizing reward and termination function parameters with various variances.

As illustrated in Sec. 3.2, the policy training objective in L2S is for the policy $\pi_{o_i}(a|s;\phi_i)$ to maximize the expected rewards over a broad range of parameter values $\phi_i \sim \tilde{q}_{\phi_i}$, ensuring robust performance across varying conditions. The parameter distribution $\tilde{q}_{\phi_i}$ for ϕ_i is configured by the user. For example, one can set $\tilde{q}_{\phi_i}$ as a Gaussian distribution $N(v_{\phi_i}, \sigma)$, where v_{ϕ_i} is the mean centered at the LLM agent's inferred parameter values for ϕ_i, and σ is the user-defined variance. As we provided information about the environment and additional knowledge that connects the semantics of real-world instructions to the robot environment and specifies the task's successful conditions (see LLM prompt section in Appendix H), the LLM gains some understanding of the environment's scale and selects reasonable (though not necessarily optimal) parameter mean values v_{ϕ_i}. By default, we set the variance σ of the parameters to be twice the maximum mean value generated by the LLM for the current task (a heuristic). We conducted experiments with varying alternative parameter value variances, while keeping the parameter mean value fixed. The results were obtained using three different random seeds and demonstrate that L2S consistently achieves optimal parameter values across the default setting and all tested variants.

LORL-Turn faucet left

For this task, we examined three different parameter variance combinations—variants 1, 2, and 3—to analyze their effects on reward and termination parameters. The first subtask in the task is trained to position the end effector around the faucet handle, with the reward function parameter defining the acceptable distance to the handle. The termination condition parameter specifies how close the end effector must be to the target position to transition to the next subtask. Results are shown in Table 8.

- **Variant 1**: The variance of the termination condition parameter is increased.
- **Variant 2**: The variance of the reward function parameter is increased.
- **Variant 3**: The variance of both parameters is increased.

ManiSkill2-Open drawer

For this task, the parameter in the termination condition of the first subtask specifies the required proximity of the robot's end effector to the target position above the drawer handle. In the variant, we

Table 8: Optimizing parameters with different variances in *LORL-turn faucet right* task. The upper table shows the parameters in Reward functions and the lower table shows the parameters in Termination functions.

Reward Func Parameters	Initial Parameters Mean Value	Initial Parameters Variance	Optimized Params/(Std)	Success Rate/(std)	Training Cost (Timesteps)
Default	[0.01]	[0.2]	[0.107]/(0.02)	0.99/(0.01)	1e6
Variant1	[0.01]	[0.2]	[0.118]/(0.04)	0.89/(0.13)	1.5e6
Variant2	[0.01]	**[0.5]**	[0.16]/(0.01)	0.97/(0.04)	1e6
Variant3	[0.01]	**[0.5]**	[0.114]/(0.01)	0.93/(0.05)	1e6
Termination Func Parameters	Initial Parameters Mean Value	Initial Parameters Variance	Optimized Params/(Std)	Success Rate/(std)	Training Cost (Timesteps)
Default	[0.01]	[0.02]	[0.013]/(0.001)	0.99/(0.01)	1e6
Variant1	[0.01]	**[0.05]**	[0.027]/(0.005)	0.89/(0.13)	1.5e6
Variant2	[0.01]	[0.02]	[0.015]/(0.0003)	0.97/(0.04)	1e6
Variant3	[0.01]	**[0.05]**	[0.031]/(0.001)	0.93/(0.05)	1e6

increase the variance of this parameter from the default value of 0.02 to 0.05 to evaluate its impact on performance. Results are shown in Table 9.

Table 9: Optimizing parameters with different variances in *ManiSkill2-Open drawer*.

Reward Func Parameters	Initial Parameters Mean Value	Initial Parameters Variance	Optimized Params/(Std)	Success Rate/(std)	Training Cost (Timesteps)
Default	[0.01]	[0.02]	[0.026]/(0.003)	0.94/(0.06)	1e6
Variant	[0.01]	**[0.05]**	[0.031]/(0.006)	0.97/(0.02)	1.5e6

F.6 Training progress evaluation

As summarized in Fig. 3, L2S significantly outperforms the baseline T2R on challenging tasks like Stack Cube. To further illustrate this performance, we present the training curves of both L2S and T2R in Fig. 11 for the PickCube and StackCube tasks. We utilized functions from the ManiSkill2 library to design evaluation functions that assess the progress of the learning agent in achieving specific key subgoals of each task. In the case of StackCube, the evaluation function gauges whether the agent has consistently mastered the abilities to grasp, lift, and stack a cube. Although T2R can generate step-wise reward functions, combining rewards across different steps proves to be difficult. In the case of PickCube or StackCube, a high grasping reward combined with a relatively lower stacking reward leads the policy to prioritize holding the cube, as ineffective stacking actions can easily result in losing contact with the cube, thus yielding a lower reward. T2R also requires significantly more training steps than L2S to achieve convergence (Fig. 5), as it learns a single monolithic policy that lacks the flexibility for easy reuse. In contrast, Fig. 11 demonstrates that L2S quickly acquires the ability to grasp a cube in StackCube by effectively reusing the grasp subtask learned during the PickCube task. Eureka faces similar challenges as T2R in generating effective step-wise reward functions, with performance declining as the complexity of the required reward functions increases.

F.7 Examples of Subtask Generation Improvement via Feedback

To illustrate the contribution of the feedback loop improving subtask generation, we take two specific subtasks generation improvement procedures from ManiSkill2 as examples.

Subtask "Move the cube above the other cube" in StackCube task

This subtask requires the end-effector to move cube A to a position above cube B while maintaining a stable grasp. **Initially**, the generated subtask moved cube A directly toward cube B using the reward term "`norm(cubeA.pos - cubeB.pos)`". However, this formulation did not ensure that cube A stayed stably above cube B.

After one feedback iteration, as shown in Figure 12, the reward term was refined to "`norm(cubeA.pos[:2] - cubeB.pos[:2]) + (cubeA.pos[2] - (self.cubeB.pos[2] + params_value))`". This improved term not only aligns the two cubes in the xy-plane, but

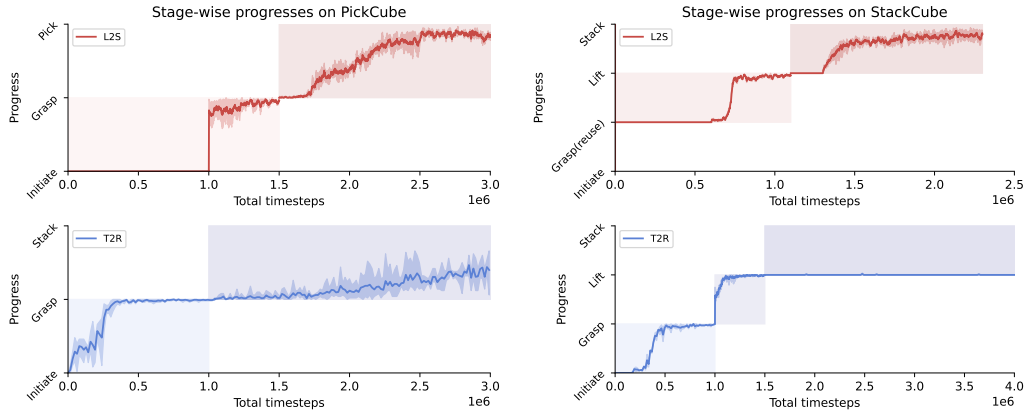

Figure 11: Left: the progress of training the task "PickCube" in ManiSkill2 measured by 2 stage-wise evaluation functions: **(1)grasp cube and (2) place cube**. Right: the progress of training the task "Stackcube" in ManiSkill2 measured by 3 stage-wise evaluation functions: **(1)grasp cube, (2) lift cube, (3) stack cube**. In sequential task learning, L2S reuses the subtasks for grasping cubes from "PickCube" in training "StackCube".

```
## Before feedback reflection
def compute_dense_reward(action, obs, params) -> float:
    tolerance = 0.01
    reward = 0.0
    if np.linalg.norm(cubeA.pos[:2] - cubeB.pos[:2]) <= tolerance:
        reward += 1.0
    reward -= 0.1 * np.linalg.norm(action)
    return reward
```

```
## After feedback reflection
def compute_dense_reward_skill_3(action, obs, params) -> float:
    reward = np.linalg.norm(cubeA.pos[:2] - cubeB.pos[:2])
        + (cubeA.pos[2] - (cubeB.pose.p[2] + params[1]))
    reward -= 0.1 * np.linalg.norm(action)
    return reward
```

Figure 12: Comparison of reward function of skill "Move the cube above the other cube" before and after the LLM feedback reflection.

also allows the height difference between them to be adjusted explicitly. Consequently, the release height of cube A onto cube B can be controlled more precisely, resulting in a higher success rate.

Subtask "Move the robot handle to open the drawer" in OpenDrawer task

This subtask controls the robot end-effector to pull the grasped cabinet drawer handle, thereby opening the drawer. **Initially**, the generated subtask assigned only a discrete reward value, `-1.0 if cabinet_handle.pos < target_pos else 0`, to the reward term `drawer_open`. Compared with a continuous reward, this formulation may provide a weaker learning signal for RL policy optimization.

After one feedback iteration, as shown in Figure 15, the reward term was refined to `-abs(cabinet_handle.pos - target_pos)`. By replacing the discrete reward with a continuous distance-based term, the revised formulation provides denser feedback, which can improve both learning efficiency and final policy performance.

F.8 Experimental results of Self-Alignment [Zeng et al., 2024]

To match training cost, we use the same number of training timesteps as in our ManiSkill2 experiments. By contrast, the Self-Alignment paper reports results obtained with `1e5` steps per iteration and more than `1e7` steps in total. Under the substantially smaller training budget used in our setting, this method does not perform well.

```
You are improving the reward and termination for ONE skill.
Original skill specification:
{
  "skill_id": 3,
  "name": "Move cube A above cube B",
  "purpose": "The purpose of this skill is to control the end-effector motion to move cube A above cube B while
  maintaining the grasp. The target is to align cube A directly above cube B. The gripper should maintain the
  grasp on cube A during the motion.",
  "success_condition": "The success condition is when the x and y coordinates of cube A's position are the same
  as cube B's, which can be checked using np.linalg.norm(self.cubeA.pose.p[:2] - self.cubeB.pose.p[:2]) <= tolerance."
}

Current reward terms:
    ...
Current reward assembly:
    ...
Current termination:
    ...
Training evidence:
- Stagnant terms:
['']
- Non-stagnant terms:
All original terms not listed above. These MUST also be audited. Non-stagnant does NOT mean correct.

Goal:
Produce the smallest necessary set of changes that makes the reward and termination correct, aligned with the intended
skill, and useful for learning.

MANDATORY reasoning order (follow internally, do NOT output reasoning):
1) Audit the implemented skill AGAINST the current skill specification FIRST:
    ...
2) If step 1 finds that the implemented skill DOES correctly match the current skill specification, you must first
diagnose why the current specification is still inadequate for RL policy training:
    ...
3) After that diagnosis, regenerate the skill specification:
    ...
...
```

Figure 13: Partial prompt for Subtask "Move the cube above the other cube". Full prompt can be found in Figure 24.

Table 10: Self-Alignment

ManiSkill2 task	Pick Cube	Open Drawer
Self-Alignment	$0.01(0.01)$	$0.12(0.03)$
Ours	$0.87(0.02)$	$0.94(0.05)$

F.9 Experimental results of League++ [Li et al., 2024]

Metric functions used in League++ To ensure a fair comparison, we restrict the LLM to choose from a set of metric functions similar to those described in their paper; the full list used in our implementation is provided in Table 11.

Table 11: Metrics functions picked in League++

Metrics Function	**Definition**
dis_to_obj	Distance between the gripper and the object
perpendicular_dis	Distance between object and gripper along the normal line
in_grasp	If the object is grasped by the gripper
drawer_opened	(LORL) The drawer is opened enough or not.
faucet_turned	(LORL) The faucet handle is turned away enough or not.
mug_pushed	(LORL) The mug is pushed away enough or not.
cube_placed	(ManiSkill2) The cube is placed in the drawer or not.

Experimental results comparing League++ with L2S As shown in Table 12, League++'s reliance on predefined Metric Functions may limit its ability to capture the full complexity of tasks or appropriately penalize suboptimal behavior, potentially reducing overall performance.

````
Block 1: audit skill implementation
```python
# Skill name: Matches the implementation. The skill is about moving cube A above cube B.
# Purpose: Matches the implementation. The purpose is to control the end-effector motion to move cube A above
cube B while maintaining the grasp.
# Success condition: Partially matches the implementation. The success condition is when the x and y coordinates of
cube A's position are the same as cube B's. However, the implementation does not consider the z coordinate (height),
which is essential for the cube A to be "above" cube B.
# Overall: The implemented skill partially matches the current skill specification. The z coordinate (height) is not
considered in the success condition.
```

Block 2: improve and regenerate skill template if necessary
```json
{
  "diagnosis": {
    "required": true,
    "why_inadequate_for_rl": "The original specification is inadequate for RL policy training because it does not
    consider the z coordinate (height) in the success condition. This can lead to a policy that aligns cube A with
    cube B in the x and y coordinates but not necessarily above cube B.",
    "allowed_failure_mode": "The policy could place cube A at the same x and y coordinates as cube B but at a lower
    or same height, which does not satisfy the 'above' condition.",
    "too_weak_or_incomplete_part": "The success condition is too weak as it does not consider the z
    coordinate (height).",
    "substantive_change_made": "The success condition is now updated to include the z coordinate (height).",
    "why_not_semantically_identical": "The new specification requires cube A to be not only aligned with cube B in
    the x and y coordinates but also to be above cube B in the z coordinate."
  },
  "final_skill_spec": {
    "skill_id": 3,
    "name": "Move cube A above cube B",
    "purpose": "The purpose of this skill is to control the end-effector motion to move cube A above cube B while
    maintaining the grasp. The target is to align cube A directly above cube B in all three dimensions. The gripper
    should maintain the grasp on cube A during the motion.",
    "success_condition": "The success condition is when the x, y, and z coordinates of cube A's position are such that
    cube A is directly above cube B. This can be checked using np.linalg.norm(self.cubeA.pose.p[:2] -
    self.cubeB.pose.p[:2]) <= tolerance and self.cubeA.pose.p[2] > self.cubeB.pose.p[2]."
  }
}
```

Block 3: generate the functions
```python
    ...
```
````

Figure 14: Partial response for Subtask "Move the cube above the other cube".

```python
## Before feedback reflection
def compute_dense_reward(action, obs, params) -> float:
    # Penalty for not reaching target drawer position
    drawer_penalty = -1.0 if link_qpos < open_target_qpos else 0.0
    # Action regularization
    action_reg = -0.1 * np.linalg.norm(action)
    # Reward is drawer penalty plus action regularization
    reward = drawer_penalty + action_reg
    return reward
```

```python
## After feedback reflection
def compute_dense_reward (action, obs, params) -> float:
    # Reward is negative distance to target drawer position plus action regularization
    reward = -np.abs(cabinet_handle.qpos - drawer_open_target_qpos) - 0.1 *
    np.linalg.norm(action)
    return reward
```

Figure 15: Comparison of reward function of skill "Move the robot handle to open the drawer" before and after the LLM feedback reflection.

G Task list

In this section, we list all tasks examined in both LORL and ManiSkill2 benchmarks separately in Table. 13 and Table. 14, accompanied by their corresponding natural language instructions. Note that these instructions constitute part of the task prompt explicitly.

H LLM prompt

A prompt used in L2S consists of following components: *introduction*, *environment description*, *additional environmental knowledge*, *tips and tricks*, *instruction hint*, and *learned subtask library*. Here we use an example of the prompt for ManiSKill2 manipulation tasks to demonstrate how each component is formatted:

```
You are improving the reward and termination for ONE skill.

Original skill specification:
{
  "skill_id": 3,
  "name": "Open Drawer",
  "purpose": "This skill controls the end-effector motion of the mobile robot to pull the grasped cabinet drawer handle,
  thereby opening the drawer. The robot involved is the MobileRobot. The target is the cabinet drawer. No tunable
  geometric parameter is needed for this skill.",
  "success_condition": "The success condition is when the qpos of the cabinet drawer is larger than the drawer open
  target qpos."
}

Current reward terms:
    ...
Current reward assembly:
    ...
Current termination:
    ...
Training evidence:
- Stagnant terms:
['drawer_open']
- Non-stagnant terms:
All original terms not listed above. These MUST also be audited. Non-stagnant does NOT mean correct.

Goal:
Produce the smallest necessary set of changes that makes the reward and termination correct, aligned with the intended
skill, and useful for learning.
...
```

Figure 16: Partial prompt for Subtask "Move the robot handle to open the drawer". Full prompt can be found in Figure 24.

````
```python
# Block 1: Audit skill implementation

# Skill name: Matches the implemented skill.
# Purpose: Matches the implemented skill.
# Success condition: Matches the implemented skill.
# Overall: The implemented skill matches the current skill specification.
```

```json
{
  "diagnosis": {
    "required": false
  },
  "final_skill_spec": {
    "skill_id": 3,
    "name": "Open Drawer",
    "purpose": "This skill controls the end-effector motion of the mobile robot to pull the grasped
    cabinet drawer handle, thereby opening the drawer. The robot involved is the MobileRobot. The target
    is the cabinet drawer. No tunable geometric parameter is needed for this skill.",
    "success_condition": "The success condition is when the qpos of the cabinet drawer is larger than
    the drawer open target qpos."
  }
}
```


```python
# Block 3: generate the functions
    ...
```
````

Figure 17: Partial response for Subtask "Move the robot handle to open the drawer".

Subtasks library prompt. A complete L2S prompt is the ordered concatenation of the above components. Additionally, we could also ask the LLM to generate response considering reusing given library of subtasks (in listing 23). Such a prompt makes it possible for L2S to reuse either subtasks generated by language model or reference subtasks given by human experts in the code generation process, thus potentially facilitate the subtask discovery and training.

Iterative subtask improvement prompt. The prompt consists of two stages. First, it audits whether the reward and termination function implementations correctly and completely realize the current skill description; if not, it revises the implementation. Second, if the implementation is already consistent with the skill description, it checks whether the subtask specification itself is incomplete, ambiguous, or insufficient for RL training, and refines the specification accordingly. The full prompt is provided here.

Table 12: Comparison of success rates for different tasks between League++ and L2S.

Tasks \Success Rate(Std)	**League++**	**L2S**
LORL-Open Drawer	0.83 (0.06)	**0.95** (0.02)
LORL-Turn Faucet Left	0.58 (0.15)	**0.99** (0.01)
LORL-Push Mug Back	0.13 (0.13)	**0.96** (0.04)
ManiSkill2-Place Cube Drawer	0.78 (0.07)	**0.92** (0.07)

Table 13: List of tasks in LORL

Single-goal Task	Instruction
Push mug backward	Move the white mug backward 0.1 meter.
Push mug left	Move the white mug left 0.1 meter.
Turn faucet left	Turn the faucet handle left $\frac{\pi}{4}$ radian distance.
Turn faucet right	Turn the faucet handle right $\frac{\pi}{4}$ radian distance.
Open drawer	Open the drawer until the position of drawer box is greater than target value.

Multi-goal Task
Push mug backward and open drawer.
Open drawer and turn faucet left.
Push mug backward and turn faucet left.
Push mug backward and open drawer and turn faucet left.

Table 14: List of tasks in ManiSkill2.

Task	Instruction
Pick cube	Pick up cube A, move it to goal position and hold it.
Stack cube	Pick up cube A and place it on top of cube B.
Open cabinet drawer	A single-arm mobile robot needs to open a cabinet drawer.
Close cabinet drawer	A single-arm mobile robot needs to close a cabinet drawer.
Open drawer, Place cube and close drawer	Open the cabinet drawer, place cube it into the drawer and close the drawer.

I Examples of reward functions and termination condition functions

In this section, we provides example pairs of generated reward functions and termination functions generated by LLM to solve given tasks. Each figure includes one dense reward function and its corresponding termination function to constitute a complete subtask. Numbers in the suffix of the function names denote the indices of the subtasks.

Turn faucet left. Fig. 25 and 26 show the generated subtasks. In this task, the language model proposes a simple yet effective two-stage solution. Specifically, the first subtask is responsible for aligning the robot arm end-effector to a 3D position on the right of the faucet handle. Then the second subtask moves robot arm end-effector and turn the faucet handle left and finally solve the task.

Open drawer. Fig. 27, 28 and 29 demonstrate the three subtass in the open drawer task decomposition. Concretely speaking, the response from LLM is:

Subtask 1: Move the robot arm end-effector to a 3D position above the drawer box handle.

Subtask 2: Align robot arm end-effector with the drawer box handle.

Subtask 3: Move robot arm end-effector to open the drawer.

The sequence for accomplishing the task could be: Subtask 1 → Subtask 2 → Subtask 3.

Stack cube. The stack cube task in the ManiSkill2 is one of the most complicated task in our experiments. To solve this task, the LLM provides a chain of subtasks from subtask 0 all the way to subtask 4 (see Fig. 30, 31, 32, 33 and 34). The corresponding response from LLM is:

The subtasks and sequence of subtasks for accomplishing each simple task are:

- Subtask 1: Navigate gripper to cubeA

```
You are an expert in robotics, reinforcement learning, task decomposing, task planning and code
generation. We are going to control a robotic arm to complete some given tasks. The robotic arm is a
7-DoF Fetch Mobile Manipulator with a two-fingered parallel gripper. The robotic arm is controlled by
small displacements of the gripper in Cartesian coordinates and the inverse kinematics are computed
internally by the MuJoCo framework. The gripper can be opened or closed in order to perform the
grasping operation of pick and place.

The action space of the robot is Box(np.array([-1, -1, -1, -np.pi, -1]), np.array([1, 1, 1, np.pi, 1]),).

| Num  | Action                                                              |  Unit         |
| ---- | ------------------------------------------------------------------- |  ------------ |
| 0    | Displacement of the end effector in the x direction dx              |  position (m) |
| 1    | Displacement of the end effector in the y direction dy              |  position (m) |
| 2    | Displacement of the end effector in the z direction dz              |  position (m) |
| 3    | Angular displacement of the end effector                            |  position (m) |
| 4    | Positional displacement per timestep of each finger of the gripper|  position (m) |

Now I want you to help me
1) decompose the robotic task into sequences of subtasks
2) write dense reward functions and termination conditions of reinforcement learning for each subtask.
3) make plan on the subtasks to finish the robotic task.

I'll give you the attributes of the environment and robotic arm itself. You can use these class attributes
to write the reward function.
```

Figure 18: Introduction.

```
The following classes provide the information about the robotic arm and all objects in the environment.

class BaseEnv(gym.Env):
    self.robot : SawyerRobot # the robot in the environment
    self.white_mug : MugObject # the white mug in the environment
    self.black_mug : MugObject # the black mug in the environment
    self.faucet : FaucetObject # the faucet object in the environment
    self.drawer : DrawerObject # the drawer object in the environment

class SawyerRobot:
    self.ee_position : np.ndarray[(3,)] # indicate the 3D position of the end-effector
    self.gripper_finger_distance : numpy.float64
            # indicate the distance between the gripper fingers away from the initial position
            # range between 0 and 0.1.
            # The closer the grippers, the smaller the value
    self.init_ee_position : np.ndarray[(3,)] # indicate the initial 3D position of the end-effector
    self.init_gripper_finger_distance : numpy.float64
            # indicate the initial distance between the gripper fingers away from the initial position,
            # range between 0 and 0.1

class MugObject:
    self.position : np.ndarray[(3,)] # indicate the 3D position of the rigid object
    self.init_position : np.ndarray[(3,)] # indicate the initial 3D position of the rigid object

class FaucetObject:
    self.faucet_handle_postion : np.ndarray[(3,)] # indicate the 3D position of the handle of faucet
    self.faucet_handle_angular_position : numpy.float64
            # indicate the angular position of the handle with respect to the faucet in radians.
            # Faucet moving clockwise makes this value smaller.
    self.init_faucet_handle_postion : np.ndarray[(3,)]
            # indicate the initial 3D position of the handle of faucet
    self.init_faucet_handle_angular_position : numpy.float64
            # indicate the initial angular position of the handle with respect to the faucet in radians.
            # Faucet moving clockwise makes this value smaller.

class DrawerObject:
    self.box_handle : np.ndarray[(3,)] # indicate the 3D position of the handle of drawer box
    self.drawer_box_position : numpy.float64 # indicate the 1D relative position of the drawer box.
                                             # The position range is between [-0.16, 0] meter.
    self.init_box_handle : np.ndarray[(3,)] # indicate the initial 3D position of the handle of drawer box
    self.init_drawer_box_position : numpy.float64 # indicate the initial 1D relative position of
                                                  # the drawer box
```

Figure 19: Environment description.

- Subtask 2: Grasp cubeA

- Subtask 3: Lift cubeA in the air

- Subtask 4: Move cubeA to the position above cubeB

- Subtask 5: Release cubeA

The sequence for b the task could be: Subtask 1 → Subtask 2 → Subtask 3 → Subtask 4 → Subtask 5

```
Additional knowledge:
1. For the robotic arm gripper and all the objects in the environment, the direction words in the
following task are defined as:
    1) "Left" or "right" means towards the positive or negative x-axis with respect to the related object
    position, respectively. X-axis is corresponding to the first value in the 3D position with form
    "np.ndarray[(3,)]". For example, x-axis of mug is "mug.position[0]".
    2) "Forward/Front" or "backward/Back" means towards the positive or negative y-axis with respect to
    the reference object position, respectively. Y-axis is corresponding to the second value in the
    3D position with form "np.ndarray[(3,)]". For example, y-axis of a mug is "mug.postion[1]".
    3) "Above" or "below" means towards the positive or negative z-axis with respect to the reference
    position, respectively. Z-axis is corresponding to the third value in the 3D position with
    form "np.ndarray[(3,)]".
    For example, z-axis of mug is "mug.position[2]".
    4) Specially, for the object faucet, "turn left" or "turn right" means turning faucet that increases
    or decreases the faucet handle angular position.
2. In order to compare the replative positions of different items in the environment, including
the robotic arm gripper and all the objects, you must first identify the attributes that represents
the 3D-positions, and then use these attributes for computation. In practice, the relative position
words in the following task are defined as:
    1) "One item is on the left or on the right of the other item" means the item is on the positive or
    negative x-axis direction with respect to the other item, respectively. X-axis is corresponding
    to the first value in the 3D position with form "np.ndarray[(3,)]".
    2) "One item is in front of or at the back of the other item" means the item is on the positive or
    negative y-axis direction with respect to the other item, respectively. Y-axis is corresponding
    to the second value in the 3D position with form "np.ndarray[(3,)]".
    3) "One item is above or below the other item" means the item is on the positive or negative z-axis
    direction with respect to the other item, respectively. Z-axis is corresponding to the third value
    in the 3D position with form "np.ndarray[(3,)]".
3. Tasks about moving mug are considered successful when mug is moved at least 0.1 meter towards the
correct direction compared with the object's initial position.
4. Tasks about turning faucet are considered successful when faucet is turned at least np.pi/4 radian
towards the correct direction compared with the object initial position.
5. Tasks about opening or closing drawer are considered successful when drawer box is fully open or
fully closes. Drawer fully open means drawer box position is smaller than -0.15 meter.
Drawer fully closed means drawer box position is greater than -0.01 meter.
```

Figure 20: Additional environment knowledge.

```
Task to be fulfilled: {instruction}.
Here is the instruction:
Please think step by step and finish the following requirements one by one in order:
    1. Tell me what does this task mean. If it is a complex task, identify how many simple task you can
    identify.
    2. Decompose a whole task into a set of possible subtasks and plan on the subtasks to finish each simple
    task. You can refer to the above examples if provided after the intruction part.
    3. Identify which example you are referring to, if any.
    4. Identify the index of subtask that terminate each simple task as you have answered above. Save the
    index of subtask in "simple_task_termination_subtask = [...]"
    5. For each subtask, design a pair of dense reward function and terminition condition function based
    on the purpose of the subtask. Write down the pair of functions one by one with the following format:
        1) Make each pair of reward function and termination function a separate python code piece
        "```python ```".
        2) Dense reward function is used in reinforcement learning, here are the requirements:
            a. Create a list "params = [...]" containing extra parameters that never exist for computing
            reward (if any). But you should not include any threshold value, reward term weight
            or attributes that already exist in the above environment information in the list "params",
            e.g, termination threshold value, reward term weight, the position information of any item.
            Make sure every parameter in list "params = [...]" is used in the dense reward function.
            b. Define the reward term one by one and explain the purpose of each reward term as comment.
            c. This function starts with `def compute_dense_reward_subtask_NUM(self, action, obs) -> float`.
            It only returns variable `reward : float`.  Replace 'NUM' with the number of subtask.
        3) Terminition condition function decides whether the subtask is successful, here are the
        requirements:
            a. Copy list "params = [...]" from dense reward function and paste it into the terminition
            condition function. Any value in list "params" shuold not be used as termination threshold.
            b. Create a list "t_params = [...]" containing the value used as termination threshold if the
            corresponding subtask is not in "simple_task_termination_subtask". Make sure every parameter
            in list "t_params = [...]" is used later in terminition condition function.
            c. Make lists "params = [...]" and "t_params = [...]" don't conflict with each other because
            they are used for different purposes.
            d. This function starts with  `def termination_subtask_NUM(self, obs) -> bool`. It only returns
            variable `done : bool`. Value of `done : bool` should be decided before it is returned.
            Replace 'NUM' with the number of subtask.
```

Figure 21: Instruction hint.

```
Instances of Few-shot Examples:
1.Task to be fulfilled: Turn an object with a handle left.
Corresponding subtasks and sequence of subtasks for accomplishing the task:
  Subtask 1: Align the robot arm end-effector to a 3D position on the right of the object handle with
  some offset.
  Subtask 2: Move robot arm end-effector and turn the object handle left.
  The sequence for accomplishing the task could be: Subtask 1 -> Subtask 2.
2.Task to be fulfilled: In the MuJoCo PickAndPlace environment, pick up a box and move it to the 3D
goal position and hold it there.
Corresponding subtasks and sequence of subtasks for accomplishing the task :
  Subtask 1. Navigate gripper to the box.
  Subtask 2. Grasp the box and move the box to the goal position and hold it.
  The sequence for accomplishing the task could be: Subtask 1 -> Subtask 2.
```

Figure 22: Few-shot example.

```
After finishing the job above, I have one more job for you.
Now we have a subtasks library which store the already trained subtasks in a list format, each element in the
list mapping  the stored subtasks number and the discription of the subtasks.
subtasks_library=[
    "1":"Navigate to the position on the left side of the faucet handle and keep some distance away
    from it. This subtask has paramters as safe distance between gripper and faucet handle that can
    be tuned.",
    "2":"Navigate gripper to the intermediate position on the forward direction of the white mug
    and maintain a safe distance from the white mug.",
    "3":"Move gripper and push the faucet to the right. This subtask can not be modified.",
    ]

What you should do is:
    1) First, think the meaning of each subtasks. You should clarify the following attributes of each subtask:
    the objects involved in the subtask, the relation between objects and the goal that the subtask finally
    should achieve.
    2) Then, with the decomposed subtask of current task and your understanding of the subtasks in subtask
    library, please think about which subtask in the library do the exact same work as some subtask(s) in
    current task and can be reused.
    3) Lastly, Please give back the pair-wise mapping from current subtask number to the subtask number in
    skills library with python JSON format. Each pair of subtask models selected must have the same
    attribute. That's means even if two subtasks are very similar but not the same, you should not select
    them because it need extra training. And you should explain the reason why you make such pairing.
```

Figure 23: Subtasks library prompt.

```
You are improving the reward and termination for ONE skill.

Original skill specification:
{{
  "skill_id": {i},
  "name": "{name}",
  "purpose": "{purpose}",
  "success_condition": "{success_condition}"
}}
Current reward terms:
{reward_terms_code}
Current reward function:
{reward_code}
Current termination:
{termination_code}
Training evidence:
- Stagnant terms:
{stagnant_table}
- Non-stagnant terms:
All original terms not listed above.
These MUST also be audited. Non-stagnant does NOT mean correct.

Goal:
Produce the smallest necessary set of changes that makes the reward and termination correct, aligned with the
intended skill, and useful for learning.
1) Audit the implemented skill AGAINST the current skill specification FIRST:
   - Check whether the current reward terms, reward function, and termination together correctly
   reflect the original skill specification.
   - This first-step audit is only about specification-to-implementation consistency: determine
   whether the implemented skill matches the current specification.
   - Use the stagnant and non-stagnant evidence only as supporting diagnostic evidence during this audit.
   - Ask: does the implemented skill, as defined by the current reward and termination, actually correspond
   to the current skill name, purpose, and success condition?
   - If the skill name, purpose, and success condition are not equally specific or do not fully agree, resolve
   the audit using the MOST behaviorally complete interpretation of the skill, not the weakest or most
   underspecified phrasing.
2) If step 1 finds that the implemented skill DOES correctly match the current skill specification, you must
first diagnose why the current specification is still inadequate for RL policy training:
   - Explicitly identify the weakness in the current skill name, purpose, or success condition.
   - State whether the weakness is due to underspecification, partial-proxy success, missing maintained
   constraints, weak trainability, behavioral incompleteness, or another concrete RL-relevant flaw.
   - State what unintended policy behavior would still be allowed by the current specification.
3) After that diagnosis, regenerate the skill specification:
   - Improve and regenerate the skill name, purpose, and success condition into a corrected version that
   is appropriate for RL policy training.
   - Treat the matched current specification as still insufficient: even if it is faithfully implemented,
   it is still the wrong skill for RL policy training.
   - The regenerated specification should preserve as much useful intent as possible, but must fix poor scope,
   weak trainability, behavioral uselessness, or other RL-relevant flaws.
   - When the original skill name, purpose, and success condition conflict, the regenerated specification MUST
   follow the more behaviorally complete interpretation of the intended skill, not the most weakly stated one.
   - Complete this regeneration before auditing individual reward terms in detail.
4) Audit ALL original reward terms under the final skill specification:
   - For EACH term, decide whether it is correct, aligned, necessary, and validly measured.
   - Check for sign bugs, wrong target construction, saturation, sparsity, bad scaling, redundancy,
   unreachable objectives, unavailable state, and mismatch with success condition.
   - Check whether the reward assembly combines terms correctly.
5) Make MINIMAL NECESSARY CHANGES:
   - Keep any original term that is already correct and useful.
   - Modify only the terms that are flawed, misaligned, redundant, too sparse, saturated, or badly scaled.
   - Remove a term only if it is truly incorrect or unnecessary.
   - Add a new term only if needed to provide missing dense progress signal or to fix a real gap.
   - Do not rewrite everything unless a large rewrite is clearly necessary.
   - If the retained reward terms are still too sparse for learning (e.g., mostly boolean success/event terms),
   add the smallest necessary dense progress term toward the FINAL skill specification.

Output format:
...

Format regularization and hard requirements:
- None of the three functions may call any of the other two, directly or indirectly. Each function must
recompute all needed quantities locally rather than reusing another generated function.
- Audit ALL original reward terms, not only stagnant ones.
- Prefer minimal edits over full redesign.
- Boolean bonuses are allowed if meaningful and reasonably scaled.
- Use only the allowed API.
- Reward function, termination function, and the FINAL skill specification must share the same task interpretation.
- Reward function may use continuous or decomposed shaping terms, but they must measure progress toward the same
success condition rather than a different task.
- Reward function must contain at least one dense pre-success shaping term.
- Reward function may use denser continuous progress measures toward success, and does not need to mirror
termination in boolean form.
- Termination function must be a boolean success test aligned with the same task definition used by reward function.
- Any success-defining geometric or relational target used in termination must be consistent with the target
used by reward.
- Do not introduce a new success condition in reward or termination that is absent from the FINAL skill specification,
and do not omit any success-relevant condition from reward or termination when supported by the available API.
- params may be used only for geometric target construction, never as reward weights.
- If reward constructs a target, termination must construct the identical target the same way.
```

Figure 24: Iterative subtask LLM-reflection improvement prompt.

```
def compute_dense_reward_0(self, action, obs, params) -> float:
    params = [0.01]
    # Reward term 1: The negative distance between robot's end-effector and the target
    # position on the right of the faucet handle.
    target_position = obs['current_state'][10:13] + np.array([params[0], 0, 0])
    distance = np.linalg.norm(obs['current_state'][:3] - target_position)
    reward = -distance

    # Reward term 2: Regularization term on the action, to encourage smaller actions
    # for smoother movements.
    action_reg = 0.1 * np.linalg.norm(action)
    reward -= action_reg

    return reward

def termination_0(self, obs, params, t_params) -> bool:
    params = [0.01]
    t_params = [0.01]
    # The subtask is successful if the robot's end-effector is close
    # enough to the target position
    target_position = obs['current_state'][10:13] + np.array([params[0], 0, 0])
    distance = np.linalg.norm(obs['current_state'][:3] - target_position)
    done = distance < t_params[0]
    return done
```

Figure 25: Subtask 0 "Move the robot finger to the right side of faucet handle." for turn faucet left.

```
def compute_dense_reward_1(self, action, obs, params) -> float:
    params = []
    # Reward term 1: The negative difference between current and initial angular
    # position of faucet handle.
    angular_difference = obs['current_state'][13] - obs['initial_state'][13]
    reward = -angular_difference

    # Reward term 2: Regularization term on the action, to encourage smaller
    # actions for smoother movements.
    action_reg = 0.1 * np.linalg.norm(action)
    reward -= action_reg

    return reward

def termination_1(self, obs, params, t_params) -> bool:
    t_params = [0.78]
    # The skill is successful if the faucet handle has been turned more than 0.78 radian
    # to the left
    angular_difference = obs['current_state'][13] - obs['initial_state'][13]
    done = angular_difference < -t_params[0]
    return done
```

Figure 26: Subtask 1 "Move the robot finger to turn faucet left" for turn faucet left.

```
def compute_dense_reward_0(self, action, obs, params) -> float:
    params = [0.01]
    # Reward term 1: The negative distance between the robot's end-effector and the
    # target position above the drawer box handle.
    reward_term_1 = -np.linalg.norm(obs['current_state'][:3] - \
        (obs['current_state'][14:17] + np.array([0, 0, params[0]])))
    # Reward term 2: Regularization term on the robot's action.
    reward_term_2 = -np.linalg.norm(action)
    # The total reward is the weighted sum of the reward terms.
    reward = 1 * reward_term_1 + 0.1 * reward_term_2
    return reward

def termination_0(self, obs, params, t_params) -> bool:
    params = [0.01]
    t_params = [0.01]
    # The subtask is considered successful if the distance between the robot's
    # end-effector and the target position is less than a threshold.
    done = np.linalg.norm(obs['current_state'][:3] - \
        (obs['current_state'][14:17] + np.array([0, 0, params[0]]))
    ) < t_params[0]
    return done
```

Figure 27: Subtask 0 for open drawer.

```
def compute_dense_reward_1(self, action, obs, params) -> float:
    params = []
    # Reward term 1: The negative distance between the robot's end-effector and the
    # drawer box handle.
    reward_term_1 = -np.linalg.norm(obs['current_state'][:3] - \
        obs['current_state'][14:17])
    # Reward term 2: Regularization term on the robot's action.
    reward_term_2 = -np.linalg.norm(action)
    # The total reward is the weighted sum of the reward terms.
    reward = 1 * reward_term_1 + 0.1 * reward_term_2
    return reward

def termination_1(self, obs, params, t_params) -> bool:
    t_params = [0.01]
    # The subtask is considered successful if the distance between the robot's
    # end-effector and the drawer box handle is less than a threshold.
    done = np.linalg.norm(obs['current_state'][:3] - \
        obs['current_state'][14:17]) < t_params[0]
    return done
```

Figure 28: Subtask 1 for open drawer.

```
def compute_dense_reward_2(self, action, obs, params) -> float:
    params = []
    # Reward term 1: The negative difference between the current state of the drawer
    # box and its goal state.
    reward_term_1 = -np.abs(obs['current_state'][17] - (-0.16))
    # Reward term 2: Regularization term on the robot's action.
    reward_term_2 = -np.linalg.norm(action)
    # The total reward is the weighted sum of the reward terms.
    reward = 1 * reward_term_1 + 0.1 * reward_term_2
    return reward

def termination_2(self, obs, params, t_params) -> bool:
    t_params = []
    # The subtask is considered successful if the drawer box is fully open.
    done = obs['current_state'][17] < -0.15
    return done
```

Figure 29: Subtask 2 for open drawer.

```
    def compute_dense_reward_0(self, action, obs, params) -> float:
        import numpy as np
        # Reward term 1: the distance between robot's gripper and cube A
        dist_to_cubeA = np.linalg.norm(self.tcp.pose.p - self.cubeA.pose.p)
        # Reward term 2: regularization of the robot's action
        action_reg = np.linalg.norm(action)
        reward = -dist_to_cubeA - 0.1 * action_reg
        return reward

    def termination_0(self, obs, params, t_params) -> bool:
        t_params = [0.01]
        dist_to_cubeA = np.linalg.norm(self.tcp.pose.p - self.cubeA.pose.p)
        done = dist_to_cubeA < t_params[0]
        return done
```

Figure 30: Subtask 0 for stack cube.

```
def compute_dense_reward_1(self, action, obs, params) -> float:
    import numpy as np
    # Reward term 1: the distance between robot's gripper and cube A
    dist_to_cubeA = np.linalg.norm(self.tcp.pose.p - self.cubeA.pose.p)
    # Reward term 2: regularization of the robot's action
    action_reg = np.linalg.norm(action)
    reward = -dist_to_cubeA - 0.1 * action_reg
    return reward

def termination_1(self, obs, params, t_params) -> bool:
    done = self.agent.check_grasp(self.cubeA)
    return done
```

Figure 31: Subtask 1 for stack cube.

```
def compute_dense_reward_2(self, action, obs, params) -> float:
    import numpy as np
    # Reward term 1: the height of cube A
    height_of_cubeA = self.cubeA.pose.p[2]
    # Reward term 2: regularization of the robot's action
    action_reg = np.linalg.norm(action)
    reward = height_of_cubeA - 0.1 * action_reg
    return reward

def termination_2(self, obs, params, t_params) -> bool:
    t_params = [0.1]
    done = self.cubeA.pose.p[2] > t_params[0]
    return done
```

Figure 32: Subtask 2 for stack cube.

```
def compute_dense_reward_3(self, action, obs, params) -> float:
    import numpy as np
    params = [0.02]
    # Reward term 1: the distance between cube A and the position above cube B
    dist_to_above_cubeB = np.linalg.norm(self.cubeA.pose.p - \
        (self.cubeB.pose.p + np.array([0, 0, params[0]])))
    # Reward term 2: regularization of the robot's action
    action_reg = np.linalg.norm(action)
    reward = -dist_to_above_cubeB - 0.1 * action_reg
    return reward

def termination_3(self, obs, params, t_params) -> bool:
    t_params = [0.01]
    params = [0.02]
    dist_to_above_cubeB = np.linalg.norm(self.cubeA.pose.p - \
        (self.cubeB.pose.p + np.array([0, 0, params[0]])))
    done = dist_to_above_cubeB < t_params[0]
    return done
```

Figure 33: Subtask 3 for stack cube.

```
def compute_dense_reward_4(self, action, obs, params) -> float:
    import numpy as np
    # Reward term 1: the openness of robot gripper
    gripper_openness = self.agent.robot.get_qpos()[-1] / \
    self.agent.robot.get_qlimits()[-1, 1]
    # Reward term 2: if coub A is on cube B
    cubeA_on_cubeB =  1 if self.check_cubeA_on_cubeB() else -1
    # Reward term 3: regularization of the robot's action
    action_reg = np.linalg.norm(action)
    reward = gripper_openness + cubeA_on_cubeB - 0.1 * action_reg
    return reward

def termination_4(self, obs, params, t_params) -> bool:
    done = not self.agent.check_grasp(self.cubeA) and \
        self.check_cubeA_on_cubeB() and check_actor_static(self.cubeA)
    return done
```

Figure 34: Subtask 4 for stack cube.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

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

   Answer: [Yes]

   Justification: The abstract and introduction accurately describe the paper's main contributions, namely inducing symbolic planning domains via language- and vision-grounded predicate discovery, together with empirical evaluation across multiple planning environments. The scope of the claims is consistent with the experiments and discussion presented in the main paper and appendix.

   Guidelines:

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

2. **Limitations**

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

Answer: [Yes]

Justification: The paper discusses key limitations of the proposed approach, including sensitivity of VLM-based predicate assertion to viewpoint, occlusion, lighting, and hallucination, as well as assumptions about controller availability and environment structure. We also describe mitigation strategies and clarify the scope of the claims in the relevant discussion sections and appendix.

Guidelines:

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

3. **Theory assumptions and proofs**

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

Answer: [Yes]

Justification: For the formal results and derivations included in the paper, the relevant assumptions and definitions are stated together with the corresponding formulations, and the supporting details are provided in the main paper and appendix. Cross-references are included so that the derivations and algorithmic formulations can be followed consistently.

Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: The paper describes the environments, task settings, prompts, optimization procedure, ablation settings, and evaluation metrics needed to reproduce the main results. Additional implementation details, prompts, and experimental descriptions are provided in the appendix to support reproducibility of the core findings.

   Guidelines:

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

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

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

   Answer: [N/A]

   Justification: We do not provide an open public release of code or data as part of the current submission for now. We will public it later.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: The paper specifies the training and evaluation environments, task construction, ablation settings, and key hyperparameters, including the optimization components used in the pipeline. Further experimental details and prompt specifications are provided in the appendix.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [Yes]

   Justification: The paper reports uncertainty and significance information for the main empirical results, including confidence intervals and/or standard deviations, and additionally provides statistical significance analysis with confidence intervals and p-values. These statistics are reported alongside the relevant experiments and comparisons.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.

- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: The paper reports efficiency-related measurements such as API calls and latency, and describes the computational components required by the pipeline, including language-model and optimization stages. The implementation and experimental sections together provide the information needed to understand the computational requirements of the reported experiments.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: To the best of our knowledge, the research complies with the NeurIPS Code of Ethics. The work is conducted primarily in simulation and benchmark environments, does not involve human subjects, and does not include deceptive or unsafe experimental procedures.

   Guidelines:

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

10. **Broader impacts**

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

    Answer: [N/A]

    Justification: Not applied to our work.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

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

11. **Safeguards**

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

    Answer: [No]

    Justification: Not applied to our work.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: Yes

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

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

    Answer: [N/A]

    Justification: Not applied to our work.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: Not applied to our work.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: Not applied to our work.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

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

Answer: [Yes]

Justification: LLMs are a core component of the proposed method and are used for predicate generation, domain construction, refinement, and related ablations. Their role is explicitly described in the method, experimental, and appendix sections, including prompt-based generation and model comparisons.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.