
LODGE: Joint Hierarchical Task Planning and Learning of Domain Models with Grounded Execution

Claudius Kienle

Intelligent Autonomous Systems Group
TU Darmstadt
Germany
claudius.kienle@tu-darmstadt.de

Benjamin Alt

AICOR Institute for Artificial Intelligence
University of Bremen
Germany
benjamin.alt@uni-bremen.de

Oleg Arenz

Intelligent Autonomous Systems Group
TU Darmstadt
Germany
oleg.arenz@tu-darmstadt.de

Jan Peters

Intelligent Autonomous Systems Group
TU Darmstadt
Germany
jan.peters@tu-darmstadt.de

Abstract

Large Language Models (LLMs) enable planning from natural language instructions using implicit world knowledge, but often produce flawed plans that require refinement. Instead of directly predicting plans, recent methods aim to learn a problem domain that can be solved for different goal states using classical planners. However, these approaches require significant human feedback to obtain useful models. We address this shortcoming by learning hierarchical domains, where low-level predicates and actions are composed into higher-level counterparts, and by leveraging simulation to validate their preconditions and effects. This hierarchical approach is particularly powerful for long-horizon planning, where LLM-based planning approaches typically struggle. Furthermore, we introduce a central error reasoner to ensure consistency among the different planning levels. Evaluation on two challenging International Planning Competition (IPC) domains and a long-horizon robot manipulation task demonstrates higher planning success rates than state-of-the-art domain synthesis and LLM-modulo planning methods, while constructing high-quality models of the domain. Resources, videos and detailed experiment results are available at <https://claudius-kienle.github.io/lodge>.

1 Introduction

Task planning has a long-standing history with applications in many domains, from flight planning [1, 2], logistics [3], to robot programming [4–6]. With the success of LLMs, recent work has also explored various ways to integrate them into task planning, among others to specify the task in natural language instead formal planning languages, or to access their world knowledge to automatically fill under-specified descriptions or clarify ambiguous instructions [7, 8]. However, the unreliability of LLMs limits their effectiveness as task planners [9]. In particular, long-horizon plans often are unreliable, as the LLM’s accuracy on extended action sequences degrades significantly, ultimately resulting in educated guesses by the LLM [7, 10]. This has led to hybrid approaches, where the LLM translates between natural language and formal language [11, 12], combined with classical planners for planning, or combining LLM-based planning with verification mechanisms to automatically detect and correct errors in generated plans [8]. However, integrating planners in task planning requires defining the planning domain, which describes the world in terms of objects, their properties, the

actions available to the agent as well as possible constraints. These domains are challenging to write correctly, even for humans. Methods that learn a domain model currently still require significant human feedback to validate or correct them [13].

To address these challenges, we propose a novel approach for hierarchical task planning and learning of domain models with grounded execution (LODGE). LODGE generates plans from natural language instructions and learns an explicit domain model during planning. This model is iteratively refined and used to suggest plan candidates, preventing the repeated suggestion of plans that make the same wrong assumptions about the domain. LLMs often struggle to generate long-horizon plans [9, 14]. To address this, we propose to hierarchically decompose planning and domain model learning. We first generate an abstract domain and plan that is subsequently decomposed into sub-plans. Hierarchical decomposition enables partial re-planning of sub-plans while maintaining alignment with a common domain across hierarchy levels, as well as reusing previously learned sub-plans cached in storage. Additionally, we verify the feasibility of sub-plans in simulation and correct symbolic state changes with observed state-transitions in simulation. A central error reasoner addresses feasibility violations and triggers re-planning of affected sub-plans. With that, our contributions are as follows:

- **Joint Planning and Domain Learning:** We introduce a framework for joint planning and domain model learning. It learns from planning mistakes by evaluating environment feedback about task plans and correcting the domain model and task plan during planning.
- **Task Decomposition:** We propose a hierarchical decomposition strategy, orchestrated by a central LLM-reasoning mechanism that simplifies complex tasks into manageable sub-plans.
- **Feasibility Verification and Retracting:** We introduce a motion verification system that tests plan feasibility in simulation, paired with a centralized error reasoner that analyzes verification failures to determine their causes and suggest fixes to either domain models or skill mappings at specific hierarchy levels.

2 Related Work

We will now discuss classical AI planners as well as their integration with LLMs for planning, problem translation or domain learning.

AI Planners Automated planning addresses the problem of generating action sequences to achieve desired goals. One of the most influential planning approaches was STRIPS [15], which introduced a formal representation of actions and state transitions [16, 17]. Building on STRIPS, the Planning Domain Definition Language (PDDL) [18, 19] was introduced to provide a standardized framework to represent planning problems and express planning tasks. Alternative approaches, such as Answer Set Programming (ASP) [20] and Linear Temporal Logic (LTL) [21, 22], are tailored for temporal, or reactive tasks. Several hierarchical planners have been proposed, which decompose the task goal into increasingly fine-grained sub-goals for short-horizon subtasks [23–28]. Hierarchical decomposition aids long-horizon planning but requires handling dependencies between sub-plans [29, 30].

LLMs as Planners Early work leveraged LLMs to generate feasible plans directly from natural language descriptions with plans formulated in natural language [4, 31] or structured with coding languages [6, 32–35]. Developing valid plans requires geometric reasoning, where the LLM has to interpret spatial relations between objects. Liang et al. [6] introduced hierarchical decomposition, allowing LLMs to define and later implement unknown functions. Researchers have also explored defining the task for the LLM via formal planning languages [36] to avoid the ambiguities of natural language task descriptions. The LLM then has to combine geometric reasoning from the natural language instruction, with symbolic reasoning, where abstract representations define the domain. For a more comprehensive survey, we refer to [37, 38]. While most LLM-based methods benefit from the flexibility of natural language instructions and avoid rigid planner syntax, they lack correctness guarantees. Validating such plans is challenging, and even state-of-the-art LLMs struggle to efficiently plan simple planning problems, as they lack explicit reasoning mechanisms and rely on pattern recognition rather than structured planning [39, 40, 10, 41, 42, 14, 43, 36].

LLMs as Problem Translators for Planning Recent research avoids reliance on LLMs for actual planning and instead combines LLMs with classical planners. One common approach is to translate

natural language instructions into formal problem definitions with an LLM. A classical planner then computes a plan given the problem definition and a **predefined domain model** [12, 36, 44]. The syntax and semantics of problem definitions can be verified by external verifiers [45] or self-critiqued by the LLM [44]. Silver et al. [46] use an LLM to write a program for a given PDDL domain that can efficiently solve new tasks. Instead of using classical planners for planning, LLMs can also be prompted to plan on the generated problem and predefined domain. LLM-planners have the advantage of providing feedback via LLM-generated plans, whereas classical planners give no feedback when a task is unsolvable [36]. The correctness of LLM-generated plans can be verified with external tools [44, 8] or task-specific critiques [8]. However, such hybrid approaches struggle with long-horizon planning and research often considers short plan sequences with not more than 10 actions [44]. Even recent LLMs struggle to propose good plans for complicated domains, making plan refinement inefficient. Additionally, these methods assume predefined domain models, which are by themselves hard to construct for many domains.

Learning Domain Models with LLMs Constructing domain models is complex, but crucial for autonomous planning systems. The construction with LLMs is particularly promising for open-world domains, but remains underexplored [47]. Some approaches verify generated domains with self-critique or classical tools [48, 49], while others generate multiple domain candidates and evaluate them using metrics or environment interaction [50–52]. Guan et al. [13] propose an LLM-based framework to construct PDDL planning domains by translating predefined skills into action definitions. Following the construction, they iteratively correct the domain with human feedback by translating between natural language and formal descriptions. Oswald et al. [53] translate natural language descriptions of planning domains to PDDL with LLMs, with focus on creating high-quality reconstructions of ground-truth PDDL domains. Mahdavi et al. [51] avoid human feedback by evaluating the correctness of plans generated from the domain. The authors propose to sample N domains with an LLM and evaluate each domain model by evaluating the correctness of plans sampled from the domain.

LODGE learns domain models in conjunction with task planning and avoids human feedback during planning. In contrast to Mahdavi et al. [51], we propose a targeted refinement of domain models after interacting with the environment, instead of measuring the correctness of sampled domains and selecting the best performing domain. Unlike Oswald et al. [53], our work does not assume access to ground truth domains during inference, making our approach more applicable to novel domains.

3 Problem Statement

We focus on long-horizon sequential plans that require geometric and symbolic reasoning. Given a natural language task instruction I , our objective is to plan a feasible sequence of skill π_1, \dots, π_k that accomplishes the intended task. The skills $\pi_i[\mathcal{V}]$ are part of a predefined skill library $\pi_i \in \Pi$ and parameterized by a set of variables \mathcal{V} that can be grounded with the known objects $o_i \in \mathcal{O}$ that exist in the domain. The implementation of these skills is unknown and only a description and parameter signatures are accessible. We assume a closed-world setting, where the initial state s_1 is fully known and includes all relevant objects and their states. The skills can be executed in a simulated environment $S(\pi_i(o_1, \dots, o_j))$ to determine successful execution and optionally return a short description of the occurred exception during execution. We denote a sequence o_1, \dots, o_j by $o_{1:j}$. Additionally, we assume a predicate library $\mathcal{F}_{\text{pred}}$, where each predicate $p_i[\mathcal{V}] \in \mathcal{F}_{\text{pred}}$ encodes a property of the current state s , such as *grasps(obj)* or *stacked(obj1, obj2)*. Each predicate p_i has a corresponding classifier $C_{p_i}(s; o_{1:j})$ with access to a simulation that evaluates whether p_i with objects $o_{1:j}$ holds in state s . We write the evaluation of all predefined predicates $\mathcal{F}_{\text{pred}}$ on state s as $C_{\mathcal{F}_{\text{pred}}}(s)$. We assume deterministic execution, where applying a sequence of skills $\pi_{1:k}$ in the same environment and from the same initial state s_1 always results in the same final state s_{k+1} .

The ground truth domain model $\mathcal{D} = \langle \mathcal{F}, \mathcal{A} \rangle$ consists of fluents \mathcal{F} , which define the state variables, and actions \mathcal{A} an actor can perform. Every action $a \in \mathcal{A}$ consists of preconditions $\text{prec}[\mathcal{V}]$ that define when a can be executed, and effects $\text{eff}[\mathcal{V}]$ that define the effect of a on the state. An action is said to be *grounded* if it is instantiated with objects o_i for each variable \mathcal{V} , otherwise we refer to it as a *lifted* action [13]. We assume a partially defined domain $\mathcal{D}_{\text{pred}} = \langle \mathcal{F}_{\text{pred}} \subseteq \mathcal{F}, \emptyset \rangle$ comprising the predefined predicates. We distinguish between skills π and actions a : Actions are part of the domain model and are used for planning. Skills on the other hand are the low-level primitives the actor can execute on the state. We later map an action a to one or more skills. LODGE leverages an LLM for various

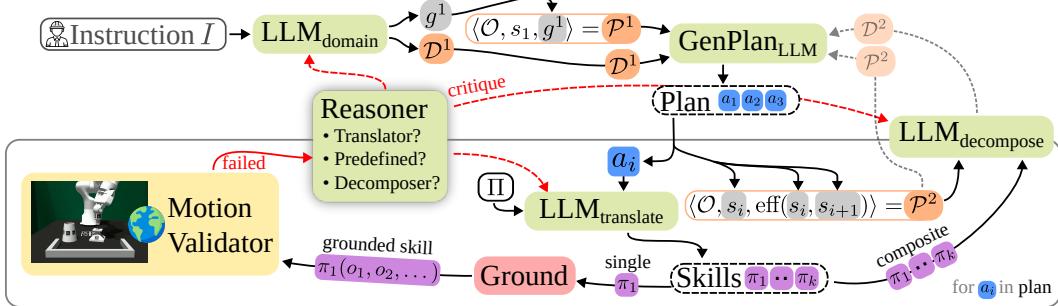


Figure 1: LODGE: A framework for joint task planning and domain model learning.

sub-problems, including action decomposition and translation of lifted actions into predefined skills. We use a temperature of 1 for all prompts and zero-shot prompting.

4 Hierarchical Task and Motion Planning with Joint Domain Model Learning

We propose LODGE, a novel hierarchical task planner for long-horizon tasks described in natural language that simultaneously constructs and refines an explicit model of the domain during planning (see Fig. 1). We propose to **decompose** (see Sec. 4.1) task planning and domain model learning hierarchically, which reduces the complexity of planning across hierarchy levels and enables more robust planning of long-horizon task sequences. We ensure the bidirectional **alignment** (see Sec. 4.2) of plans and domains across hierarchy levels, which is required to combine the generated subplans at different hierarchy levels to one valid hierarchical plan. Subplans are validated against the domain model, and executed in a simulated environment to verify the correctness of the domain model against the simulation. We propose a **central motion feasibility reasoner** (see Sec. 4.3) that is triggered if misalignments between the domain and simulator are detected, diagnoses them, corrects the domain, and initiates re-planning of the affected sections of the plan.

4.1 Jointly learning hierarchical plans and domain models

Joint planning and domain model learning enables the planner to learn from planning mistakes. Given the user instruction I , predefined domain $\mathcal{D}_{\text{pred}}$ and initial state s_1 , we prompt the LLM to complete the domain \mathcal{D}^1 , notably define the necessary lifted actions $a_{1:k}$ and define a goal state g^1 that would be needed to fulfill the instruction,

$$\mathcal{D}^1, g^1 = \text{LLM}_{\text{domain}}(I, C_{\mathcal{F}_{\text{pred}}}(s_1), \mathcal{D}_{\text{pred}}, \Pi).$$

The LLM also has access to the skill library Π containing the skills supported by the agent. The generated domain \mathcal{D}^1 contains new actions a_i and newly defined predicates $p_{j,\text{new}} \notin \mathcal{F}_{\text{pred}}$. The motion verification (Sec. 4.3) verifies that the predefined predicates for every grounded action of a candidate plan hold. Newly added predicates, on the other hand, are not assessed during motion verification, since no corresponding classifier $C_{p_{\text{new}}}$ is available. The predicted goal state g^1 and domain \mathcal{D}^1 are checked for syntax and semantic errors [45].

Instead of directly generating one action for each of the predefined skills Π , we explicitly instruct the LLM to define high-level actions where needed. This significantly simplifies domain generation. We observed that the LLM produces more accurate definitions when freely generating only a few high-level actions, than directly generating all required low-level actions. Therefore, enforcing the use of predefined skills at topmost or intermediate planning levels increases complexity of the domain to generate, leading to decreased accuracy in the domains proposed by the LLM.

We use the domain model \mathcal{D}^1 and problem $\mathcal{P}^1 := \langle \mathcal{O}, s_1, g^1 \rangle$ to subsequently generate a plan candidate $a_{1:k}^1$

$$a_{1:k}^1 = [a_1^1, \dots, a_k^1] = \text{GenPlan}_{\text{LLM}}(\mathcal{D}^1, \mathcal{P}^1).$$

We rely on LLM-generated plans because classical planners produce no feedback when the domain is unsolvable. Additionally, planners can exploit flaws in the generated domain and cannot respect weak user instructions, like temporal ordering of sub-actions [13]. However, we still speculatively

run a planner in parallel and, should a valid plan exist, supplement the prompt for $\text{GenPlan}_{\text{LLM}}$ with the planner-generated plan as a plan suggestion. Ablation 6 inspects how using the plan found by classical planners influences task planning performance.

Once $\text{GenPlan}_{\text{LLM}}$ proposes a plan candidate $a_{1:k}^1$, we iterate over every action a_j^1 of that plan to either *map* it to one of the predefined skills $\pi_i \in \Pi$ or further *decompose* it into a sub-plan $a_{1:n}^2$. Action decomposition enables to plan at an abstract level for long-horizon tasks that would require long skill sequences. Since the LLM-suggested plan candidates become less accurate with increased plan length, decomposing the task reduces the plan lengths at every level, which improves the correctness of plan candidates significantly.

Mapping and Translation To decide whether a lifted action a_i^1 matches a predefined skill π_j , we prompt the $\text{LLM}_{\text{translate}}$ with the skill library Π the action a_i^1 to propose a sequence of predefined skills to implement a_i^1

$$\pi_{1:k} = [\pi_1, \dots, \pi_k] = \text{LLM}_{\text{translate}}(a_i^1, \Pi, \mathcal{O}).$$

We instruct the LLM to parameterize the skills with the variables of the action a_i^1 where appropriate. This results in a lifted skill sequence, that can be grounded with the objects $o \in \mathcal{O}$ given used in the grounded action a_i^1 . If this skill sequence consists of only one skill, we mark a_i^1 as leaf node and start motion verification (Sec. 4.3). We otherwise *decompose* a_i^1 .

Decomposition The goal of decomposing an action a_i^1 is to find a lower-level action sequence $a_{1:n}^2 = [a_1^2, \dots, a_n^2]$ that implements the high-level action a_i^1 , e.g. decomposing *Grasp* into *Approach*, *CloseGripper*, and *Lift*. The action a_i^1 in the action sequence causes a state transition from s_i before the action to the state s_{i+1} after the action. The preconditions of a_i^1 are a subset of s_i , and the effect defines the state change from s_i to s_{i+1} [45]. To decompose a_i^1 , we define a new problem definition

$$\mathcal{P}^2 := \langle \mathcal{O}, s_i, \text{eff}(s_i, s_{i+1}) \rangle, \text{ whereby} \quad (1)$$

$$\text{eff}(s_i, s_{i+1}) := \{p(o_{1:k}) \mid p \in \mathcal{P}, o_{1:k} \subseteq \mathcal{O}, C_p(s_i; o_{1:k}) \neq C_p(s_{i+1}; o_{1:k})\}. \quad (2)$$

Analog to the $\text{LLM}_{\text{domain}}$, the LLM is tasked to complete the domain \mathcal{D}^2 , including introducing a set of lower-level actions that are required to implement a_i^1

$$\mathcal{D}^2 = \text{LLM}_{\text{decomp}}(a_i^1, \mathcal{P}^2, \mathcal{D}_{\text{pred}}, \Pi, \pi_{1:k}).$$

In contrast to $\text{LLM}_{\text{domain}}$, we pass the entire problem \mathcal{P}^2 and lifted action a_i^1 , instead of the natural language instruction I and start state s_1 . We additionally inject the proposed skill sequence $\pi_{1:k}$ generated during *Mapping and Translation*. Passing the skill sequence helps to align the decision of $\text{LLM}_{\text{translate}}$ with the content of decomposition.

4.2 Knowledge Preservation and Hierarchy Level Alignment

Dividing the planning problem hierarchically presents two key challenges: preserving knowledge across different levels of abstraction and ensuring alignment between the planning problems at the different hierarchy levels to ultimately produce a coherent hierarchical plan.

To maintain knowledge consistency, predefined predicates $\mathcal{F}_{\text{pred}}$ and objects \mathcal{O} are retained across hierarchy levels and made available for action decomposition. If new predicates are added during domain generation at one hierarchy level, these predicates are also available at all lower planning levels. Retaining the predefined predicates is crucial as the Motion Verification (see Sec. 4.3) relies on them to verify feasibility.

Mitigating goal overshoots and side effects The decomposition of a high-level action a with effects $\text{eff}(a)$ generates a low-level action sequence $a_{1:k}$ with joint effects $\text{eff}(a_{1:k})$. In order for the high-level action to remain aligned with the low-level action sequence, the effects $\text{eff}(a)$ must be identical to the joint effects $\text{eff}(a_{1:k})$. It holds that $\text{eff}(a) \subseteq \text{eff}(a_{1:k})$, as the goal state of the lower-level problem \mathcal{P} equals the upper-level effects $\text{eff}(a)$. However, the joint effects $\text{eff}(a_{1:k})$ can be a *superset* of the upper-level effects $\text{eff}(a)$ such that more predicates change than initially assumed in \mathcal{F} . We group these predicates $\text{eff}(a_{1:k}) - \text{eff}(a)$ into the two groups depending on their cause: *overshoots* and *side effects*. Overshoots are predicates that define a state change on objects that

are parameters of the grounded action a . Side effects on the other hand describe state changes on objects that are not listed in the parameters of the action. For example, when the high level action $\text{pick-up}(\text{object})$ has the effect $\text{grasp}(\text{object})$, but during decomposition, the low-level action sequence leads to a joint effect of $[\text{grasp}(\text{object}), \text{door-open}(\text{drawer}), \text{closed-gripper}]$, $\text{eff}(a_{1:k})$ contains two predicates not given in $\text{eff}(a)$. The predicate `closed-gripper` is due to an *overshoot* and `door-open(drawer)` a *side effect* of the decomposition.

These *misalignments* between hierarchy levels can break plan correctness at the upper level. After action a , the upper-level state reflects $\text{eff}(a)$, but the actual state after executing the lower-level sequence $a_{1:k}$ has changed by $\text{eff}(a_{1:k})$. Considering the example from above, if the action `close-gripper` follows next, requiring `not(closed-gripper)`, the plan seems valid at the upper level but fails after decomposition because the lower-level effect sets `closed-gripper` to false.

We detect misalignments and prompt $\text{LLM}_{\text{decomp}}$ to correct action a 's effects, aligning them with its decomposed sequence $a_{1:k}$. Fixing effect overshoots is simpler than addressing side effects, since the side effects operate on objects not defined in the upper-level action. Addressing side effects consequently requires the $\text{LLM}_{\text{decomp}}$ to add new variables to the upper-level action. Smaller models like GPT4o-mini handle overshoots well but struggle correcting misalignments caused by side effects.

4.3 Verifying Skills in Simulation, Reasoning about Mistakes, and Retracting

$\text{LLM}_{\text{domain}}$ and $\text{LLM}_{\text{decomp}}$ generate the domain model based on their world knowledge and provided input. As a consequence, the domain model is highly contingent on the world knowledge of the LLM. Modeling errors and the lack of planning capabilities of the LLM, thus, can result in invalid plans. We therefore verify the plan in simulation to check its feasibility and the correctness of the domain model.

Motion Verification Given a grounded leaf action $a(o_{1:k})$, the mapped predefined skill π , and the current state s_i , we first verify whether the preconditions of the action hold in the current state, such that $\text{pred}(a) \subseteq C_{\mathcal{F}_{\text{pred}}}(s_i; o_{1:k})$. We then execute π in simulation and verify that it executed without an error message. The simulation is in state s_{i+1} after execution. We lastly verify that the effects observed in simulation equal the effects of a , such that $\text{eff}(a) = \text{eff}(s_i, s_{i+1})$ (see Eq. 2). We verify that the preconditions and effects of predefined predicates hold in the simulation. Predicates introduced by the LLM on the other hand will not be validated as no matching predicate classifier C exists for them. Instead, we assume custom predicates used in an action a to be defined correctly and rely on the LLM to detect incorrectly defined custom predicates without motion verification.

Error Reasoning and Retracting A failed verification for an action a_i in a plan $a_{1:k}$ can be caused by an incorrectly defined domain model, a mistake in the plan candidate, or an incorrect mapping to a predefined skill. We propose a centralized error reasoner $\text{LLM}_{\text{reasoner}}$ that analyzes the occurred verification failure to determine its cause. The reasoner receives the chat history of previous LLM calls at all hierarchy levels and a summary of the motion verification, either comparing the expected effect given in the domain model with the observed effect given by the classifiers $C_{\mathcal{F}_{\text{pred}}}$, or the occurred error message raised while executing the skill π in simulation. The reasoner then determines the cause for the misalignment. The output of $\text{LLM}_{\text{reasoner}}$ is either the action a_i that should be corrected, or one of the previous skills π indicating the mapping from action to skill being the reason for the misalignment. We also prompt the reasoner to suggest a fix, either of the action, or the mapping. Once decided, we retract to the level that introduced the lifted action, or the mapping and re-prompt the related LLM with the suggested fix given by the reasoner.

5 Experiments

We evaluate the effectiveness of LODGE on the IPC domains *logistics* and *household* proposed by Guan et al. [13] in Section 5.1 and compare it against state-of-the-art domain generation approaches as well as related task planners. We additionally apply our task planner to a long-horizon robotic assembly task from the FurnitureBench Benchmark [54] in Section 5.2 to qualitatively evaluate its performance.

Env	Model	Method	Domain Errors				Tokens		
			# Defs	# Pred	# Eff	# Retries	In-C	In-T	Out
Logistics	4o-mini	GuanL [13]	3.7	5.7	1.4	—	32k	70k	10k
		LODGE (ours)	2.1	0.2	0	16	38k	128k	13k
	4.1-mini	GuanL [13]	1	2	0	—	19k	22k	3k
		LODGE (ours)	0.3	0	0	5	28k	64k	8k
Household	4o-mini	GuanL [13]	6.7	12.6	8.5	—	110k	399k	52k
		LODGE (ours)	6.1	1.7	0	19	43k	199k	13k
	4.1-mini	GuanL [13]	3.8	7.3	1.3	—	118k	245k	32k
		LODGE (ours)	2.5	0.8	0.2	14	65k	188k	18k

Table 1: Accuracy and efficiency of generated domain models with GuanL [13] and LODGE. We evaluate 10 seeds of GuanL and learn the domain model for all tasks from scratch with LODGE. LODGE produces more accurate domain models, with fewer definitions missing (# *Defs*), as well as fewer errors in the actions’ predicates (# *Pred*) and effects (# *Eff*). *In-C* states the cached input tokens, while *In-T* records the total input tokens used.

5.1 Planning Benchmark

We perform quantitative evaluation of LODGE on the IPC domains *household* and *logistics* introduced by Guan et al. [13]. Tasks in both domains are long-horizon and the average plan has a length of 23 skills. The household domain contains 22 different skills, while the logistics domain contains 6. Details about the two environments are given in Appendix A and in the work of Guan et al. [13]. We evaluate LODGE’s ability in learning accurate domain models as well as its ability to compute correct plans. While we could reuse the learned domain for subsequent tasks of the same domain, we explicitly delete learned domains and artifacts, to evaluate how robustly our approach can develop domains from scratch. We avoid prompt caching between tasks to not influence LLM responses by previously planned tasks. Additionally, we limit the overall number of re-planning iterations to 20. We use the task instructions, domain descriptions and skill descriptions from Guan et al. [13]. The definition of a predefined skill looks like the following:

```
def heat_food_with_pan(food: str, pan: str):
    """This action enables the robot to heat food which is heatable with a pan. The food should be
    placed on the pan, and the pan needs to be placed on a stove burner before executing this action.
    Note that the food is no longer pickupable after it has been heated."""

```

We evaluate the impact of omitting specific descriptions in Ablation 6.

Domain Learning We evaluate the correctness of the learned domains on the two IPC domains and compare them to the domains generated by the method proposed by Guan et al. [13], referred to as *GuanL*. Guan et al. construct the domain using an LLM and correct it with human feedback. We compare our method to GuanL’s domain construction method. As we are interested in comparing the quality of generated domains automatically with LLMs, we leave out the human feedback correction proposed by Guan et al. [13].

We categorize the errors as either missing definitions (# *Defs*), e.g. predicate or action definitions, or incorrect preconditions (# *Pred*) or effects (# *Eff*). While Guan et al. [13] generate the entire domain model, LODGE constructs a task-centric domain model. To compare the two domain models, we identify the errors per task and then average them over the tasks. The errors per task are defined by retrieving the skills required to solve the task and summing up the errors in the matching action definitions of the domain model. To determine the robustness of both approaches, we generate the domain with GuanL 10 times and average the results. LODGE constructs a new domain for every new task, which naturally evaluates robustness as we do not retain domains across tasks.

Table 1 evaluates the domain correctness of LODGE with GuanL [13]. Domains generated with LODGE have fewer missing definitions and significantly fewer mistakes in the action preconditions and effects, particularly with GPT-4.1-mini. One failure mode of LODGE are missing actions that are required to complete the task. This is often due to planning failures at early stages, when future actions have not been added to the domain yet.

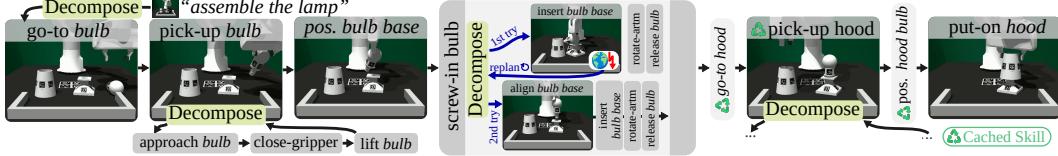


Figure 2: The hierarchical planning of LODGE for the FurnitureBench lamp-assembly environment, including decomposition of *pick-up bulb* and re-planning within *screw-in bulb*.

While excessive retrying with LLMs would gradually improve the performance, it also comes with heavy API and compute usage. We limit LLM interactions by allowing a maximum of 20 re-planning steps per task. Table 1 compares the token usage of our approach to GuanL and shows the average number of re-planning iterations per task. LODGE requires moderately more tokens than GuanL [13] in the logistics domain. The increased token usage is primarily due to many re-planning iterations in few tasks using GPT4.1-mini, and the failed tasks for GPT4o-mini, which all used the maximum of 20 re-planning iterations. The significantly reduced token usage of LODGE on the Household domain is mainly due to learning the part of the domain model relevant to solve the task, while GuanL generates the entire domain model. Note that we discard learned domains and planning artifacts across the tasks of an environment. Retaining them would greatly reduce the token usage (see Abl. 6).

Task Planning We compare LODGE to different task planning methods to evaluate its task planning ability. The *LLM Planner* baseline generates a plan sequence given natural language instruction I , domain description and predefined skills Π with descriptions. *LLM Planner + Simulation* additionally has access to the same simulator LODGE uses to evaluate plan candidates. Simulation feedback about plan candidates will be reprompted a maximum of 20 times such that the LLM can propose a corrected plan. We lastly evaluate using the domains from GuanL for planning. We manually translate natural language instructions into goal definitions using the predicates defined in the domain of GuanL and programmatically translate the start state into the set of predicates used in the domains. We manually correct syntax issues present in the domains of GuanL to enable planning with a classical planner. We do not apply human feedback to correct domains [13].

Table 2 shows the number of successfully planned tasks on the IPC domains using GPT-4.1-mini as LLM for all planners. LODGE outperforms the *LLM Planner* and GuanL baselines. While LODGE performs only slightly better than *LLM Planner + Simulation*, our approach simultaneously learned the domain model by interacting with the simulation. *LLM Planner + Simulation* corrected the plans directly and did not learn, eschewing benefits for future planning tasks. Additionally, we found that LODGE generates more efficient plans for planning-heavy tasks. On the logistics domain, LODGE found plans that are four skills shorter than the plans of *LLM Planner + Simulation* on the same tasks. This can be motivated by the domain model and classical planner, which LODGE uses to sample plans. All domains generated by GuanL were still unsolvable without human correction. Upon manual evaluation, we found that many domains used duplicate predicates for one property of the state, e.g. *appliance-on* and *appliance-off*. The actions often only operated on one predicate and left out the other, making the domain unsolvable.

5.2 FurnitureBench

We evaluate LODGE on the task planning for a robotics assembly task of the FurnitureBench Benchmark [54]. The IPC domains investigated earlier are tailored to work with formal language task planning, e.g. by defining different skills for grasping from a furniture and grasping from a receptacle. Related work often denotes such domains as *PDDL domains*. Evaluating LODGE on the FurnitureBench assembly investigates how well LODGE can plan tasks for domains with a skill library less tailored to task planning. Additionally, while most skills in the IPC domain are only defined when executed in the right order, causing an error message to be returned when executed in a different state, the skills in the FurnitureBench domain are general-purpose low-level robot skills.

Only considering the plan executability as feedback signal in such environments as in [51] cannot help to detect modelling errors in the domain. Detecting simulation feasibility violations for this environment therefore mostly requires to detect unsatisfied preconditions or misaligned effects rather than errors caused during skill execution. We evaluate LODGE with GPT4.1-mini for this domain.

Figure 2 depicts the progress of planning the skill sequence to assemble a *lamp*, which consists of three parts: *bulb*, *base* and *hood*. The Figure also depicts how the decomposition of the task significantly reduces the plan length at the topmost layer. Additionally, the initially decomposed *grasp-part bulb* is reused during its second invocation *grasp-part hood*, not requiring any LLM calls to again decompose that action. A video of the planning, including the re-planning until a valid plan was found, is available on the website of the paper.

6 Ablation

Using plans from a classical planner Using plans found by classical planners during domain learning presents a trade-off: The plan could exploit a flaw in the domain model that LLM-planners do not exploit. However, weaker LLMs struggle to suggest accurate plan candidates, even when re-prompting them with feedback about the issue in the plan. Table 3 shows how using plans found by a classical planner impacts the performance of LODGE. The increased success rate for GPT4o-mini and decreased success rate for GPT4.1-mini highlights this trade-off. We therefore use the classical planner for GPT4o-mini, while relying on GPT4.1-mini to suggest plan candidates. Supplying the domain description to the LLM does seem to help for GPT4o-mini, while it reduces the success rate for GPT4.1-mini. It remains open to future work to inspect in detail how leaving out different parts of information, e.g. function descriptions, impacts learning domain models and planning.

Retaining domains across tasks The learned domain model can be retained for future tasks on the same domain. Retaining the domain and plan artifacts can significantly reduce the LLM usage during planning, thereby speeding up planning. We tested injecting the domain model of previous tasks in the initial prompt of LLM_{domain} along with the other information. Table 4 shows the impact of retaining the domain model on the token usage for successfully planned task. Retaining domains reduces the number of re-planning iterations until LODGE finds a valid plan. Retaining the domain does not affect unsuccessful tasks, as they always use the maximum of 20 re-plans.

7 Conclusion

We propose LODGE, a hierarchical task planner that jointly learns the domain model while developing the plan. The planner decomposes the task hierarchically, initially learning an abstract domain model and plan that is iteratively refined into sub-domains and sub-plans. We introduce a novel central error reasoner that verifies the feasibility of plan candidates in simulation and analyzes infeasible plans to detect and correct misalignments between the simulation and learned domain model. We evaluate LODGE on two IPC domains and one FurnitureBench environment. Our experiments show that LODGE successfully learns task-centric domain models for complex IPC domains and generates more accurate domain models than related work, while jointly solving more tasks successfully than existing planning methods. The evaluation on the FurnitureBench environment shows that LODGE

DD	Planner	GPT4o-mini	GPT4.1-mini
✓	✓	7 / 21	12 / 21
✓	✗	1 / 21	11 / 21
✗	✓	2 / 21	15 / 21
✗	✗	1 / 21	20 / 21

Table 3: Impact of using domain descriptions (DD) and plans from a classical planner (Planner) on planning success in the logistics domain.

	Logistics		Household	
	w/o	w	w/o	w
# Retries	4	3	8	6
Uncached Input Tokens	27k	17k	47k	39k
Total Input Tokens	54k	31k	130k	113k
Output Tokens	8k	6k	12k	8k

Table 4: Impact of retaining (w) or deleting (w/o) the domain model on token usage for successful tasks.

can solve tasks for real-world task planning domains that are not tailored to formal languages. While LODGE provides strong performance on hierarchical domain learning and planning, there are several directions for future work. First, we assume access to a library of predefined predicates, which is required to evaluate the state of the simulation. Future work could investigate combining LODGE with approaches for predicate learning [55] such that predicates must not be known in advance. Second, the decomposition of tasks requires alignment of the hierarchy levels during planning. Realigning two planning levels is non-trivial due to goal overshoots and side effects. Future work could investigate on more sophisticated approaches to realignment that work robustly even for smaller LLMs. Lastly, the impact of auxiliary information supplied to LODGE, like domain or function descriptions, is open to future investigation.

8 Acknowledgement

This work was supported by the German Federal Ministry of Education and Research (project RobInTime, grant 01IS25002B).

References

- [1] A. Fukunaga, G. Rabideau, S. Chien, and D. Yan, “Aspen: A framework for automated planning and scheduling of spacecraft control and operations,” in *Proc. International Symposium on AI, Robotics and Automation in Space*. Citeseer, 1997, pp. 181–187. [Online]. Available: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=159badc67e92166261574d866a9793f9c80c05ef>
- [2] J. Liu, F. Yang, and J. Li, “Research of AI Planning for Space Flight Control Based on PDDL,” in *Proceedings of the 27th Conference of Spacecraft TT&C Technology in China*, R. Shen and W. Qian, Eds. Berlin, Heidelberg: Springer, 2015, pp. 359–369.
- [3] J. García, J. E. Florez, A. Torralba, D. Borrado, C. L. López, A. García-Olaya, and J. Sáenz, “Combining linear programming and automated planning to solve intermodal transportation problems,” *European Journal of Operational Research*, vol. 227, no. 1, pp. 216–226, May 2013. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0377221712009496>
- [4] B. Ichter, A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog, D. Ho, J. Ibarz, A. Irpan, E. Jang, R. Julian, D. Kalashnikov, S. Levine, Y. Lu, C. Parada, K. Rao, P. Sermanet, A. T. Toshev, V. Vanhoucke, F. Xia, T. Xiao, P. Xu, M. Yan, N. Brown, M. Ahn, O. Cortes, N. Sievers, C. Tan, S. Xu, D. Reyes, J. Rettinghouse, J. Quiambao, P. Pastor, L. Luu, K.-H. Lee, Y. Kuang, S. Jesmonth, N. J. Joshi, K. Jeffrey, R. J. Ruano, J. Hsu, K. Gopalakrishnan, B. David, A. Zeng, and C. K. Fu, “Do As I Can, Not As I Say: Grounding Language in Robotic Affordances,” in *Proceedings of The 6th Conference on Robot Learning*. PMLR, Mar. 2023, pp. 287–318, iSSN: 2640-3498. [Online]. Available: <https://proceedings.mlr.press/v205/ichter23a.html>
- [5] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter, “Inner Monologue: Embodied Reasoning through Planning with Language Models,” Jul. 2022, arXiv:2207.05608 [cs]. [Online]. Available: <http://arxiv.org/abs/2207.05608>
- [6] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, “Code as Policies: Language Model Programs for Embodied Control,” in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, May 2023, pp. 9493–9500. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10160591?casa_token=WFKbBaJAs6IAAAAA:KPr_ULH6fkAWMMMyLaS01pZ2_xkQEajIgZyrD6wkN1jKE-wfvtx3DOwk8Gmb26BqUCzNuS_gLgQ
- [7] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg, “Text2Motion: from natural language instructions to feasible plans,” *Autonomous Robots*, vol. 47, no. 8, pp. 1345–1365, Dec. 2023. [Online]. Available: <https://link.springer.com/10.1007/s10514-023-10131-7>
- [8] A. Gundawar, K. Valmeekam, M. Verma, and S. Kambhampati, “Robust Planning with Compound LLM Architectures: An LLM-Modulo Approach,” Nov. 2024, arXiv:2411.14484 [cs]. [Online]. Available: <http://arxiv.org/abs/2411.14484>

[9] S. Kambhampati, K. Valmeekam, L. Guan, M. Verma, K. Stechly, S. Bhambri, L. Saldyt, and A. Murthy, “LLMs Can’t Plan, But Can Help Planning in LLM-Modulo Frameworks,” Jun. 2024, arXiv:2402.01817. [Online]. Available: <http://arxiv.org/abs/2402.01817>

[10] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, “Large Language Models Still Can’t Plan (A Benchmark for LLMs on Planning and Reasoning about Change),” Nov. 2022. [Online]. Available: <https://openreview.net/forum?id=wUU-7XTL5XO>

[11] E. Gestrin, M. Kuhlmann, and J. Seipp, “NL2Plan: Robust LLM-Driven Planning from Minimal Text Descriptions,” May 2024, arXiv:2405.04215 [cs]. [Online]. Available: <http://arxiv.org/abs/2405.04215>

[12] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, “LLM+P: Empowering Large Language Models with Optimal Planning Proficiency,” Sep. 2023, arXiv:2304.11477. [Online]. Available: <http://arxiv.org/abs/2304.11477>

[13] L. Guan, K. Valmeekam, S. Sreedharan, and S. Kambhampati, “Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 79 081–79 094, Dec. 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/f9f54762ccb4fe4dbffdd4f792c31221-Abstract-Conference.html

[14] X. Huang, W. Liu, X. Chen, X. Wang, H. Wang, D. Lian, Y. Wang, R. Tang, and E. Chen, “Understanding the planning of LLM agents: A survey,” Feb. 2024, arXiv:2402.02716. [Online]. Available: <http://arxiv.org/abs/2402.02716>

[15] A. I. Center, “Shakey the robot,” 1984. [Online]. Available: <https://www.cs.sfu.ca/~vaughan/teaching/415/papers/shakey.pdf>

[16] J. Carbonell, O. Etzioni, Y. Gil, R. Joseph, C. Knoblock, S. Minton, and M. Veloso, “PRODIGY: an integrated architecture for planning and learning,” *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 51–55, Jul. 1991. [Online]. Available: <https://dl.acm.org/doi/10.1145/122344.122353>

[17] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, “SHOP2: An HTN planning system,” *Journal of artificial intelligence research*, vol. 20, pp. 379–404, 2003. [Online]. Available: <https://www.jair.org/index.php/jair/article/view/10362>

[18] M. Ghalleb, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, “PDDL - The Planning Domain Definition Language,” *Technical Report, Tech. Rep.*, 1998. [Online]. Available: https://www.researchgate.net/profile/Craig-Knoblock/publication/2278933_PDDL_-_The_Planning_Domain_Definition_Language/links/0912f50c0c99385e19000000/PDDL-The-Planning-Domain-Definition-Language.pdf

[19] M. Fox and D. Long, “PDDL2. 1: An extension to PDDL for expressing temporal planning domains,” *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003. [Online]. Available: <https://www.jair.org/index.php/jair/article/view/10352>

[20] G. Brewka, T. Eiter, and M. Truszczyński, “Answer set programming at a glance,” *Communications of the ACM*, vol. 54, no. 12, pp. 92–103, Dec. 2011. [Online]. Available: <https://dl.acm.org/doi/10.1145/2043174.2043195>

[21] H. Kress-Gazit, G. Fainekos, and G. Pappas, “Temporal-Logic-Based Reactive Mission and Motion Planning,” *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, Dec. 2009. [Online]. Available: <http://ieeexplore.ieee.org/document/5238617/>

[22] F. Nawaz, S. Peng, L. Lindemann, N. Figueira, and N. Matni, “Reactive Temporal Logic-based Planning and Control for Interactive Robotic Tasks,” Apr. 2024, arXiv:2404.19594. [Online]. Available: <http://arxiv.org/abs/2404.19594>

[23] L. P. Kaelbling and T. Lozano-Pérez, “Hierarchical task and motion planning in the now,” in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 1470–1477, ISSN: 1050-4729.

[24] T. Silver, R. Chitnis, J. Tenenbaum, L. P. Kaelbling, and T. Lozano-Pérez, “Learning symbolic operators for task and motion planning,” in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 3182–3189.

[25] Z. Liang, Y. Mu, H. Ma, M. Tomizuka, M. Ding, and P. Luo, “Skilldiffuser: Interpretable hierarchical planning via skill abstractions in diffusion-based task execution,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2024, pp. 16 467–16 476.

[26] C. Chen, F. Deng, K. Kawaguchi, C. Gulcehre, and S. Ahn, “Simple hierarchical planning with diffusion,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=kXHEBK9uAY>

[27] H. Guo, F. Wu, Y. Qin, R. Li, K. Li, and K. Li, “Recent trends in task and motion planning for robotics: A survey,” *ACM Comput. Surv.*, vol. 55, no. 13s, Jul. 2023. [Online]. Available: <https://doi.org/10.1145/3583136>

[28] I. Georgievski and M. Aiello, “Htn planning: Overview, comparison, and beyond,” *Artificial Intelligence*, vol. 222, pp. 124–156, 2015.

[29] A. Ajay, S. Han, Y. Du, S. Li, A. Gupta, T. Jaakkola, J. Tenenbaum, L. Kaelbling, A. Srivastava, and P. Agrawal, “Compositional foundation models for hierarchical planning,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 22 304–22 325, 2023.

[30] C. Qi, D. Wang, H. Muñoz-Avila, P. Zhao, and H. Wang, “Hierarchical task network planning with resources and temporal constraints,” *Knowledge-Based Systems*, vol. 133, pp. 17–32, 2017.

[31] S. S. Raman, V. Cohen, E. Rosen, I. Idrees, D. Paulius, and S. Tellek, “Planning With Large Language Models Via Corrective Re-Prompting,” Nov. 2022. [Online]. Available: <https://openreview.net/forum?id=cMDMRBe1TKs>

[32] K. Burns, A. Jain, K. Go, F. Xia, M. Stark, S. Schaal, and K. Hausman, “GenCHiP: Generating Robot Policy Code for High-Precision and Contact-Rich Manipulation Tasks,” Apr. 2024, arXiv:2404.06645. [Online]. Available: <http://arxiv.org/abs/2404.06645>

[33] S. Wang, M. Han, Z. Jiao, Z. Zhang, Y. N. Wu, S.-C. Zhu, and H. Liu, “LLM3:Large Language Model-based Task and Motion Planning with Motion Failure Reasoning,” Aug. 2024, arXiv:2403.11552. [Online]. Available: <http://arxiv.org/abs/2403.11552>

[34] Y. Jin, D. Li, Y. A. J. Shi, P. Hao, F. Sun, J. Zhang, and B. Fang, “RobotGPT: Robot Manipulation Learning from ChatGPT,” Dec. 2023, arXiv:2312.01421. [Online]. Available: <http://arxiv.org/abs/2312.01421>

[35] S. S. Kannan, V. L. N. Venkatesh, and B.-C. Min, “SMART-LLM: Smart Multi-Agent Robot Task Planning using Large Language Models,” Mar. 2024, arXiv:2309.10062 [cs]. [Online]. Available: <http://arxiv.org/abs/2309.10062>

[36] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, “On the Planning Abilities of Large Language Models - A Critical Investigation,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 75 993–76 005, Dec. 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/efb2072a358cef75886a315a6fcf880-Abstract-Conference.html

[37] V. Pallagani, B. C. Muppasani, K. Roy, F. Fabiano, A. Loreggia, K. Murugesan, B. Srivastava, F. Rossi, L. Horesh, and A. Sheth, “On the prospects of incorporating large language models (llms) in automated planning and scheduling (aps),” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 34, 2024, pp. 432–444.

[38] H. Li, Z. Chen, J. Zhang, and F. Liu, “Lasp: Surveying the state-of-the-art in large language model-assisted ai planning,” *arXiv preprint arXiv:2409.01806*, 2024.

[39] S. Kambhampati, “Can Large Language Models Reason and Plan?” *Annals of the New York Academy of Sciences*, p. nyas.15125, Mar. 2024, arXiv:2403.04121 [cs]. [Online]. Available: <http://arxiv.org/abs/2403.04121>

[40] K. Stechly, K. Valmeekam, and S. Kambhampati, “Chain of Thoughtlessness? An Analysis of CoT in Planning,” Jun. 2024, arXiv:2405.04776. [Online]. Available: <http://arxiv.org/abs/2405.04776>

[41] K. Valmeekam, K. Stechly, and S. Kambhampati, “LLMs Still Can’t Plan; Can LRM? A Preliminary Evaluation of OpenAI’s o1 on PlanBench,” Sep. 2024, arXiv:2409.13373 [cs]. [Online]. Available: <http://arxiv.org/abs/2409.13373>

[42] K. M. Collins, C. Wong, J. Feng, M. Wei, and J. B. Tenenbaum, “Structured, flexible, and robust: benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks,” May 2022, arXiv:2205.05718. [Online]. Available: <http://arxiv.org/abs/2205.05718>

[43] H. Wei, Z. Zhang, S. He, T. Xia, S. Pan, and F. Liu, “Plangenllms: A modern survey of llm planning capabilities,” 2025. [Online]. Available: <https://arxiv.org/abs/2502.11221>

[44] Z. Zhou, J. Song, K. Yao, Z. Shu, and L. Ma, “ISR-LLM: Iterative Self-Refined Large Language Model for Long-Horizon Sequential Task Planning,” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, May 2024, pp. 2081–2088. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10610065>

[45] R. Howey, D. Long, and M. Fox, “VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL,” in *16th IEEE International Conference on Tools with Artificial Intelligence*, Nov. 2004, pp. 294–301, iSSN: 1082-3409. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/1374201>

[46] T. Silver, S. Dan, K. Srinivas, J. B. Tenenbaum, L. P. Kaelbling, and M. Katz, “Generalized Planning in PDDL Domains with Pretrained Large Language Models,” Dec. 2023, arXiv:2305.11014 [cs]. [Online]. Available: <http://arxiv.org/abs/2305.11014>

[47] M. Tantakoun, X. Zhu, and C. Muise, “Llms as planning modelers: A survey for leveraging large language models to construct automated planning models,” *arXiv preprint arXiv:2503.18971*, 2025.

[48] P. Smirnov, F. Joublin, A. Ceravola, and M. Gienger, “Generating consistent PDDL domains with Large Language Models,” Apr. 2024, arXiv:2404.07751 [cs]. [Online]. Available: <http://arxiv.org/abs/2404.07751>

[49] A. Ishay and J. Lee, “LLM+AL: Bridging Large Language Models and Action Languages for Complex Reasoning about Actions,” Feb. 2025, arXiv:2501.00830 [cs]. [Online]. Available: <http://arxiv.org/abs/2501.00830>

[50] Z. Yu, Y. Yuan, T. Z. Xiao, F. F. Xia, J. Fu, G. Zhang, G. Lin, and W. Liu, “Generating symbolic world models via test-time scaling of large language models,” *arXiv preprint arXiv:2502.04728*, 2025.

[51] S. Mahdavi, R. Aoki, K. Tang, and Y. Cao, “Leveraging Environment Interaction for Automated PDDL Generation and Planning with Large Language Models,” Jul. 2024, arXiv:2407.12979. [Online]. Available: <http://arxiv.org/abs/2407.12979>

[52] M. Hu, T. Chen, Y. Zou, Y. Lei, Q. Chen, M. Li, Y. Mu, H. Zhang, W. Shao, and P. Luo, “Text2World: Benchmarking Large Language Models for Symbolic World Model Generation,” Feb. 2025, arXiv:2502.13092 [cs]. [Online]. Available: <http://arxiv.org/abs/2502.13092>

[53] J. Oswald, K. Srinivas, H. Kokel, J. Lee, M. Katz, and S. Sohrabi, “Large Language Models as Planning Domain Generators,” Apr. 2024, arXiv:2405.06650 [cs]. [Online]. Available: <http://arxiv.org/abs/2405.06650>

[54] M. Heo, Y. Lee, D. Lee, and J. J. Lim, “FurnitureBench: Reproducible real-world benchmark for long-horizon complex manipulation,” *The International Journal of Robotics Research*, p. 02783649241304789, 2023, publisher: SAGE Publications Ltd STM. [Online]. Available: <https://doi.org/10.1177/02783649241304789>

[55] M. Han, Y. Zhu, S.-C. Zhu, Y. N. Wu, and Y. Zhu, “InterPreT: Interactive Predicate Learning from Language Feedback for Generalizable Task Planning,” May 2024, arXiv:2405.19758 [cs]. [Online]. Available: <http://arxiv.org/abs/2405.19758>

Category	Logistics	Household
# of tasks	21	24
# of actions	6	22
# of params and literals	54	271
avg plan length	23.2	21.6
avg types of skills	6	7.9

Table 5: Details about IPC domains

A IPC Domains

Details about the IPC planning domains used in Experiment 5.1 are listed in Table 5.

A.1 Logistics

Task 1: Transport package package_0 to location location_2

```
load_truck('package_0', 'truck_1')
fly_plane('plane_0', 'location_1', 'location_0')
unload_truck('package_0', 'truck_1')
load_plane('package_0', 'plane_0')
fly_plane('plane_0', 'location_0', 'location_1')
unload_plane('package_0', 'plane_0')
drive_truck('truck_0', 'location_2', 'location_1')
load_truck('package_0', 'truck_0')
drive_truck('truck_0', 'location_1', 'location_2')
unload_truck('package_0', 'truck_0')
```

Task 2: Transport package package_3 to location location_3, package package_4 to location location_0, package package_1 to location location_2, package package_0 to location location_0 and package package_2 to location location_4

```
load_truck('package_1', 'truck_1')
load_plane('package_4', 'plane_0')
load_plane('package_0', 'plane_0')
drive_truck('truck_1', 'location_4', 'location_5')
unload_truck('package_1', 'truck_1')
load_plane('package_1', 'plane_0')
fly_plane('plane_0', 'location_5', 'location_2')
unload_plane('package_4', 'plane_0')
load_truck('package_4', 'truck_0')
load_plane('package_3', 'plane_0')
unload_plane('package_1', 'plane_0')
unload_plane('package_0', 'plane_0')
load_truck('package_0', 'truck_0')
drive_truck('truck_0', 'location_2', 'location_0')
unload_truck('package_4', 'truck_0')
unload_truck('package_0', 'truck_0')
drive_truck('truck_0', 'location_0', 'location_1')
load_truck('package_2', 'truck_0')
drive_truck('truck_0', 'location_1', 'location_2')
unload_truck('package_2', 'truck_0')
load_plane('package_2', 'plane_0')
fly_plane('plane_0', 'location_2', 'location_5')
unload_plane('package_3', 'plane_0')
load_truck('package_3', 'truck_1')
unload_plane('package_2', 'plane_0')
load_truck('package_2', 'truck_1')
drive_truck('truck_1', 'location_5', 'location_3')
unload_truck('package_3', 'truck_1')
drive_truck('truck_1', 'location_3', 'location_4')
unload_truck('package_2', 'truck_1')
```

A.2 Household

Task 1: put apple_2 on side_table_2

```
go_to("dining_table_1")
put_on("mug_1", "dining_table_1")
```

```

go_to("fridge_1")
open("fridge_1")
pick_up("lunch_box_2", "fridge_1")
go_to("dining_table_1")
put_on("lunch_box_2", "dining_table_1")
open_small("lunch_box_2")
pick_up_from("apple_2", "lunch_box_2")
go_to("side_table_2")
put_on("apple_2", "side_table_2")

```

Task 2: heat pizza_1 with pan_2, and put it on plate_2

```

go_to("dining_table_1")
put_on("mug_1", "dining_table_1")
go_to("cabinet_2")
open("cabinet_2")
pick_up("pizza_box_1", "cabinet_2")
go_to("countertop_2")
put_on("pizza_box_1", "countertop_2")
open_small("pizza_box_1")
go_to("drawer_1")
pick_up("pan_2", "drawer_1")
go_to("countertop_2")
put_on("pan_2", "countertop_2")
transfer("pizza_1", "pizza_box_1", "pan_2")
pick_up("pan_2", "countertop_2")
go_to("stove_burner_2")
put_on("pan_2", "stove_burner_2")
heat("pizza_1", "pan_2")
go_to("cabinet_3")
open("cabinet_3")
pick_up("plate_2", "cabinet_3")
go_to("countertop_2")
put_on("plate_2", "countertop_2")
go_to("stove_burner_2")
pick_up("pan_2", "stove_burner_2")
go_to("countertop_2")
put_on("pan_2", "countertop_2")
transfer("pizza_1", "pan_2", "plate_2")

```

B Example Prompts

System Prompt

```

You are a planning expert tasked with developing a world model for planning based on a user instruction.

--- 

<!-- when generating content for the sections listed below, follow the specified format exactly. -->
### Explanation
<!-- task specific explanation and chain-of-thought reasoning -->

### Change/Add Action(s)
1. {action-name-1}: {(add|edit|delete)}
   - Description: {description what happens during the action}
   - PDDL Definition:
      ````pddl
 {pddl_action_definition}
      ````

### Change/Add Predicate(s)
- {(predicate_name} {predicate_args...}): {predicate_description}

### Change Initial State
<!-- add predicates that should be changed, without text, leave 'None' if no change -->
({predicate_4}): {(true|false|remove)}
({predicate_8}): {(true|false|remove)}

### Change Goal State
<!-- add predicates that should be changed, without text, leave 'None' if no change -->
({predicate_7}): {(true|false|remove)}

### Final thoughts

```

B.1 LLM_{domain}

```

### User Instruction
place fork_1, spoon_1 and knife_1 on dining_table_1, please take the fork first and the knife last

### Predicates
- (at-agent ?a - agent ?loc - furniture_appliance): Agent is at a specific furniture piece or appliance.
- (holding ?a - agent ?obj - household_object): Agent is holding an object.
...
...

### Types
household_object furniture_appliance agent - object
...

### Objects
drawer_1 ... - drawer fridge_1 - fridge ...

### Initial State
(at-agent robot dining_table_1) (holding robot mug_1) ...

### Skill Library
go_to_a_furniture_piece_or_an_appliance, pick_up_an_object_on_or_in_a_furniture_piece_or_an_appliance,
...

### Your Task
1. Define the goal: Based on the user instruction, create a PDDL goal that reflects the objective the user wants to accomplish.
2. List of predicates: Inspect the predicates that will be used to describe the state of the world and the relationships between entities in the domain.
3. Define actions:
    - Based on the goal and available skills in the skill library, define a set of PDDL actions that enable planning toward the goal.
    - Preferably define high-level actions that abstract over one or more low-level skills to support hierarchical planning.
    - Each PDDL action should:
        - Have a clear and descriptive name
        - Include a general-purpose description (not instance-specific)
        - Include a PDDL definition: (:action <action_name> :parameters <parameters> :precondition <precondition> :effect <effect>)
    - You are encouraged to:
        - Define high-level composite actions when they simplify planning.
        - Define individual PDDL actions for each skill you intend to use.
    - Avoid unnecessary actions - only include those essential to achieving the goal.

```

B.2 LLM_{translate}

```

### Predefined Skills
```python
def go_to_a_furniture_piece_or_an_appliance(furniture_or_appliance: str):
 """This action enables the robot to navigate from one normally immovable piece of furniture to another (e.g., dining tables, side tables, cabinets, and sinks) or an appliance (e.g., refrigerators, coffee makers, microwaves, and washers)."""
 ...

def pick_up_an_object_on_or_in_a_furniture_piece_or_an_appliance(object_id: str, furniture_or_appliance: str):
 """This action enables the robot to pick up an object in/on a large piece of furniture (e.g., dining tables, drawers, cabinets) or an appliance (e.g., dishwashers and refrigerators). The furniture piece or appliance should be opened if it is openable. The object to pick up should not be stacked on top of other household items."""
 ...

```
### Objects
drawer_1 drawer_2 ... - drawer fridge_1 - fridge ...

### Previous skill executed
; no previous skill

### PDDL Action Definition
put_object_on_furniture: add
- Description: Agent places a held object on a specified furniture piece or appliance. Agent must be at that furniture piece.
- PDDL Definition:
    ```pddl
 (:action put_object_on_furniture
 :parameters (?a - agent ?obj - household_object ?furn - furniture_appliance)
 :precondition (and (holding ?a ?obj) (at-agent ?a ?furn)))
    ```


```

```

        :effect (and (not (holding ?a ?obj)) (agent-hand-empty ?a) (at-object ?obj ?furn))
    )
```


 ### Task
 Your task is to map the PDDL action ‘put_object_on_furniture’ to an implementation using the predefined skills provided.
 1. Analyze the PDDL action’s preconditions and effects
 2. Identify which predefined skill(s) can accomplish the same effect as the PDDL action
 3. Create a sequence of skill calls that implements the PDDL action. You may include additional skills not mentioned in the PDDL action if they are necessary

 Output the action sequence inside a markdown Python code block:

 Important Rules:
 - Use only the predefined skills listed above
 - Select skills that achieve the same end result as the PDDL action’s effect
 - You may need to use multiple skills to properly implement a single PDDL action
 - Assume the preconditions are already satisfied - i.e., you don’t need to include skills to make them true again (e.g., no need to grasp_object if (holding ?a ?item) is already true).
 - However, you must include all skills required to implement the effects, including any intermediate steps required by the predefined skills, even if the PDDL action does not mention them explicitly (e.g., if the effect is a clean floor, and the cleaning skill requires the vacuum to be turned on first, you must include turning on the vacuum before cleaning).

 Output Format
 - First, explain your reasoning
 - Then output a Python list containing the sequence of function calls that implements the PDDL action. Use following variables to reference pddl parameters, or strings if you need additional arguments where no pddl parameter exists
```
python
from variables import a_agent, obj_household_object, furn_furniture_appliance
```

```

### B.3 LLM<sub>decompose</sub>

```

You are given a high-level PDDL action and supporting context. Your task is to decompose this high-level action into a set of meaningful, lower-level PDDL actions that result in the same effect.

 ### High-level Action ‘(prepare_and_place_potato robot potato_1 plate_2)’
 prepare_and_place_potato: add
 - Description: Composite high-level action that slices, heats, and places the potato on a plate using the correct predicate.
 - PDDL Definition:
```
pddl
(:action prepare_and_place_potato
  :parameters (?a - agent ?p - potato ?pl - plate)
  :precondition (and (holding ?a ?p) (not (heated ?p)) (not (sliced ?p)))
  :effect (and (sliced ?p) (heated ?p) (at-receptacle ?p ?pl) (agent-hand-empty ?a) (not (holding ?a ?p)))
)
```


 ### Predicates
 - (at-agent ?a - agent ?loc - furniture_appliance): Agent is at a specific furniture piece or appliance.
 - (holding ?a - agent ?obj - household_object): Agent is holding an object.
 ...

 ### Types
 household_object furniture_appliance agent - object
 ...

 ### Objects
 drawer_1 drawer_2 ... - drawer fridge_1 - fridge ...

 ### Initial State
 (at-object cup_1 drawer_4) (at-object lamp_1 side_table_2) ...

 ### Goal State
 (at-receptacle potato_1 plate_2) (heated potato_1) ...

 ### Predefined Skill
 go_to_a_furniture_piece_or_an_appliance, pick_up_an_object_on_or_in_a_furniture_piece_or_an_appliance,
 ...

 ### Suggested Decomposition

```

```

```pddl
put_an_object_on_or_in_a_furniture_piece_or_an_appliance(p_potato, 'cutting_board_1')
go_to_a_furniture_piece_or_an_appliance('cabinet_1')
...
put_an_object_on_or_in_a_furniture_piece_or_an_appliance(pl_plate, 'cabinet_4')
```

Instructions
Follow the steps below to complete the decomposition:
1. Describe the Initial State
2. Understand the High-Level Action: Examine the :precondition and :effect. Identify what state change it induces.
3. Plan the Transition: Determine how the state should evolve from the :precondition to the :effect using grounded, lower-level actions. Refer to the suggested decomposition for inspiration. You may generalize, refine, or restructure it as needed for correctness and abstraction. Use reasoning to bridge the gap.
4. Define Lower-Level Actions: Construct new actions that together implement the high-level action.
 - Only include actions that result in state change.
 - Do not include meta-actions like confirm, complete, sense, or computation-related steps.
 - You may define new predicates if needed
 - Define individual PDDL actions for each skill you intend to use.
5. Specify the Goal State: List the predicates that reflect the intended outcome of the high-level action.
```

```

B.4 LLM_{reasoner}

```

You are given a decomposition hierarchy and a record of skills executed in a simulated environment. The last skill has failed during execution. Your goal is to identify why the observed effect of the simulation diverged from the expected effect of that skill. The simulation and skill implementations are correct and fixed - you cannot modify them. Your focus is on reasoning about the planning model and its action decomposition.

---

### Context:

Initial World State:
(at-object cup_1 drawer_4) (at-object lamp_1 side_table_2) ...

Decomposition Hierarchy
- (put_down_object robot mug_1 drawer_1): put_an_object_on_or_in_a_furniture_piece_or_an_appliance('mug_1', 'drawer_1')
- (go_to robot fridge_1): go_to_a_furniture_piece_or_an_appliance('fridge_1')
- (pick_up_object robot potato_1 fridge_1)

Executed Python Skill for PDDL Action 'pick_up_object':
pick_up_an_object_on_or_in_a_furniture_piece_or_an_appliance(obj_household_object,
loc_furniture_appliance)

Execution Failure:
Door of fridge_1 is not open

---

Your Task:
1. Summarize the state before executing the failed action.
2. Identify the cause of the deviation
  - could executing additional skills fix the error?: State the skills that should have been executed for the action.
  - if not, are any expected effects missing from the PDDL domain definition?: Adapt the PDDL action.
  - or, were any skills redundant, missing, or incorrectly ordered?
Note that:
  - The simulation and predefined skills are correct and unchangeable
  - Do not assume the current expected effects are correct - critically assess and revise them as needed
  - Also add negated predicates to the PDDL action effects and don't just delete them
```

```

## Assistant response

```

1. State Before Executing the Failed Action

- The robot is at 'fridge_1'.
- The 'fridge_1' is closed (since there is no predicate '(is-open fridge_1)'.
- The potato ('potato_1') is inside 'fridge_1'.
- The robot's hand is empty.
```

```

```

- Other environment details: multiple open furniture and appliances, but importantly, 'fridge_1' is not
  open.

### 2. Cause of Deviation and Analysis

(...)

### Final Notes

- The failure was not due to skill implementation errors but due to an incomplete PDDL model ignoring
  the prerequisite of door state.
- No skills are redundant or incorrectly ordered per se, but the planner must be informed of the opening
  requirement to plan the 'open' action prior to 'pick_up'.
- This reasoning applies to all picking from openable furniture/appliances (fridge, cabinets, drawers,
  etc.)..

---

### Updated PDDL 'pick_up_object' Action Definition

```
:pddl
(:action pick_up_object
 :parameters (?a - agent ?obj - household_object ?loc - furniture_appliance)
 :precondition (and
 (at-agent ?a ?loc)
 (at-object ?obj ?loc)
 (agent-hand-empty ?a)
 (not (holding ?a ?obj))
 (imply (openable ?loc) (is-open ?loc))
)
 :effect (and
 (holding ?a ?obj)
 (not (at-object ?obj ?loc))
 (not (agent-hand-empty ?a))
)
)
```

```

User prompt to get decision of reasoner

Determine the most probable fix type based on the following options:
 - the pddl action (pick_up_object robot potato_1 fridge_1) should be implemented with multiple skills: 'multiple-skills'
 - the pddl action can be implemented with one skill, but the action definition must be corrected: 'pddl-fix'

Additionally, reference the predefined Python skill or PDDL action that must be corrected.

Provide your final decision in the following format:
 - Type of Fix: '<chosen-fix-type>'
 - Correction: '<corrected-skill-or-action>'

Example 1:

- Type of Fix: 'pddl-fix'
 - Correction: '(go-to arm banana)'

Example 2:

- Type of Fix: 'multiple-skills'
 - Correction: 'grasp_part("banana")'

Final decision of reasoner

```

- Type of Fix: 'pddl-fix'
- Correction: 'pick_up_object'

```

B.5 GenPlan_{LLM}

```

### Predicates
- (at-agent ?a - agent ?loc - furniture_appliance): Agent is at a specific furniture piece or appliance.
- (holding ?a - agent ?obj - household_object): Agent is holding an object.
...

### Actions
1. slice_potato: add
  - Description: Slices a potato item.
  - PDDL Definition:
    ``pddl

```

```

(:action slice_potato
  :parameters (?a - agent ?p - potato)
  :precondition (and (holding ?a ?p) (not (sliced ?p)))
  :effect (sliced ?p)
)
```
...
4. prepare_and_place_potato: edit
- Description: Composite high-level action that slices, heats, and places the potato on a plate using the correct predicate.
- PDDL Definition:
  ````pddl
  (:action prepare_and_place_potato
    :parameters (?a - agent ?p - potato ?pl - plate)
    :precondition (and (holding ?a ?p) (not (heated ?p)) (not (sliced ?p)))
    :effect (and (sliced ?p) (heated ?p) (at-receptacle ?p ?pl) (agent-hand-empty ?a) (not (holding ?a ?p)))
  )
````

Objects
drawer_1 drawer_2 ... - drawer fridge_1 - fridge ...

Initial State
(at-agent robot drawer_1) (holding robot mug_1) ...

Goal State
(at-receptacle potato_1 plate_2) (sliced potato_1) (heated potato_1)

Output Format:
Your output should contain two parts:

1. Reasoning:
 Given the information provided, you should compute the plan to solve the problem.
 Add a short paragraph to explain your reasoning process.

2. Final Plan:
 Provide the best plan. Format it like this:
 If you think additional PDDL actions are needed, add them to the plan too.

````plan
(first-action arg1 arg2)
(unknown-action arg1 arg2)
...
(last-action arg1 arg2)
````
```