Lifelong Experience Abstraction and Planning

Peiqi Liu[1,2]*, **Joshua B. Tenenbaum**[1], **Leslie Pack Kaelbling**[1], **Jiayuan Mao**[1]
[1] Massachusetts Institute of Technology
[2] EECS, Peking University
peiqiliu@stu.pku.edu.cn, {jiayuanm,jbt,lpk}@mit.edu

Abstract

We present LEAP (Lifelong Experience Abstraction and Planning), a framework for continual behavior learning in embodied agents through interaction with the environment and guidance from humans. LEAP addresses the challenge of representing flexible knowledge about tasks and environments — ranging from constraints and subgoal sequences to action plans and high-level goals — in a unified framework. At its core, LEAP builds on the Crow Definition Language (CDL), a behavior rule language that integrates imperative programming with declarative planning by allowing agents to express both executable subroutines and subgoal hierarchies. Leveraging large language models (LLMs), LEAP translates diverse human instructions into CDL programs, generates planning-compatible code, and abstracts reusable behavior rules from successful executions to support future generalization. LEAP maintains a library of such CDL programs, enabling the agent to accumulate and refine its behavioral repertoire over time. We evaluate LEAP on the VirtualHome benchmark, demonstrating its ability to represent a wide variety of human instructions and its capacity to continually improve task performance through experience and interaction.

1 Introduction

To operate effectively in everyday human environments, an embodied agent must continually adapt—learning to use new tools, generalize across scenes, and acquire skills that may not have been present in its training data. Such learning can arise from diverse sources, including direct interaction with the environment and natural language guidance from humans. Human-provided instructions often convey both common-sense knowledge (e.g., how to cook a dish) and environment-specific information (e.g., what's in the fridge). While this guidance significantly enhances an agent's ability to plan and act, over-reliance on human input in all situations is costly. This raises a key challenge: How can we build agents that continually learn from environmental interactions and human instructions, abstract reusable behavior knowledge, and reduce their dependence on humans over time?

We address this challenge by introducing LEAP (Lifelong Experience Abstraction and Planning), a framework for embodied continual behavior learning. LEAP enables agents to acquire and reuse flexible behavior knowledge from diverse natural language instructions and environmental feedback. At the core of our framework is the Crow Definition Language (CDL; Mao et al., 2024), a behavior rule language that unifies imperative programming (e.g., action sequences) with declarative planning (e.g., subgoals and constraints), allowing the agent to represent both detailed execution steps and high-level task abstractions. However, two key challenges remain: (1) how to robustly translate diverse, ambiguous, and partially specified natural language into CDL programs, and (2) how to extract generalizable behavior rules that can transfer to new environments and goals, rather than simply recording task-specific action traces.

*Work done while Peiqi Liu was a visiting student at MIT.

ICML 2025 Workshop on Programmatic Representations for Agent Learning.

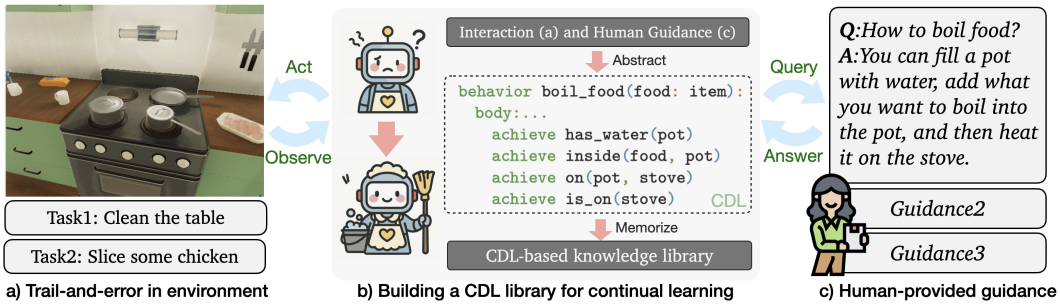

Figure 1: Lifelong Experience Abstraction and Planning (LEAP) is a framework for continual behavior learning through interaction with the environment and guidance from humans. LEAP abstracts valuable behavior knowledge from interaction (a) and human guidance (c) into reusable behavior rules (b).

To address these challenges, LEAP leverages large language models (LLMs) to parse human instructions into CDL programs, equipped with task decomposition and correction mechanisms that improve translation accuracy and sample efficiency. LEAP further abstracts generalized behavior programs from successful executions and stores them in a symbolic CDL library, enabling retrieval and adaptation for future tasks. This allows the agent to learn incrementally over time, continually refining its planning capabilities while reducing reliance on new human input.

We evaluate LEAP on a new human-in-the-loop benchmark based on the VirtualHome environment (Puig et al., 2018), featuring 210 long-horizon tasks across three distinct household scenes. We extend the original benchmark annotations to include detailed success metrics based on both goal satisfaction and critical action execution. Experimental results demonstrate that LEAP successfully handles diverse, unstructured instructions and continually improves its performance across environments. Notably, learning and planning with LEAP outperforms methods relying solely on human instructions, showing the system's ability to generalize and reuse knowledge across tasks.

In summary, our key contributions are: 1) An algorithm that translates diverse natural language instructions into structured behavior representations, 2) A mechanism for abstracting and storing reusable behavior rules for continual agent learning, and 3) A new human-in-the-loop benchmark of 210 challenging long-horizon tasks in three household environments for systematically evaluating continual behavior learning in embodied agents.

2 Related Work

Long-horizon embodied task planning. LLMs have been widely used in the design of embodied agents. An approach is to have LLMs directly generate action sequences (Huang et al., 2022; Wang et al., 2023b; Song et al., 2023; Joublin et al., 2024; Li et al., 2024b). However, this approach struggles with realistic tasks that involve hundreds of objects and complex object relations. To improve on this, some works have explored better prompting techniques (e.g., "chain-of-thought" and feedback) (Mu et al., 2023; Zhu et al., 2023) or the use of scene graphs (Liu et al., 2024; Ni et al., 2024). Another line of research focuses on translating task and environment knowledge into programs, typically using one of two representations: *1. Planning Domain Definition Language (PDDL)*, Liu et al. (2023); Guan et al. (2023); Wong et al. (2024); Smirnov et al. (2024) represent task as logical goals of the final state but cannot capture action orderings or constraints; *2. Python functions*, Liang et al. (2023); Wang et al. (2023a, 2024b) translate task completion into hierarchical functions, but do not involve search, making them less effective for tasks with implicit constraints. In this work, our framework leverages CDL programs — a behavior rule language that integrates imperative programming with declarative planning — to represent reusable behavior rules.

Embodied agent benchmarks. A number of embodied planning benchmarks have been proposed to evaluate agents' task planning abilities in embodied environments (Shridhar et al., 2020; Srivastava et al., 2022; Li et al., 2024a; Choi et al., 2024; Li et al., 2024b; Yang et al., 2025), providing task definitions, corresponding simulators (Kolve et al., 2017; Puig et al., 2018; Li et al., 2022; Puig et al., 2023), and evaluation metrics. However, few of them incorporate human interaction. TEACh (Padmakumar et al., 2022) provides agents with pre-collected human–human dialogues but does not support online agent-human communication. Wang et al. (2024a) and Chang et al. (2025) model the human as a cooperative agent but do not allow the human to provide guidance that helps the

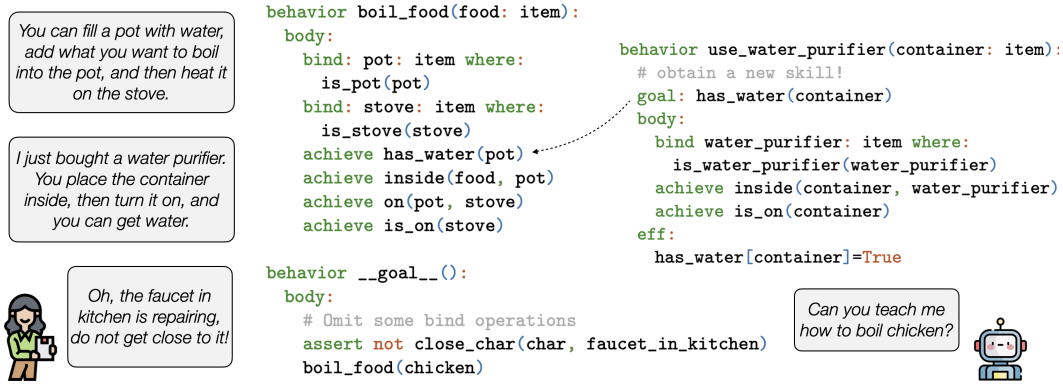

Figure 2: CDL can represent a diverse set of behavior knowledge.

agent improve. In our work, we create a human-in-the-loop benchmark based on the VirtualHome environment (Puig et al., 2018), featuring an LLM-based human teacher who provides diverse natural language guidance when queried by agents, allowing them to learn through interaction.

Library-based continual and lifelong learning. This refers to learning from a non-stationary and potentially endless stream of tasks, where the agent improves over time by reusing and expanding a library of accumulated knowledge. Zheng et al. (2024); Wang et al. (2024c) stores action histories and workflows, which are straightforward, but may be redundant, and lack generalization when the environment changes. Wong et al. (2024) build a PDDL library for planning, and Wang et al. (2024b) construct a Python function library — both methods rely solely on trial-and-error within the environment. Our method goes further by incorporating diverse human guidance, abstracting them into CDL programs, and storing them in a structured CDL library as pairs of subtask descriptions and corresponding CDL programs. This design facilitates reuse and enables fast adaptation to new environments.

3 Preliminaries

Problem formulation. We consider a system with three primary components: the environment, the agent, and the human. The environment is a tuple of $\langle \mathcal{X}, \mathcal{U}, \mathcal{T} \rangle$, where $\mathcal{X}$ is the state space, $\mathcal{U}$ is the primitive action space (See Appendix), and $\mathcal{T}$ is a deterministic transition function $\mathcal{T} : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$. We assume the state $\mathcal{X}$ is represented in an object-centric format, which includes a set of objects (the agent also considered as an object) and their features. For example, a simple scene where an agent is standing close to a table can be represented as an abstract state containing two objects, A and B, with the features $agent(A)$, $table(B)$, $close(A, B)$, $close(B, A)$. The agent has access to this abstract state, but the environment is partially observable. At each timestep, the agent can observe the features of objects that are directly visible (excluding objects in other rooms or inside closed containers). Initially, the agent only knows about objects that cannot be moved (e.g., sinks). It can discover new objects when they are close to the targets and when they open a new container.

Based on the environment, a collection of tasks are defined. Each task is described by a natural language instruction, corresponding to a success rate estimator $g(\{x_t, u_t\}_{t=1}^{H}) \to [0, 1]$. Given an initial state $x_0 \in \mathcal{X}$ and a natural language instruction, the target of the agent is to generate a sequence of actions $\{u1, u2, ..., u_H\}$ such that $g(\{x_t, u_t\}_{t=1}^{H})$ is 1 (where $x_t = \mathcal{T}(x_t, u_t)$). When the agent fails to complete a task for multiple attempts. The agent can request assistance from a human. The human, who is assumed to have full observability of the environment, provides guidance in format-free natural language based on complete information about the task and environment, helping the agent complete the task.

Behavior rule representation. To flexibly represent and reuse behavior knowledge such as action sequences and subgoal decomposition rules, we leverage CDL (Mao et al., 2024) as the representation language for the behavior library. CDL uses the same state representation as the problem formulated above, consisting of a set of objects and their features. The primary part of a CDL program is a set of behavior rules (e.g., *boil_food* in Figure 1). In general, the behavior rule contains two sections, a goal section, which is a logical expression over object features (e.g., the goal of behavior "*open*" is *opened(x)*), and a body section, which defines how to achieve that goal from the current state. In this work, we consider five types of statements in the body. 1) *bind statements*, which nondeterministically

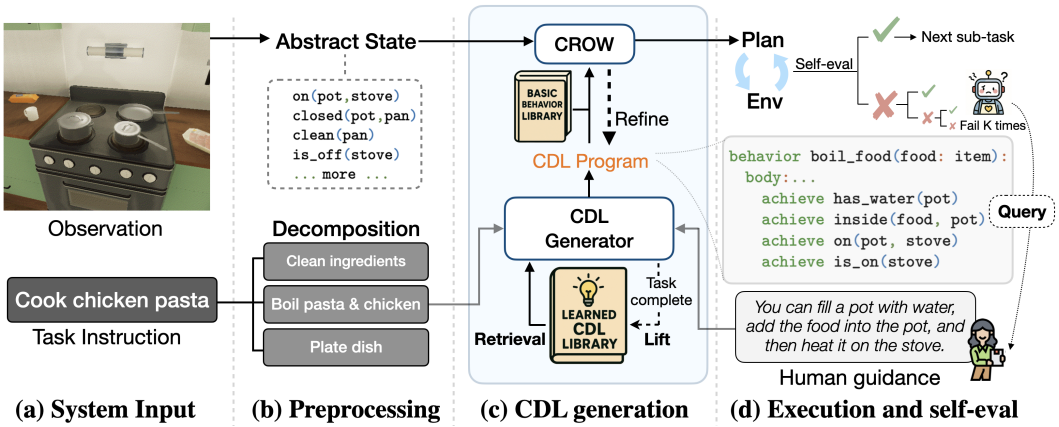

Figure 3: Overview pipeline of LEAP. Given a new task (a), LEAP first decomposes the task into subgoals (b), and performs retrieval-augmented generation of behavior rules of individual subgoals (c). Finally, it executes the plan with self-evaluation (d).

select an object satisfying a given constraint; 2) *achieve statements*, which should be recursively refined with another behavior rule whose goal matches the specified condition; 3) *assert statements*, which specify a condition that must hold along execution (i.e., a constraint that must continuously hold true); 4) *primitive actions* and 5) *procedure calls* to behavior rules. In addition to these two main sections, a behavior rule may also include an optional *effect* section, which describes the expected changes to the abstract state resulting from the execution of the behavior. In CDL, a goal can also be specified as a special "top-level" behavior rule (e.g., Figure 2). Accompanying CDL is a general-purpose planner, Crow, which interprets CDL programs and abstracted states to output plans. The planner uses a hierarchical search algorithm that supports a wide range of operations, from imperative policy execution to planning and constraint-based variable binding, all depending on the behavior specification.

4 Lifelong Experience Abstraction and Planning

LEAP is a framework for continual behavior learning and interaction. In this paper, we use a household environment (Puig et al., 2018) as an example domain. In its lifetime, a LEAP agent maintains a library of behavior rules represented in CDL. It is initialized with a given set of basic behavior rules that allow the agent to move itself in the environment and perform simple pick-and-place operations. Next, the agent receives a stream of tasks, each consisting of an initial state and a goal instruction. The agent not only solves each task but also extracts reusable behaviors from the experience and updates its behavior library.

Figure 3 illustrates how LEAP solves a single task. Upon receiving the initial observation and a task instruction, as shown in (a), the agent begins by decomposing the task into subtasks (b) and translating each subtask into a CDL program (c). During this translation process, the agent has access to the behavior library that stores knowledge abstracted from previous tasks. As a generalizable representation, CDL can adapt across diverse environments and goals. Shown in (d), the generated CDL programs are then interpreted as action sequences to execute the task. After executing each action sequence, an LLM evaluates whether the subtask has been successfully completed. If so, the agent proceeds to the next subtask; otherwise, it replans and retries the current one. If the agent fails to generate a valid plan after multiple attempts, it queries a human for additional guidance, which is then incorporated into the CDL generation process. Once the entire task is completed, the newly generated CDL programs are added to the library to improve future planning. Algorithm 1 illustrates the overall system in pseudocode.

4.1 State Representations and The Behavior Rule Library

State representations. At the beginning of each task, since we consider a partially observable setting, the agent only knows the stationary objects. The features of all movable objects are initially set to None. Two special features, *unknown(a)* and *checked(a,b)*, are used to handle partial observability. *unknown(a)* indicates that object a has not been discovered (all movable objects are initially marked

Algorithm 1 Pipeline Overview

Input: task description, guidance (Optional)
Initialize: empty the CDLPool at the start of each task

```
subtasks ← Decomposition(task description)                    ▷ Section 4.2
for each subtask do
    demos ← Library-Retrieval(subtask)                        ▷ Section 4.3
    CDL ← Refine(CDLGenerator(subtask, demos, guidance))      ▷ Section 4.3
    ActionSeq ← CROW(CDL)                                     ▷ Section 4.4
    if subtask completed after executing ActionSeq then
        CDLPool ← CDLPool ∪ CDL
if whole task completed then
    Library-Lift(CDLPool)                                     ▷ Section 4.4
```

as *unknown*). *checked(a,b)* means the agent has explored around object b but did not find object a there (See all features in Appendix A.2).

Behavior rule library. The agent is provided with a set of basic behavior rules that can be used to achieve certain goals under constraint-free conditions. For example, the behavior *turn_on* can achieve the goal specified by predicate *is_on(x)* when there is no additional constraints are present (e.g., the task do not prohibit the agent get close to *x*). However, this basic behavior rule set is often insufficient. It cannot handle situations involving additional constraints — such as when a necessary tool is unavailable or a particular action is prohibited. Moreover, some goals may not be achievable using any of the current basic behaviors (e.g., no basic behavior rules can achieve *cooked(x)*). Therefore, it is necessary for the agent to learn new behavior rules. We organize the behavior rule library into two parts: a predefined behavior rule library and a learned behavior rule library. The predefined rules are directly provided to the agent when interpreting the goal. The learned rules are stored as $\langle$subtask description, behavior rule$\rangle$ pairs, which can be retrieved as references when generating new behavior rules. Details are provided in Section 4.3.

4.2 Subtask Decomposition

Directly generating all behavior rules needed to solve a long-horizon, multi-stage task is challenging: LEAP uses an LLM-based module that decomposes a task into multiple subtasks. For example, in Figure 3, the task "cook chicken pasta" is decomposed into: cleaning the ingredients, boiling the food, and plating the prepared food. The task decomposition module is useful in two cases: *(1) Insufficient information*: when the agent lacks the necessary information for planning, such as the location of relevant objects. For example, consider the task "Put the items on the table in their proper places." If the agent does not know which items are on the table, it cannot make a reliable plan. In this case, a subtask such as "explore what items are on the table" should be added. *(2) long-horizon planning*: when the task includes multiple complex stages, such as "cook chicken pasta". Generating all behavior rules at once can be inefficient—if an early step fails and replanning is needed, the design effort for the later parts is wasted. In general, the number of subtasks is less than 4.

4.3 Retrieval-Augmented Behavior Rule Generation

Next, the agent generates a "top-level" goal behavior rule (*behavior __goal__()*) that accomplishes the task, along with additional rules that may be used to achieve subgoals within the behavior rule, using an LLM. In addition, we use a simple rule to query the human for additional guidance: humn guidance is requested when the agent fails to generate a valid plan or when the generated plan fails to complete the subtask after 5 attempts.

Relevant behavior rule retrieval. Recall that the behavior rule library stores a collection of subtask descriptions paired with CDL programs. When generating a new program, the agent retrieves up to K relevant examples by computing cosine similarity between sentence embeddings of the current subtask and those in the library, using the paraphrase-MiniLM-L6-v2 model from SentenceTransformer. The value of K is a tunable hyperparameter. In our experiments, we set $K = 200$.

Behavior rule generation. We prompt the LLM to generate new CDL behavior rules based on the following steps. We guide GPT-4o to reason through intermediate states required for task completion using "Chain-of-Thought" (CoT) prompting. The input to the LLM consists of: 1) the subtask description and optionally additional human-provided guidance; 2) CDL syntax, including predicates,

keywords, and behavior definitions; 3) CoT examples, comprising two basic CoT-guided examples; 4) K pairs of subtask descriptions and behavior rules retrieved from the library. The LLM outputs codes that use the keyword *achieve* to represent subgoals and *assert* to specify constraints. Direct actions and behavior calls are also supported. Figure 2 gives a concrete example.

Refinement mechanism. Generated behavior rules may contain syntax and logical errors. When errors exist, instead of directly resampling, we employ a refinement mechanism that handles different types of errors separately. If the program contains syntax errors, such as incorrect keyword usage or missing parameters, we design rule-based methods to automatically parse the erroneous parts and rectify them. For logic-related issues where the CDL program is syntactically correct but the interpreter fails to generate a plan, we introduce an LLM-based debugger to refine the program. The debugger is prompted with the current plan, the planning state, and a list of common logical errors along with their corresponding corrections to guide the refinement process.

Exploration rules. An important class of newly generated behavior rules is exploration behaviors for object search. For task-relevant objects that are initially unknown to the agent, we prompt LLMs to predict likely locations based on the objects' category names. This knowledge is encoded as additional behavior rules with the goal of "finding an object of a given type." Details are provided in the appendix. After generating all behavior rules for task completion and object search, a plan is constructed under the assumption that the unknown object can be found at the predicted location. This corresponds to the "most-likely-observation" heuristic used in partially observable MDP algorithms. Importantly, even if the object is not at the predicted location, the agent will continue to observe and replan, allowing it to adapt and search in alternative locations.

4.4 Execution and self-evaluation

Plan execution. Given the set of basic, exploration, and task-relevant behavior rules, the CROW algorithm will find a plan that achieves the goal of the current subtask. The plan is a sequence of actions that can be directly executed in the environment. During execution, the agent receives new observations about nearby objects, which will be used to update its internal state representation. Replanning is triggered whenever the initial plan is invalidated — for example, if the agent assumes the chicken is in the fridge but fails to find it after opening it, or if a predicted action becomes non-executable.

Self-evaluation. Since we can not directly receive subtask completion signal from the environment, after executing the generated action sequence for a subtask, the agent uses an LLM to evaluate whether the subtask has been fully completed. The LLM receives the action and observation history, and determines whether to proceed to the next subtask or to replan the current one.

CDL programs lifting. Each time the agent successfully completes an entire task, the CDL programs used to generate plans for each subtask are paired with their corresponding subtask descriptions and added to the CDL library for future use.

5 VirtualHome-HG

We introduced a new dataset, VirtualHome-HG (Human Guidance), built on the VirtualHome simulator (Puig et al., 2018, 2021). The dataset defines 210 diverse tasks across three different annotated scenes. Specifically, the dataset includes 93 cooking tasks, 33 cleaning tasks, 27 laundry tasks, and 57 rearrangement tasks. On average, each scene contains 376 distinct items spanning 157 categories. The benchmark supports both symbolic execution and visual simulation via the VirtualHome environment. The action space consists of discrete actions such as walk, grab, and put (see the full list in the Appendix 6).

Evaluation metric. We evaluate task completion from two perspectives: task completion rate and key action execution rate. To compute the task completion rate, first, we define a ground truth goal for each task, represented also in CDL. We compute an "oracle" plan based on this ground truth goal, a fully-observed environment state, and a set of human-defined behavior rules to find a plan. The length of this plan, denoted as $\mathcal{L}_{init}$, reflects the reference number of steps required to complete the task. Upon the agent finishing its execution, based on the final goal state, we compute another plan based on ground truth goals and behavior rules, whose length is $\mathcal{L}_t$. The task completion rate is computed as: $\mathcal{C}_{\text{goal}} = \max\{(\mathcal{L}_{\text{init}} - \mathcal{L}_t)/\mathcal{L}_{\text{init}}, 0\}$. When the agent has successfully solved the task, $\mathcal{C}_{goal} = 1$. We also compute a key-action execution rate: We manually annotate key actions required for each task and calculate how many of these actions were executed, denoted as $\mathcal{C}_{\text{action}} = \mathcal{N}_{\text{execute}}/\mathcal{N}_{\text{required}}$.

Method	Simple		Multi-stage		Ambiguous		Constraint		Total	
	WOG	WG	WOG	WG	WOG	WG	WOG	WG	WOG	WG
LLM Policy	70.4%	70.9%	49.5%	51.0%	46.2%	51.0%	37.2%	36.9%	59.1%	59.3%
LLM+P	88.5%	84.2%	44.5%	60.7%	48.0%	52.9%	47.0%	60.4%	67.8%	70.1%
Code as Policy	80.4%	89.6%	48.6%	64.1%	39.8%	50.3%	29.3%	42.5%	61.7%	69.9%
Voyager	84.7%	89.4%	61.0%	65.5%	47.8%	55.5%	34.5%	48.3%	70.1%	76.4%
Ours (ActSeq)	86.0%	90.4%	51.0%	66.3%	49.4%	53.4%	44.0%	55.5%	69.9%	74.0%
Ours (Full)	**92.6%**	**94.9%**	**66.5%**	**69.5%**	**50.5%**	**64.8%**	**49.9%**	**65.3%**	**75.6%**	**80.1%**

Table 1: **Main results.** WOG: without human guidance; WG: with human guidance.

Required actions can have logical combinations, such as $(a_1\ then\ a_2)\ or\ (a_3\ then\ a_4)$ (i.e., execute $a1$ followed by $a2$, or $a3$ followed by $a4$). The overall completion rate is a weighted sum of the two types of completion rates $\mathcal{C} = 2/3 \cdot \mathcal{C}_{action} + 1/3 \cdot \mathcal{C}_{goal}$.

Human guidance. The ideal setting for evaluating agent learning from humans would involve a real human providing real-time responses to agents. However, this approach is not only costly but also challenging to standardize, making it difficult to ensure fair comparisons across different methods and users. To address this, we annotate high-quality guidance within the dataset, describing how to complete a task step by step using unstructured natural language. Notably, this language is written to mimic human communication, such as how a parent might teach their child, without specifically using robotic actions or referencing specific instances. We then introduce an LLM-based human agent with access to these guidance annotations. When asked a question, the human agent responds in a natural, human-like manner based on the provided guidance.

6 Experiments

We compare our method with three baselines without library learning and two baselines with library learning: **LLM Policy** directly uses an LLM to generate action sequences based on task information and predefined action rules. **LLM+P** (Liu et al., 2023) uses an LLM to translate the task into logical goals of the final state, then employs a planner to generate a plan. **Code-as-Policy** (Liang et al., 2023) uses an LLM to decompose the task, translates each subtask into Python functions, and executes them to obtain a plan.

In the second group, we have **Voyager-like** (Wang et al., 2024b) which additionally stores pairs of subtask descriptions and corresponding Python functions for successful tasks based on Code-as-Policy and **Ours(ActSeq)** which replaces stored pairs of subtask descriptions and CDL programs with pairs of subtask descriptions and action-observation sequences.

We evaluate each method on all 210 tasks in our benchmark without human guidance (WOG) and with human guidance (WG). Both settings permit the agent to query the human about the location of unknown objects after 5 failed exploration attempts. In the WG setting, the agent can also ask for human guidance after 5 failed attempts to generate or execute a plan. We further select 4 subsets of tasks to evaluate specific capabilities of each method:

Simple Set: Includes 78 single-stage tasks, typically requiring fewer than 15 actions. This set evaluates the agent's basic task completion ability.

Multi-stage Set: Comprises 30 multi-stage tasks requiring 30 to 150 actions, designed to evaluate the agent's ability in long-horizon reasoning and multi-stage problem solving.

Ambiguous Set: Consists of 57 tasks with highly ambiguous descriptions, such as "Make fried bananas." These tasks are difficult for LLMs to solve using only common-sense reasoning and rely heavily on additional human guidance. This set evaluates the agent's ability to understand and utilize human guidance.

Constraint Set: Includes 30 tasks with strong implicit size constraints, requiring the agent to reason about size relationships between objects. For example, a task may involve selecting a cup smaller than the coffee machine to successfully prepare coffee.

Some tasks are not included in any of the four subsets, as they do not clearly meet the criteria defined above. However, all 210 tasks are included in the total evaluation.

6.1 Results

Overview. LLM Policy performs poorly even on simple tasks, due to its difficulty in reasoning over many objects and their complex relationships. LLM+P struggles with multi-stage tasks because it cannot decompose tasks and is limited by the representational ability of PDDL, which can not specify action orderings or constraints. Code-as-Policy and Voyager translate tasks into imperative Python functions, perform poorly on tasks with implicit constraints (e.g., action *putin(a,b)* implicitly requires $size(a) < size(b)$). Since they cannot search alternatives (e.g., trying different cups). Our method addresses these limitations and is capable of solving complex tasks with diverse human instructions.

Our framework better leverages human guidance. As shown in Table 1, our framework achieves the highest performance and the largest improvement in Ambiguous Set after receiving human guidance (14.3%), compared to LLM+P(4.9%), Code-as-Policy (10.5%), and Voyager(7.7%). These tasks typically involve an ambiguous task description, making it difficult for LLMs to generate solutions using only common-sense. Considering "Make a cup of coffee using a coffee machine," where all methods receive guidance, including the information "use a cup of proper size." Code-as-Policy and Voyager achieve success rates under 20% because they fail to infer that "proper" refers to constraints $size(cup) < size(coffee\_machine)$, and select too large cups. LLM+P, despite supporting search, achieves only a 58% success rate due to its lack of task decomposition. Our framework achieves a 100% success rate. It translates natural language into CDL programs that represent constraint specification, and finds plans with search, task decomposition, and self-evaluation.

CDL library stores reusable knowledge and benefits future planning. As illustrated in Figure 4, the performance of our method demonstrates continous and faster improvement compared other library-based approaches (Voyager and Ours(ActSeq)). In contrast, baseline methods without library learning show no increase in success rates, indicating that the tasks do not becoming easier. To further demonstrate the benefits of constructing a CDL library, we conducted ablation experiments (Table 2, comparison between w/o CDL Library and Full) and a focused evaluation on cooking tasks (Figure 5). As Table 2 shows, removing the CDL library resulted in a noticeable decrease in success rates across all types of task.

Cooking tasks often share common knowledge (e.g., cleaning or boiling), so we design focused experiments to show how the CDL library enhances performance. Tasks are grouped into 3 levels: 24 easy, 36 medium, and 30 hard. In the continual learning setup (Figure 5), the agent first learns from human-guided easy and medium tasks before attempting harder ones, with no human help during testing. Prior experience significantly improves performance — by 12% on medium and 17% on hard tasks — outperforming direct human guidance, which yields only 5% and 6% gains, respectively.

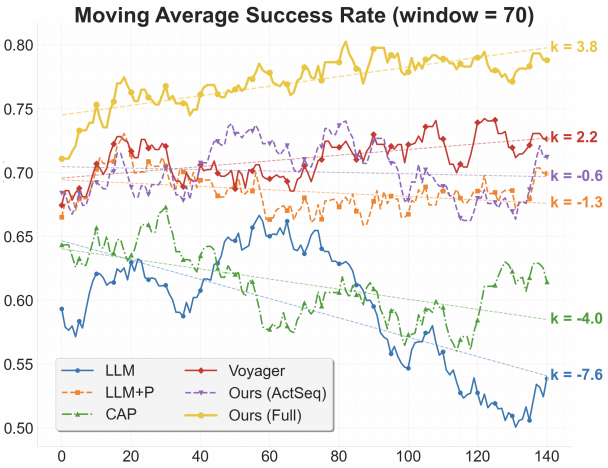

Figure 4: Moving average success rates of different methods across 210 tasks, all performed without human guidance. Baseline methods without the library show no improvement, suggesting that the tasks do not become easier over time. In contrast, our method shows clear improvement, demonstrating its ability to learn and adapt.

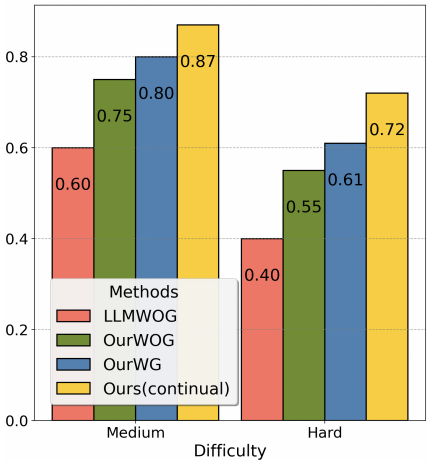

Figure 5: Success rate in the cooking test for different settings. "Continual" refers to the agent having completed easier tasks with human guidance and built a CDL library, without human input during the test.

	Normal	Multi-stage	Ambiguous	Constraint	Total	Gen. Time
w/o CDL Library	87.7%	69.0%	61.0%	56.2%	76.6%	1339
w/o Refinement	93.6%	**70.1**%	61.2%	**68.3**%	79.6%	1598
w/o Decomposition	93.0%	56.4%	62.7%	48.8%	77.4%	**871**
Full	**94.9**%	69.5%	**64.8**%	65.3%	**80.1**%	1428

Table 2: Results of ablation experiments. Gen. Time denotes the total time taken to generate CDL programs for all 210 tasks using each method.

Storing CDL programs is better than storing action history. Storing interaction history is a common strategy in continual learning (Zheng et al., 2024; Wang et al., 2024c). However, as shown in Table 1 and Figure 4, storing CDL programs yields significantly better results in all kinds of tasks. Our insight is that action history is highly dependent on the specific scene. When the scene changes, stored actions may refer to target objects that no longer exist or actions that can no longer be executable. Additionally, statistical results show that the average length of action history (4,889) from a single task is ten times longer than the corresponding CDL program (492) and often includes many failed attempts. These observations suggest that abstraction is essential.

Sub-task decomposition facilitates feasible CDL program generation. Table 2 shows that incorporating subtask decomposition leads to increased overall task performance, particularly for Multi-stages tasks and Constraint tasks. The improvement in Constraint tasks occurs because such tasks always consist of multiple stages, such as placing specific items at different locations. The performance gain from subtask decomposition comes from simplifying CDL behavior rule generation by handling one subtask at a time.

Refinement is more efficient than resampling. The CDL interpreter provides valuable feedback, allowing the refinement mechanism to correct errors more efficiently. As shown in Table 2, incorporating the refinement mechanism reduces CDL generation time by approximately 10% while achieving even better overall performance.

7 Conclusion

We present LEAP, a framework for continual behavior learning that can learn from both human guidance and interaction with the environment. Our experiments demonstrate that LEAP can better utilize diverse human guidance and achieve better continual learning ability than prior PDDL-based and Python-function-based methods. Although this work focuses on embodied planning tasks, the core ideas of LEAP can also be applied to other domains requiring lifelong learning.

Limitations. A limitation of LEAP is the increased number of LLM queries due to subtask decomposition and self-evaluation. Furthermore, our simple human-querying strategy leaves open how to ask more precise, informative questions.

References

Matthew Chang, Gunjan Chhablani, Alexander Clegg, Mikael Dallaire Cote, Ruta Desai, Michal Hlavac, Vladimir Karashchuk, Jacob Krantz, Roozbeh Mottaghi, Priyam Parashar, Siddharth Patki, Ishita Prasad, Xavier Puig, Akshara Rai, Ram Ramrakhya, Daniel Tran, Joanne Truong, John M Turner, Eric Undersander, and Tsung-Yen Yang. PARTNR: A benchmark for planning and reasoning in embodied multi-agent tasks. In *ICLR*, 2025.

Jae-Woo Choi, Youngwoo Yoon, Hyobin Ong, Jaehong Kim, and Minsu Jang. Lota-bench: Benchmarking language-oriented task planners for embodied agents. In *ICLR*, 2024.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. In *NeurIPS*, 2023.

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

Frank Joublin, Antonello Ceravola, Pavel Smirnov, Felix Ocker, Joerg Deigmoeller, Anna Belardinelli, Chao Wang, Stephan Hasler, Daniel Tanneberg, and Michael Gienger. Copal: corrective planning of robot actions with large language models. In *ICRA*, 2024.

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017.

Chengshu Li, Fei Xia, Roberto Martín-Martín, Michael Lingelbach, Sanjana Srivastava, Bokui Shen, Kent Elliott Vainio, Cem Gokmen, Gokul Dharan, Tanish Jain, Andrey Kurenkov, Karen Liu, Hyowon Gweon, Jiajun Wu, Li Fei-Fei, and Silvio Savarese. igibson 2.0: Object-centric simulation for robot learning of everyday household tasks. In *CoRL*, 2022.

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Wensi Ai, Benjamin Martinez, et al. Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation. *arXiv preprint arXiv:2403.09227*, 2024a.

Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Li Erran Li, Ruohan Zhang, et al. Embodied agent interface: Benchmarking llms for embodied decision making. In *NeurIPS*, 2024b.

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. In *ICRA*, 2023.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

Yuchen Liu, Luigi Palmieri, Sebastian Koch, Ilche Georgievski, and Marco Aiello. Delta: Decomposed efficient long-term robot task planning using large language models. *CoRR*, abs/2404.03275, 2024.

Jiayuan Mao, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Hybrid declarative-imperative representations for hybrid discrete-continuous decision-making. In *WAFR*, 2024.

Yao Mu, Qinglong Zhang, Mengkang Hu, Wenhai Wang, Mingyu Ding, Jun Jin, Bin Wang, Jifeng Dai, Yu Qiao, and Ping Luo. Embodiedgpt: Vision-language pre-training via embodied chain of thought. In *NeurIPS*, 2023.

Zhe Ni, Xiaoxin Deng, Cong Tai, Xinyue Zhu, Qinghongbing Xie, Weihang Huang, Xiang Wu, and Long Zeng. Grid: Scene-graph-based instruction-driven robotic task planning. In *IROS*, 2024.

Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. Teach: Task-driven embodied agents that chat. In *AAAI*, 2022.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *CVPR*, 2018.

Xavier Puig, Tianmin Shu, Shuang Li, Zilin Wang, Yuan-Hong Liao, Joshua B. Tenenbaum, Sanja Fidler, and Antonio Torralba. Watch-and-help: A challenge for social perception and human-{ai} collaboration. In *ICLR*, 2021.

Xavier Puig, Eric Undersander, Andrew Szot, Mikael Dallaire Cote, Tsung-Yen Yang, Ruslan Partsey, Ruta Desai, Alexander William Clegg, Michal Hlavac, So Yeon Min, et al. Habitat 3.0: A co-habitat for humans, avatars and robots. *arXiv preprint arXiv:2310.13724*, 2023.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *CVPR*, 2020.

Pavel Smirnov, Frank Joublin, Antonello Ceravola, and Michael Gienger. Generating consistent pddl domains with large language models. *arXiv preprint arXiv:2404.07751*, 2024.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *ICCV*, 2023.

Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, et al. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *CoRL*, 2022.

Chenxu Wang, Boyuan Du, Jiaxin Xu, Peiyan Li, Di Guo, and Huaping Liu. Demonstrating humanthor: A simulation platform and benchmark for human-robot collaboration in a shared workspace. *arXiv preprint arXiv:2406.06498*, 2024a.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *TMLR*, 2024b. ISSN 2835-8856.

Yuki Wang, Gonzalo Gonzalez-Pumariega, Yash Sharma, and Sanjiban Choudhury. Demo2code: From summarizing demonstrations to synthesizing code via extended chain-of-thought. *NeurIPS*, 2023a.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with LLMs enables open-world multi-task agents. In *NeurIPS*, 2023b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c.

Lionel Wong, Jiayuan Mao, Pratyusha Sharma, Zachary S Siegel, Jiahai Feng, Noa Korneev, Joshua B Tenenbaum, and Jacob Andreas. Learning grounded action abstractions from language. In *ICLR*, 2024.

Rui Yang, Hanyang Chen, Junyu Zhang, Mark Zhao, Cheng Qian, Kangrui Wang, Qineng Wang, Teja Venkat Koripella, Marziyeh Movahedi, Manling Li, et al. Embodiedbench: Comprehensive benchmarking multi-modal large language models for vision-driven embodied agents. *arXiv preprint arXiv:2502.09560*, 2025.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *ICLR*, 2024.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023.

A Dataset

A.1 Environment

Our benchmark is built on the VirtualHome simulator (Puig et al., 2018). We select three distinct scenes, enhance them with additional objects, introduce new environment states, and annotate the simulator to support a broader range of tasks defined in our benchmark, such as cooking and doing laundry. Figure 6 visualizes these scenes.

Figure 6: Visualization of the selected VirtualHome scenes.

A.2 State representation

VirtualHome includes a set of built-in features, and the original simulator supports transitions between these states. To construct our benchmark, we select a subset of these features — removing onces such as *Facing* and *Cream* — and add two new states *cooked* and *has_water* to better support cooking-related tasks. In addition, to design more realistic tasks with size constraints, we introduce the *size* feature. All state features (Table 3), relationships features (Table 4), and property features (Table 5) are listed below.

Feature	Description
is_on(x: item)	The item is switched on
is_off(x: item)	The item is switched off
plugged(x: item)	The item is plugged in
unplugged(x: item)	The item is unplugged
open(x: item)	The item is open
closed(x: item)	The item is closed
dirty(x: item)	The item is dirty
clean(x: item)	The item is clean
cut(x: item)	The item has been cut
inhand(x: item)	The item is in hand
has_water(x: item)	The item contains water (new add)
cooked(x: item)	The item is cooked (new add)

Table 3: State features

Feature	Description
on(x: item, y: item)	Item x is on item y
inside(x: item, y: item)	Item x is inside item y
close(x: item, y: item)	Item x is close to item y
holds_rh(x: character, y: item)	The character holds item y with right hand
holds_lh(x: character, y: item)	The character holds item y with left hand
close_char(x: character, y: item)	The character is close to item y
inside_char(x: character, y: item)	The character is inside item y

Table 4: Relationship features

Feature	Description
surfaces(x: item)	The item has a surface
grabbable(x: item)	The item can be grabbed
hangable(x: item)	The item can be hung
recipient(x: item)	The item can receive contents
cuttable(x: item)	The item can be cut
pourable(x: item)	The item can be used for pouring
can_open(x: item)	The item can be opened
has_switch(x: item)	The item has a switch
containers(x: item)	The item can contain other items
has_plug(x: item)	The item has a plug
movable(x: item)	The item can be moved
is_food(x: item)	The item is food
size(x: item) $\rightarrow$ int64	Returns the size of the item (new add)

Table 5: Property features

A.3 Action space

Table 6 lists all valid actions considered in our benchmark. Notably, the action *exp* and *obs* are newly introduced to address the partially observable setting in our tasks, enabling agents to actively explore the environment.

Controller	Description
walk_executor(x: item)	Makes the character walk toward the item
switchoff_executor(x: item)	Turns off the specified item (e.g., a lamp)
switchon_executor(x: item)	Turns on the specified item (e.g., a lamp)
put_executor(x: item, y: item)	Places item x onto item y (e.g., put cup on table)
putin_executor(x: item, y: item)	Puts item x inside item y (e.g., put apple into fridge)
grab_executor(x: item)	Grasps the item (e.g., pick up a cup)
wash_executor(x: item)	Washes the item (e.g., wash a dish)
scrub_executor(x: item)	Scrubs the item (e.g., scrub a pot)
rinse_executor(x: item)	Rinses the item with water (e.g., rinse a plate)
open_executor(x: item)	Opens the item (e.g., open a door)
close_executor(x: item)	Closes the item (e.g., close a fridge)
pour_executor(x: item, y: item)	Pours the contents of item x into item y
plugin_executor(x: item)	Plugs in the item (e.g., plug in a toaster)
plugout_executor(x: item)	Unplugs the item (e.g., unplug a toaster)
cut_executor(x: item)	Cuts the item (e.g., cut a carrot)
touch_executor(x: item)	Touches the item (e.g., touch a button)
type_executor(x: item)	Types on the item (e.g., keyboard)
push_executor(x: item)	Pushes the item (e.g., push a chair)
pull_executor(x: item)	Pulls the item (e.g., pull a drawer)
exp(x: item, y: item)	Expresses a relation or interaction between items (new add)
obs(x: item, q: string)	Makes an observation or query about the item (new add)

Table 6: The action space we considered in this work

A.4 Examples

A.4.1 Task examples

The first example is a simple task that illustrates the basic components of a task definition. Each task in our dataset includes a Task Name, Task Description, Guidance, Required Action, Keystate Logic, and corresponding key behavior rules. The Guidance is provided to the LLM-based human agent to enable them to offer assistance. The Required Action is used for key action evaluation. Keystate Logic specifies the logical composition of the keystate, which can be used to compute task completion rate.

```
Task name: Wash windows
Task description: Wipe all the windows in the house by towel.
Guidance: Hold a towel. And wipe all the windows in the house.
Required Action:
S0_Actions: wipe_executor(window_63) and wipe_executor(window_86) and
wipe_executor(window_348)
S1_Actions: wipe_executor(window_2156) and wipe_executor(window_191)
and wipe_executor(window_310)
S2_Actions: wipe_executor(window_2109) and wipe_executor(window_40)
and wipe_executor(window_181) and wipe_executor(window_287) and
wipe_executor(window_346)
Keystate Logic: k1

behavior k1():
    body:
        bind towel: item where:
            is_towel(towel)
        achieve inhand(towel)
```

The second example is a task in Constraint Set:

```
Task name: Make coffee
Task description: Make a cup of coffee using the coffee maker.
Guidance: Put a proper sized cup into the coffee machine and start it.
 The coffee machine is already filled with water and coffee beans.
Simply place the cup and start the machine.
Required Action: None
Keystate Logic: k1

behavior k1():
    body:
        bind coffe_maker: item where:
            is_coffe_maker(coffe_maker)
        bind cup: item where:
            is_cup(cup)
        achieve inside(cup, coffe_maker)
        achieve is_on(coffe_maker)
```

The third example presents a multi-stage, long-horizon task.

```
Task name: Cook chicken pasta
Task description: Make a plate of chicken pasta
Guidance: To boil pasta, fill a pot with water. Put the pot on the
stove and turn on the stove. Then put the pasta into the pot. To cook
chicken, add some oil in a fryingpan and heat it on the stove. Then
put the chicken into the fryingpan. Lastly, put the pasta and chicken
in a plate.
Required Action: None
Keystate Logic: ((k1 then k2) or (k2 then k1)) then (k3 or k4)

behavior k1():
    body:
        bind pasta: item where:
            is_dry_pasta(pasta)
        bind stove: item where:
            is_stove(stove)

        symbol pot_with_water=exists pot:item: is_pot(pot) and
        has_water(pot)
        if not pot_with_water:
            bind pot: item where:
                is_pot(pot)
```

```
            achieve has_water(pot)
            achieve inside(pasta, pot)
            achieve on(pot, stove)
            achieve is_on(stove)

        else:
            bind pot: item where:
                is_pot(pot) and has_water(pot)
            achieve inside(pasta, pot)
            achieve on(pot, stove)
            achieve is_on(stove)

behavior k2():
    body:
        bind chicken: item where:
            is_food_chicken(chicken)
        bind fryingpan: item where:
            is_fryingpan(fryingpan)
        bind stove: item where:
            is_stove(stove)
        bind oil: item where:
            is_oil(oil)
        achieve inside(oil,fryingpan)
        achieve on(fryingpan, stove)
        achieve is_on(stove)
        achieve inside(chicken, fryingpan)

behavior k3():
    body:
        bind plate: item where:
            is_plate(plate)
        bind pasta: item where:
            is_dry_pasta(pasta)
        bind chicken: item where:
            is_food_chicken(chicken)
        achieve inside(pasta, plate)
        achieve inside(chicken, plate)

behavior k4():
    body:
        bind plate: item where:
            is_plate(plate)
        bind pasta: item where:
            is_dry_pasta(pasta)
        bind chicken: item where:
            is_food_chicken(chicken)
        achieve on(pasta, plate)
        achieve on(chicken, plate)
```

A.4.2 Agent-Human communication examples

In this section, we present several examples of question-answering interactions recorded in the experimental logs:

- **Question:** Can you teach me how to boil the vegetables?
  Answer: First, clean the vegetables. Then, fill a pot with water and heat it on the stove. Finally, place the vegetables in the pot.
- **Question:** Can you teach me how to iron the skirt?
  Answer: Place the skirt on the ironing board. Turn on the iron, then gently move it back and forth over the skirt.

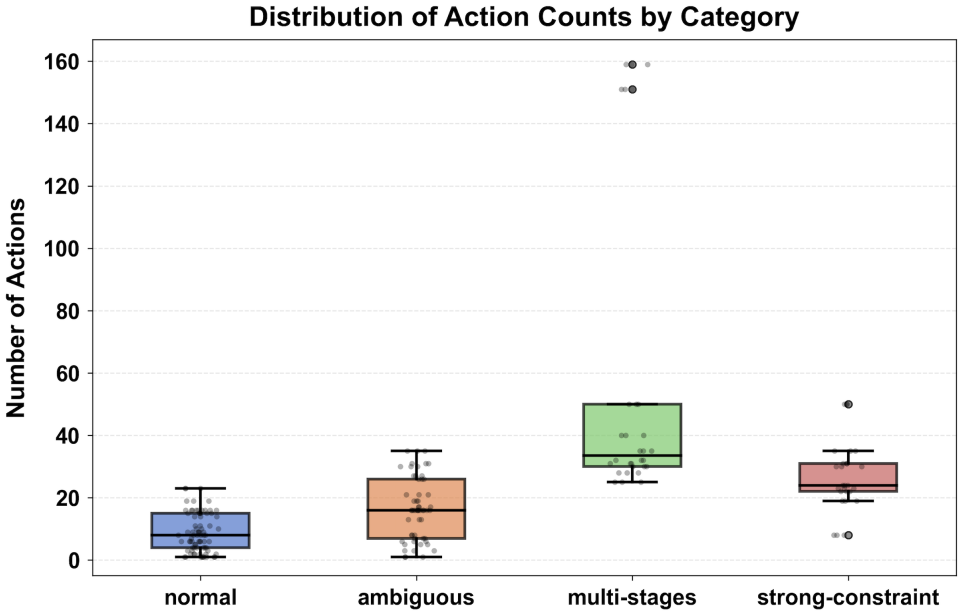

Figure 7: Statistics of the required action lengths for the four experiment task sets

- **Question:** Can you teach me how to add coffee grounds and water to the coffee maker?
  Answer: The coffee maker is already filled with water and coffee beans. Simply place a cup under the spout and start the machine.
- **Question:** Can you teach me how to fill a pot with water?
  Answer: Hold the pot under a faucet, then turn on the faucet to fill it with water.
- **Question:** Can you teach me how to slice chicken?
  Answer: Place the chicken on a cutting board. Hold a knife firmly and slice the chicken into pieces.

A.5 Dataset Statistics

Our benchmark consists of 210 tasks spanning three different scenes, including 93 cooking tasks, 33 cleaning tasks, 27 laundry tasks, and 57 rearrangement tasks. These tasks cover a wide range of complexity, from the simplest requiring only a single action to the most complex involving up to 159 actions.

In addition to the task type, we further categorize the tasks into four sets based on their properties: 78 normal tasks, 57 ambiguous tasks, 30 multi-stage tasks, and 27 strong-constraint tasks. On average, each task requires 19.8 actions (median: 16.0; standard deviation: 25.3). As shown in Figure 7, normal tasks exhibit the lowest and most consistent action counts (mean: 8.9; range: 1–23), while ambiguous tasks show moderate complexity with notable variability (mean: 16.5; range: 1–35). Multi-stage tasks are the most complex, requiring significantly more actions (mean: 58.1; range: 25–159) and demonstrating the highest variation. Strong-constraint tasks also reflect moderate complexity but with more controlled variation (mean: 26.9; range: 8–50). This hierarchical structure of task complexity offers a comprehensive testbed for evaluating performance across varying difficulty levels and diverse scenarios, effectively capturing the breadth of real-world task challenges.

B Implementation details

B.1 LLM Prompts

In this work, we use gpt-4o-2024-08-06 as the LLM component for all methods. Below, we present the prompts used for each module.

Subtask Decomposition for a given task, we prompt the LLM to decide whether the task should be decomposed into multiple subtasks or executed directly. The few-shot prompt is composed of the following elements:

1. A task decomposition principle block that provides high-level guidance, encouraging minimal subtask count for simple tasks.
2. A sequence of examples, each including:
   - A natural language task description;
   - A list of completed subtasks (or none if the current subtask is the first subtask);
   - Optional human guidance;
   - A structured output: a list of next subtask or `No decomposition`.
3. A final instruction block that reinforces the output formatting rules, including when to use `No decomposition` and how to number subtasks if decomposition is necessary.
4. A dynamically filled section with the current task information, including the current task description, completed subtasks, and human guidance.

CDL Generator The prompt design for this core module is described in the main text (Section 4.3) and is omitted here.

CDL Debugger (Logical Error) The syntax error can be directly corrected by rule-based functions, and this LLM-based debugger is used to correct logical errors. The prompt is composed of the following elements:

1. A task description section that includes the current subtask, the overall task description, completed subtasks, and additional observations if available.
2. The incorrect CDL program and corresponding error message (from CROW).
3. A set of examples illustrates common errors and their corrected versions
4. A final instruction block that requires the model to revise the CDL program directly.

Here we provide one of the examples given to the CDL debugger that corrects logical errors:

```
- Error example:
behavior put_apple_on_table(apple:item,table:item):
    body:
        achieve inhand(apple)
        achieve on(apple,table)

Error Analysis: In this example, using achieve inhand(apple) means the
 apple must remain in hand until the end of the behavior. However,
achieve
on(apple, table) indicates that the apple should be placed on the
table.
This leads to a contradiction. The apple cannot be both held and
placed at the same time. The solution is to remove the unnecessary
inhand(apple)
condition, as on(apple, table) is the actual intended goal of this
behavior rule.

- Corrected behavior rule:
behavior put_apple_on_table(apple:item,table:item):
    body:
        achieve on(apple,table)
```

Self-Eval for a given subtask, we prompt the LLM to decide whether the subtask has been completed based on a structured summary of actions taken, CDL program, and collected observations. The prompt is composed of the following elements:

1. A task overview section that encourages evaluating based on action effects and known information, assuming all actions succeed and ignoring timing constraints.

2. A dynamically filled section that includes:
    - The current subtask to evaluate;
    - A list of actions already taken observed effects;
    - The next intended subtask;
    - The overall task description;
    - Optional human guidance;
    - The CDL program.
3. A final instruction block that specifies the required output format, including a binary decision (`Yes`/`No`) indicating the result of the self-evaluation. If the answer is `No`, an explanation is also provided.

B.2 Exploration behavior template

The LLM first selects the most likely location of $target\_instance$, denoted as $LLM\_chosen\_loc\_instance$, which is selected from the discovered and not checked objects. The relevant information is then embedded into the template below, where $LLM\_chosen\_loc\_cat$ and $LLM\_chosen\_loc\_id$ denote the category and ID of $LLM\_chosen\_loc\_instance$, respectively. This exploration behavior can be used directly while interpreting an CDL program:

```
behavior find_{target_instance}_around_{LLM_chose_loc}({
target_instance}:item):
    goal: not unknown({target_instance})
    body:
        bind {LLM_chose_loc_cat}_instance:item where:
            is_{LLM_chose_loc_cat}({LLM_chose_loc_cat}_instance) and
            id[{LLM_chose_loc_cat}_instance]=={LLM_chose_loc_cat_id}
        achieve close_char(char,{LLM_chose_loc_cat}_instance)
        if can_open({LLM_chose_loc_cat}_instance):
            achieve_once open({LLM_chose_loc_cat}_instance)
            exp({target_instance},{LLM_chose_loc_cat}_instance)
        else:
            exp({target_instance},{LLM_chose_loc_cat}_instance)
    eff:
        unknown[{target_instance}]=False
        close[{target_instance},{LLM_chose_loc_cat}_instance]=True
        close[{LLM_chose_loc_cat}_instance,{target_instance}]=True
```

B.3 Behavior rule library

B.3.1 Basic behavior rule library

LEAP is provided with a set of basic behavior rules that can be used to achieve certain goals under constraint-free conditions. Here we provide some examples :

```
# Example 1: This behavior rule can be used to get close to an obj.
behavior walk_to(obj: item):
  goal: close_char(char, obj)
  body:
    achieve not unknown(obj)
    walk_executor(obj)
    let inhand_objects=findall o: item where: inhand(o)

  eff:
    inside_char[char, :] = False
    close_char[char, :] = False
    foreach o in inhand_objects:
      inside[o, :]= False
      close[o, :] = False
      close[:, o] = False
```

```
    foreach icf in (findall t:item: obj_inside_or_on(obj, t)):
      close_char[char, icf] = True
      if not (can_open(icf) and closed(icf)):
        unknown[icf] = False
      foreach o in inhand_objects:
        close[o, icf] = True
        close[icf, o] = True

    close_char[char,obj] = True
    foreach o in inhand_objects:
      close[o, obj] = True
      close[obj, o] = True

# Example 2: This behavior rule can pick up an obj.
behavior grab(obj: item):
  goal: inhand(obj)
  body:
    assert grabbable(obj)
    achieve not unknown(obj)
    achieve has_a_free_hand(char)
    achieve close_char(char, obj)
    grab_executor(obj)
  eff:
    inhand[obj] = True
    on[obj, :] = False
    inside[obj, :] = False
    close[obj, :] = False
    close[:, obj] = False
    if exists item1: item : holds_lh(char, item1):
      holds_rh[char, obj] = True
      has_a_free_hand[char] = False
    else:
      holds_lh[char, obj] = True

# Example 3: This behavior rule can be used to turn off an obj.
behavior switch_off(obj: item):
  goal: is_off(obj)
  body:
    assert has_switch(obj)
    achieve not unknown(obj)
    achieve has_a_free_hand(char)
    achieve close_char(char, obj)
    switchoff_executor(obj)
  eff:
    is_off[obj] = True
    is_on[obj] = False
```

B.4 Learned behavior rule library

Here we provide some examples that learned by LEAP through interaction with the environment and guidance from human:

```
# Example 1
Subtask description: Wash the clothes in the washing machine.
CDL program:
behavior start_washing(clothes:item, washing_machine:item):
    body:
        achieve inside(clothes, washing_machine)

behavior operate_washing_machine(washing_machine:item):
    body:
        achieve closed(washing_machine)
        achieve is_on(washing_machine)
```

```
behavior __goal__():
    body:
        bind basket: item where:
            is_basket_for_clothes(basket)
        # Select the basket for clothes

        bind washing_machine: item where:
            is_washing_machine(washing_machine)
        # Select the washing machine

        foreach clothes: item:
        # Load all dirty clothes from the basket into the washing
        machine
            if is_clothes(clothes) and inside(clothes, basket):
                start_washing(clothes, washing_machine)

        operate_washing_machine(washing_machine)
        # Start the washing machine

# Example 2
Subtask description: Boil the egg.
CDL program:
behavior boil_egg(egg:item, pot:item, stove:item):
    body:
        achieve has_water(pot)
        achieve inside(egg, pot)
        achieve on(pot, stove)
        achieve is_on(stove)

behavior __goal__():
    body:
        bind egg: item where:
            is_food_egg(egg)
        # Select the egg

        bind pot: item where:
            is_pot(pot)
        # Select a pot

        bind stove: item where:
            is_stove(stove)
        # Select a stove

        boil_egg(egg, pot, stove)
        # Boil the egg in the pot on the stove

# Example 3
Subtask description: Slice the bread and toast it.
CDL program:
behavior slice_bread(bread: item, cutting_board: item, knife: item):
    body:
        achieve on(bread, cutting_board) # Place the bread on the
        cutting board
        achieve cut(bread) # Slice the bread

behavior toast_bread(bread: item, toaster: item):
    body:
        achieve inside(bread, toaster)
        # Place the sliced bread inside the toaster
        achieve closed(toaster)
        # Close the toaster
        achieve is_on(toaster)
        # Turn on the toaster to toast the bread

behavior __goal__():
```

```
    body:
        bind bread: item where:
            is_food_bread(bread) and cuttable(bread)
        # Select a cuttable bread

        bind cutting_board: item where:
            is_cutting_board(cutting_board)
        # Select a cutting board

        bind knife: item where:
            is_knife(knife)
        # Select a knife

        bind toaster: item where:
            is_toaster(toaster)
        # Select a toaster

        slice_bread(bread, cutting_board, knife)
        # Slice the bread
        toast_bread(bread, toaster)
        # Toast the sliced bread
```