RoboAgent: Chaining Basic Capabilities for Embodied Task Planning

Supplementary Material

6. Details of the Scheduler and Capabilities

In this section we present a detailed introduction of the input and output format of each VLM calling. Fig. 5 shows the input and output format of the Exploration Guidance (EG) capability. It takes the object query generated by the scheduler as the exploration target, and reads in a candidate list of all the objects in the scene that can serve as the navigation goal (detailed in Sec 8.1). In addition, it incorporates a simple memory module that records the historical outputs associated with the same query object, thereby preventing repeated attempts that previously resulted in failure. Whenever EG receives a new query that differs from the previous one, the exploration history is reset to an empty list. The output of EG is a possible location of the target object, which is then passed to AD to be transformed into concrete exploration actions.

Fig. 6 shows the input and output format of the Object Grounding (OG) capability. It follows the standard practice in open-vocabulary grounding tasks and generates JSON-style annotations as output. It also implicitly converts free-form object queries into a clear object category (the "label" field in the annotations), thereby facilitating subsequent planning of the scheduler. Note that the bounding box coordinates provided by OG are actually not used during inference, as current simulators do not require object location information when performing interactions. We have OG output these coordinates to better supervise its object recognition performance during training, as well as to prepare for future integration with a low-level controller.

Fig. 7 shows the input and output format of the Scene Description (SD) capability. Basically, it translates the image observation into textual representations for the subsequent calling of AD. To ensure that it focuses on the target object to be manipulated, its prompt includes an object query, which is the object category output by the previous OG calling. The output of SD describes the on/in relationships between the target object and other objects, as well as the properties of the target object.

In our implementation of the Action Decoding (AD) capability, we further divide it into two modes: one for exploration sub-plans (Fig. 8) and another for manipulation sub-plans (Fig. 9). The former is responsible for converting the exploration direction output by EG into an action sequence, typically involving only navigation and open actions (when inspecting the inside of a container). The latter converts the manipulation command provided by the scheduler into an action sequence, which requires analyzing the specific state of the target object (for instance, to put `Apple` to `Fridge`, it needs to first `open` the `Fridge` if `Fridge` is `closed`). Therefore, it receives the description generated by SD. Additionally, during task execution, we record each navigation action and each pick/place action performed by the agent, thereby maintaining the agent's inventory and location. These two variables are also provided as inputs to the manipulation AD capability to assist in generating atomic actions with the correct object references. It should be noted that the prompts of AD are simulator-specific. Fig. 8 and 9 use ALFWorld [94] as an example. When testing in other simulators, the action space description in the prompts should be adapted accordingly.

Fig. 10 shows the input and output format of the Experience Summarization (ES) capability. The input contains the manipulation command issued by the scheduler, along with the actions and their execution outcomes (success or not) produced by the previous AD calling. The output is a summary of this action history.

Fig. 11 shows the input and output format of the scheduler. The prompt contains a description of the 6 predefined capabilities (AD is split into "exploration_planner" and "manipulation_planner" for easier understanding) and their corresponding arguments. The scheduler maintains a memory of historical queries and feedback. We retain only the feedback from OG (whether the target object is found) and ES (whether the manipulation sub-plan is successful). Feedback from other modules (EG and SD) is primarily used by other capabilities within the same scheduler iteration and does not have much impact on subsequent process of the scheduler. The scheduler's output consists of two components: the Chain-of-Thought (CoT) and the invoked capabilities. The former includes an analysis of completed and remaining sub-plans based on the history, while the latter comprises a sequence of capability names paired with their corresponding query arguments. We note that the scheduler does not need to provide arguments for AD (exploration) or SD, as their queries come directly from the outputs of the previous EG and OG invocations, respectively.

7. Details of the Training Data

7.1. The Training Set of ALFRED

As stated in Sec 4.1, the training of our model is conducted on the training split of ALFRED [93]. ALFRED provides 2.4k distinct training tasks, which are grouped into seven categories (pick and place, stack and place, pick two and place, clean and place, heat and place, cool and place, examine in light). Based on the AI2-THOR 2.0 [48]

The Input and Output Format of Exploration Guidance (EG)

Prompt:
Suppose you are a helpful robotic agent in an indoor environment. Your task is to find {object query} in the house, based on common house layouts and object placements. Currently, you can observe the following objects in the house: {list of available objects}. You need to output an exploration direction in the exact form of <relation> <object>, where <relation> is chosen from [target, in, on, near], and <object> is an object from the given object list. Previously, you have tried the following exploration directions: {exploration history}. Do not output the directions in this list again since they all failed.
Output example:
On CounterTop 1

Figure 5. The input and output format of the EG capability.

The Input and Output Format of Object Grounding (OG)

Prompt:
<image>
Locate {object query} in the image. If you can find it, output the bounding box in json format (including the object class as "label" of the bounding box); if you cannot find it, output no
Output example:
[{"bbox_2d": [193, 104, 308, 253], "label": "alarm clock"}]

Figure 6. The input and output format of the OG capability.

simulator, ALFRED constructs 120 different scenes (30 each for kitchens, bathrooms, bedrooms, and living rooms). Grounding each task to different scenes yields a total of 6.4k task instances. For each task instance, ALFRED generates expert trajectories using the Fast Forward algorithm [29] and manually annotates about 3 high-level task descriptions and step-by-step action instructions for each trajectory. The resulting 20k high-level task descriptions serve as the task instructions for the problem of Embodied Task Planning (ETP). Based on these data, we construct our training sets for different training stages, which will be detailed in the following subsections.

7.2. Stage 1

As discussed in Sec. 3.3, the key to constructing the first-stage training data lies in converting expert trajectories into sequences of capability invocations and corresponding queries. To begin with, we pre-process the task instructions provided by ALFRED using a large language model (LLM) (GPT-4.1-mini). Specifically, we feed each instruction along with its corresponding task category into the LLM, prompting it to parse the instruction and extract the key objects involved in the task. For example, for an instruction "Move the knife from the counter to the microwave table" belonging to the "pick and place" category, the LLM will output {object: knife (hint: on the counter), receptacle: microwave table}. We note that ALFRED's human-written instructions contain substantial noise, *i.e.*, the described content does not always match the actual task requirements. Therefore, we manually filter the parsing outputs and discard the instructions whose extracted object descriptions are inconsistent with the real target object categories. This results in a cleaned set of approximately 15k instructions. In addition, ALFWorld [94] provides some instruction templates for each ALFRED task type (*e.g.*, "heat some <object> and put it in <receptacle> " for "heat and place"). By substituting the placeholders with the target object categories, we generate one additional instruction for each training task instance, expanding the size of the instruction set to 21k. Although these template-based instructions lack the linguistic diversity of human-written ones, they offer precise and unambiguous descriptions of the corresponding task goals.

After processing the instructions, we now turn to the expert trajectories. ALFRED provides a high-level PDDL plan for each training task, which has a similar pattern with the plan in ETP problems. For each human-written instruction, we align its expert plan to the action space of EB-ALFRED, while the plan corresponding to templated instructions are aligned to the action space of ALFWorld (details of the action spaces will be discussed in Sec. 8.1). By mixing plans drawn from the two action spaces during training, we hope that the AD capability learns more generalizable action knowledge.

Next, for each training instruction, we decompose its corresponding expert trajectory into alternating exploration

The Input and Output Format of Scene Description (SD)

Prompt:
<image>
This is an egocentric image observed by a robotic household agent. Please describe the {object query} in the scene.
Output example:
There is a laptop on the desk. The laptop is open.

Figure 7. The input and output format of the SD capability.

The Input and Output Format of Action Decoding (AD)

Prompt:
Suppose you are a helpful robotic agent in an indoor environment. You are able to perform the following actions: 1. go to <object> 2. open <object> 3. close <object>
The <object> is an object name composed of an object class and an object ID (e.g., Apple 1, DiningTable 2).
Now, your task is to {exploration query}. Please complete this task by performing one or a series of actions. You should output a list of actions, e.g. [go to Fridge 1], or [go to Cabinet 2, open Cabinet 2]
Output example:
[find a DeskLamp 1]

Figure 8. The input and output format of the AD capability for exploration sub-plans (using ALFWorld's action space).

and manipulation sub-plans following the procedure described in Sec. 3.3, and further convert the sub-plans into a sequence of capability invocations. The object descriptions extracted by the LLM are then used as queries for EG and OG. Each manipulation sub-plan is classified to one of several predefined categories (grasp <object> , put <object> to <receptacle> , turn on <object> , slice <object> , heat <object> with <tool> , cool <object> with <tool> , clean <object> with <tool> , put <object> to <receptacle> and grasp <receptacle>). By substituting the placeholders with the parsed object descriptions, we obtain the queries for AD and ES. Through this process, the expert plan is transformed into a corresponding sequence of structured capability invocations. Along with each set of invocations, we also construct a corresponding CoT with templates like "The task is to {instruction}. I have already {done manipulation sub-plans}. Next, I should {remaining manipulation sub-plans}."

To assess the reliability of the transformed capability sequences, we examine whether they can achieve the task goal assuming that all capabilities provide correct outputs. To this end, we construct "perfect" implementations of each capability using the ground-truth information available in the simulator: perfect SD is built upon the object locations and properties in the scene graph (SG); perfect OG comes from the segmentation masks; perfect AD comes from the decomposed expert plan; perfect ES returns the action feedback given by the simulator. The ground-truth outputs for EG can also be obtained from the SG. However, we note that accurately predicting the location of an object in a partially observable environment is intrinsically challenging (*e.g.*, finding a knife in a kitchen containing 10+ cabinets and 10+ drawers). To better reflect this difficulty, we inject perturbations into EG's outputs: with a certain probability, it gives a random exploration direction instead of the ground-truth one. Using these perfect (or perturbed) capability implementations, we attempt to execute the capability invocation sequences in the simulator. Among the 21k instructions, 16k complete successfully. Failures arise from two major sources. First, the ALFWorld/EB-Habitat simulators are not fully aligned with ALFRED, causing the expert plan derived from ALFRED to be occasionally infeasible in these environments. Second, excessive random exploration may sometimes reach the maximum step limit before the plan has been fully executed. We construct the scheduler's training data using only the task instances whose capability invocation sequence succeeds.

During the verification procedure, we simultaneously record the input and ground-truth output for each capability invocation, generating the training set for each type of capability. In total, we build datasets of size 130k/157k/74k/203k/60k for EG/OG/SD/AD/ES, respectively, with each training sample formatted as shown in Fig. 5-10. For the scheduler, the 16k successful capability invocation sequences provide 179k samples of scheduler callings, each formatted as in Fig. 11. We split all these samples into training and validation sets using an 80/20 ratio, and perform the first expert-SFT stage based on them.

The Input and Output Format of Action Decoding (AD)

Prompt:
Suppose you are a helpful robotic agent in an indoor environment. You are able to perform the following actions: 1. go to <object> 2. open <object> 3. close <object> 4. use <object> (which means to turn on the object) 5. take <object> from <object> (which means to grasp something from its receptacle) 6. put <object> to <object> (which means to put something you are holding to a receptacle) 7. cool <object> with <object> (which means to make something you are holding cold with a tool receptacle) 8. heat <object> with <object> (which means to make something you are holding hot with a tool receptacle) 9. clean <object> with <object> (which means to make something you are holding clean with a tool receptacle)
The <object> is an object name composed of an object class and an object ID (e.g., Apple 1, DiningTable 2). You need to perform a sequence of actions to complete a given task. You should output a list of actions in the given format, e.g. [take Cup 1 from CounterTop 2], [open Cabinet 1, put Apple 3 to Cabinet 1, close Cabinet 1].
Now, you are holding {agent inventory}. You are at {agent location}. The environment infomation is: {scene information}
Your task is to {manipulation query}. Please generate a list of actions in the given format to complete this task.
Output example:
[open the Fridge 1, put down the object in hand, close the Fridge 1, open the Fridge 1, pick up the BreadSliced, close the Fridge 1]

Figure 9. The input and output format of the AD capability for manipulation sub-plans (using ALFWorld's action space).

The Input and Output Format of Experience Summarization (ES)

Prompt:
<image>
Suppose you are a helpful robotic agent in an indoor environment. Your task is to {manipulation query}. Here is a list of actions you have performed and their corresponding environment feedbacks: action history
Your current egocentric observation is shown in the image. Please summarize the progress made and analyze the reasons for the failures (if any).
Output example:
You successfully grasp SoapBar 1.

Figure 10. The input and output format of the ES capability.

7.3. Stage 2

To construct the training data for the second stage, we deploy the model trained in the first stage on the training tasks and record all the capability invocations. Then we follow the pipeline described in Sec. 3.4 to generate corrective ground-truth outputs for each invocation. As for data augmentation, we apply the following strategies: (1) With the help of an LLM (GPT-4.1-mini again), we generate synonyms for every object category appearing in ALFRED, and randomly replace the object IDs in the candidate list of EG's input with these synonyms (*e.g.*, replacing "Armchair 1" with "Couch 1"). (2) We generate descriptive phrases for each object category and randomly replace the object queries in OG's input with them (*e.g.*, replacing "locate the cabinet" with "locate the tall storage unit with doors"). (3) For each atomic action, we generate several semantically equivalent rephrasings and randomly replace the action names in both the input and output of AD (*e.g.*, replacing "pick up the <object>" with "grasp the <object>"). (4) We further use the LLM to produce additional object categories and construct synthetic samples for AD by substituting the original object categories with the newly generated ones (*e.g.*, replacing "put Apple 1 to Fridge 1" with "put Shirt 1 to WashingMachine 1"). The goal of these augmentations is to improve the generalization of the capabilities involved. They prevent the model from overfitting to ALFRED's limited set of object and action names and encourage it to learn the semantic knowledge. In total, we build a dataset of 820k samples for stage 2, and split it with an 80/20 ratio for training and validation.

7.4. Stage 3

The third training stage focuses on improving the performance of the scheduler. Since the scheduler operates on textual inputs/outputs and does not interact with the environment directly, we do not utilize the simulator when constructing the training data for this phase. Instead, we

The Input and Output Format of the scheduler

Prompt:
Suppose you are a helpful robotic agent in an indoor environment. You have the following abilities and you can invoke them by function calling:
1. exploration_guidance(object_information): given the name or the description of an object, output a direction for exploration
2. exploration_planner(): explore the environment accrording to the exploration direction
3. object_grounding(object_information): given the name or the description of an object, find it in the egocentric view of the robot
4. scene_description(): describe the egocentric observation of the robot
5. manipulation_planner(subtask): given a subtask instruction (a subtask is defined as a part of the task that can be completed within the scene observed from the robot's current egocentric view), complete it by performing atomic actions
6. experience_summarization(subtask): summarize the previous subtask execution experience
You need to complete a given task by sequentially generating ability queries. When querying the ability of exploration_guidance, object_grounding, manipulation_planner, experience_summarization, you need to give them a proper input argument. After querying object_grounding or experience_summarization, you will get feedbacks of the grounding results or execution results. At each step, you will be given the history of queries you made and feedbacks you received. You need to output your reasoning process after "Think: ". Then, you need to output your ability query (or queries) after "Query: ", or simply output "Stop" if you believe the task is completed.
Now, here comes the task:
Task: {task instruction}
History: {query and feedback history}
Output example:
Think: The task is "To chill lettuce and place it on the counter". I am at the beginning of the task. I should: grasp lettuce, chill lettuce, put lettuce to counter. First, I need to find lettuce.
Query: 1. exploration_guidance(lettuce)
2. exploration_planner()
3. object_grounding(lettuce)

Figure 11. The input and output format of the scheduler.

synthesize the feedback for the invoked capabilities. For each training task, we first obtain its ground-truth capability invocation sequence (Sec. 7.2). To construct the scheduler's input (Fig. 11), we need to generate feedback for every OG and ES calling. With perfect capability execution, each OG feedback would indicate that the target object is found, and each ES feedback would indicate successful manipulation. However, the trained capabilities do not always provide correct outputs. To model this, we introduce random errors to the capabilities. For each OG invocation, we assign a certain probability of returning a "target not found" feedback. In such cases, the scheduler need to repeat the previous (EG, AD, OG) invocations until the target is located. Similarly, for each ES invocation, we randomly return a "failure" feedback and a plausible failure reason with a certain probability. Upon receiving such feedback, the scheduler must roll back to an earlier sub-plan and re-invoke the corresponding capabilities. Through this process, we augment the original expert capability sequence ($[\{\hat{q}_1^j\}_{j=1}^{\hat{n}_1}, ..., \{\hat{q}_{T_1}^j\}_{j=1}^{\hat{n}_{T_1}}]$) with an error-recovery mechanism, producing longer and more realistic interaction trajectories ($[\{q_1^j\}_{j=1}^{n_1}, f_1, ..., \{q_{T_2}^j\}_{j=1}^{n_{T_2}}, f_{T_2}]$) that better reflect the actual behaviors of the capabilities.

We adopt the augmentation strategy proposed in Stage 2 when synthesizing the feedback. In addition, we construct 20 new task categories and about 3k new task instances by combining ALFRED's 7 original task types (*e.g.*, "transfer one hot tomato and one cold tomato to the side table"). For each newly created task, we derive its ground-truth capability-invocation sequence by merging the sequences of its constituent tasks, and then synthesize training samples for scheduler using the error-injection procedure described above. To prevent the newly constructed tasks from being too difficult for the scheduler (so that it cannot generate any reasonable outputs during reinforcement learning), we incorporate their expert invocation sequences (without error recovery but including CoT reasoning, approximately 46k samples) into the SFT dataset of Stage 2. These additional scheduler training signals provide a suitable starting point for the Reinforcement Fine-tuning (RFT) phase.

Finally, stage 3 yields 25k task episodes and 360k sam-

ples (in the format of Fig. 11, but without CoT), which are again split into training and validation sets using an 80/20 ratio. The details of the RFT training based on these samples are presented in Sec. 9.

8. Details of the Benchmarks

8.1. ALFWorld and EB-ALFRED

ALFWorld [94] and EB-ALFRED [125] are used as the primary evaluation environments. The ALFWorld benchmark provides a wrapper of ALFRED's low-level actions and implements a text-based action interface through the TextWorld [19] engine. It follows ALFRED's original data split. The training set retains 3.6k out of ALFRED's 6.4k training tasks (removing all tasks in the "stack and place category" as well as those involving the "slice" action). Its evaluation set is drawn from ALFRED's validation split and contains 134 tasks in unseen scenes and 140 tasks in seen scenes during training (both with novel object initialization). ALFWorld supports two agent modes. The first is a vision-based setting, in which the agent receives an egocentric image (with a default resolution of 300×300) at each time step, mirroring the original ALFRED environment.[2] The second is a text mode, in which the agent receives a textual description of the outcome of the previous action along with the set of currently observable objects. The maximum number of action steps per episode is set to 50.

EmbodiedBench [125] integrates 4 simulation environments to provide a comprehensive evaluation of embodied agents across planning, navigation, and manipulation competencies. Among them, EB-ALFRED is derived from the LoTa-ALFRED [17] benchmark and introduces a wrapper for the ALFRED environment that differs from that of ALFWorld. It offers 300 evaluation tasks, all sourced from ALFRED's validation set of seen scenes. These tasks span all 7 task categories of ALFRED, and are reorganized into 6 splits (base, visual appearance, spatial relationship, complex instruction, common sense, long horizon) to capture distinct characteristics of the task instructions. EB-ALFRED adopts a setting of vision-based observation with a default resolution of 500×500. The maximum number of action steps per episode is set to 30, and an episode also terminates if the agent outputs 10 invalid actions.

Table 7 summarizes several key differences between ALFWorld and EB-ALFRED. Specifically: (1) **Task diversity**. EB-ALFRED includes a richer set of task types than ALFWorld, *e.g.*, "put *a bowl with a knife in it* to the countertop", "put a *sliced* apple to the countertop". (2) **Action granularity**. In ALFWorld, "heat", "cool", and "clean" are implemented as atomic actions. In contrast, EB-ALFRED requires the agent to realize these effects through more primitive operations (*e.g.*, *cooling* an apple by *putting* it to the fridge and then *picking* it up). EB-ALFRED also provides a "slice" action, which ALFWorld lacks because it contains no tasks involving sliced objects. (3) **Navigation behavior**. ALFWorld restricts navigation to "go to" actions targeting large, immovable receptacles. EB-ALFRED, however, allows the agent to approach any object in the environment via a "find" action. This makes ALFWorld place greater emphasis on exploration: the agent must reason about the potential locations of the target object in order to find it. (4) **Instruction design**. ALFWorld uses templated instructions in which the target object category is explicitly mentioned. In contrast, EB-ALFRED uses human-written (or GPT-augmented) instructions that are longer, syntactically richer, and often refer to target objects more indirectly (*e.g.*, by describing their appearance or location instead of naming their category). (5) **Evaluation environments**. Assuming that the training is performed on ALFRED's training tasks, then ALFWorld evaluates the agent in both seen and unseen scenes during training, whereas EB-ALFRED's evaluation is conducted in seen scenes. Overall, EB-ALFRED emphasizes task diversity and instruction complexity, while ALFWorld places more focus on efficient exploration and generalization to novel scenes.

Our goal is to train a model that can bridge the aforementioned discrepancies and operate seamlessly across ALFWorld and EB-ALFRED. To achieve this, as described in Sec. 7.2, we use ALFRED's training set (which is a superset of ALFWorld's training set) and include both the human-annotated instructions from ALFRED and the templated instructions from ALFWorld. To address the differences in atomic actions, we provide AD with training data that encompass different action spaces. To handle the differences in navigation behavior, we construct two variants of the candidate list in the EG input: a "receptacle-only" setting and an "all-objects" setting. Both the receptacle list and the object list can be extracted from the set of available actions.

8.2. Adaptation to EB-Habitat

EB-Habitat is another benchmark for ETP designed by EmbodiedBench [125]. It is constructed upon the Language Rearrangement task proposed by LLaRP [102]. EB-Habitat adopts the same observation format as EB-ALFRED, and its 300 testing tasks are partitioned into the same 6 splits to capture the diversity of instruction styles. However, since EB-Habitat and EB-ALFRED are built on different simulators with substantially different sets of object categories and task types, it serves as a challenging out-of-distribution (OOD) evaluation environment for our trained model.

When evaluating on EB-Habitat (Table 5 of the main

[2] Some previous works on ALFWorld's vision-based environment assumes a textual feedback of each action (which may contain a list of newly observed objects), while we only leverage a binary signal (whether the action is available or not) as action feedback during inference.

Table 7. A comparison between ALFWorld [94] and EB-ALFRED [125]. We slightly modify the names of the atomic actions (*e.g.*, replacing "find" with "go to") to enable a clearer comparison of the differences.

	Task Types	Action Space	Nav. Target	Instruction	Test Scene
ALFWorld	6	go to, pick, put, turn on, open, close, heat, cool, clean	only receptacles	template-based	seen&unseen
EB-ALFRED	7	go to, pick, put, turn on, open, close, slice	all objects	human-written	seen

paper), we keep the VLM parameters fixed and introduce the following modifications for the capability invocation pipeline. For AD, the action descriptions in the prompt (Figs. 8 and 9) are replaced with the action space of EB-Habitat ("navigate", "pick", "place", "open", "close"). Similar to ALFWorld, EB-Habitat only permits navigation actions toward receptacles, so we set the candidates in the prompt of EG to the list of receptacles present in the scene. In practice, we find that the primary obstacles to OOD generalization stem from the visual domain: EB-Habitat provides less realistic environment rendering than ALFRED and also contains many object categories that never appear in ALFRED. As a result, capabilities that rely on image inputs exhibit degraded performance. To mitigate this issue, we introduce several additional adjustments. Noting that EB-Habitat involves relatively few object-state changes and that action failures occur in a highly similar pattern (mostly "target object out of reach"), we omit SD (*i.e.*, make it return an empty string when invoked) and remove the image input from ES. For OG, the agent first checks whether the object query belongs to the set of object categories provided by EB-Habitat (which can be parsed from the set of valid actions). If it does, the model directly returns that the target object is found. If it does not, the model performs the standard open-vocabulary grounding procedure, after which the detected object label is mapped to the most similar string in the category set. This adjustment compensates for the model's limited recognition ability at the cost of increased number of action steps (*e.g.*, attempting to grasp an apple even when it is not visible now).[3]

8.3. Adaptation to Text Environments

In Tables 3 and 6 of the main paper, we also evaluate the model on two text environments: ALFWorld-Text and LoTa-WAH. ALFWorld-Text is already introduced in Sec. 8.1. To adapt the VLM agent to receive textual observations, we make the following adaptations to the capabilities. (1) The candidate list of EG's input is parsed from the initial observation, which lists all the receptacles in the room. (2) OG is implemented by checking whether the queried object appears in the object list extracted from the most recent observation. (3) SD is implemented by outputting "There is a {object query}" and appending the property descriptions extracted from the latest observation (if any). (4) The image input for ES is removed. Comparing the results in Tables 2 and 3, we observe that the ETP task is easier in the text-based environment than in the visual environment. This indicates that visual recognition of objects and their properties remains one of the key factors limiting planning performance. This is also validated by the error analysis in Sec. 11.

LoTa-Bench [17] is another text-based benchmark for embodied planning. It organizes two simulators for evaluation, ALFRED [93] and VirtualHome [70]. Here, we focus on the latter, LoTa-WAH, since EB-ALFRED basically subsumes the former. Compared with ALFWorld-Text, LoTa-WAH exhibits a different set of object categories and does not provide updated environment observations. Instead, it only reports the complete list of object categories at the beginning of a task and indicates whether each executed action succeeds or fails at each timestep. Given these characteristics, we omit SD (which directly returns an empty string) and OG (which directly returns "the target object is found"), and remove the image input from ES. For EG, we note that LoTa-WAH allows navigation actions toward any object, but the number of objects in the scene is large. Accordingly, we retain only all receptacles and the objects having high semantic similarity with the object query (measured using all-MiniLM-L6-v2 [77]) in the input candidate list. For AD, we modify the prompt to match the action space defined in LoTa-WAH. LoTa-WAH uses subgoal success rate (SSR) as the evaluation metric, defined as the proportion of the predefined goal conditions that are satisfied at the end of an episode.

9. Preliminaries on the RFT Stage

Reinforcement learning (RL) aims to learn a policy π that generates an action distribution given a state: $a_t \sim \pi(\cdot|s_t)$. After taking each action, it will receive a reward $R(s_t, a_t)$. The most classical training objective is the expected return

[3] We also observe that EB-Habitat occasionally presents situations where the agent is close to an object but the object is not visible in the rendered view, which further motivates the above modification to the OG capability.

of the policy:

$$\eta(\pi) = \mathbb{E}_{(s_0,a_0,...,s_T,a_T)\sim\pi}[\sum_{t=0}^{T}\gamma^t R(s_t,a_t)], \quad (9)$$

where γ is a decay factor, T is the maximum length of the episode, $\tau = (s_0, a_0, ..., s_T, a_T)$ is a trajectory collected by rolling out π in the environment. With a parameterized policy π_θ, Policy Gradient [99] shows that the gradient of η can be computed with:

$$\nabla_\theta\eta(\pi) = \mathbb{E}_{(s_0,a_0,...,s_T,a_T)\sim\pi}[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(s_t|a_t)A_\pi(s_t,a_t)], \quad (10)$$

where the advantage $A_\pi(s_t,a_t) = Q_\pi(s_t,a_t) - V_\pi(s_t)$, and $Q_\pi(s_t,a_t) = R(s_t,a_t) + \gamma V_\pi(s_{t+1})$, $V_\pi(s_t) = \mathbb{E}_{(a_t,s_{t+1}...,s_T,a_T)\sim\pi}[\sum_{t'=t}^{T}\gamma^{t'-t}R(s_{t'},a_{t'})]$. However, this limits the policy training to using strict on-policy data ($(s_0, a_0, ..., s_T, a_T) \sim \pi$). To allow gradient optimization with slightly off-policy data, [42] shows that:

$$\eta(\pi') = \eta(\pi) + \mathbb{E}_{(s_0,a_0,...,s_T,a_T)\sim\pi'}\sum_{t=0}^{T}\gamma^t A_\pi(s_t,a_t). \quad (11)$$

TRPO further gives a surrogate for the right hand side:

$$\eta(\pi') - \eta(\pi) = \mathbb{E}_{s\sim d_{\pi'}}\mathbb{E}_{a\sim\pi'(\cdot|s)}A_\pi(s,a) \quad (12)$$

$$\approx \mathbb{E}_{s\sim d_\pi}\mathbb{E}_{a\sim\pi(\cdot|s)}\frac{\pi'(a|s)}{\pi(a|s)}A_\pi(s,a). \quad (13)$$

where d_π is the (unnormalized) state distribution under π: $d_\pi(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + ... + \gamma^T P(s_T = s)$. Now, π' in Eq (13) can be optimized using trajectories generated with an old policy π. TRPO implements this by performing constraint optimization to avoid π' from deviating too far from π, while PPO [83] achieves this through a clipping mechanism:

$$J_{\text{PPO}}(\pi') = \mathbb{E}_{s\sim d_\pi, a\sim\pi(\cdot|s)}\min[r(a,s)A_\pi(s,a), \text{clip}(r(a,s), 1+\epsilon, 1-\epsilon)A_\pi(s,a)], \quad (14)$$

where $r(a,s) = \frac{\pi'(a|s)}{\pi(a|s)}$ is adaptively clipped according to the signal of A_π. PPO calculates the advantage value by General Advantage Estimation [82], which involves the training of a value model along with the policy. GRPO [86] further eliminates the need for this value model by using the group-wise average return as the baseline:

$$J_{\text{GRPO}}(\pi') = \mathbb{E}_{s\sim d_\pi, a^i\sim\pi(\cdot|s)}\sum_{i=1}^{G}\min[r(a,s)\hat{A}_\pi(s,a^i), \text{clip}(r(a,s), 1+\epsilon, 1-\epsilon)\hat{A}_\pi(s,a^i)], \quad (15)$$

where $\hat{A}_\pi(s,a^i)$ is the return of action a^i normalized with the group's mean and standard deviation.

Our proposed Expert-Induced Policy Optimization (EIPO) follows Eq (11) but substitute π with an expert policy π^*. So, the optimization target is transformed to:

$$\eta(\pi') - \eta(\pi^*) = \mathbb{E}_{s\sim d_{\pi'}}\mathbb{E}_{a\sim\pi'(\cdot|s)}A_{\pi^*}(s,a) \quad (16)$$

$$\approx \mathbb{E}_{s\sim D}\mathbb{E}_{a\sim\pi(\cdot|s)}\frac{\pi'(a|s)}{\pi(a|s)}A_{\pi^*}(s,a), \quad (17)$$

where D is a static dataset of policy input. By introducing π^*, EIPO bypasses the estimation of returns and state values under policy π through value models or Monte Carlo sampling. Instead, it adopts the more stable expert advantage function A_{π^*} as the optimization objective. This approach more clearly reflects the credit of each action and provides more effective gradient signals consequently. Furthermore, as discussed in Sec. 3.5, we incorporate PPO-style probability ratio clipping and GRPO-style group normalization into EIPO. Note that GRPO's normalization uses the group mean as an estimate of the state value under policy π to approximate the advantage $A_\pi(s,a^i)$. In contrast, EIPO's normalization employs the group mean as a baseline for A_{π^*} to reduce the variance of the policy gradient and introduce positive gradient signals (detailed in Sec. 3.5). Following recent works like [62], we omit the standard deviation normalization used in GRPO and only subtract the group average.

The proposed EIPO can be applied to any policy as long as an expert π^* can be formulated. We present two examples in this work. The first is a conventional action policy, in which the model directly outputs atomic actions. The second is the scheduler policy in our capability-driven planning framework, where the model outputs capability invocations. For the action policy, we train the model using online data. In each iteration, the model rolls out $G = 8$ complete trajectories in each of the batch_size=16 randomly selected training environments. During planning, we maintain the lists of completed and remaining sub-plans for each timestep. For each state s along each trajectory, suppose the number of remaining sub-plans is n_s, then an expert action policy should be able to complete the task within n_s rounds of output (where each output is a list of actions completing a sub-plan). Assuming that completing each sub-plan yields a reward of $+1$, the corresponding expert value for that state, $V_{\pi^*}(s)$, can thus be computed as $\sum_{i=0}^{n_s-1}\gamma^i$. Meanwhile, by examining the difference in the number of remaining sub-plans between consecutive states along the trajectory, we can determine the immediate reward $R(s,a)$ for each action a. Based on the reward for a and the expert value for s, we obtain the advantage value for each state-action pair by $A_{\pi^*}(s,a) = \gamma V_{\pi^*}(s') + R(s,a) - V_{\pi^*}(s)$, where s' is the next state in the trajectory. After group-based normalization, this is used as the objective for policy gradient

and model update. The training is conducted for 150 iterations with an initial learning rate of 1e-6. We implement the online training pipeline on ALFWorld's text environment using the framework provided by GiGPO [25]. The experimental results are presented in Sec. 10.3.

In our training stage 3, we apply the EIPO algorithm for the scheduler. Treating the scheduler VLM as a policy, we define the state s_t as all previously generated queries and the corresponding feedback up to step $t-1$, and the action a_t as the sequence of queries produced at the current scheduler calling. To avoid the substantial time cost of policy rollouts during training (which involves image rendering and capability invocations), we sample input states from an offline dataset, as shown in Eq (17). This deviates from the original objective in Eq (16), which ideally samples states from the distribution induced by the current policy π. However, the trajectory synthesis process described in Sec. 7.4 enables us to generate a large and diverse dataset (covering scenarios where capability invocations succeed or fail), which helps mitigate the problem of distribution shift to some extent. The training proceeds for 120 iterations. In each iteration, we sample a batch of 512 states to serve as prompts. For each prompt, the model generates $G = 8$ responses, each containing both the CoT reasoning and the capability queries (as in Fig. 11). The computation of A_{π^*} follows a procedure similar to that used for the action policy. We again maintain the lists of completed and remaining sub-plans for each state, and assume that an expert scheduler would be able to resolve the remaining n_s sub-plans with n_s rounds of output. The reward scheme assigns $+1$ for the completion of each manipulation sub-plan. This choice is motivated by the observation that the successful execution of exploration sub-plans often depends heavily on the performance EG and OG capabilities; thus even a correct scheduler output may not directly complete an exploration sub-plan in practice. Given this reward definition, we can compute V_{π^*} for any input state s. However, in the offline setting, obtaining the next state s' resulting from an action is challenging, since no environment or capability interaction is available. To estimate $R(s, a)$ and $V_{\pi^*}(s')$, we adopt a simple approximation: when the scheduler's output matches that of the expert scheduler, we follow the expert policy to determine s' (a state where a new sub-plan is completed). When the output does not match, we assume that the number of remaining sub-plans stays unchanged and the immediate reward is 0. We find that this approximation yields satisfactory empirical performance (as in Table 4), and leave the incorporation of a world model and the design of a more delicate reward mechanism for future work. After obtaining A_{π^*} for each state-action pair (*i.e.*, prompt-output pair), we perform the computation of policy gradient and the update of model parameters following the same procedure as standard GRPO.

10. More Experimental Results

10.1. Impact of Training Data

Our proposed training pipeline leverages all tasks in ALFRED's training set to develop an agent capable of operating in both the ALFWorld and EB-ALFRED environments. Since ALFWorld uses only a subset of ALFRED's tasks, we re-run the 3-stage training procedure using only ALFWorld's training tasks, in order to enable a more controlled comparison with previous baselines on ALFWorld. The results are presented in Table 8. Without additional tasks, Our model still reaches a success rate of 67.2% after the 3-stage optimization, which is better than all previous methods in Table 2. On the other hand, by comparing with the results in Table 4, we observe that incorporating the additional ALFRED tasks, despite that they belong to task categories different from ALFWorld, yields a roughly 10% performance improvement. This suggests that our capability-based framework is able to acquire generalizable planning strategies from a diverse set of tasks and may possess promising scalability. Moreover, introducing synthetic tasks (constructed by combining the original ALFWorld tasks) for EIPO also leads to performance gains, further demonstrating the effectiveness of our data collection strategy in Stage 3.

10.2. Impact of the Capability-Driven Framework

To further demonstrate the advantage of the capability-driven planning framework, we compare it with a plain planner that directly generates actions. Specifically, this baseline uses the same Qwen2.5-VL-3B backbone as our method. At each timestep, it receives the current observation image, the task instruction, and the interaction history (including past actions and the environment feedback of success/failure), and outputs a CoT reasoning process followed by one or more actions. This can be viewed as an integration of the scheduler and AD capability in our framework. To train this model, we adopt a two-stage pipeline. First, SFT is performed using expert trajectories. The trained model is then utilized to collect trajectories on the training tasks, which serves as the dataset for an EIPO-based RFT stage. Here, we adopt an approach analogous to the scheduler policy described at the end of Sec. 9 to compute the number of remaining sub-plans and the V_{π^*} value for each state along the trajectory. We then construct a set of optimal actions for each state based on the expert scheduler and expert AD. During training, we determine the next state s' by checking whether the model's predicted action falls within the optimal action set, which then allows us to compute the advantage A_{π^*} for optimization. Note that we omit the DAgger-SFT stage, as it is designed for enhancing the capabilities. Similar to Sec. 10.1, the training is conducted on ALFWorld's training tasks, and the evaluation re-

Table 8. The results of solely using ALFWorld for training.

SFT-expert	SFT-DAgger	EIPO	SR
✓	✗	✗	42.5
✓	✓	✗	59.7
✓	✓	w.o. syn. task	67.2
✓	✓	w. syn. task	68.6

Table 9. The results of an VLM planer with/without explicit capability invocation.

Cap.	SFT	EIPO	SR
✗	✓	✗	40.3
✗	✓	✓	48.5
✓	✓	✗	59.7
✓	✓	✓	68.6

sults are reported in Table 9. Under both the SFT-only and SFT+RFT settings, the planner equipped with capabilities consistently outperforms the version that directly generates actions. We observe that the planner without capabilities tends to more frequently produce invalid actions, misinterpret the task progress (*e.g.*, attempting to heat an object despite having failed to pick it), and fail to locate the target object due to insufficient exploration or inaccurate recognition. In addition, the results in Table 9 also demonstrate the generality of the proposed EIPO algorithm across different types of planners, which will be elaborated more concretely in the next subsection.

The benefits of introducing capabilities can be understood from two aspects. First, it enables a clearer and more reliable reasoning process, such as the explicit exploration and object localization. Second, it allows the incorporation of additional fine-grained supervisory signals into the training process.

10.3. Impact of the RFT Algorithm

To more thoroughly analyze the effect of the EIPO algorithm, we compare it with the GRPO [86] baseline under two settings: an online action policy and an offline scheduler policy. For the action policy, the results in Table 10 extend those shown in Fig. 4 of the main text. As described in Sec. 9, we train models within the verl-agent framework provided by GiGPO [25], using Qwen2.5-1.5B-Instruct as the base model and keeping all hyperparameters consistent. Experiments are conducted in the text-based ALFWorld environment. For a complete episode collected by the model, GRPO assigns a reward solely based on the final signal of task success or failure. This terminal reward is shared by all steps within the episode and used as the optimization target. GiGPO further computes a per-step return for each action and constructs both step-level and episode-level groups for normalization. In contrast, our method directly uses the expert advantage A_{π^*} for each action as the optimization target. We also experiment with constructing both episode-level and step-level normalization groups, analogous to GiGPO. The results demonstrate that the more stable and accurate optimization target of expert advantage enables EIPO to outperform GRPO, which relies on estimating policy returns through sampled rollouts. Moreover, incorporating step-level grouping does not yield additional performance gains. A possible explanation is that EIPO's per-action advantage computation already addresses much of the credit assignment problem, reducing the need for fine-grained normalization through additional grouping.

Table 10. A comparison between GRPO and EIPO for training LLM-based (Qwen2.5-1.5B) planner on ALFWorld. The reported results are averaged across 3 independent runs.

objective	SR
rollout return (episode-level group)	72.8±3.6
rollout return (episode&step-level group)	86.7±1.7
expert advantage (episode-level group)	**94.8**±3.9
expert advantage (episode&step-level group)	92.7±3.9

Table 11. A comparison between GRPO and EIPO for the training stage 3 of our pipeline.

objective	ALFWorld	EB-ALFRED
none	73.1	64.3
reward	69.4	65.0
expert advantage	**77.6**	**67.0**

For the scheduler, GRPO based on episode-level returns is difficult to apply because we employ an offline dataset to avoid expensive rollouts. As a baseline for EIPO, we consider an alternative algorithm that uses per-action rewards as the optimization target: the model receives a reward of 1 if its output matches the expert scheduler's output, and 0 otherwise. This approach resembles GRPO on single-turn QA datasets, ignoring the long-term consequences of actions and the multi-turn planning process. As shown in Table 11, EIPO achieves consistently better performance than this reward-based baseline in both evaluation environments. We also note that the gain is particularly pronounced on longer-horizon tasks (*e.g.*, the long horizon split of EB-ALFRED and the exploration-intensive tasks in ALFWorld). This indicates that EIPO provides a practical and effective objective for offline policy optimization of VLM-based agents.

10.4. Efficiency

Introducing capability invocations into the planning process may raise concerns regarding computational efficiency. To assess this, we execute our capability-driven planning pipeline for 3 times on 6 ALFWorld tasks (one from each task category) and report the average statistics in Table 12. For comparison, we also evaluate the planner that directly generates actions (described in Sec. 10.2) under the same

Table 12. Statics of VLM inference for capability-based and action-based planner.

	time	# call	# token in	# token out
action-based	35.6s	16.3	11.3k	1.1k
capability-based	36.3s	35.1	10.1k	1.1k

experimental conditions. All results are obtained on a single NVIDIA RTX 4090.

The results show that incorporating capabilities indeed increases the number of VLM calls. However, the per-call token cost is lower, as many capabilities have short prompts and concise outputs. The overall runtime of the two methods is comparable. It is worth noting that the runtime depends not only on the number of VLM calls but also on the number of interactions with the simulator. For many complex tasks, action-based approaches may spend a large number of tokens and considerable time on blind exploration or invalid actions, which usually result in task failure. In such cases, the capability-based approach offers advantages in both performance and efficiency.

10.5. Plan visualizations

We present the CoT traces and capability invocations generated by RoboAgent on several tasks in Fig. 16-22 to qualitatively analyze its planning capabilities. In Fig. 16-18, the agent successfully completes a complex task in EB-ALFRED involving more than 20 action steps, with the scheduler accurately analyzing the task progress and invoking appropriate capabilities to sequentially solve each subtask. Fig. 19 illustrates a task in EB-ALFRED that involves open-vocabulary object reference. The OG capability successfully grounds the query "round kitchen table" to the dining table, facilitating subsequent planning and manipulation. Fig. 20-21 depict a task in ALFWorld where the agent cannot directly "go to" small objects. Our EG and OG capabilities assist the agent in efficiently exploring the receptacles in the scene, ultimately locating the target object (cup). Fig. 22 shows an example in the text-based environment, where OG, SD, and ES provide feedback by parsing the textual observations.

10.6. Real-World Demonstration

Following the reviewer's suggestion, we attempt to deploy RoboAgent in a real-world task setting. To this end, we construct a simple environment using toy kitchen utensils. The scene involves multiple tabletops to simulate the partial observability commonly encountered in household applications. We employ the Qwen3-VL-3B model trained on ALFRED. The task instruction, the list of all receptacles, and manually captured images are provided as inputs to the model, which generates a sequence of actions. The actions are then manually executed by a human operator. The experimental results are illustrated in Fig. 12. This example provides a preliminary demonstration of the feasibility of RoboAgent when applied to real-world observations.

11. Error Analysis

One advantage of our capability-driven task planning framework is that it enables fine-grained failure analysis. We manually examine all failed tasks in EB-ALFRED and ALFWorld and present their attribution in Fig. 13 and 14. For EB-ALFRED, approximately half of the failures come from the visual grounding stage. In some cases, OG fails to recognize the target object in the agent's field of view (Object Recognition), which may occur when the object is small or occluded. In other cases, EG and OG may not fully understand the open-vocabulary object queries (Open-Vocabulary Referring), causing the agent to miss the target object, interact with a wrong object, or output invalid actions due to wrong object category label. About 18% of failures are due to the scheduler incorrectly parsing the instruction, which typically occurs when the instruction involves complex sentence structures, irrelevant information, or long object queries. Approximately 17% of failures originate from low-level control errors, potentially due to simulator implementation issues (*e.g.*, when the agent is near an object but cannot interact with it, or when multiple instances of the same object category appear in the field of view and the agent cannot specify the intended target). Around 9% of failures result from ambiguities in the instruction itself, such as the agent choosing a soap bottle rather than a spray bottle when instructed to get a "bottle", or selecting a dining table instead of a side table for the query "table." Another 3% of errors arise from the ignored action preconditions, *e.g.*, put something to the cabinet when the cabinet is closed, stemming from inaccurate outputs of SD or missing actions in AD. Finally, 1% of errors are due to the ES capability (History Summarization), where it fails to correctly report the outcome of the previous actions and gives false assumptions for subsequent planning.

In ALFWorld, approximately 43% of errors also stem from object recognition. Although ALFWorld does not involve open-vocabulary descriptions, the restriction to moving toward receptacles sometimes places the target object at the edge of the observed image, increasing the difficulty of object grounding. About 20% of failures are due to exploration, where the model exhausts the allowed action steps without finding the target object, indicating room for improvement in EG.[4] Another 20% of failures are due to low-level control, similarly related to simulator implementation issues. The remaining 17% arise from ignored action preconditions. We note that, in ALFWorld, the action of "heat something with microwave" often fails when the

[4]We also observe that, in some tasks, the target object cannot be seen even when all feasible positions are explored.

Figure 12. A demonstration of real-world deployment. We present some of the input images and output actions. A human operator carried out the actions and captured the images with camera.

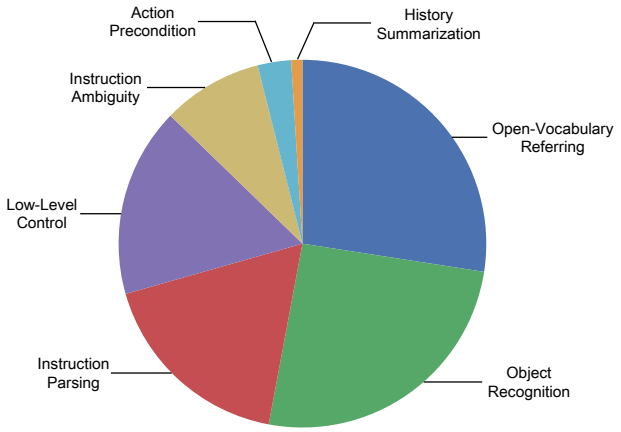

Figure 13. Error distribution on EB-ALFRED.

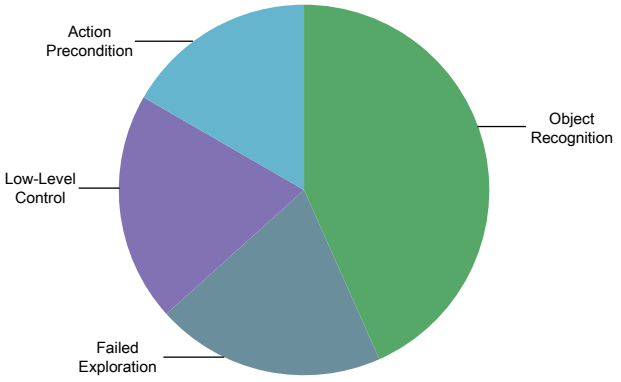

Figure 14. Error distribution on ALFWorld.

microwave is open, suggesting that AD still needs to learn certain simulator-specific action rules.

In Fig.15, we present the visualization of some typical failure cases for the scheduler and the capability.

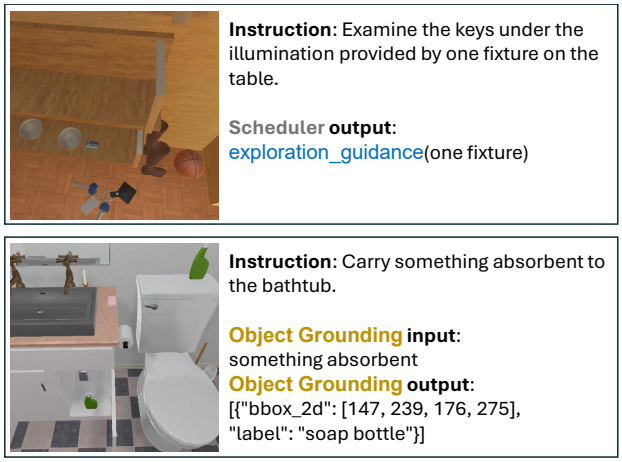

Figure 15. A visualization of two typical failure cases of RoboAgent. In the task above, the scheduler ignores the key information "illumination" from the instruction, and does not provide a complete and correct query to the invoked capabilities. In the task below, OG fails to understand the open-vocabulary query, resulting in the detection of an incorrect target object (soap bottle, instead of towel).

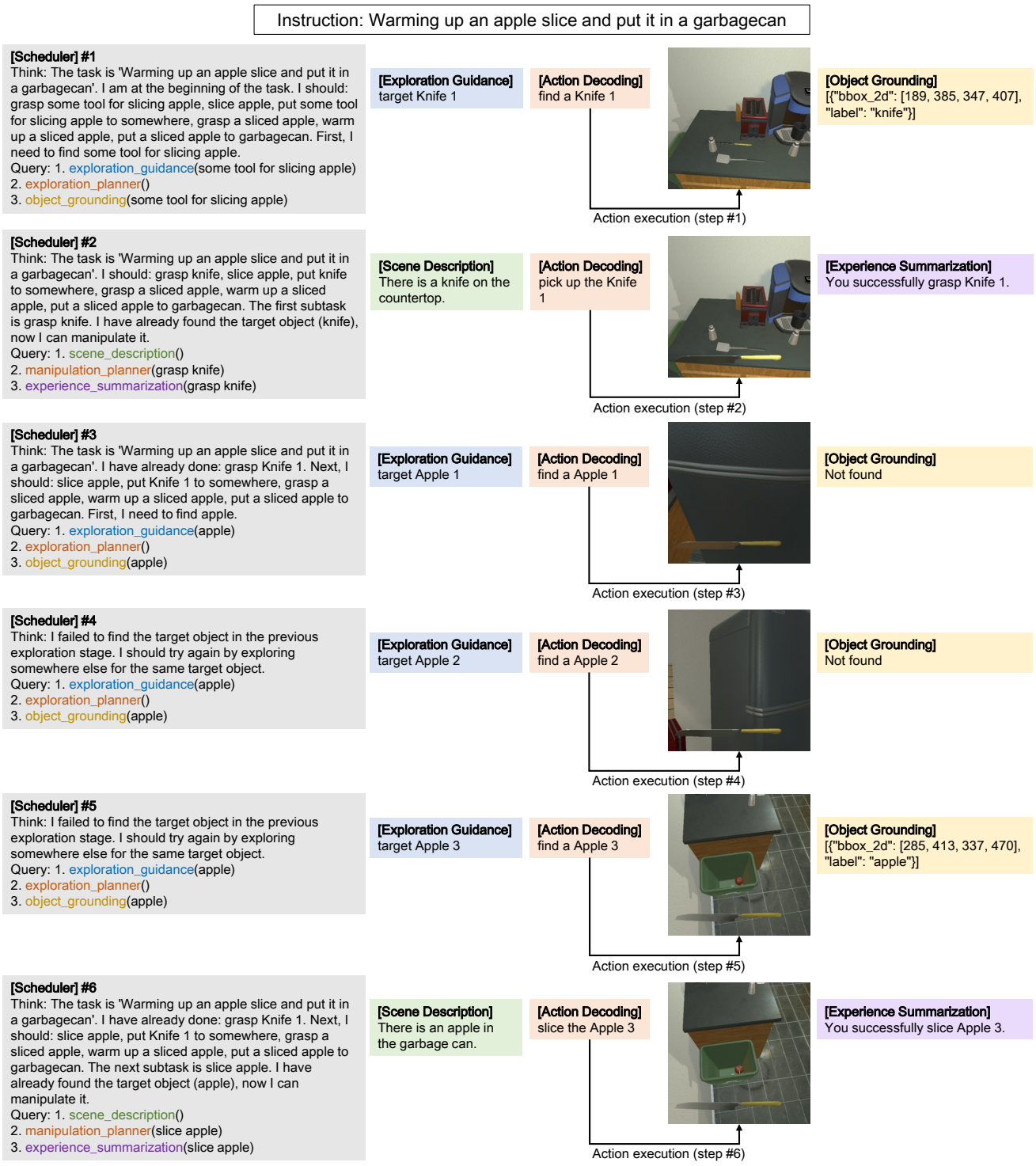

Figure 16. A visualization of the generated plan on EB-ALFRED (long horizon split), part 1.

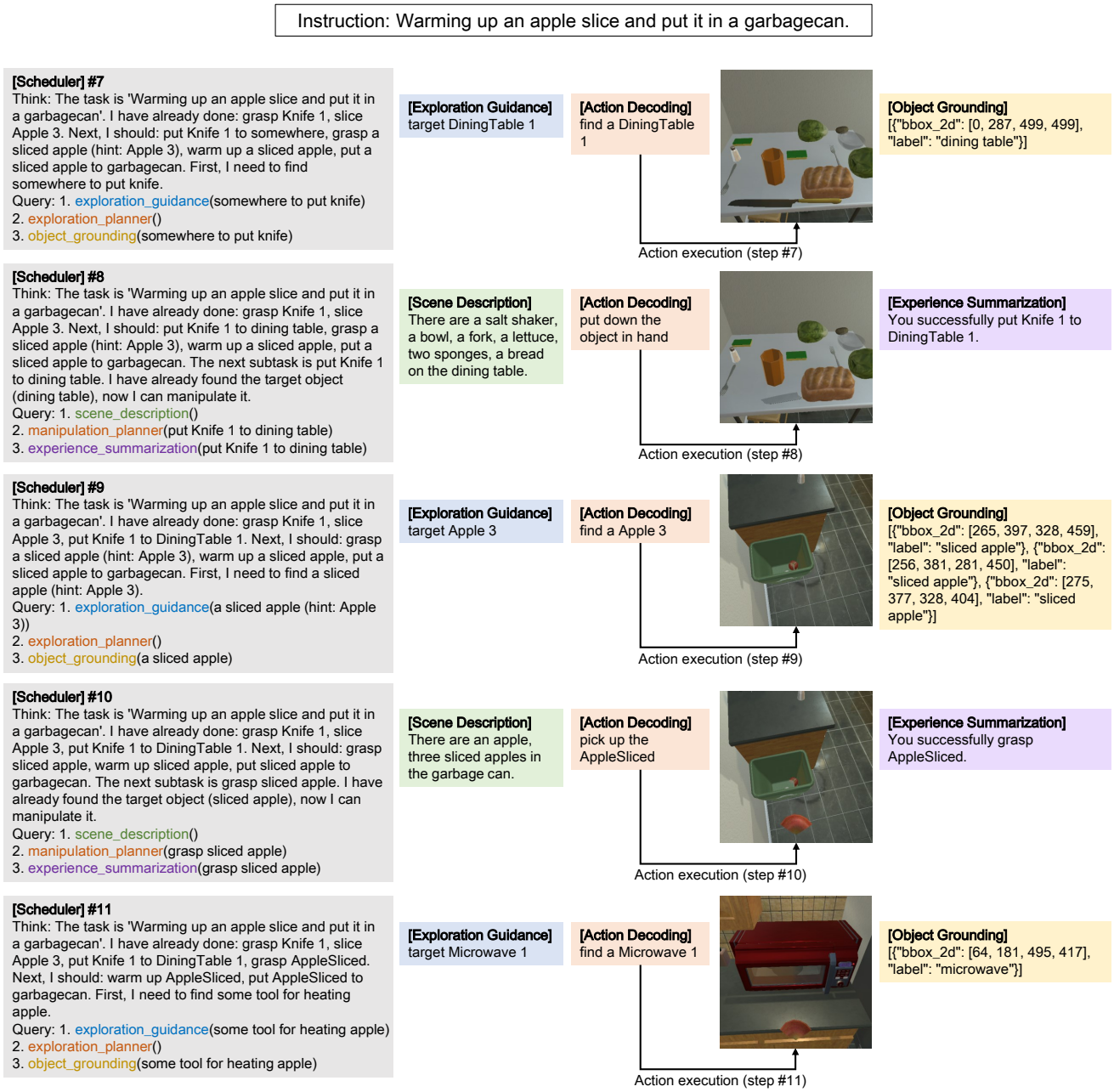

Figure 17. A visualization of the generated plan on EB-ALFRED (long horizon split), part 2.

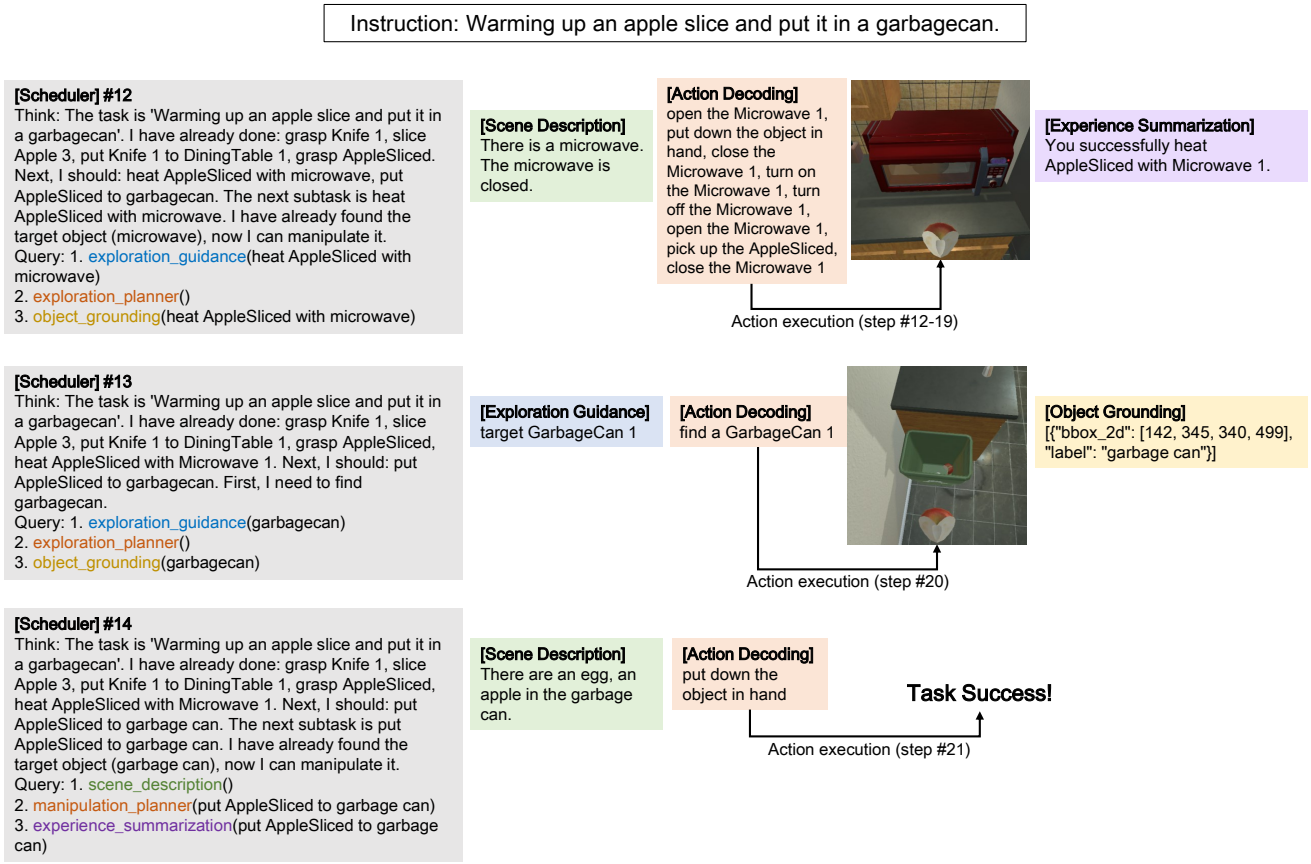

Figure 18. A visualization of the generated plan on EB-ALFRED (long horizon split), part 3.

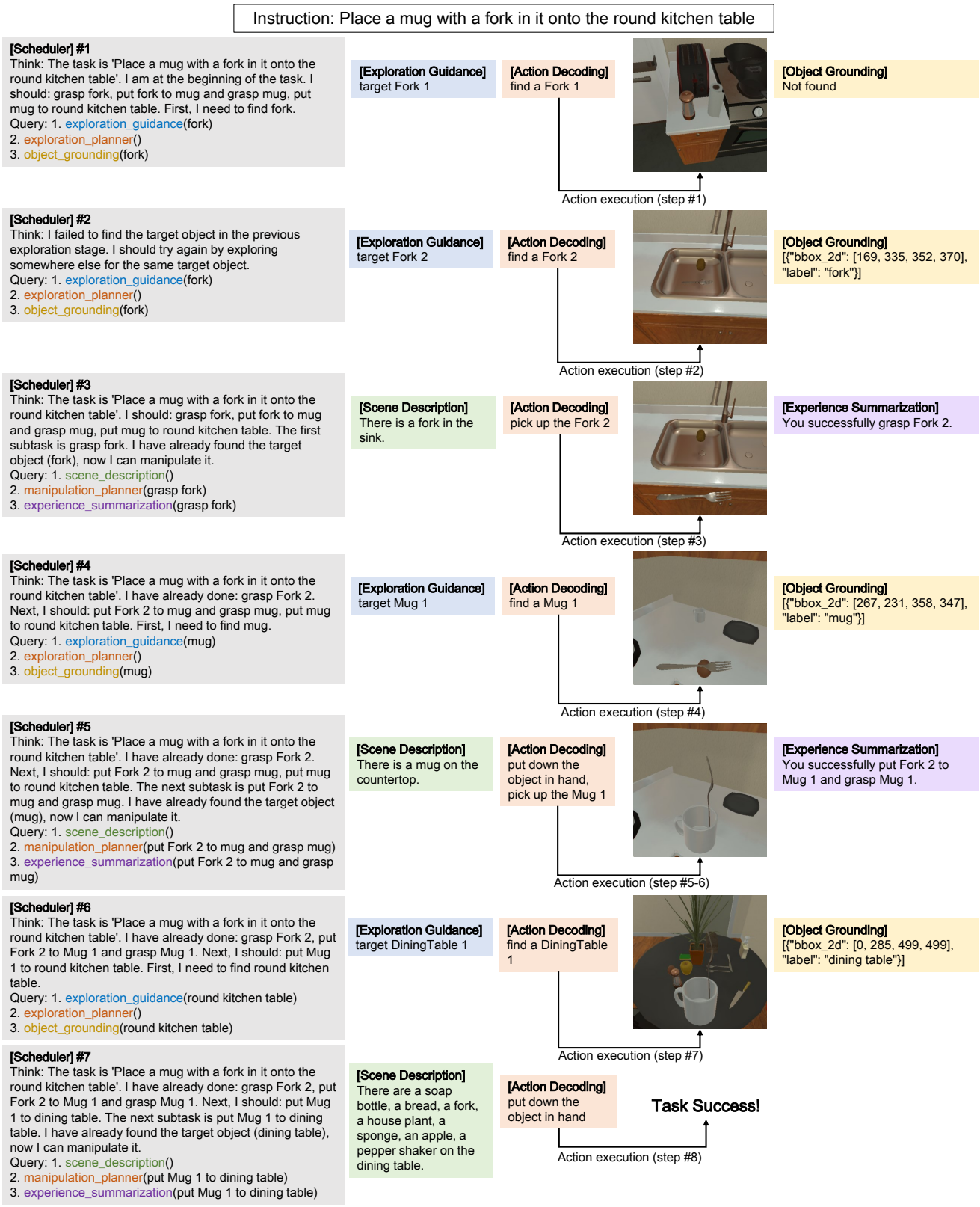

Figure 19. A visualization of the generated plan on EB-ALFRED (visual appearance split).

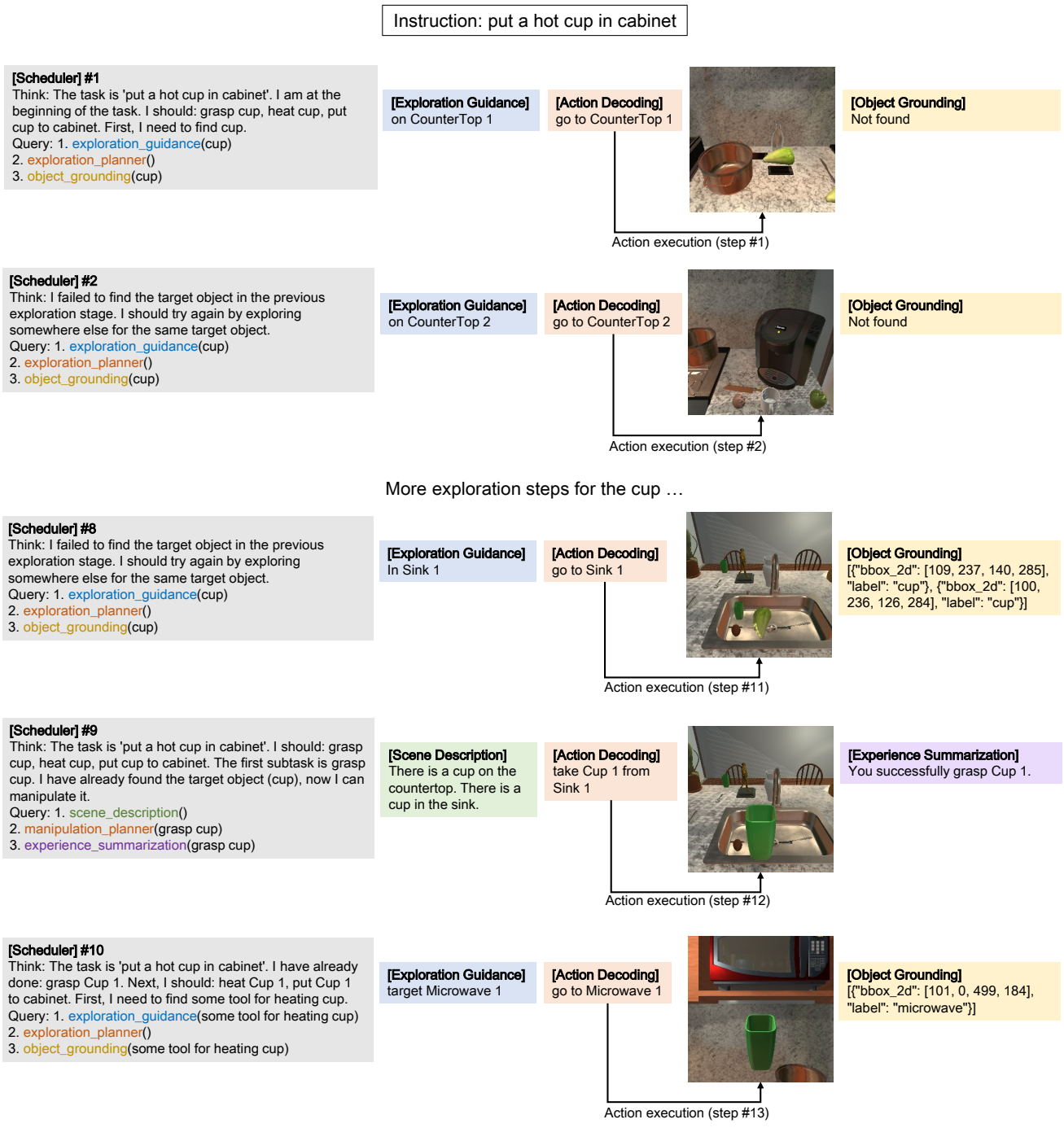

Figure 20. A visualization of the generated plan on ALFWorld's visual environment, part 1.

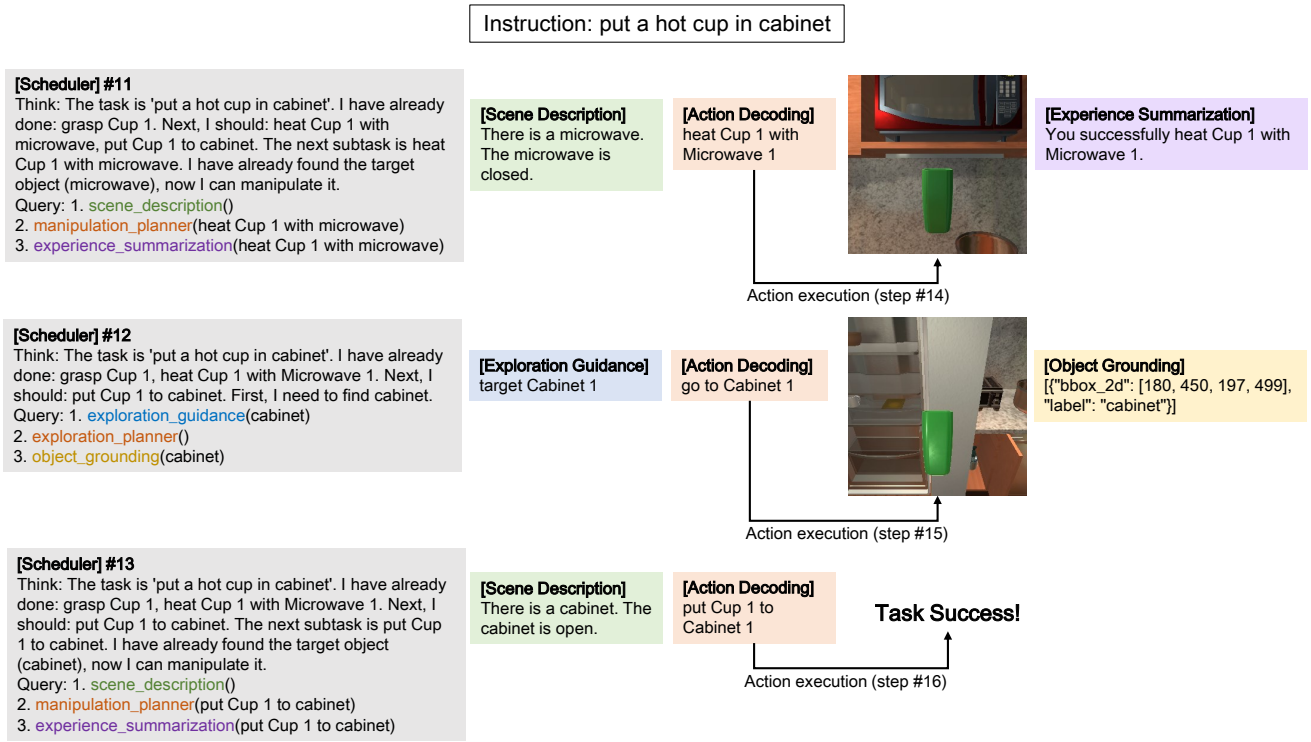

Figure 21. A visualization of the generated plan on ALFWorld's visual environment, part 2.

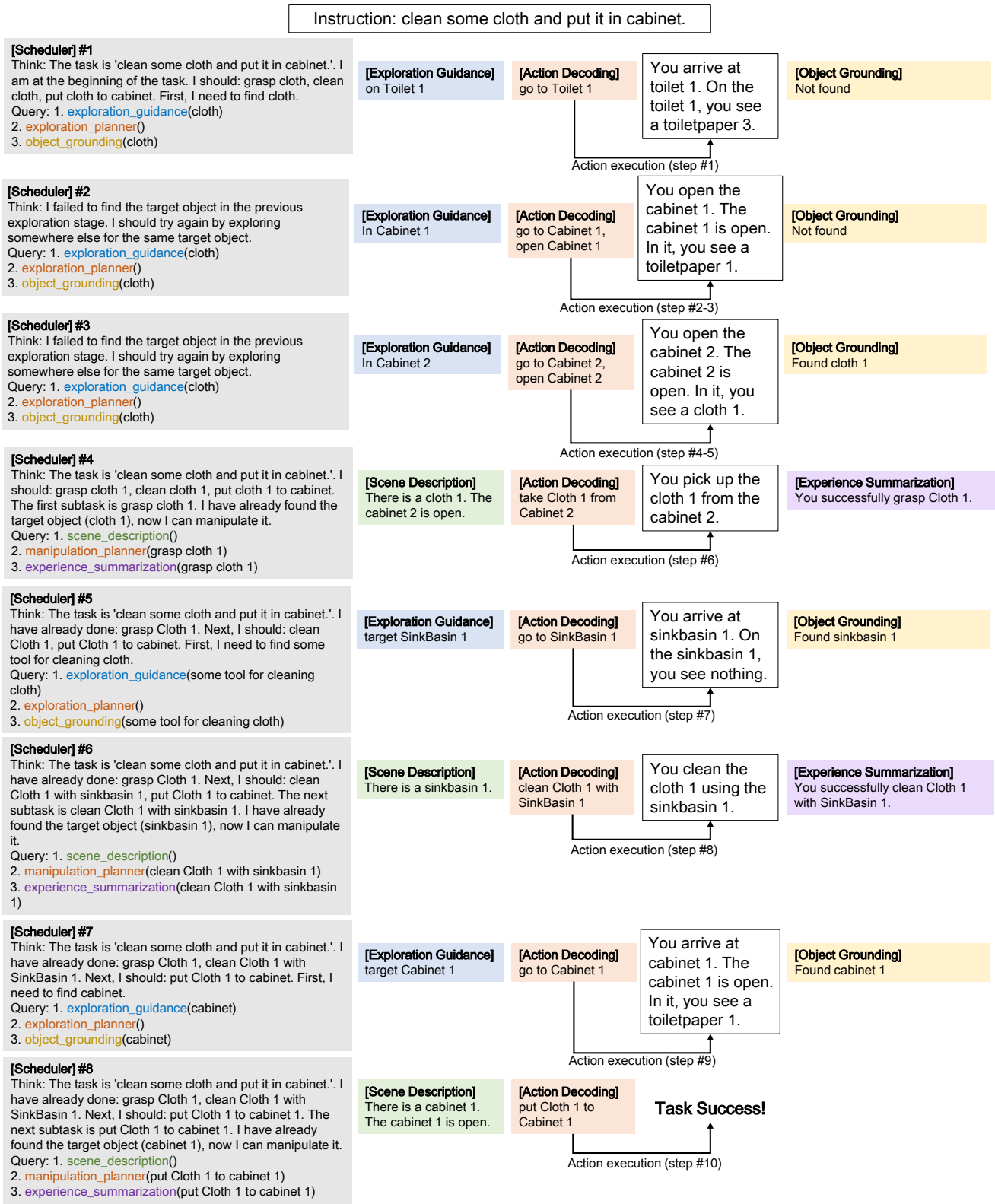

Figure 22. A visualization of the generated plan on ALFWorld's textual environment.