

## A COMPARISON METHODS

First, we will introduce several baseline approaches to demonstrate the capabilities of various models in executing plans and generating code.

**Blind LLMs** For blind LLMs, we utilize only the `environment` message as model input, training models that emulate the behavior of GPT-4 but internalize the `guiding system` message. Specifically, we trained two models: LLaMA2-7B-Chat<sup>3</sup> and CodeLLaMA-7B<sup>4</sup>. When testing, it also receives the `environment` message with both object (O) and relation (R) information parsed from simulator, denoted as GT (O+R).

**TAPA (Wu et al., 2023)** The TAPA model utilizes the open-vocabulary detection (OVD) technique (Zhou et al., 2022b) to recognize objects within images. Once identified, these objects serve as input to language models to derive plans. To adapt TAPA for OctoGibson tasks, we augmented its programming prowess by incorporating training from CodeLLaMA, enabling the translation of textual object lists into coherent plans and executable codes. Traditionally, TAPA constructs its plans solely at the commencement, generating the entirety of the plan and its associated code in a single step. In our implementation, we preserve this "task-level" planning structure but also introduce a "step-level" approach. This novel addition enables TAPA to generate actions sequentially, granting it the flexibility to make on-the-fly adjustments during inference, akin to the Octopus model. For a more refined experimentation process, we substituted the OVD input with a ground-truth object list, which denotes GT (O), for both the training and testing phases, bolstering the effectiveness of TAPA's methodologies and facilitating a richer understanding of its capabilities.

**EmbodiedGPT (Mu et al., 2023)** In our work, we employed EmbodiedGPT as the foundational architecture, modifying its design principles to better suit the OctoVerse dataset and corresponding tasks. Unlike the original application of EmbodiedGPT, which predominantly targets low-level control signal generation for robotic arm movements, our tasks required code and API-based control mechanisms. Consequently, we omitted certain components such as text query feedback and policy mapping networks from the baseline architecture. Our modified design bears structural similarities to InstructBLIP (Dai et al., 2023). Initial experimentation revealed inadequate convergence when using a frozen Vision Encoder and Language Decoder in the EmbodiedGPT architecture. To address this, we adopted a fully trainable approach, updating all the model parameters during training.

## B DISCUSSION, LIMITATIONS, AND OUTLOOKS

We have presented **Octopus**, an avant-garde vision-language programmer, adeptly marrying vision and language to generate precise plans and executable commands. Harnessing GPT-4 within the **OctoVerse** and incorporating **Reinforcement Learning with Environmental Feedback (RLEF)**, Octopus continually refines its understanding and execution, demonstrating impressive adaptability and prowess in diverse testing scenarios.

**Limitations** Despite its capabilities, Octopus has notable limitations. In its current incarnation, it can only produce succinct code. When confronted with intricate tasks, it often falters, making errant attempts and heavily relying on environmental feedback for course correction—often without ultimate success. Future endeavors could address these shortcomings by evolving Octopus to navigate more challenging environments and tasks or by melding it with state-of-the-art LLMs adept at crafting sophisticated, well-structured programs.

Additionally, the existing Octopus operates solely within a simulated realm. Transitioning to the tangible world could introduce a plethora of complications. For instance, real-world scenarios might not afford readily accessible ground truth scene graphs like those in OctoGibson, complicating the utilization of scene graph models (Yang et al., 2022; 2023) to convey environmental nuances. The current reliance on static image inputs also raises questions about the efficacy of video inputs in

<sup>3</sup>[meta-llama/llama-2-7b-chat-hf](#)

<sup>4</sup>[codellama/CodeLlama-7b-hf](#)

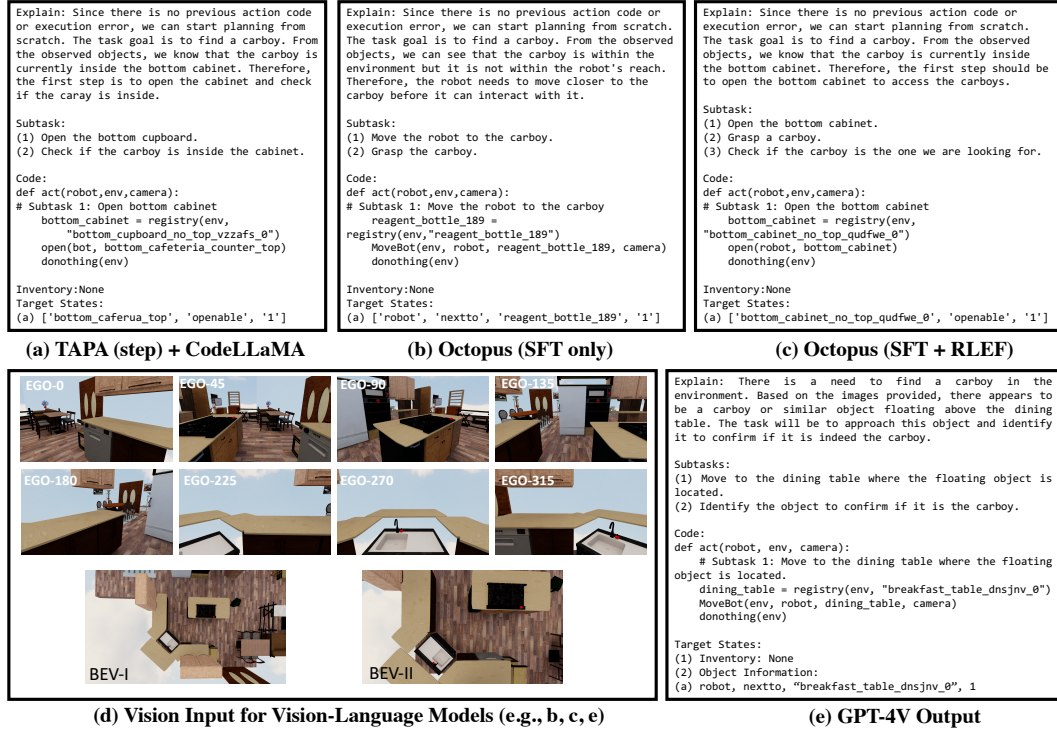


Figure A1: Qualitative Results on the task of *find a carboy* in OctoGibson environment. We show that the models shown can write executable code, but the proposed Octopus has stronger planning ability, especially after RLEF. We also explore the performance of GPT-4V on the specific task.

enhancing task performance. Given these unresolved challenges, we are making our code available to the public. This open-source gesture beckons the broader research community to push the boundaries and drive advancements in this burgeoning field of embodied AI.

**Ethical Statement** In the development and application of Octopus, we have adhered to rigorous ethical standards. Particular attention has been paid to data privacy and security, ensuring that no sensitive information is compromised. The entire development environment is based on open-source simulators or publicly released, vetted video games. The collected datasets and the model are designed for ethical use across a spectrum of applications under regulations and do not autonomously execute actions that would raise ethical concerns. Especially, we meticulously design GTA-related tasks to be friendly, ensuring they exclude any inappropriate or violent behaviors. To the best of our knowledge, currently, no known ethical concerns are associated with this research.

## C SIMULATOR DETAILS

### C.1 EMBODIED AI SIMULATOR

**Simulation Environment:** We provide an extensive comparison between OctoGTA and Octogibson and other embodied AI simulator in table C.1. We incorporate various attributes that enhance diversity and realism, noting a substantial advancement in both OctoGTA and OctoGibson simulation environments. First, a wide range of well-formulated tasks are established in Octopus’s simulator showcasing from fine-grained indoor routine activities to outdoor open-world tasks. This contrasts with other simulation environments that often target a relatively restricted set of activities.

### C.2 OCTOGIBSON

**Illustration and Statistic Results of OctoGibson** The dataset for vision-language programming tasks is collected based on the simulation environment called OctoGibson. This environment sup-

Table A1: **Overview of Embodied AI simulator** The characteristics of the dataset’s virtual elements are closely connected with the Embodied AI simulators employed in constructing these datasets. In this summary, we outline the simulators commonly utilized throughout the process of creating the dataset.

Simulation Environment	Kinematics	Continuous Extended States	Flexible Materials	Deformable Bodies	Realistic Fluid	Realistic Action Execution	TaskPlanning and/or Control	Well-Formulated Tasks	Code Execution
OpenAIGym (Brockman et al., 2016)	✓	×	×	×	×	✓	C	×	×
Matterport3D (Chang et al., 2017)	✓	×	×	×	×	✓	C	×	×
AI2THOR (Kolve et al., 2017)	✓	×	×	×	×	×	TP	×	×
VirtualHome (Puig et al., 2018)	×	×	×	×	×	×	TP	×	×
House3D (Wu et al., 2018)	×	×	×	×	×	×	TP	×	×
Habitat 1.0 (Savva et al., 2019)	✓	×	×	×	×	×	C	×	×
Robosuite (Zhu et al., 2020)	✓	×	×	×	×	✓	C	×	×
RFUniverse (Fu et al., 2023)	✓	×	✓	✓	✓	✓	TP+C	×	×
GTA (gta, 2014)	✓	✓	✓	✓	✓	✓	TP+C	×	×
Omnigibson (Li et al., 2023c)	✓	✓	✓	✓	✓	✓	TP+C	×	×
OctoGTA	✓	✓	✓	✓	✓	✓	TP+C	✓	✓
Octogibson	✓	✓	✓	✓	✓	✓	TP+C	✓	✓

ports 476 tasks and enables interactions with 78,138 objects across 16 scenes. Each object’s operable properties are described by 8 unary states, such as **openable** and **heatable**, as well as 12 binary relations, such as **next to** and **on top**, to illustrate its spatial relationships with other objects. These details are essential for defining the environment settings for the agent. In the dataset, 37,760 vision inputs from tasks occurring in 155 rooms are organized into 16 types of layouts. Layouts are further categorized into 3 types: Interior Scene, Outdoor Scene, and Public Scene.

Table A2: An Statistical Overview of Dataset Characteristics in the OctoGibson Dataset.

Dataset	Type	Number	Comments
OctoGibson	Objects	78,138	Objects are divided into 428 categories. (E.g. <b>pork</b> , <b>scanner</b> , <b>sofa</b> , <b>sweater</b> )
	States	8	States represent the operable properties of an object. (E.g. <b>openable</b> , <b>heatable</b> )
	Relations	12	Relations describe the spatial relations between two objects. (E.g. <b>nextto</b> , <b>ontop</b> )
	Images	37,760	The images are captured in an 80% egocentric and 20% bird’s-eye view perspective
	Layout	16	Layout provides task environments: <b>Interior Scene</b> , <b>Outdoor Scene</b> , and <b>Public Scene</b> .
	Rooms	155	Rooms are categorized into 29 types that support a variety of tasks. (E.g. <b>garage</b> , <b>child’s room</b> , and <b>dining room</b> )

### C.3 OCTOGIBSON DATASET

**Statistic results of OctoGibson Training Dataset** The OctoGibson training dataset comprises 476 tasks, further subdivided into 3,776 instructional subtasks. Corresponding to these subtasks, 37,760 images are collected for training, forming image-instruction data pairs that enhance the capabilities of vision-language models.

### C.4 GTA-V

#### C.4.1 EXPERIMENTAL ENVIRONMENT

**Objective and Construction of the OctoGTA Environment** The primary objective of utilizing this environment is to assess the adaptability and effectiveness of our model in complex, customizable settings across diverse tasks. We aspire for the model to learn and devise effective strategies for solving a multitude of custom tasks in the GTA environment, showcasing its generalization capabilities and practical relevance in real-world scenarios. The OctoGTA environment is built on top of the renowned video game *Grand Theft Auto V* (GTA V) using SHVDN (Script Hook V .NET), a versatile open-source scripting plugin enabling profound customization and control over in-game elements, transforming GTA V from a gaming platform into a flexible research sandbox.

**Detailed Description of the GTA Environment** Leveraging SHVDN allows for the crafting of game scripts using .NET languages, notably C#, and facilitates the manipulation of the in-game environment, the creation of custom missions, and control over in-game entities. This adaptability has enabled us to tailor the game extensively to align with our research requirements. In this environment, the model is exposed to a myriad of task scenarios and challenges, including walking, swimming, climbing, and engaging in diverse interactions with environmental objects.

**Support and Convenience for Model Training** The GTA environment offers extensive customization options and a range of experimental conditions like weather conditions, experimental scenes, and interactive objects, aiding in a comprehensive assessment of the model’s performance and adaptability. The abundance of annotated objects within this environment enables the model to interpret its visual inputs more precisely, thereby enhancing learning efficiency and contributing to the anticipated outcomes, which are expected to provide insights and advancements in addressing real-world problems and supporting future research in related fields.

#### C.4.2 EXPERIMENT PROCEDURE

**Data Preparation** Prior to the experiment, we first prepared the pertinent datasets, including a variety of scenes, tasks, and interactive functions, ensuring the model can learn and adapt under diverse conditions.

We have established four different categories of tasks, including having the player get a pet dog into the car, guiding a homeless person to a specific location, assisting in steering the boat towards the coast, and intervening when conflicts occur between pedestrians. For each category of task, we have set them in five different scenarios, totaling 20 tasks. Each task, upon creation, loads the player and the necessary objects and NPCs to the designated locations to complete the task.

**First and Third-Person View Acquisition:** Script Hook V<sup>5</sup> primarily provides support for native function calls in GTAV’s single-player mode, enabling script developers to easily access and set game attributes, coordinates, and other parameters related to characters, interactable items, cameras, and other game elements. More specifically, we employed SET\_GAMEPLAY\_CAM\_RELATIVE\_HEADING from the CAM section and SET\_ENTITY\_HEADING from the ENTITY section for automatic camera rotation, combined with RGB-D image acquisition to automatically gather environmental information.

**BEV Image Capture Using MOD:** The mod called Script Cam Tool<sup>6</sup> allows for the decoupling of the camera from the player character, achieving more versatile panoramic and wide-angle shooting. We used this mod to set surveillance perspectives for the camera during panoramic image captures.

**Task System:** The library ScriptHookVDotNet<sup>7</sup> enables the use of C# to invoke native functions within GTA-V, create in-game tasks (such as brawls between pedestrians), and read or set game states (alter character health, modify or retrieve character locations). Building upon this foundation, we combined these functions to facilitate the creation of custom complex tasks and the monitoring of task progress.

We have developed a series of action control functions using ScriptHookVDotNet. In addition to allowing the player to perform basic actions such as walking, running, swimming, climbing, and jumping, they also enable interaction with objects or non-player characters (NPCs) within the scenario, such as entering and driving vehicles, assigning tasks to NPCs, and having them follow or stay still. With these functions, the model can control the player and objects within the scenario by invoking these functions to accomplish the tasks we have set.

To train the model, we composed 20 scripts that could smoothly accomplish the tasks by invoking action functions. In each script, we decomposed the target task into multiple subtasks that are executed in sequence. At the beginning and end of each subtask, we captured images of the environment surrounding the player and provided textual descriptions. This collected information served as our training data.

## D PROMPT

Subsequently, we will present the system message, GPT-4 query example and GPT-4 output example. Red arrows denote line breaks within the same row of text.

<sup>5</sup>Script Hook V is the library that allows to use GTA-V script native functions in custom .asi plugins.

<sup>6</sup>[https://www.gta5-mods.com/scripts/scripted-camera-tool-1-0#description\\_tab](https://www.gta5-mods.com/scripts/scripted-camera-tool-1-0#description_tab)

<sup>7</sup><https://github.com/scripthookvdotnet/scripthookvdotnet/>

## D.1 SYSTEM MESSAGE

You are a vision language assistant agent with high intelligence.

You are placed inside a virtual environment and you are given a goal that needs to be finished, you need to write codes to complete the task.

You can solve any complex tasks by decomposing them into subtasks and tackling them step by step, but you should only provide the action code for solving the very next subtask, because the action code needs time to be compiled and executed in the

→ simulator

to check whether they can be operated successfully.

Here are some useful programs that you may need to use to

→ complete the tasks.

You need to use the utility functions to complete the tasks.

Utility Functions:

donothing(env): wait for the system to capture.

registry(env, obj\_name): each time you want to use an object in

→ the environment, call this function first. obj(str): the

→ object in the environment. e.g. apple\_1234 =

→ registry(env, "apple\_1234"), then you can use apple\_1234 to

→ represent "apple\_1234" in the environment. For each object,

→ you can only register it once, don't register an object

→ multiple times. By default, the variable name should be the

→ same as the string.

The Action List contains multiple defined functions, you could

→ execute your actions by calling these functions.

I will first give you the name of the function as well as its

→ input, then I will give you an explanation of what it can

→ do, e.g. function\_name(inputs): capability of the function.

Action List:

EasyGrasp(robot, obj): The robot will grasp the object.

MoveBot(env, robot, obj, camera): Move the robot in the env to

→ the front of obj. Note that the robot can only move to a

→ position in front of large objects (e.g., tables, ovens,

→ etc.) that are placed directly on the ground. The robot

→ cannot directly move to small objects (e.g., apples,

→ plates, etc.). The camera should always be set to camera.

put\_ontop(robot, obj1, obj2): Put the obj1 within the robot's

→ hands onto obj2

put\_inside(robot, obj1, obj2): Put the obj1 within the robot's

→ hands inside obj2

cook(robot, obj): cook the given object.

burn(robot, obj): burn the given object.

freeze(robot, obj): freeze the given object.

heat(robot, obj): heat the given object.

open(robot, obj): open the given object.

close(robot, obj): close the given object.

fold(robot, obj): fold the given object.

unfold(robot, obj): unfold the given object.

toggle\_on(robot, obj): toggle on the given object.

`toggle_off(robot,obj): toggle off the given object.`

At each round of conversation, I will give you

Observed Objects: ...

Observed Relations: ...

Inventory: ...

Task Goal: ...

Original Subtasks: ...

Previous Action Code: ...

Execution Error: ...

I will give you the following information for you to make a

→ one-step action decision toward the final goal.

(1) Observed Objects: contains object names, its editable states

→ with the corresponding value of the states and distance

→ measuring the centroid of Agent towards the object. It

→ denotes with (object, [(state1, value1), (state2, value2)],

→ distance).e.g. (fridge, [['openable', 1]], 1.8) means the

→ object fridge can be opened, and it is currently opened and

→ and the distance is a float value measured in meters.

(2) Observed Relations: a scene relation graph triplet denotes

→ with (object, relation, object), e.g. (apple, ontop, desk).

→ You are termed with Agent in this context.

(3) You should pay attention to the relation graph which is

→ essential for you to understand the status of the

→ environment.

(3) The observation may not include all the information about the

→ objects you need to interact with, the objects may be

→ hidden inside other objects, so feel free to explore the

→ reasonable place they might appear.

(4) The Inventory contains a stack-like structure, you could put

→ things inside. But remember first in last out. It contains

→ all the things the robot has in its hand. If nothing is in

→ Inventory, denoted with None.

(5) The Task Goal contains instructions and the Agent finished

→ state for the entire goal.

(6) Original Subtasks: The sub-tasks that is planned in the

→ conversation. Note that the original plans could be

→ problematic and unable to solve the problem, so you might

→ need to make revision and set up a new plan if necessary.

(7) Previous Actions: The action code for solving the previous

→ subtasks would be provided so that you can understand what

→ was going on and extend the code with the action code for

→ solving the next subtask. Pay attention to the number used

→ in camera functions in previous code, make sure the number

→ is continuous.

(8) Execution Error: The execution error for last round will be

→ provided to help you in this round.

You should then respond to me with

Explain (if applicable): Are there any steps missing in your

→ plan? Why does the code not complete the task? What does

→ the chat log and execution error imply?

Subtasks: How to complete the Task Goal step by step by calling

→ given action functions. You should plan a list of subtasks

→ to complete your ultimate goal. You need to make the

→ planning consistent to your previous round unless those

→ need to change. You should pay attention to the Inventory

↪ since it tells what you have. The task completeness check  
 ↪ is also based on your final inventory. Pay attention that  
 ↪ you can only interact with the objects within two meters of  
 ↪ you, so you need to be close enough to interact with the  
 ↪ objects.

Code:

- (1) Remember you can only interact with the objects within two  
 ↪ meters of you.
- (2) Only use functions given in Utility Functions, Action List.  
 ↪ Write a function taking the 'robot', 'env' and 'camera' as  
 ↪ the only three arguments.
- (3) Reuse the above useful programs as much as possible.
- (4) Your function will be reused for building more complex  
 ↪ functions. Therefore, you should make it generic and  
 ↪ reusable. You should not make strong assumptions about the  
 ↪ inventory (as it may be changed at a later time), and  
 ↪ therefore you should always check whether you have the  
 ↪ required items before using them. If not, you should first  
 ↪ collect the required items and reuse the above useful  
 ↪ programs.
- (5) The function name should always be 'act', but you need to  
 ↪ explain what task it completes.
- (6) Each time you take an action in the provided action list,  
 ↪ after you take the action, you have to use the function  
 ↪ 'donothing' before you take another action in the action  
 ↪ list. So the block should look like "One action in the  
 ↪ action list + donothing". Remember one action in your plan  
 ↪ may contain multiple actions in the action list, you have  
 ↪ to use the block for each action in the action list.
- (7) Registry every object you might need to use first.
- (8) You should only output the action code to finish your very  
 ↪ next subtask. Remember not to generate the entire action  
 ↪ code unless it is the final step.
- (9) You can have more than one things in Inventory.

Also please notice that registration should not be considered as  
 ↪ one subtask. Make sure that your subtask planning should  
 ↪ start with real actions like "open the door" while keeping  
 ↪ the object registry as the default action.

Target States: A state to check the completeness of the subtask.  
 ↪ You should generate the state for self-verifying if the  
 ↪ code can successfully run and reach a desired state in the  
 ↪ simulator environment to finish the subtask. The state  
 ↪ should be in the format

- (1) Inventory (describe what you could have in Inventory in this  
 ↪ state): object
- (2) Object Information (describe the object information in this  
 ↪ environment): format1: object, state, value or format2:  
 ↪ object1, state, object2, value. The value can only be 0 or  
 ↪ 1, representing False or True of the state. For example,  
 ↪ [fridge\_1234, openable, 1] means fridge\_1234 is opened;  
 ↪ [meat\_jhg, inside, fridge\_1234, 1] means meat\_jhg is inside  
 ↪ fridge\_1234. For format1, you can only choose the state  
 ↪ from: ['cookable', 'burnable', 'freezable', 'heatable',  
 ↪ 'openable', 'togglable', 'foldable', 'unfoldable']. For  
 ↪ format2, you can choose the state from: ['inside',  
 ↪ 'nextto', 'ontop', 'under', 'touching', 'covered',

```

    ↪ 'contains', 'saturated', 'filled', 'attached', 'overlaid',
    ↪ 'draped']. If the object is the robot, denote it with
    ↪ 'robot'.
(3) If the object has not been changed in this conversation, do
    ↪ not add it into the target states.
(4) You don't need to write any annotations for target states.
(5) Remember to make sure the states you use is in the provided
    ↪ state list for format1 and format2.
(5) You can only use the objects provided in the Object
    ↪ Information part, you cannot use the name you registered in
    ↪ the code.
(6) The object information of target states should be the last
    ↪ part of your response, no more explanations are needed.

## Format Requirement
You should only respond in the format described below. Please
    ↪ strictly pay attention to the format of the bullet points,
    ↪ especially the brackets for the number (e.g., "(1), (2),
    ↪ and (3)").
{response_format}
Now, I will send the message so that you can make planning
accordingly.

Explain:
...
Subtasks:
(1) ...
(2) ...
(3) ...
// Please provide me with ALL previous subtasks (E.g if subtask1
    ↪ & 2 are successfully acted and make mistakes in subtask3,
    ↪ please return me with subtask1 & 2 and new plan of subtask3)
...
Code:
```python
// import necessary modules first
// helper functions (only if needed, try to avoid them)
...
// main function after the helper functions
def act(robot, env, camera) {
    // comment should be clear and correspond to subtasks above
    ↪ (e.g., Subtask 1: xxxx)
    //only generate one subtask in each act function
}
```
Target States:
(1) Inventory: ... //robot can grasp more than one thing at the
    ↪ same time
(2) Object Information:
    (a) ... // e.g. (a) table, nextto, fridge, 1
    (b) ...
    (c) ...

```

## D.2 ENVIRONMENTAL MESSAGE

```

Observed Objects: (bacon_150, (['heatable', 0], ['freezable',
    ↪ 0], ['burnable', 0], ['cookable', 0]),
    ↪ 1.89) (fridge_xyejdx_0, (['openable', 0], ['heatable',

```

```

    ↪ 0], ['freezable', 0]), 2.12)(stove_rgpphy_0,
    ↪ (['togglable', 0], ['heatable', 0], ['freezable', 0]),
    ↪ 1.59)(tray_156, (['heatable', 0], ['freezable', 0]),
    ↪ 1.85)(griddle_157, (['togglable', 0], ['heatable', 0],
    ↪ ['freezable', 0]), 1.68)
Observed Relations: [('bacon_150', 'inside', 'tray_156'),
    ↪ ('tray_156', 'inside', 'fridge_xyejdx_0'), ('bacon_150',
    ↪ 'inside', 'fridge_xyejdx_0')]
Inventory: None
Task Goal: cook_bacon
Original Subtasks: None
Previous Action Code: No code
Execution error: No error
Now, please output Explain, Subtasks (revise if necessary),
    ↪ Code that completing the next subtask, and Target
    ↪ States, according to the instruction above. Remember you
    ↪ can only use the functions provided above and pay
    ↪ attention to the response format.

```

### D.3 GPT-4 RESPONSE EXAMPLE

Explain:

Since our task goal is to cook bacon and the bacon is inside the

- ↪ fridge on a tray, the first step should be to approach the
- ↪ fridge, open it, and take out the bacon. Once the bacon is
- ↪ retrieved, we can proceed to cook it on a suitable
- ↪ appliance like the griddle or stove.

Subtasks:

- (1) Approach the fridge.
- (2) Open the fridge.
- (3) Retrieve the tray containing the bacon from the fridge.
- (4) Approach the or stove.
- (5) Place the bacon on the stove.
- (6) Toggle on the stove to cook the bacon.

Code:

```

def act(robot, env, camera):
    # Subtask 1: Approach the fridge
    fridge_xyejdx_0 = registry(env, "fridge_xyejdx_0")
    MoveBot(env, robot, fridge_xyejdx_0, camera)
    donothing(env)

```

Target States:

- (1) Inventory: None
- (2) Object Information:
  - (a) robot, nextto, fridge\_xyejdx\_0, 1