

MALMM: Multi-Agent Large Language Models for Zero-Shot Robotic Manipulation

Harsh Singh^{*†1}, Rocktim Jyoti Das^{†1}, Mingfei Han¹, Preslav Nakov¹, Ivan Laptev¹

Abstract—Large Language Models (LLMs) have demonstrated remarkable planning abilities across various domains, including robotic manipulation and navigation. While recent work in robotics deploys LLMs for high-level and low-level planning, existing methods often face challenges with failure recovery and suffer from hallucinations in long-horizon tasks. To address these limitations, we propose a novel multi-agent LLM framework, *Multi-Agent Large Language Model for Manipulation* (MALMM). Notably, MALMM distributes planning across three specialized LLM agents, namely high-level planning agent, low-level control agent, and a supervisor agent. Moreover, by incorporating environment observations after each step, our framework effectively handles intermediate failures and enables adaptive re-planning. Unlike existing methods, MALMM does not rely on pre-trained skill policies or in-context learning examples and generalizes to unseen tasks. In our experiments, MALMM demonstrates excellent performance in solving previously unseen long-horizon manipulation tasks, and outperforms existing zero-shot LLM-based methods in RLBench by a large margin. Experiments with the Franka robot arm further validate our approach in real-world settings.

I. INTRODUCTION

Robotic manipulation has seen impressive advancements, enabling agents to handle increasingly complex tasks with greater precision and efficacy. Current solutions, however, often struggle with generalization, in particular when using imitation learning for policy training [1], [2]. Such methods typically excel at specific tasks but lack the adaptability to handle new tasks. One major drawback of imitation learning is the labor-intensive and time-consuming process for data collection, which limits the scalability of resulting policies. Moreover, training task-specific manipulation policies typically require thousands of training episodes [1], [2], making the approach computationally expensive and inefficient. To cope with the generalization, robotics policies should demonstrate a deeper understanding of their environment. This involves recognizing and grounding relevant objects and understanding the relationships between them [3]. Equipped with this knowledge, policies can then plan and execute actions more efficiently while adapting to changes in the environment and new task requirements.

Recent advancements in LLMs have demonstrated remarkable generalization and reasoning capabilities across diverse domains such as commonsense, mathematical, and symbolic reasoning [4]. These models, particularly when scaled to billions of parameters, exhibit emergent abilities to break

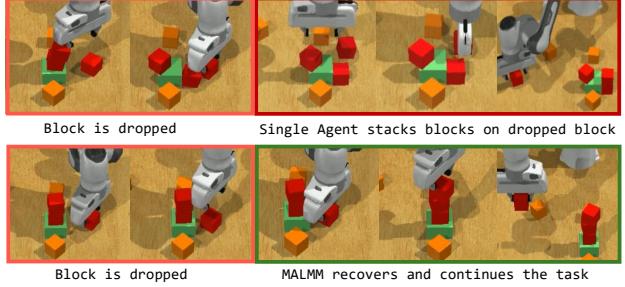


Fig. 1: Examples of executing “*Stack four blocks at the green target area*” task by the Single Agent LLM (top) and our Multi-Agent MALMM framework (bottom). MALMM recovers after dropping one block and continues stacking above the target area, while the Single Agent mistakenly continues stacking blocks on top of the dropped block.

down complex tasks into simpler steps through structured reasoning techniques such as *chain of thought* [5]. LLMs have already shown promise in high-level task planning in different domains, suggesting their potential for flexible and versatile robotic manipulation. Yet, recent LLM-based methods for robotic planning face multiple challenges. One important issue is the tendency of LLMs to produce incorrect high-level plans and low-level control. Additionally, LLMs suffer from hallucinations in long-context generation [6], which is often observed in closed-loop LLM systems. As a result, they may disregard geometric constraints and the parameters of predefined functions or may even lose the sight of the goal.

In this work, we propose a Multi-Agent Large Language Model for Manipulation (MALMM) to leverage the collective intelligence and the specialized skills of multiple agents for complex manipulation tasks. Our framework incorporates agents dedicated to high-level planning, low-level code generation and a supervisor that oversees transitions between other agents and tools. We show that through the use of multiple specialized agents in a multi-agent setup, we are able to mitigate the hallucination issues observed in the case of a single agent, as shown in Fig. 1.

Our contributions can be summarized as follows:

- We introduce the first multi-agent LLM framework for robotic manipulation MALMM, equipped with specialized agents that bring collaborative and role-specific capabilities to handle unseen, diverse, and complex manipulation tasks.
- We demonstrate the advantages of the proposed multi-

^{*}Corresponding author.

[†]Equal contribution.

¹Mohamed bin Zayed University of Artificial Intelligence, UAE
email: firstname.lastname@mbzui.ac.ae,

Project Page: <https://malmm1.github.io/>

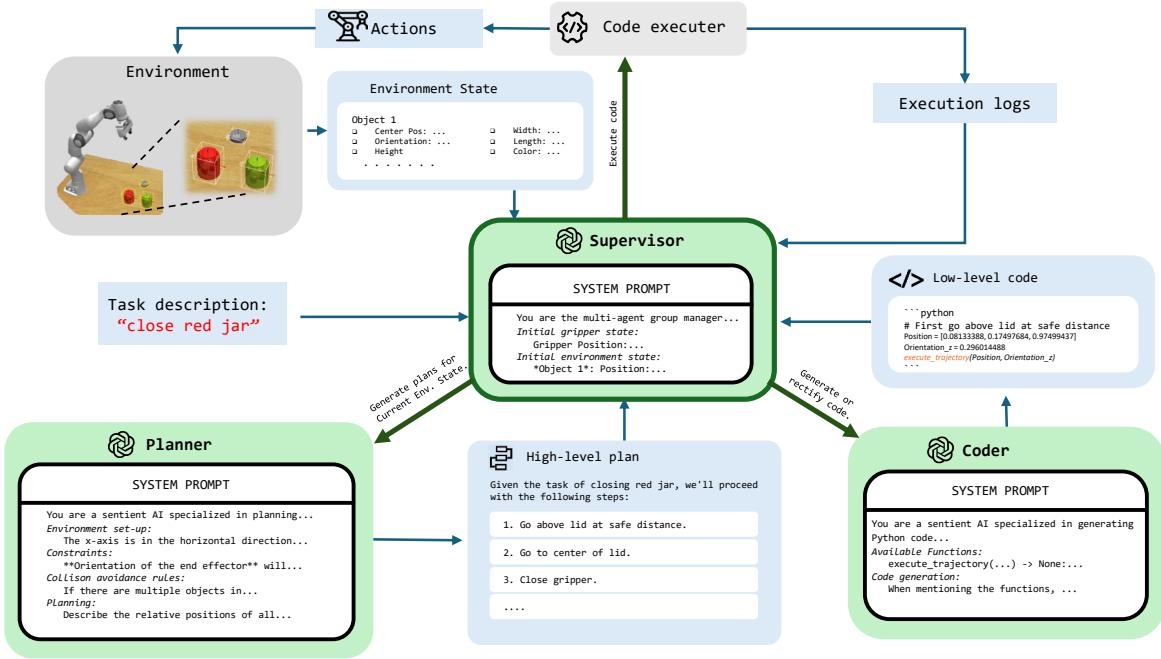


Fig. 2: An overview of our multi-agent system, **MALMM**, which consists of three LLM agents—**Planner**, **Coder**, and **Supervisor**—and a **Code executor** tool. Each agent operates with a specific system prompt defining its role: (1) the **Planner** generates high-level plans and replans in case of intermediate failures, (2) the **Coder** converts these plans into low-level executable code, and (3) the **Supervisor** coordinates the system by managing the transitions between the **Planner**, the **Coder**, and the **Code executor**.

agent framework through systematic ablation studies on tasks with varying horizons and complexity.

- We evaluate MALMM in challenging zero-shot settings both in a simulation and in the real world, and we show substantial improvements over state-of-the-art methods.

II. RELATED WORK

A. Language Grounded Robotics

Language instructions enable the definition of complex robotics tasks with compositional goals [7] and support scalable generalization to new tasks [8]. The literature around language grounding is vast, ranging from classical tools such as lexical analysis, formal logic, and graphical models to interpreting language instructions [7], [9]. Recently, much effort has focused on adopting the impressive capabilities of LLMs to language grounding in robotics [8]. Additionally, recent advancements have benefited from pre-trained LLMs thanks to their open-world knowledge, tackling more challenging tasks such as 3D robotic manipulation and leveraging code generation capabilities to produce high-level, semantically-rich procedures for robot control.

B. LLM for Robotics

Language Models have been used for various robotics purposes including the definition of reward functions [10], task planning [11], [12], [13], failure summarization and guiding language-based planners [14], and policy program synthesis [15]. VoxPoser [16] and Language to Rewards [17]

used LLMs for generating reward regions for assisting external trajectory optimizers in computing trajectory. Our work is most related to methods using LLMs for manipulation planning. Most of such work [12], [16] relies on pre-trained skills, motion primitives, and trajectory optimizers and has focused primarily on high-level planning. The closest to our approach is *Language Models as Zero-Shot Trajectory Generators* [11], which deployed a language model to generate high-level plans and then convert these plans into low-level control. However, LLMs suffer from hallucinations, which affect long-horizon task planning. Moreover, [11] assumed the correct execution of each step and did not account for occasional failures or unforeseen changes in the environment. Our evaluation shows sizable improvements of MALMM over [11] thanks to its multiple specialized agents and intermediate environment feedback.

C. LLM-Based Multi-Agents in Robotics

Recently, several studies have focused on using LLMs for sequential decision-making and complex reasoning tasks [18]. There is also an emerging field of using multiple agents driven by LLMs to solve complex tasks, including robotic manipulation, in a collaborative manner [19]. In most of the work in robotics, LLMs have been used in multi-robot collaboration and communication. Moreover, Reinforcement Learning (RL) policy agents are also used in collaboration with LLMs, which limits their generalization to new tasks and environments [20]. In contrast, we propose a multi-agent framework, MALMM, which incorporates three role-specific

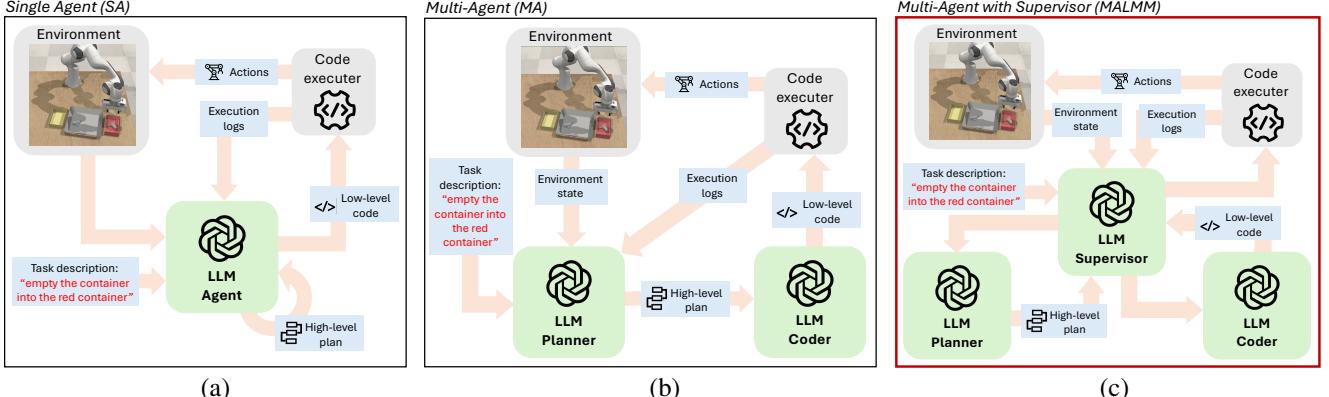


Fig. 3: Agents for robotic manipulation: The figure illustrates three LLM-based manipulation frameworks: **SA**, **MA**, and **MALMM**, with the different number of agents in each framework. All three frameworks begin by receiving an input command and the initial environment observation. Each framework iteratively generates a high-level plan along with corresponding low-level code. After each intermediate step, the frameworks use updated environment observation to detect failures and replan as needed until the task is completed.

LLM agents and enables zero-shot execution of previously unseen robotic manipulation tasks.

III. METHOD

LLMs have recently emerged as a universal tool for a variety of tasks, including robotic navigation and manipulation [21], [16], [15], [11]. Such models exhibit surprising generalization abilities and support zero-shot solutions for new tasks. While several recent methods have explored LLMs for high-level task planning [22], [12], other methods attempted to bring the power of LLMs to the low-level trajectory planning [15], [11]. We follow these works and leverage LLMs to precisely control the end-effector for solving complex and previously unseen manipulation tasks, without relying on any predefined skill library for sub-tasks in the manipulation process.

Trajectory planning is a complex problem that requires reasoning about the precise shape, position, and semantics of the objects as well as understanding object interactions. To solve this problem, LLM agents can be instructed to handle information about objects and robots by *text prompts*. To harness the complexity of the problem, here we propose to address the manipulation problem with multiple LLM agents. Inspired by recent work on multi-agent LLMs, we design agents specialized in different aspects of the manipulation problem and connect them into a framework. Below, we describe our proposed **MALMM** framework in detail (see Fig. 2).

A. Preliminaries

Our goal is to design a LLM framework capable of solving previously unseen manipulation tasks defined by natural language instructions. We assume access to a robotic environment supporting manipulation actions of closing the gripper, opening the gripper, and changing the gripper's position and orientation around the z axis. Environment observations are obtained as described in Sec. III-D.

B. MALMM: Multi-Agent Manipulation

The core motivation of our work is to investigate how a multi-agent framework can leverage the collaborative and role-specific reasoning capabilities of multiple LLM agents to complete complex manipulation tasks. Below, we first introduce a single-agent architecture and then we propose its multi-agent extensions. The single and multi-agent architectures considered in our work are illustrated in Fig. 3.

Single Agent (SA). SA is adopted from [11] by environment-specific (RLBench [23] or real-world) prompt tuning. Prompting LLMs to interpret natural language instructions in the form of executable primitive actions has been shown successful for a variety of tasks, including image understanding [24] and numerical reasoning [25]. Similarly, LLMs can be prompted to interpret embodied tasks, e.g., *open a wine bottle* and convert them into a sequence of primitive manipulation actions. A version of such a system with a single LLM agent is outlined in Fig. 3(a). Here, an LLM is first prompted to break down the language instruction into a sequence of steps. It then uses its code generation capabilities to translate each step into executable Python code, using predefined functions to control the end-effector. This code is then sent to a Python interpreter that executes the steps in the environment. After each step, the LLM receives new observations from the environment and proceeds in a loop with planning and code generation until meeting termination criteria.

Multi-Agent (MA). A Single Agent performing multiple roles struggles to excel in all of them. To address this issue, we propose two specialized LLM agents with shorter role-specific contexts: the *Planner* and the *Coder*, see Fig. 3(b). The *Planner* breaks down the language instructions into a sequence of manipulation steps while the *Coder* iteratively translates these steps into an executable Python code. After each intermediate step, the *Planner* detects potential failures

and re-plans according to new observations of the environment.

Multi-Agent with a Supervisor. Our final multi-agent architecture, MALMM extends MA with a *Supervisor* agent that coordinates the *Planner*, the *Coder*, and the *Code Executor*, as shown in Fig. 3(c). The *Supervisor* decides which agent or tool to activate next based on the input instructions, the roles of the individual agents, the environment observations, and the entire chat history of the active agents.

C. Multi-Agent Prompting

Each agent is provided with a task-agnostic system prompt. The agents rely solely on their internal world knowledge for reasoning and decision-making. For prompt construction, we draw inspiration from [11] and its study of LLM-based trajectory generation for robotic manipulation. We adapt the prompt according to our environments (RLBench and real-world). Note that unlike other recent work [15], [16], we do not provide the agents with any examples for in-context learning and apply MALMM to new tasks without any changes, i.e., in a zero-shot mode.

Each agent’s prompt is specifically designed to suit its role. Since LLMs require step-by-step reasoning to solve tasks, the *Planner* is prompted to generate steps that define the intermediate goals needed to complete the task. MALMM perceives the environment observations directly from the simulator or by analyzing the RGBD (see Sec. III-D). Therefore, the *Planner* is given a detailed description of the environment’s coordinate system, enabling it to interpret the directions from the gripper’s perspective. Given the limited exposure of LLMs to grounded physical interaction data in their training, LLM agents often fail to account for potential collisions. To address this, we include generic collision-avoidance rules in the *Planner* prompts. Moreover, to handle intermediate failures, primarily due to collisions or missed grasps, we prompt the *Planner* to evaluate action based on the previous and current environment observations after each intermediate step, and to replan if necessary. The prompt used for the *Planner* agent is shown in Figure 10.

The prompt for the *Coder*, as shown in Figure 11, includes information about the expected input and output for all available functions—*execute_trajectory()*, *open_gripper()*, *close_gripper()*, and *check_task_completion()*—as well as guidelines to avoid syntactic and semantic errors, which are common in code generated by LLMs. Finally, the *Supervisor* agent, as presented in Figure 9, is prompted to manage the workflow, coordinating the transitions between the LLM agents, the *Planner* the *Coder*, and the *Code Executor* to ensure successful task completion.

D. Environment Observations

LLMs trained on textual inputs cannot directly perceive or interpret 3D environments. Our agents receive information about the environment either from the internal state space of a simulator or from RGBD.

State-space observations. In this setup, the LLM agents have direct access to the simulator’s state information. The

observations are provided as 3D bounding boxes (object dimensions, center position, and orientation), along with object colors and the gripper’s position, orientation, and open/closed state. The execution logs for *empty container* task presented in Appendix-E show input observation format as CURRENT ENVIRONMENT STATE for MALMM. The same format is also used for Single Agent and Multi-Agent presented in Section III-B.

Visual observations. To apply MALMM in real-world settings, we restrict observations to the front-facing RGB-D sensor (as illustrated in Fig. 4), from which we obtain both RGB images and 3D point clouds. We then use pre-trained foundation models to extract information about scene objects. To this end, we employ *gpt-4-turbo* to derive a list of objects relevant to the instruction text and RGB image, and subsequently utilize LangSAM [26] to produce segmentation masks for these objects, such as *block* or *red jar*. We then segment the 3D object point clouds by projecting the 2D segmentation masks into the 3D space. To compute accurate object-centric grasping poses, we apply the M2T2 [27] model and predict grasps given 3D point clouds of source objects. We use the obtained gripper poses to control the gripper during grasping. To facilitate object placement, we estimate the 3D bounding box of the target object in the environment. For example, in the task *close the red jar*, the target object would be the *red jar*. We extract the 3D bounding box directly from the object’s point cloud, and use it to guide the placement process. We leverage these visual observations while conducting experiments in both simulated and real-world settings. Please refer to Appendix-D for details on obtaining visual observations.

IV. EXPERIMENTS

We evaluate the accuracy of MALMM in a zero-shot settings, i.e. when solving diverse set of previously unseen tasks defined by a short text description.

A. Implementation Details

We use *gpt-4-turbo*¹ [28] to drive the LLM agents in all our experiments. Additionally, we report the results of MALMM using LLaMA-3.3-70B [29] to demonstrate the performance of our framework with an open-source LLM model. For developing the multi-agent framework, we used AutoGen [30], which is an open-source programming library for building AI agents and facilitates collaboration between multiple agents to solve complex tasks. To perform zero-shot evaluation, we do not fine-tune our agents, and we use no training data for in-context learning. We initially developed our prompts for the *Stack Blocks* task and used them for other tasks without any task-specific tuning. MALMM generates 3D waypoints, while the trajectories are computed and executed using a motion planner, following the approach commonly used in RLBench. Our code, prompts, and additional results are available from the project webpage [31].

¹The experiments were conducted in June 2024 using the gpt-4-turbo-2024-04-09 snapshot.

TABLE I: Success rate for zero-shot evaluation on RLBench [23]: The table highlights the best-performing method for each task in **bold** and the second-best-performing method is underlined. **Symbol:** \dagger denotes LLaMA-3.3-70B as a base model. All other methods use GPT-4-Turbo as a base model.

Methods	Basketball in Hoop	Close Jar	Empty Container	Insert in Peg	Meat off Grill	Open Bottle	Put Block	Rubbish in Bin	Stack Blocks	Avg
CAP [15]	0.00	0.00	0.00	0.08	0.00	0.00	0.76	0.00	0.00	0.09
VoxPoser [16]	0.20	0.00	0.00	0.00	0.00	0.00	0.36	<u>0.64</u>	<u>0.32</u>	0.17
Single Agent (SA) [11]	0.52	0.40	0.36	<u>0.24</u>	0.44	0.80	<u>0.92</u>	0.48	0.20	0.50
MALMM (\dagger)	0.84	0.88	0.60	0.80	0.64	0.84	0.84	0.56	0.32	0.70
MALMM	0.88	0.84	0.64	0.68	0.92	0.96	1.00	0.80	0.56	0.81

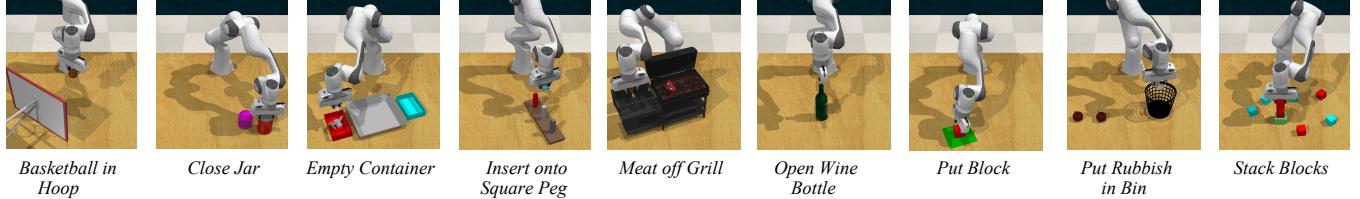


Fig. 4: Illustration of the **nine** RLBench [23] tasks used in our evaluation, featuring diverse tasks with varying task horizons and different object shapes.

B. Environment and Tasks.

We conduct the simulation in CoppeliaSim, interfaced via PyRep, using Franka Panda robot with a parallel gripper. The setup incorporates RLBench [23], a robot learning benchmark that provides various language-conditioned tasks with specified success criteria. For evaluation, we sampled 9 RLBench tasks with 25 samples per task, featuring diverse object poses, object shapes, and task horizons. Fig. 4 shows snapshots of the nine considered tasks.

In the real-world setup, we evaluated five tasks on a tabletop using a 7-DOF Franka Emika Panda Research 3 robot equipped with a parallel jaw gripper. Three of these tasks are identical to those that use vision-based observations from the simulator, while two are new. We use an Intel RealSense D435i RGB-D camera to capture the frontal view and the panda-py [32] library to control the robot arm.

C. Baselines.

We compare our approach to three state-of-the-art zero-shot LLM-based manipulation methods: Code as Policies (CAP) [15], VoxPoser [16], and SA [11]. For CAP², we adapt its official implementation for use with RLBench. We used the official implementation of VoxPoser³ without any modifications. Our SA baseline is a version of “Language Models as Zero-Shot Trajectory Generators” [11] with environment-specific (RLBench or real-world) prompt tuning. Additionally, for fair comparison between the SA and MALMM, the prompts we provided to both of them were equivalent. The only difference is that in the case of MALMM, the information and the instructions in the Single-Agent prompt are distributed between the *Supervisor*, the *Planner*, and the *Coder* according to their respective roles. All the baselines use *gpt-4-turbo* as an LLM. The prompts

for Single Agent and MALMM are provided in Appendix-A.1 and Appendix-A.2 respectively.

D. Results for Zero-Shot Evaluation

Table I presents the results for the three baseline methods, along with our proposed MALMM, across 9 different tasks. From the table, we observe that MALMM outperforms all baselines across all 9 tasks, including long-horizon tasks such as *stack blocks* and *empty container*, as well as tasks involving complex shapes, such as *meat off grill* and *rubbish in bin*. Moreover, the Code as Policies is able to generate a successful trajectory for only two tasks. This limited success is because the original Code as Policies implementation relied on few-shot examples to perform well on tasks involving regularly shaped objects. In our evaluation, we replaced these few-shot examples with coordinate definitions and detailed instructions about the functions available for the LLM to call. However, Code as Policies completely failed in this zero-shot setting, which was also reported in [11].

VoxPoser [16], which generates 3D voxel maps for value functions to predict waypoints, successfully generated trajectories for three tasks with good accuracy. For two of these tasks, it was the second-best performing method. However, its performance did not generalize well to other tasks.

Both the SA and our proposed multi-agent framework, MALMM, successfully generated trajectories for all 9 tasks. However, MALMM consistently outperformed the SA approach by using different agents for specific roles, enabling it to generate accurate high-level plans and low-level code while mitigating hallucinations. In addition to the *gpt-4-turbo* experiments, we evaluated MALMM using the open-source LLaMA-3.3-70B [29]; the results are presented in Table I. Although there is a drop in performance compared to *gpt-4-turbo*, MALMM, with LLaMA-3.3-70B, outperforms the existing baselines by a sizable margin. We provide qualitative comparison between Single Agent and MALMM for the

²<https://code-as-policies.github.io/>

³<https://voxposer.github.io/>

TABLE II: Ablation study assessing the impact of different components in MALMM: environment feedback, *Planner* (**P**), *Coder* (**C**), and *Supervisor* (**S**).

Agents	Environment Feedback	P	C	S	Stack Blocks	Empty Container
Single Agent (SA)	x	x	x	x	0.08	0.12
Single Agent (SA)	✓	x	x	x	0.20	0.36
Multi-Agent (MA)	✓	✓	✓	x	0.36	0.48
MALMM	✓	✓	✓	✓	0.56	0.64

Stack Blocks task in Appendix-B.

E. Multi-Agent Ablation

We performed ablation for each of the components (agents) in MALMM in order to evaluate how each of them contributes to the overall performance. We considered two tasks, namely *stack blocks* and *empty container*, and report the results in Table II.

We first analyzed the importance of the intermediate environment feedback. To this end, we considered the Single Agent (SA) setting and removed the environment feedback provided after each intermediate step. In this setup, LLM generates the full manipulation plan at once and executes it without revisions. As shown in Table II, the SA without environment feedback exhibits 12% and 24% drop in performance for the *stack blocks* and *empty container* tasks, respectively. By analyzing the failure cases of both methods, we observed that the environment feedback provided after each intermediate step crucially affected the agent's ability to detect unforeseen situations and recover from failures such as collisions and inaccurate grasping.

We next validated the advantage of the Multi-Agent architecture with separate LLM agents for planning and code generation. As shown in Table II, the MA, consisting of a dedicated *Planner* and *Coder*, demonstrated 16% and 12% performance improvement over SA for the two tasks respectively. This can be attributed to the inherent limitations of LLMs in managing very long context conversations [6]. In the SA setup, where a single LLM is responsible for both the high-level planning and the low-level code generation, the agent must handle an extensive context, particularly for long-horizon tasks. This often leads to errors such as failing to account for collisions with other objects, omitting the input arguments for the predefined functions, using variables before they were initialized, and even forgetting the specified goal. In contrast, the MA system mitigates these issues by dividing the workload among specialized agents. The *Planner* and the *Coder* agents in the MA setting focus on specific roles through specialized prompts and communicate with each other, thus reducing the likelihood of errors and hallucinations, in particular for longer tasks.

Our initial Multi-Agent system pre-defines the cyclic sequence of the *Planner*, the *Coder*, and the *Coder Executor*, see Fig. 3(b), assuming that each agent correctly completes its task. However, hallucinations may occur even within multi-agent systems [33]. For example, the *Coder* may miss the variable initialization resulting in compilation errors or incomplete sub-goal code generation, such as producing

code only for approaching the object without grasping it. In such situations, the *Coder* may need to be re-executed in order to correct possible errors before passing the control to the *Planner*. To automate this process, we introduced a *Supervisor* agent that dynamically re-routes the execution process to the next agent based on the input instruction, the entire communication history of all active agents, and the role descriptions of all agents rather than following a fixed sequence. This adaptive approach is at the core of our MALMM framework, and it improves the performance of the dual-agent MA setup by 20% in the '*stack blocks*' task and 16% in the '*empty container*' task, respectively, as shown in Table II.

F. MALMM is Better at Long-Horizon Planning

To validate the effectiveness of MALMM in long-horizon tasks, we created three variations of the '*stack blocks*' task, each with a different number of blocks, and compared the performance of MALMM to the Single Agent setup. The results in Fig. 5 indicate that while the Single Agent setup struggles with stacking 3 and 4 blocks, MALMM substantially outperforms SA, in particular for tasks that require longer planning. Fig. 1 illustrates an example of failure recovery with key steps by MALMM compared to SA on the stack block task.

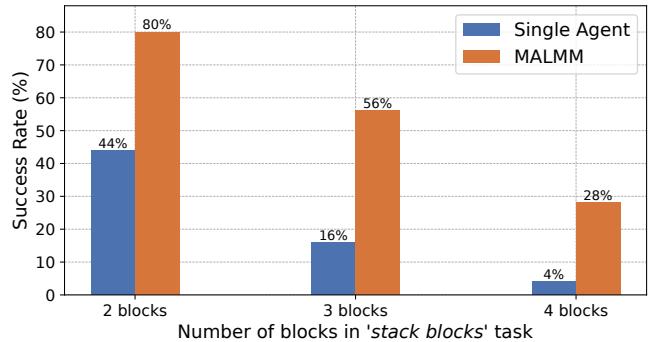


Fig. 5: Comparison of **Single Agent** vs. **MALMM** for variations of the *stack blocks* task that require stacking 2, 3, or 4 blocks on top of each other.

G. Results for Vision-Based Observations

In order to approach real-world settings where direct access to the environment states is not available, we next perform experiments in simulation restricted to vision-based observations in the form of 3D point clouds. We evaluate the performance of MALMM and compare it to the Single Agent setup on three tasks: (ii) *put block*, (ii) *rubbish in bin*, and (iii) *close jar*. Consistently with our previous state-based experiments, the results in MALMM in Table III show sizable improvements over the Single-Agent baseline across all three tasks and confirm the advantage of our proposed Multi-Agent framework.

By comparing the results in Tables I and III we observe degradation in performance when switching to vision-based observations. This can be attributed to inaccuracies of the



Fig. 6: Illustration of the **five** real-world tasks used in our evaluation.

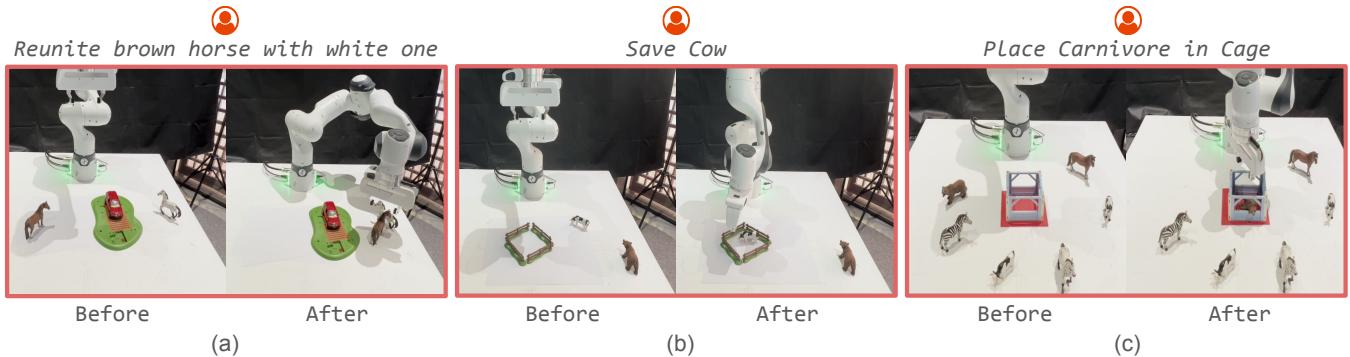


Fig. 7: MALMM performs zero-shot manipulation on three unseen tasks in a real world, each guided by *high-level* user instructions. (a) “*Reunite the brown horse with the white one.*” The environment contains a brown horse, a white horse, a road, and a car; the goal is to place the brown horse near the white horse. (b) “*Save Cow.*” The environment contains a bear, a cow, and a fenced enclosure; the goal is to place the cow inside the enclosure. (c) “*Place Carnivore in Cage.*” The environment contains two horses, two cows, one zebra, a bear, and a cage; the goal is to place the bear inside the cage. In each scenario, the left image shows the initial arrangement and the right image shows the final arrangement after MALMM completes the instructed task.

vision-based estimators such as 3D bounding box detection and grasp estimation. Note that our current vision pipeline makes use of single-view scene observations. A parallel work, Manipulate Anything [34] showed that a relatively straightforward extension to multi-view settings can reduce the impact of occlusions and yield higher accuracy.

TABLE III: Comparison of the Single Agent and MALMM in a simulated environment with vision-based observations.

Agents	Close Jar	Put Block	Rubbish in Bin
Single Agent [11]	0.24	0.68	0.40
MALMM	0.56	0.84	0.52

H. Results for the Real-World Experiments

For the real-world robotics setup, we evaluated five tasks as shown in Fig. 6: *close jar* (put lid on top of jar), *put block* (place a block in the red target area), *put rubbish in bin* (place rubbish in the bin), *put case* (place an earbuds case in the red target area), and *put jar in bin* (place a jar in the bin) – in ten different initial states, each with both MALMM and the Single Agent. As shown in Table IV, consistently with our simulation results, MALMM outperforms the Single Agent on all five tasks by a sizable margin. It achieved a 40% higher success rate for both *put case* and *rubbish in bin*, 30% higher success rate for *put block*, and 20% higher success rate for

close jar and *jar in bin*. To further demonstrate zero-shot capabilities of our method, Fig. 7 demonstrates successful performance of MALMM for three new tasks, each defined by *high-level* user instruction.

TABLE IV: Comparison of Single Agent and MALMM in a real-world Franka robot arm environment.

Agents	Close Jar	Put Block	Rubbish in Bin	Put Case	Jar in Bin
Single Agent [11]	2/10	3/10	2/10	3/10	3/10
MALMM	4/10	6/10	6/10	7/10	5/10

V. DISCUSSION

A. Limitations

Despite its string advantages over SA, MALMM has several limitations. First, MALMM relies on three *gpt-4-turbo* agents, making it costly to operate. Using open-source LLMs is possible at the cost of reduced accuracy (cf. Table I). Second, like other LLM-based planners, MALMM depends on manual prompt engineering, which impacts its performance. However, advancements in prompting [5] can reduce these efforts. Finally, MALMM requires accurate bounding box estimation to determine the correct grasp positions, but as the complexity of the objects increases, the 3D bounding boxes alone may not provide enough information for precise

grasping. Our experiments with a vision pipeline in simulation and in a real-world scenario suggest that using pretrained grasping and placement models, such as M2T2 [27], could improve the performance for complex manipulation tasks. Even though MALMM performs over $6\times$ better than the SA on the long-horizon task (stacking 4 blocks), it achieves only a 28% success rate, revealing limitations in handling rich object interactions. Additionally, the vision-based pipeline is not robust to severe occlusions due to its single-view limitation.

B. Conclusion and Future Work

We explored the use of LLM agents for solving previously unseen manipulation tasks. In particular, we proposed the first multi-agent LLM framework for robotics manipulation MALMM and demonstrated its advantages over single-agent baselines. Our method uses task-agnostic prompts and requires no in-context learning examples for solving new tasks. Extensive evaluations, both in simulation and real-world settings, demonstrated excellent results for MALMM for a variety of manipulation tasks. In future work, we will explore richer object interactions and more complex tasks, including articulated objects. We also aim to use vision-language foundation models to incorporate contextual scene details beyond object bounding boxes.

REFERENCES

- [1] A. Goyal, J. Xu, Y. Guo, V. Blukis, Y.-W. Chao, and D. Fox, “Rvt: Robotic view transformer for 3d object manipulation,” *arXiv preprint arXiv:2306.14896*, 2023.
- [2] T. Gervet, Z. Xian, N. Gkanatsios, and K. Fragkiadaki, “Act3d: 3d feature field transformers for multi-task robotic manipulation,” in *Conference on Robot Learning*. PMLR, 2023, pp. 3949–3965.
- [3] J. Gao, B. Sarkar, F. Xia, T. Xiao, J. Wu, B. Ichter, A. Majumdar, and D. Sadigh, “Physically grounded vision-language models for robotic manipulation,” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 12462–12469.
- [4] E. Davis, “Mathematics, word problems, common sense, and artificial intelligence,” *Bulletin of the American Mathematical Society*, vol. 61, no. 2, pp. 287–303, 2024.
- [5] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. hsin Chi, F. Xia, Q. Le, and D. Zhou, “Chain of thought prompting elicits reasoning in large language models,” *ArXiv*, vol. abs/2201.11903, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:246411621>
- [6] A. Maharana, D.-H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fang, “Evaluating very long-term conversational memory of llm agents,” *ArXiv*, vol. abs/2402.17753, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:268041615>
- [7] J. Thomason, S. Zhang, R. J. Mooney, and P. Stone, “Learning to interpret natural language commands through human-robot dialog,” in *International Joint Conference on Artificial Intelligence*, 2015. [Online]. Available: <https://api.semanticscholar.org/CorpusID:6745034>
- [8] E. Jang, A. Irpan, M. Khansari, D. Kappler, F. Ebert, C. Lynch, S. Levine, and C. Finn, “Bc-z: Zero-shot task generalization with robotic imitation learning,” *ArXiv*, vol. abs/2202.02005, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:237257594>
- [9] T. Kollar, S. Tellex, D. K. Roy, and N. Roy, “Grounding verbs of motion in natural language commands to robots,” in *International Symposium on Experimental Robotics*, 2010. [Online]. Available: <https://api.semanticscholar.org/CorpusID:5821491>
- [10] Y. Wang, Z. Sun, J. Zhang, Z. Xian, E. Biyik, D. Held, and Z. M. Erickson, “Rl-vlm-f: Reinforcement learning from vision language foundation model feedback,” *ArXiv*, vol. abs/2402.03681, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:267499679>
- [11] T. Kwon, N. D. Palo, and E. Johns, “Language models as zero-shot trajectory generators,” *IEEE Robotics and Automation Letters*, vol. 9, pp. 6728–6735, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:264289016>
- [12] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. R. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter, “Inner monologue: Embodied reasoning through planning with language models,” in *Conference on Robot Learning*, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:250451569>
- [13] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, “Llm+ p: Empowering large language models with optimal planning proficiency,” *arXiv preprint arXiv:2304.11477*, 2023.
- [14] Z. Liu, A. Bahety, and S. Song, “Reflect: Summarizing robot experiences for failure explanation and correction,” *ArXiv*, vol. abs/2306.15724, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:259274760>
- [15] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. R. Florence, and A. Zeng, “Code as policies: Language model programs for embodied control,” *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:252355542>
- [16] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei, “Voxposer: Composable 3d value maps for robotic manipulation with language models,” *ArXiv*, vol. abs/2307.05973, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:259837330>
- [17] W. Yu, N. Gileadi, C. Fu, S. Kirmani, K.-H. Lee, M. G. Arenas, H.-T. L. Chiang, T. Erez, L. Hasenlever, J. Humplík, B. Ichter, T. Xiao, P. Xu, A. Zeng, T. Zhang, N. M. O. Heess, D. Sadigh, J. Tan, Y. Tassa, and F. Xia, “Language to rewards for robotic skill synthesis,” *ArXiv*, vol. abs/2306.08647, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:259164906>
- [18] S. Bubeck, V. Chandrasekaran, R. Eldan, J. A. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y.-F. Li, S. M. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, “Sparks of artificial general intelligence: Early experiments with gpt-4,” *ArXiv*, vol. abs/2303.12712, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:257663729>
- [19] Z. Mandi, S. Jain, and S. Song, “Roco: Dialectic multi-robot collaboration with large language models,” *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 286–299, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:259501567>
- [20] I. Dasgupta, C. Kaeser-Chen, K. Marino, A. Ahuja, S. Babayan, F. Hill, and R. Fergus, “Collaborating with language models for embodied reasoning,” *ArXiv*, vol. abs/2302.00763, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:253180684>
- [21] C. Huang, O. Mees, A. Zeng, and W. Burgard, “Visual language maps for robot navigation,” *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10608–10615, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:252846548>
- [22] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, “Language models as zero-shot planners: Extracting actionable knowledge for embodied agents,” *ArXiv*, vol. abs/2201.07207, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:246035276>
- [23] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, “Rlbench: The robot learning benchmark & learning environment,” *IEEE Robotics and Automation Letters*, vol. 5, pp. 3019–3026, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:202889132>
- [24] D. Surís, S. Menon, and C. Vandrick, “Vipergpt: Visual inference via python execution for reasoning,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 11888–11898.
- [25] W. Chen, X. Ma, X. Wang, and W. W. Cohen, “Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks,” *arXiv preprint arXiv:2211.12588*, 2022.
- [26] L. Medeiros, “Language segment-anything,” <https://github.com/luca-medeiros/lang-segment-anything>, 2024, accessed: 2024-09-12.
- [27] W. Yuan, A. Murali, A. Mousavian, and D. Fox, “M2t2: Multi-task masked transformer for object-centric pick and place,” in *Conference on Robot Learning*, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:263629874>
- [28] OpenAI, “GPT-4 technical report,” 2023.
- [29] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, and A. Y. et al, “The llama 3 herd

- of models,” *ArXiv*, vol. abs/2407.21783, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:271571434>
- [30] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, “Autogen: Enabling next-gen llm applications via multi-agent conversation,” 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:263611068>
- [31] “MALMM project webpage,” <https://malmm1.github.io/>.
- [32] J. Elsner, “Taming the panda with python: A powerful duo for seamless robotics programming and integration,” *SoftwareX*, vol. 24, p. 101532, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2352711023002285>
- [33] S. Banerjee, A. Agarwal, and S. Singla, “Llms will always hallucinate, and we need to live with this,” 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:272524830>
- [34] J. Duan, W. Yuan, W. Pumacay, Y. R. Wang, K. Ehsani, D. Fox, and R. Krishna, “Manipulate-anything: Automating real-world robots using vision-language models,” *ArXiv*, vol. abs/2406.18915, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:270764457>
- [35] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *kdd*, vol. 96, no. 34, 1996, pp. 226–231.

APPENDIX

This appendix provides additional details for the MALMM approach. We present complete prompts in Appendix-A, qualitative results in Appendix-B, RLbench tasks in Appendix-C, details on the visual observation pipeline in Appendix-D and an example of a full execution log in Appendix-E.

A. Prompts

1) Single Agent Prompts: The prompt for the Single Agent baseline is shown in Figure 8. There are four variables in this prompt: *[INSERT TASK]* for task instruction, *[INSERT EE POSITION]* for end effector initial position, *[INSERT EE ORIENTATION]* for end effector initial orientation and *[STATE]* for environment state or observation.

Single Agent Prompt

You are a sentient AI specialized in generating a sequence of steps and Python code for the robot arm end-effector to complete a given task. The end effector is in a environment and informations about the objects in the environment including the end-effector are provided in terms of their positions and orientations. You must remember that this conversation is a monologue, and that you are in control. I am not able to assist you with any questions, and you must output the plan and code yourself by making use of the common sense, general knowledge, and available information.

PLANNER:

ENVIRONMENT SET-UP:

- The 3D coordinate system of the environment is as follows:
 1. The x-axis is in the horizontal direction, increasing to the right.
 2. The y-axis is in the depth direction, increasing away from you.
 3. The z-axis is in the vertical direction, increasing upwards.
 4. Workspace is in the table positioned at a z-level of 0.75199986. The workspace ranges from x: -0.075 to 0.575 and y: -0.455 to 0.455.

CONSTRAINTS:

1. **Orientation of the end effector** will always be **z orientation of object to grasp or destination to place**.
 2. Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise rotation. The rotation values should be in radians.
 3. The <safe distance> in the z direction is 0.1 units.
 4. The <release distance> in the z direction is 0.02 units above the top surface of the destination area or object, please estimate this distance.
 5. **Remember you can only grasp the object from its *CENTER*. Not from any other position. So to grasp the object, end effector has to be lowered down to center of object**.

<CURRENT ENVIRONMENT STATE>:

This variable describes the current state of the environment in term of position, orientation and dimensions of all the objects in the environment. This information will be printed after every code execution, you have to replan next steps based on this information. The robot is required to assess the current position, orientation, and dimensions of all objects in the environment (If they are moved or not to desired place). Replan the next steps of the plans based on the current environment state. Check if the previous step was successful based on previous target object position/orientation and most recent target object position/orientation. If the previous step was not successful, replan the next steps based on the current environment state.

COLLISION AVOIDANCE:

- If there are multiple objects in the environment:
 1. Make sure to consider the widths, lengths, and heights of other objects so that robot arm end effector does not collide with other objects or table.
 2. This information may help to generate additional trajectories and add specific waypoints (calculated from the given objects' information) to avoid collision with other objects and the table.

COLLISION FREE OBJECT INTERACTION RULES:

1. Position the gripper <safe distance> above the target object.
2. Move to the **CENTER** of the target object. If *CENTER* position is (x,y,z) then grasping position will also be (x,y,z) Do not add any height margin.
3. Grasp the target object.
4. Raise the gripper <safe distance> above the target object.
5. Move to <safe distance> above the destination area.
6. Lower the gripper to the destination area.
7. Release the object (drop) at <release distance> (**(0.02 units)**) above the top surface of destination area.
8. Raise the gripper <safe distance> above the destination area.

PLANNING:

1. Describe the relative positions of all objects in the environment, including their spatial relationships, alignments, and groupings.
2. Provide a detailed, step-by-step plan for the given task.
3. Generate the code one step at a time. After four step is executed by the code executor, give feedback about success of the step based on the relative positions and orientations of the objects.

CODE GENERATOR:

AVAILABLE FUNCTIONS:

You are able to call any of the following Python functions, if required, as often as you want:
 1. `execute_trajectory(position: list[float], orientation: float) -> None`: This function will execute the trajectory on the robot arm end-effector based on position and orientation, and will also not return anything. It takes list position of 3 elements and one float orientation value as input.
 2. `open_gripper() -> None`: This function will open the gripper on the robot arm, and will also not return anything.
 3. `close_gripper(object_name: str) -> None`: This function will close the gripper on the robot arm, and will also not return anything. It takes the name of the object as input.
 4. `task_completed() -> None`: Call this function only when the task has been completed. This function will also not return anything. If there is **any error in the code or planning consecutively for five times**, **then also call this function**.
 When calling any of the functions, make sure to stop generation after each function call and wait for it to be executed, before calling another function and continuing with your plans.

CODE GENERATION:

When generating the code for the trajectory, do the following:
 1. When mentioning the functions, specify the required parameters and clearly define them in the same code block before passing it to code executor. For `execute_trajectory`, define the position and orientation lists prior to it and mention object name in `close_gripper(object_name)` from <CURRENT ENVIRONMENT STATE>.
 2. Orientation parameter will always be z orientation of object to grasp or destination to be place.
 3. *Do not generate the code all in one go for all the steps; instead, generate it step by step*. After generating, provide this step-level code to the code executor and wait for the reply. If the generated code is incorrect and the code executor encounters an error during execution, *correct it*.
 4. Mark any code clearly with the ```python and ``` tags.

*Remember you can only grasp the object from its **CENTER**. Not from any other position. So to grasp the object, the end effector has to be lowered down to the center of the object*.

Use the robot arm end effector to "[INSERT TASK]" in the environment.
 The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the orientation [INSERT EE ORIENTATION], and the gripper is open.
 The positions and orientations of all objects in the environment as follows:
 <CURRENT ENVIRONMENT STATE>:
 "[INSERT CURRENT STATE ENVIRONMENT]"

Fig. 8: Prompt for Single Agent.

2) *MALMM Prompts*: The Supervisor prompt of MALMM, shown in Figure 9, contains the same four variables as the Single Agent prompt described in Appendix-A.1: *[INSERT TASK]*, *[INSERT EE POSITION]*, *[INSERT EE ORIENTATION]*, and *[STATE]*. Additionally, the prompts for Planner and Coder in MALMM are depicted in Figures 10 and 11 respectively.

Supervisor Prompt

You are the Multi-Agent Supervisor, responsible for overseeing the collaboration between two agents: the Planner and the Coder. Below are the role descriptions for both agents, along with the task instructions and the initial state information of the robot's end-effector and all objects in the environment:

Planner Role Description:
The Planner's task is to determine the steps needed to complete the given task. After each step is executed, the Planner evaluates the progress and provides the plan for the remaining steps.

Coder Role Description:
The Coder is responsible for writing code based on the plan provided by the Planner.

Your task is to use the robot arm's end-effector to perform the following task: "*[INSERT TASK]*" in the environment.

The robot arm's end-effector is currently positioned at *[INSERT EE POSITION]*, with an orientation of *[INSERT EE ORIENTATION]*, and the gripper is open.

The position, orientation, height, weight, length, and color of all objects in the environment are as follows:
Current Environment State:
[STATE]

Fig. 9: Prompt for Supervisor of MALMM

Planner Prompt

You are a sentient AI specialized in generating a sequence of steps for the robot arm end-effector to complete a given task. The end effector is in a environment and information about the objects in the environment including the end-effector are provided in terms of their positions and orientations. You must remember that this conversation is a monologue, and that you are in control. I am not able to assist you with any questions. You must output the plan yourself by making use of the available information, common sense, and general knowledge.

ENVIRONMENT SET-UP:

The 3D coordinate system of the environment is as follows:

1. The x-axis is in the horizontal direction, increasing to the right.
2. The y-axis is in the depth direction, increasing away from you.
3. The z-axis is in the vertical direction, increasing upwards.
4. Workspace is in the table positioned, at a z-level of 0.75199986. The workspace ranges from x: -0.075 to 0.575 and y: -0.455 to 0.455.

The robot arm is in a top-down set-up, with the end-effector facing down onto a tabletop. The end-effector is therefore able to only rotate about the z-axis, from -pi to pi radians. The end-effector gripper has two fingers, and they are currently parallel to the x-axis.

CONSTRAINTS:

1. **Orientation of the end effector** will always be **z orientation of object to grasp or destination to place**.
2. Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise rotation. The rotation values should be in radians.
3. The <safe distance> in the z direction is 0.1 units.
4. The <release distance> in the z direction is 0.02 units above the top surface of the destination area or object, please estimate this distance.
5. **Remember you can only grasp the object from its *CENTER*. Not from any other position. So to grasp the object, end effector has to be lowered to center of object**

<CURRENT ENVIRONMENT STATE>:

This variable describes the current state of the environment in terms of position, orientation and dimensions of all the objects in the environment. This information will be printed after every code execution, you have to replan next steps based on this information. The robot is required to asses the current position, orientation, and dimensions of all objects in the environment (If they are moved or not to desired place). Replan the next steps of the plans based on the current environment state. Check if the previous step was successful based on previous target object's position/orientation and most recent target object position/orientation. If the previous step was not successful, replan the next steps based on the current environment state.

COLLISION AVOIDANCE:

If there are multiple objects in the environment:

1. Make sure to consider the widths, lengths, and heights of other objects so that robot arm end effector does not collide with other objects or table.
2. This information may help to generate additional trajectories and add specific waypoints (calculated from the given objects' information) to avoid collision with other objects and the table.

COLLISION-FREE OBJECT INTERACTION RULES:

1. Position the gripper <safe distance> above the target object.
2. Move to the *CENTER* of the target object. If *CENTER* position is (x,y,z) then grasping position will also be (x,y,z) Do not add any height margin.
3. Grasp the target object.
4. Raise the gripper <safe distance> above the target object.
5. Move to <safe distance> above the destination area.
6. Lower the gripper to the destination area.
7. Release the object (drop) at <release distance> (**(0.02 units)**) above the top surface of destination area.
8. Raise the gripper <safe distance> above the destination area.

PLANNING:

1. Describe the relative positions of all objects in the environment, including their spatial relationships, alignments, and groupings.
2. Provide a detailed, step-by-step plan for the given task.
3. Instruct the code generator to produce the code one step at a time. After four step is executed by the code executor, give feedback about success of the step based on the relative positions and orientations of the objects.

Fig. 10: Prompt for Planner of MALMM

Coder Prompt

You are a sentient AI specializing in generating Python code to control a robot arm's end-effector.

AVAILABLE FUNCTIONS:

You must remember that this conversation is a monologue, and that you are in control. I am not able to assist you with any questions, and you must output the final code yourself by making use of the available information, common sense, and general knowledge. You are, however, able to call any of the following Python functions, if required, as often as you want:

1. `execute_trajectory(position: list[float], orientation: float) -> None`: This function will execute the trajectory on the robot arm end-effector based on position and orientation, and will also not return anything. It takes list position of 3 elements and one float orientation value as input.
2. `open_gripper() -> None`: This function will open the gripper on the robot arm, and will also not return anything.
3. `close_gripper(object_name: str) -> None`: This function will close the gripper on the robot arm, and will not return anything. It takes the name of the object as input.
4. `task_completed() -> None`: Call this function only when the task has been completed. This function will also not return anything. If there is **any error in the code or planning consecutively for five times**, **then also call this function**.

When calling any of the functions, make sure to stop generation after each function call and wait for it to be executed before calling another function and continuing with your plans.

CODE GENERATION:

When generating the code for the trajectory, do the following:

1. When mentioning the functions, specify the required parameters and clearly define them in the same code block before passing it to code executor. For `execute_trajectory`, define the position and orientation lists prior to it and mention object name in `close_gripper(object_name)` from <CURRENT ENVIRONMENT STATE>.
2. Orientation parameter will always be z orientation of object to grasp or destination to be placed.
3. *Do not generate the code all in one go for all the steps; instead, generate it step by step*. After generating, provide this step-level code to the code executor and wait for the reply. *Pass the response to the planner*. If the generated code is incorrect and the code executor encounters an error during execution, *correct it and then submit it to the code executor again*.
4. Mark any code clearly with the ````python` and ````` tags.

*Remember you can only grasp the object from its **CENTER**. Not from any other position. So to grasp the object, the end effector has to be lowered to the center of the object*.

Fig. 11: Prompt for Coder of MALMM

3) *Multi-Agent Prompts*: The Multi-Agent baseline consists of two agents: the Planner and the Coder. The prompts for each agent are the same as those used by MALMM, as shown in Figure 10 for the Planner and in Figure 11 for the Coder.

B. Single Agent vs MALMM Qualitative results

Figure 12 shows qualitative result comparing Single Agent and MALMM. The sequence of frames on the top row refers to Single Agent, while the bottom row refers to MALMM. In the case of Single Agent, two blocks were initially stacked correctly but while approaching the third block, the gripper collided with the stacked blocks causing the second block to fall. This subsequently led to hallucinations and ultimately stacking the blocks away from the target area. In contrast, although MALMM dropped the third block while attempting to place it on top of the stacked blocks on the target area, it was able to successfully detect and rectify the failure, ultimately completing the task by stacking all the blocks on the target area.

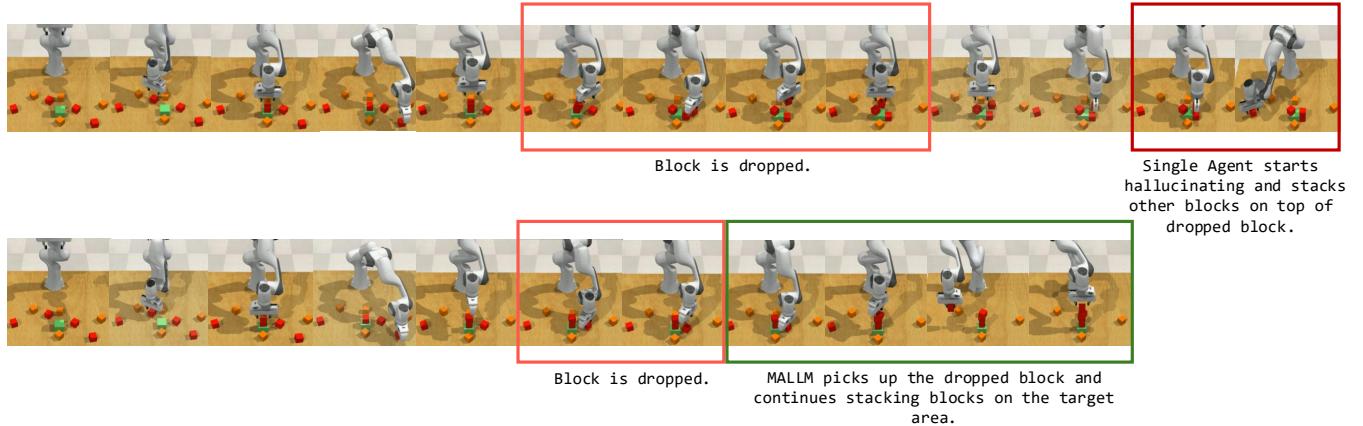


Fig. 12: Qualitative results comparing the Single Agent approach and MALMM are presented. The figure illustrates examples of the task "Stack four blocks at the green target area" performed by the Single Agent LLM (top) and our MALMM approach (bottom). While both approaches experience failures, MALMM demonstrates recovery by resuming the stacking process after dropping a block and continuing above the target area. In contrast, the Single Agent mistakenly stacks additional blocks on top of the dropped block.

C. RLBench Tasks

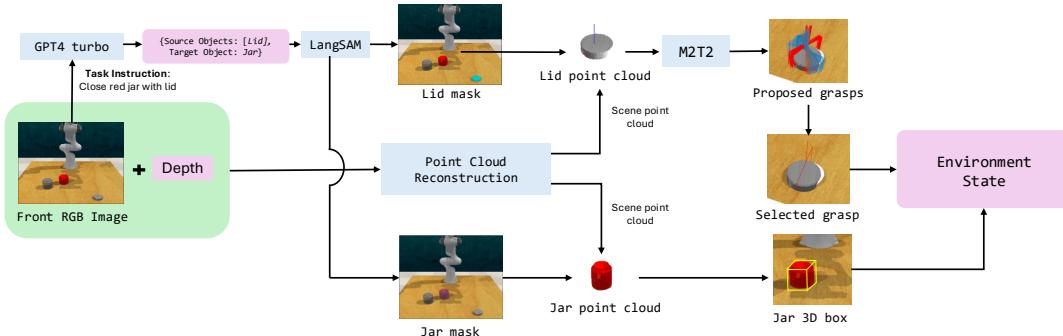
We evaluated on nine RLBench tasks which are listed in Table V along with the task instruction and success criteria.

TABLE V: Details of the RLBench tasks used for evaluation.

Task Instruction	Details
<i>Basketball In Hoop</i>	<i>Task Description:</i> Put basketball in hoop. <i>Success Criteria:</i> Basketball passes through hoop.
<i>Close Jar</i>	<i>Task Description:</i> Close the colored jar with a lid. <i>Success Criteria:</i> Lid is on top of the colored jar.
<i>Empty Container</i>	<i>Task Description:</i> Pick all the objects from the large container and put them into the colored container. <i>Success Criteria:</i> All objects from the large container are now in the colored container.
<i>Insert In Peg</i>	<i>Task Description:</i> Insert the square ring into the colored peg. <i>Success Criteria:</i> The square ring is in the colored peg.
<i>Meat Off Grill</i>	<i>Task Description:</i> Pick the meat (chicken or steak) from the grill and place it into the designated area. <i>Success Criteria:</i> Meat is on the designated area.
<i>Open Bottle</i>	<i>Task Description:</i> Remove the cap of the wine bottle. <i>Success Criteria:</i> Cap of the wine bottle is removed.
<i>Put Block</i>	<i>Task Description:</i> Put the block in the target area. <i>Success Criteria:</i> The block is in the target area.
<i>Rubbish In Bin</i>	<i>Task Description:</i> Put the rubbish in the bin. <i>Success Criteria:</i> Rubbish is in the bin.
<i>Stack Blocks</i>	<i>Task Description:</i> Stack a specified number of colored blocks on the target block. <i>Success Criteria:</i> Specified number of blocks are stacked on top of the target block.

D. Obtaining Visual Observations

To obtain the visual observations, we use RGBD frames from the front view. First, we apply *gpt-4-turbo* [28] on the RGB image and task instruction, prompting it to generate appropriate names for the source and target objects, which is helpful for LangSAM to segment. Second, we generate 3D point clouds from the input RGBD frames. Then, we segment the 3D point clouds of the source and target objects by projecting the 2D masks generated by LangSAM [26] into 3D. Next, we employ the M2T2 [27] model to identify potential grasp poses on the source objects. We filter these grasp poses to retain only those with a top-down gripper orientation. Finally, we apply DBSCAN [35] clustering to the numerous grasp poses and select the central pose. For the placement, we use the 3D bounding box of the target object to identify the placement area. Thus, the selected grasp poses of the objects to be moved, along with the 3D bounding box of the target object, are used as the environment state. Figure 13 illustrates the process of selecting the environment state, as described for the *close jar* task with the instruction *close red jar with lid*. In addition, to dynamically update the grasp and placement states, we introduce a history-aware mechanism to mitigate jitter in the extracted visual states (i.e., grasp and placement poses). Specifically, during each execution, if the translation distance is less than 0.01 meters or the change in gripper orientation is below 30 degrees, we retain the visual state from the previous step. This approach helps maintain stability during manipulation.


 Fig. 13: Process of obtaining visual observation, illustrated using the *close jar* task.

E. Execution Logs

This section presents an example execution of our MALMM framework for the *empty container* task. The initial setup is shown after the *Task Instruction*, followed by images of changes captured after successful code execution.

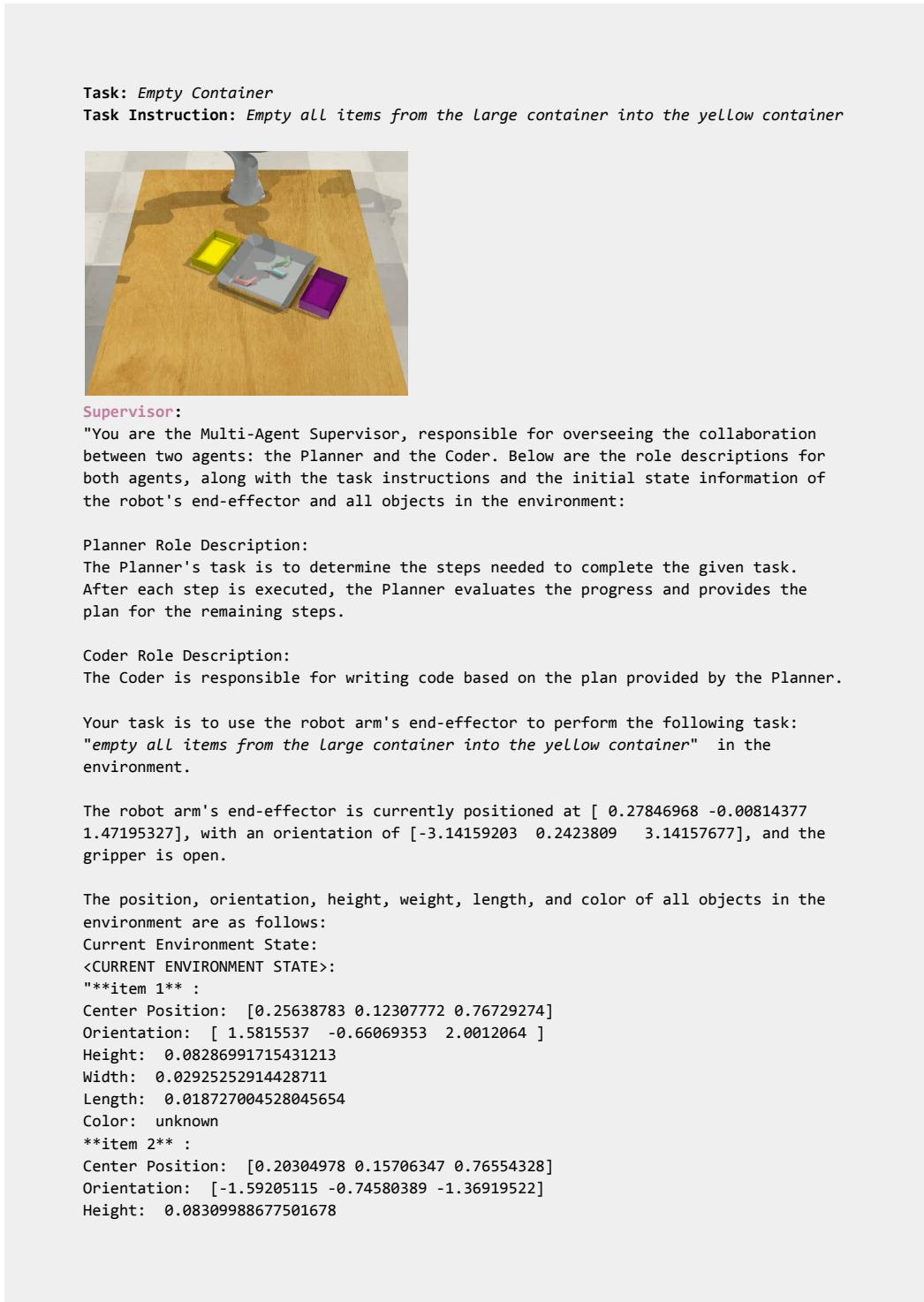


Fig. 14: Execution logs for MALMM method for the task *empty container*.

```

Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.31352305 -0.00940853  0.77087897]
Orientation: [-1.56326413  0.23990303  2.20398927]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
"

```

Next speaker: [Planner](#)

[Planner](#) (to [Supervisor](#)):

Task: Empty the large container and move all items to the yellow container.

To streamline this process, we will execute the operations in a step-by-step manner for each object found in the large container. Here's the plan:

```

### Step 1: Move to Item 1
- **Target**: Item 1
- **Details**: Item 1 is located at [0.25638783, 0.12307772, 0.76729274].

```

```

#### Movement Plan:

```

Fig. 15: Execution logs for MALMM method for the task *empty container*.

```

1. Move to a safe height (+0.1 units in the z-direction from its current height)
above Item 1.
2. Change the orientation of the end-effector to match the z-orientation of Item 1.
3. Lower the end-effector to the center height of Item 1, which is 0.76729274.

### Step 2: Grasp Item 1
- Grasp Item 1 using the gripper.

### Step 3: Move to Yellow Container
- **Target**: Yellow Container
- **Details**: Center Position [0.14428537, -0.13835038, 0.77662492]. 

#### Movement Plan:
1. Lift Item 1 to the safe height (+0.1 units).
2. Move the end-effector with Item 1 to a safe height (+0.1 units in z-direction)
above the center of the Yellow Container.
3. Change the orientation of the end-effector to match the z-orientation of the
Yellow Container.
4. Lower the end-effector to a height just above the top surface of the objects in
the container (release distance of 0.02 units above 0.77662492).

### Step 4: Release Item 1
- Release Item 1 into the Yellow Container.

```

Repeat these steps for Item 2 and Item 3 with appropriate adjustments for their specific dimensions and locations.

Start executing the plan now. Focus on achieving the movement and grasp of Item 1 first, and proceed with the subsequent steps as each phase completes successfully. This ensures systematic progress and allows for adjustments if needed between the steps. Capturing feedback after each main action (grasp, move, release) is crucial to adapt to any changes in the environment or to any errors in execution.

Next speaker: **Coder**

Coder (to **Supervisor**):

Let's start by moving to a position above Item 1 and adjusting the orientation to prepare for grasping.

```

```python
Define the position above Item 1 and its orientation
position_above_item_1 = [0.25638783, 0.12307772, 0.76729274 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_1 = 2.0012064 # z-orientation of Item 1

Execute trajectory to the specified position above Item 1
execute_trajectory(position_above_item_1, orientation_item_1)
```

```

Fig. 16: Execution logs for MALMM method for the task *empty container*.

```
Execution of this code will move the robot arm end-effector above Item 1.
```

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above Item 1 and its orientation
position_above_item_1 = [0.25638783, 0.12307772, 0.76729274 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_1 = 2.0012064 # z-orientation of Item 1

# Execute trajectory to the specified position above Item 1
execute_trajectory(position_above_item_1, orientation_item_1)
{'lang': 'python', 'filename': None, 'use_docker': False}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

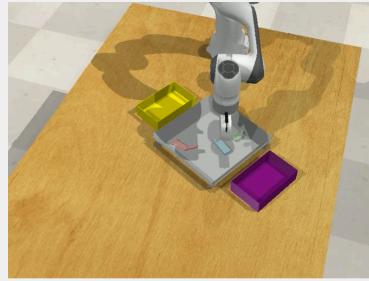
**item 1** :
Center Position: [0.26242024 0.11987246 0.76251084]
Orientation: [ 1.42751968 -0.66114056 1.3509903 ]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [0.20288718 0.15586215 0.76546675]
Orientation: [-1.61480486 -0.86026013 -1.43872547]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30273923 -0.00877987 0.75928092]
Orientation: [-1.69029105 0.06539517 1.47153783]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
```

Fig. 17: Execution logs for MALMM method for the task *empty container*.

```

Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [0.25604063 0.1224228 0.86869109]
No object has been grasped

```



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

The robot arm end-effector has successfully moved to a position above Item 1. Now, let's lower the end-effector to the center height of Item 1 to prepare for grasping.

```

```python
Define the exact center position of Item 1
position_item_1_center = [0.26242024, 0.11987246, 0.76251084] # updated center
position from the feedback
orientation_item_1 = 1.3509903 # updated z-orientation

Execute trajectory to the center position of Item 1
execute_trajectory(position_item_1_center, orientation_item_1)
```

```

This will align the end-effector with the center of Item 1 to enable grasping.

Fig. 18: Execution logs for MALMM method for the task *empty container*.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the exact center position of Item 1
position_item_1_center = [0.26242024, 0.11987246, 0.76251084] # updated center
position from the feedback
orientation_item_1 = 1.3509903 # updated z-orientation

# Execute trajectory to the center position of Item 1
execute_trajectory(position_item_1_center, orientation_item_1)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

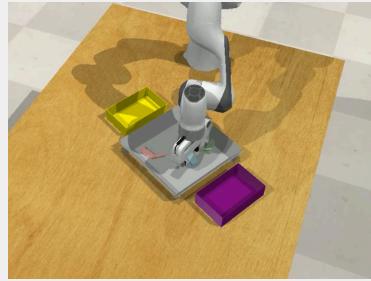
**item 1** :
Center Position: [0.26242912 0.11986454 0.76252615]
Orientation: [ 1.42761958 -0.66101032 1.34984219]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [0.20888169 0.161502 0.76569635]
Orientation: [-1.18745136 -0.85008377 -0.7391547 ]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30275935 -0.00879942 0.75926876]
Orientation: [-1.69133103 0.06280121 1.4741801 ]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
```

Fig. 19: Execution logs for MALMM method for the task *empty container*.

```

Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [0.27312514 0.12201677 0.77491474]
No object has been grasped

```



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

The end-effector is now positioned at the center of Item 1. Let's proceed to grasp Item 1.

```

```python
Close the gripper to grasp Item 1
close_gripper("item 1")
```

```

This command will activate the gripper to securely hold Item 1. After this, we'll move the end-effector with the grasped item to the yellow container.

Next speaker: [Code Executor](#)

```

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Close the gripper to grasp Item 1
close_gripper("item 1")
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

```

Fig. 20: Execution logs for MALMM method for the task *empty container*.

```

exitcode: 0 (execution succeeded)
Code output:

**item 1** :
Center Position: [0.26161361 0.12198494 0.76284611]
Orientation: [ 1.50454521 -0.19488159  1.48120713]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [0.22874464 0.16926542 0.77116591]
Orientation: [-0.92153287 0.04600464 0.64431977]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30276319 -0.00882074  0.75925827]
Orientation: [-1.6911993 0.06105263 1.47464418]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [0.26214865 0.11984584 0.76198256]
Grasped Object: item 1

```

Fig. 21: Execution logs for MALMM method for the task *empty container*.



Next speaker: **Coder**

Coder (to **Supervisor**):

Item 1 has been successfully grasped. Now we will move the end-effector, holding Item 1, to the yellow container and prepare to release it there.

```
```python
Define the position above the yellow container and its orientation
position_above_yellow_container = [0.14428537, -0.13835038, 0.77662492 + 0.1] # adding 0.1 to the z-coordinate for safe height
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

Execute trajectory to the position above the yellow container
execute_trajectory(position_above_yellow_container, orientation_yellow_container)
```

```

This command will relocate the end-effector to a position over the yellow container. Upon successful completion, we will proceed with the release of Item 1 into the container.

Next speaker: **Code Executor**

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above the yellow container and its orientation
position_above_yellow_container = [0.14428537, -0.13835038, 0.77662492 + 0.1] # adding 0.1 to the z-coordinate for safe height
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

# Execute trajectory to the position above the yellow container
execute_trajectory(position_above_yellow_container, orientation_yellow_container)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):
```

exitcode: 0 (execution succeeded)

Fig. 22: Execution logs for MALMM method for the task *empty container*.

```

Code output:

**item 1** :
Center Position: [ 0.14527979 -0.13991456  0.87741494]
Orientation: [-1.50911832 -0.02035595 -1.64567244]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown

**item 2** :
Center Position: [0.24045222 0.17546031 0.76531255]
Orientation: [-1.48467231 0.46156439 1.37311101]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown

**item 3** :
Center Position: [ 0.30272269 -0.00888073  0.75927299]
Orientation: [-1.69017982 0.05295125 1.47075367]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown

**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow

**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple

**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown

**End-effector**:
robot arm end-effector Position: [ 0.14431079 -0.13794859  0.87656611]
Grasped Object: item 1

```

Fig. 23: Execution logs for MALMM method for the task *empty container*.



Next speaker: **Coder**

Coder (to **Supervisor**):

The end-effector, holding Item 1, is now positioned above the yellow container.
Let's proceed to release Item 1 into the container.

```
```python
Open the gripper to release Item 1 into the yellow container
open_gripper()
```
```

This action will place Item 1 inside the yellow container. After confirming the successful release, we can continue with the next items.

Next speaker: **Code Executor**

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Open the gripper to release Item 1 into the yellow container
open_gripper()
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
```

Code Executor (to **Supervisor**):

```
exitcode: 0 (execution succeeded)
Code output:

**item 1** :
Center Position: [ 0.1430622 -0.14138995  0.7625128 ]
Orientation: [-1.45696712 -0.03870786 -1.69939542]
Height:  0.08286991715431213
Width:   0.02925252914428711
Length:  0.018727004528045654
Color:   unknown
**item 2** :
Center Position: [ 0.24045578  0.17546412  0.76531279]
```

Fig. 24: Execution logs for MALMM method for the task *empty container*.

```

Orientation: [-1.485111  0.46158794  1.37342489]
Height:  0.08309988677501678
Width:  0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30271551 -0.00887695  0.75927991]
Orientation: [-1.69037807  0.05254418  1.47010922]
Height:  0.09559351205825806
Width:  0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:  0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:  0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height:  0.050115831196308136
Width:  0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.14381902 -0.13824694  0.8761875 ]
No object has been grasped

```



Fig. 25: Execution logs for MALMM method for the task *empty container*.

```

Next speaker: Planner

Planner (to Supervisor):
Item 1 has been successfully released into the yellow container. Now, let's move to
the next item.

### Step 5: Move to Item 2
Following similar steps to ensure safe handling and avoiding collisions, let us move
above Item 2 and prepare for grasping.

- **Target**: Item 2
- **Details**: Item 2 is located at [0.24045578, 0.17546412, 0.76531279].  

##### Movement Plan:  

1. Move to a safe height (+0.1 units in the z-direction from its current height)
above Item 2.  

2. Adjust the orientation of the end-effector to match the z-orientation of Item 2.

```python
Define the position above Item 2 and its orientation
position_above_item_2 = [0.24045578, 0.17546412, 0.76531279 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_2 = 1.37342489 # updated z-orientation

Execute trajectory to the specified position above Item 2
execute_trajectory(position_above_item_2, orientation_item_2)
```

This command will relocate the end-effector above Item 2. After moving to the
appropriate position, we will proceed with the sequence to grasp Item 2.

-----

```

Next speaker: Coder

Coder (to Supervisor):

```

```python
Define the position above Item 2 and its orientation
position_above_item_2 = [0.24045578, 0.17546412, 0.76531279 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_2 = 1.37342489 # z-orientation of Item 2

Execute trajectory to the specified position above Item 2
execute_trajectory(position_above_item_2, orientation_item_2)
```

This command will move the robot arm end-effector above Item 2. Once in position, we
will lower to the center height of Item 2 for grasping.

```

Fig. 26: Execution logs for MALMM method for the task *empty container*.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above Item 2 and its orientation
position_above_item_2 = [0.24045578, 0.17546412, 0.76531279 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_2 = 1.37342489 # z-orientation of Item 2

# Execute trajectory to the specified position above Item 2
execute_trajectory(position_above_item_2, orientation_item_2)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

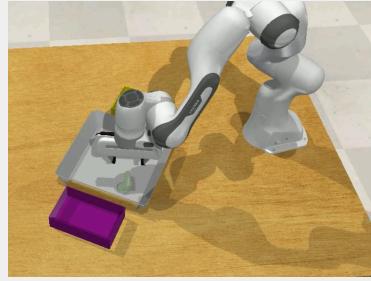
**item 1** :
Center Position: [ 0.14306277 -0.14138883  0.76250517]
Orientation: [-1.45664239 -0.03869065 -1.69928336]
Height:  0.0828699171543123
Width:   0.02925252914428711
Length:  0.018727004528045654
Color:  unknown
**item 2** :
Center Position: [ 0.24044535 0.17545241 0.76531231]
Orientation: [-1.48365438 0.46132487 1.37236249]
Height:  0.08309988677501678
Width:   0.04162578284740448
Length:  0.024330079555511475
Color:  unknown
**item 3** :
Center Position: [ 0.30272231 -0.00903048  0.75927842]
Orientation: [-1.68989277 0.0401011  1.46921694]
Height:  0.09559351205825806
Width:   0.04475623369216919
Length:  0.01606738567352295
Color:  unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:   0.1255001425743103
Length:  0.05075031518936157
Color:  yellow
**container 2** :
Center Position: [ 0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
```

Fig. 27: Execution logs for MALMM method for the task *empty container*.

```

Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [0.24014723 0.17452209 0.8648594 ]
No object has been grasped

```



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

The robot arm end-effector is now positioned above Item 2. Next, I'll lower the end-effector to the exact center of Item 2 to prepare for grasping.

```

```python
Define the exact center position of Item 2 for grasping
position_item_2_center = [0.24044535, 0.17545241, 0.76531231] # updated center
position from the feedback
orientation_item_2 = 1.37236249 # updated z-orientation

Execute trajectory to the center position of Item 2
execute_trajectory(position_item_2_center, orientation_item_2)
```

```

This step will align the end-effector with the center of Item 2, making it ready for grasping.

Next speaker: [Code Executor](#)

Fig. 28: Execution logs for MALMM method for the task *empty container*.

```

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the exact center position of Item 2 for grasping
position_item_2_center = [0.24044535, 0.17545241, 0.76531231] # updated center
position from the feedback
orientation_item_2 = 1.37236249 # updated z-orientation

# Execute trajectory to the center position of Item 2
execute_trajectory(position_item_2_center, orientation_item_2)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

**item 1** :
Center Position: [ 0.14308642 -0.14137222  0.76250553]
Orientation: [-1.45808053 -0.03841349 -1.69689798]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown

**item 2** :
Center Position: [0.24041151 0.17541966 0.76531327]
Orientation: [-1.47995949 0.46067515 1.36954308]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown

**item 3** :
Center Position: [ 0.30272567 -0.00912107  0.75928366]
Orientation: [-1.68991935  0.03379511  1.46791077]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown

**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow

**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple

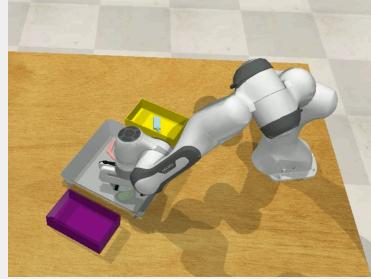
```

Fig. 29: Execution logs for MALMM method for the task *empty container*.

```

**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [0.2393128 0.1751644 0.76459098]
No object has been grasped

```



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

The end-effector is now correctly positioned at the center of Item 2. Let's proceed to grasp Item 2.

```

```python
Close the gripper to grasp Item 2
close_gripper("item 2")
```

```

This command will activate the gripper to securely hold Item 2. After confirming the successful grasp, we'll move the end-effector with the grasped item to the yellow container.

Next speaker: [Code Executor](#)

```

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Close the gripper to grasp Item 2
close_gripper("item 2")
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

```

Fig. 30: Execution logs for MALMM method for the task *empty container*.

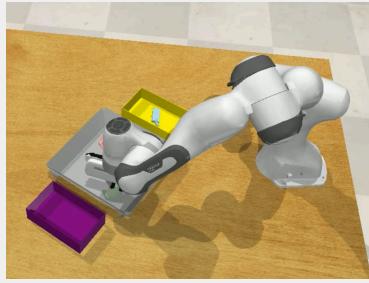
```

exitcode: 0 (execution succeeded)
Code output:

**item 1** :
Center Position: [ 0.14308627 -0.14137173  0.76250553]
Orientation: [-1.45808089 -0.03840774 -1.69689798]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [0.24609478 0.17843167 0.7654205 ]
Orientation: [-1.43815088 0.25025159 1.35532677]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30273569 -0.00911842  0.75928581]
Orientation: [-1.69069362  0.03281237  1.46990514]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [0.23991209 0.17532751 0.76476395]
Grasped Object: item 2

```

Fig. 31: Execution logs for MALMM method for the task *empty container*.



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

Item 2 has been successfully grasped. Now, let's move the end-effector, holding Item 2, to the yellow container and prepare to release it there.

```
```python
Define the position above the yellow container and its orientation for Item 2
position_above_yellow_container_item_2 = [0.14428537, -0.13835038, 0.77662492 + 0.1]
adding 0.1 to the z-coordinate for safe height

Execute trajectory to the position above the yellow container with Item 2
execute_trajectory(position_above_yellow_container_item_2,
orientation_yellow_container)
```

```

This command will relocate the end-effector to a position over the yellow container. Upon successful completion, we will proceed with the release of Item 2 into the container.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above the yellow container and its orientation for Item 2
position_above_yellow_container_item_2 = [0.14428537, -0.13835038, 0.77662492 + 0.1]
# adding 0.1 to the z-coordinate for safe height

# Execute trajectory to the position above the yellow container with Item 2
execute_trajectory(position_above_yellow_container_item_2,
orientation_yellow_container)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):
```

exitcode: 1 (execution failed)

Fig. 32: Execution logs for MALMM method for the task *empty container*.

```

Code output:
Error: name 'orientation_yellow_container' is not defined

-----
Next speaker: Coder

Coder (to Supervisor):
It appears I forgot to define the `orientation_yellow_container` used in the
movement plan. Let me correct this and execute the trajectory again.

```python
Define the orientation for the yellow container
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

Execute trajectory to the position above the yellow container with Item 2
execute_trajectory(position_above_yellow_container_item_2,
orientation_yellow_container)
```

This correction will ensure the end-effector moves correctly above the yellow
container with Item 2.

-----
Next speaker: Code Executor

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the orientation for the yellow container
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

# Execute trajectory to the position above the yellow container with Item 2
execute_trajectory(position_above_yellow_container_item_2,
orientation_yellow_container)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 1 (execution failed)
Code output:
Error: name 'position_above_yellow_container_item_2' is not defined

-----
Next speaker: Coder

Coder (to Supervisor):

```

Fig. 33: Execution logs for MALMM method for the task *empty container*.

It seems I also mismanaged the variable definitions in the context of the previous error fix. Let's provide the complete definition and execute the movement again.

```
```python
Define the position above the yellow container and its orientation
position_above_yellow_container_item_2 = [0.14428537, -0.13835038, 0.77662492 + 0.1]
adding 0.1 to the z-coordinate for safe height
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

Execute trajectory to the position above the yellow container with Item 2
execute_trajectory(position_above_yellow_container_item_2,
orientation_yellow_container)
```

```

This code will move the end-effector to the correct position above the yellow container, ready to release Item 2.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above the yellow container and its orientation
position_above_yellow_container_item_2 = [0.14428537, -0.13835038, 0.77662492 + 0.1]
# adding 0.1 to the z-coordinate for safe height
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

# Execute trajectory to the position above the yellow container with Item 2
execute_trajectory(position_above_yellow_container_item_2,
orientation_yellow_container)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

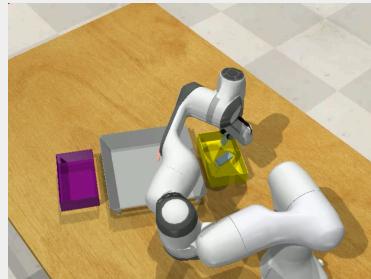
**item 1** :
Center Position: [ 0.14308822 -0.14136653  0.76249969]
Orientation: [-1.45792902 -0.03835068 -1.69666004]
Height:  0.08286991715431213
Width:  0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [ 0.14182729 -0.15575357  0.88039124]
Orientation: [ 1.38485324 -0.06536578 -1.74580479]
Height:  0.08309988677501678
Width:  0.04162578284740448
Length: 0.024330079555511475
Color: unknown
```

Fig. 34: Execution logs for MALMM method for the task *empty container*.

```

**item 3** :
Center Position: [ 0.3027004 -0.0092243  0.75927955]
Orientation: [-1.68999493  0.02021662  1.46634865]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.14730179 -0.15146361  0.87970412]
Grasped Object: item 2

```



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

Fig. 35: Execution logs for MALMM method for the task *empty container*.

The end-effector, holding Item 2, is now correctly positioned above the yellow container. Let's proceed to release Item 2 into the container.

```
```python
Open the gripper to release Item 2 into the yellow container
open_gripper()
```

```

This action will place Item 2 inside the yellow container. After confirming the successful release, we can continue with the last item, Item 3.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Open the gripper to release Item 2 into the yellow container
open_gripper()
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

**item 1** :
Center Position: [ 0.14301753 -0.14140184  0.76249993]
Orientation: [-1.45650399 -0.04159273 -1.69872046]
Height:  0.08286991715431213
Width:   0.02925252914428711
Length:  0.018727004528045654
Color:  unknown
**item 2** :
Center Position: [ 0.12674096 -0.14680521  0.77946627]
Orientation: [ 1.73217309 -0.01521828 -0.65067261]
Height:  0.08309988677501678
Width:   0.04162578284740448
Length:  0.024330079555511475
Color:  unknown
**item 3** :
Center Position: [ 0.30269942 -0.00921898  0.7592541 ]
Orientation: [-1.68962145  0.01996957  1.46679688]
Height:  0.09559351205825806
Width:   0.04475623369216919
Length:  0.01606738567352295
Color:  unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:   0.1255001425743103
```

Fig. 36: Execution logs for MALMM method for the task *empty container*.

```

Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.14393187 -0.13809551 0.87573469]
No object has been grasped

```



Next speaker: [Planner](#)

[Planner](#) (to [Supervisor](#)):

Item 2 has been successfully released into the yellow container. Now, let's proceed to the final item, Item 3.

Movement Plan for Item 3:

- **Target**: Item 3
- **Details**: Item 3 is located at [0.30269942, -0.00921898, 0.7592541].

Move to Item 3:

1. Move to a safe height (+0.1 units in the z-direction from its current height) above Item 3.
2. Adjust the orientation of the end-effector to match the z-orientation of Item 3.

```python

Fig. 37: Execution logs for MALMM method for the task *empty container*.

```

Define the position above Item 3 and its orientation
position_above_item_3 = [0.30269942, -0.00921898, 0.7592541 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_3 = 1.46679688 # z-orientation of Item 3

Execute trajectory to the specified position above Item 3
execute_trajectory(position_above_item_3, orientation_item_3)
```

```

This command positions the end-effector above Item 3. Once this is successfully completed, we'll continue with the sequence to grasp Item 3.

Next speaker: **Coder**

Coder (to **Supervisor**):

```

```python
Define the position above Item 3 and its orientation
position_above_item_3 = [0.30269942, -0.00921898, 0.7592541 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_3 = 1.46679688 # z-orientation of Item 3

Execute trajectory to the specified position above Item 3
execute_trajectory(position_above_item_3, orientation_item_3)
```

```

This code will enable the robot arm end-effector to move above Item 3, aligning it correctly for subsequent grasping. Once in position, we will proceed with lowering to the center height of Item 3 for grasping.

Next speaker: **Code Executor**

```

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above Item 3 and its orientation
position_above_item_3 = [0.30269942, -0.00921898, 0.7592541 + 0.1] # adding 0.1 to
the z-coordinate for a safe height
orientation_item_3 = 1.46679688 # z-orientation of Item 3

# Execute trajectory to the specified position above Item 3
execute_trajectory(position_above_item_3, orientation_item_3)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

```

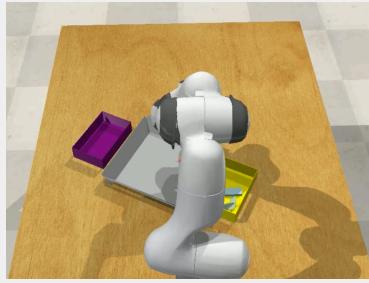
Fig. 38: Execution logs for MALMM method for the task *empty container*.

```

**item 1** :
Center Position: [ 0.14379495 -0.14135085  0.76250517]
Orientation: [-1.45657134 -0.05273249 -1.697667 ]
Height:  0.08286991715431213
Width:  0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [ 0.08712924 -0.13331485  0.76473057]
Orientation: [1.67019391 0.69773251 1.56450593]
Height:  0.08309988677501678
Width:  0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30268082 -0.00930745  0.75925148]
Orientation: [-1.68963873  0.00939034  1.46507275]
Height:  0.09559351205825806
Width:  0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:  0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:  0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height:  0.050115831196308136
Width:  0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.26563698 -0.01240687  0.84598607]
No object has been grasped

```

Fig. 39: Execution logs for MALMM method for the task *empty container*.



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

The robot arm end-effector is now well-positioned above Item 3. We'll proceed by lowering the end-effector to the exact center of Item 3 to prepare for grasping.

```
```python
Define the exact center position of Item 3 for grasping
position_item_3_center = [0.30268082, -0.00930745, 0.75925148] # updated center
position from the feedback
orientation_item_3 = 1.46507275 # updated z-orientation

Execute trajectory to the center position of Item 3
execute_trajectory(position_item_3_center, orientation_item_3)
```

```

This step will align the end-effector with the center of Item 3, making it ready for grasping.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the exact center position of Item 3 for grasping
position_item_3_center = [0.30268082, -0.00930745, 0.75925148] # updated center
position from the feedback
orientation_item_3 = 1.46507275 # updated z-orientation

# Execute trajectory to the center position of Item 3
execute_trajectory(position_item_3_center, orientation_item_3)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
```

[Code Executor](#) (to [Supervisor](#)):

```
exitcode: 0 (execution succeeded)
Code output:
```

Fig. 40: Execution logs for MALMM method for the task *empty container*.

```

**item 1** :
Center Position: [ 0.14379486 -0.14135034  0.76250517]
Orientation: [-1.45656896 -0.05272721 -1.69767213]
Height:  0.08286991715431213
Width:  0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [ 0.08702552 -0.13337131  0.76461375]
Orientation: [1.66767275 0.69895399 1.57531953]
Height:  0.08309988677501678
Width:  0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.30268118 -0.00933307  0.75927961]
Orientation: [-1.6899581  0.00720141  1.46473527]
Height:  0.09559351205825806
Width:  0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:  0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height:  0.20000019669532776
Width:  0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height:  0.050115831196308136
Width:  0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.29555097 -0.00663747  0.76224905]
No object has been grasped

```

Fig. 41: Execution logs for MALMM method for the task *empty container*.



Next speaker: [Coder](#)

[Coder](#) (to [Supervisor](#)):

The end-effector is now correctly positioned at the center of Item 3. Let's proceed to grasp Item 3.

```
```python
Close the gripper to grasp Item 3
close_gripper("item 3")
```
```

This command will activate the gripper to securely hold Item 3. After the grasping is confirmed, we'll move the end-effector holding Item 3 to the yellow container.

Next speaker: [Code Executor](#)

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Close the gripper to grasp Item 3
close_gripper("item 3")
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
```

[Code Executor](#) (to [Supervisor](#)):

```
exitcode: 0 (execution succeeded)
Code output:
```

```
**item 1** :
Center Position: [ 0.14379486 -0.14134988  0.76250517]
Orientation: [-1.45656753 -0.05272545 -1.69767296]
Height:  0.08286991715431213
Width:   0.02925252914428711
Length:  0.018727004528045654
Color:   unknown
**item 2** :
Center Position: [ 0.0871923  -0.13323432  0.76476294]
```

Fig. 42: Execution logs for MALMM method for the task *empty container*.

```

Orientation: [1.66801322 0.69813508 1.55739307]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.29552701 -0.0087701  0.75902712]
Orientation: [-1.68933785 -0.08074107  1.46892059]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.3055259 -0.00908552  0.76144767]
Grasped Object: item 3

```



Fig. 43: Execution logs for MALMM method for the task *empty container*.

Next speaker: **Coder**

Coder (to **Supervisor**):

Item 3 has been successfully grasped. Now, let's move the end-effector, holding Item 3, to the yellow container and prepare to release it there.

```
```python
Define the position above the yellow container and its orientation for Item 3
position_above_yellow_container_item_3 = [0.14428537, -0.13835038, 0.77662492 + 0.1]
adding 0.1 to the z-coordinate for safe height
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

Execute trajectory to the position above the yellow container with Item 3
execute_trajectory(position_above_yellow_container_item_3,
orientation_yellow_container)
```

This command will relocate the end-effector to a position over the yellow container. Upon successful completion, we will proceed with the release of Item 3 into the container.
```

Next speaker: **Code Executor**

```
>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Define the position above the yellow container and its orientation for Item 3
position_above_yellow_container_item_3 = [0.14428537, -0.13835038, 0.77662492 + 0.1]
# adding 0.1 to the z-coordinate for safe height
orientation_yellow_container = -1.57081306 # z-orientation of the yellow container

# Execute trajectory to the position above the yellow container with Item 3
execute_trajectory(position_above_yellow_container_item_3,
orientation_yellow_container)
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):
```

exitcode: 0 (execution succeeded)

Code output:

```
**item 1** :
Center Position: [ 0.14379337 -0.14134274  0.76251006]
Orientation: [-1.45677412 -0.05271862 -1.69775331]
Height: 0.08286991715431213
Width: 0.02925252914428711
Length: 0.018727004528045654
Color: unknown
**item 2** :
Center Position: [ 0.08708401 -0.13334724  0.76472932]
```

Fig. 44: Execution logs for MALMM method for the task *empty container*.

```

Orientation: [1.67025661 0.69984609 1.56752503]
Height: 0.08309988677501678
Width: 0.04162578284740448
Length: 0.024330079555511475
Color: unknown
**item 3** :
Center Position: [ 0.15460324 -0.13777663 0.87436849]
Orientation: [ 1.68682742 0.17788626 -1.6716038 ]
Height: 0.09559351205825806
Width: 0.04475623369216919
Length: 0.01606738567352295
Color: unknown
**container 1** :
Center Position: [ 0.14428537 -0.13835038 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.14478338 -0.13840255 0.87690151]
Grasped Object: item 3

```

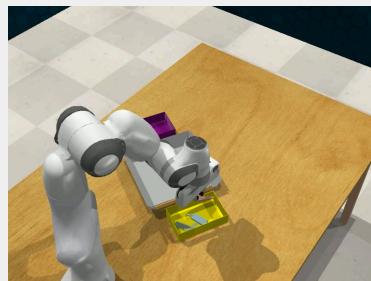


Fig. 45: Execution logs for MALMM method for the task *empty container*.

```

Next speaker: Coder

Coder (to Supervisor):
The end-effector, holding Item 3, is now positioned above the yellow container.
Let's proceed to release Item 3 into the container.

```python
Open the gripper to release Item 3 into the yellow container
open_gripper()
```

This action will place Item 3 inside the yellow container. After confirming the
successful release, we will conclude the task.

-----

```

Next speaker: Code Executor

```

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...
# Open the gripper to release Item 3 into the yellow container
open_gripper()
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
Code Executor (to Supervisor):

exitcode: 0 (execution succeeded)
Code output:

**item 1** :
Center Position: [ 0.14355005 -0.14146221  0.76248848]
Orientation: [-1.45776689 -0.04456368 -1.69519341]
Height:  0.08286991715431213
Width:  0.02925252914428711
Length: 0.018727004528045654
Color:  unknown
**item 2** :
Center Position: [ 0.08707935 -0.13334341  0.76466411]
Orientation: [1.67015123 0.69829035 1.56843448]
Height:  0.08309988677501678
Width:  0.04162578284740448
Length: 0.024330079555511475
Color:  unknown
**item 3** :
Center Position: [ 0.15378314 -0.13552269  0.77213782]
Orientation: [ 1.61661255  0.24450047 -2.26560116]
Height:  0.09559351205825806
Width:  0.04475623369216919
Length: 0.01606738567352295
Color:  unknown
**container 1** :

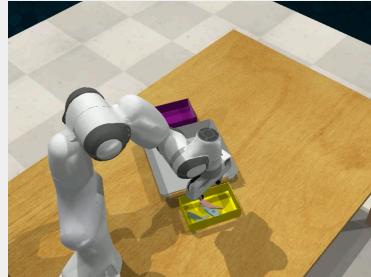
```

Fig. 46: Execution logs for MALMM method for the task *empty container*.

```

Center Position: [ 0.14428537 -0.13835038  0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: yellow
**container 2** :
Center Position: [0.37851185 0.30339384 0.77662492]
Orientation: [-1.57081509 -1.08325779 -1.57081306]
Height: 0.20000019669532776
Width: 0.1255001425743103
Length: 0.05075031518936157
Color: purple
**large container** :
Center Position: [0.26139459 0.08251426 0.77600056]
Orientation: [-3.05039459e-04 1.61741351e-04 -4.87538993e-01]
Height: 0.050115831196308136
Width: 0.31764334440231323
Length: 0.31764310598373413
Color: unknown
**End-effector**:
robot arm end-effector Position: [ 0.14364897 -0.13831139  0.87574887]
No object has been grasped

```



Next speaker: [Planner](#)

[Planner](#) (to [Supervisor](#)):

Item 3 has been successfully released into the yellow container. With this, all items from the large container have been successfully relocated to the yellow container.

This completes the task of emptying the large container into the yellow container. You may review the final placements and orientations of the items within the yellow container for your records or further actions. Thank you for the collaboration. If there are any other tasks or adjustments needed, please let me know!

Fig. 47: Execution logs for MALMM method for the task *empty container*.

Next speaker: **Coder**

Coder (to **Supervisor**):

```
```python
Task completed
task_completed()
```
```

This command signifies that the task has been completed successfully. All specified items have been moved from the large container to the yellow container. Thank you! If you need further assistance or additional tasks to be performed, I am here to help.

Next speaker: **Code Executor**

>>>>> EXECUTING CODE BLOCK 0 (inferred language is python)...

```
# Task completed
task_completed()
{'lang': 'python', 'filename': None, 'use_docker': 'python:None'}
```

Code Executor (to **Supervisor**):

```
exitcode: 0 (execution succeeded)
Code output:
```

Stopping the process.....

Fig. 48: Execution logs for MALMM method for the task *empty container*.