Language Models as Zero-Shot Trajectory Generators

Teyun Kwon[1], Norman Di Palo[1], Edward Johns[1]

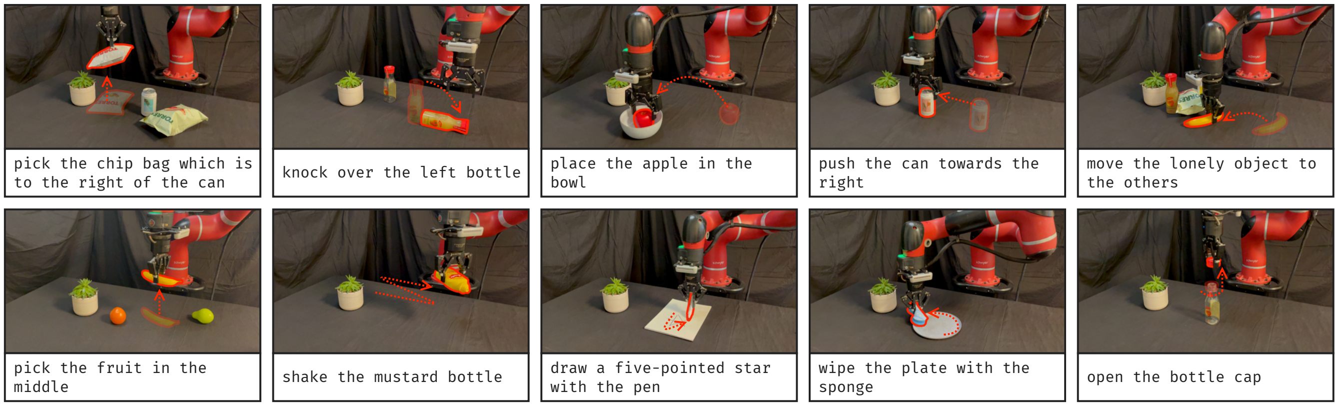

Fig. 1: A selection of the tasks we use to study if a single, task-agnostic LLM prompt can generate a dense sequence of end-effector poses, when given only object detection and segmentation models, and no in-context examples, motion primitives, pre-trained skills, or external trajectory optimisers.

***Abstract*— Large Language Models (LLMs) have recently shown promise as high-level planners for robots when given access to a selection of low-level skills. However, it is often assumed that LLMs do not possess sufficient knowledge to be used for the low-level trajectories themselves. In this work, we address this assumption thoroughly, and investigate if an LLM (GPT-4) can directly predict a dense sequence of end-effector poses for manipulation tasks, when given access to only object detection and segmentation vision models. We designed a single, task-agnostic prompt, without any in-context examples, motion primitives, or external trajectory optimisers. Then we studied how well it can perform across 30 real-world language-based tasks, such as "*open the bottle cap*" and "*wipe the plate with the sponge*", and we investigated which design choices in this prompt are the most important. Our conclusions raise the assumed limit of LLMs for robotics, and we reveal for the first time that LLMs do indeed possess an understanding of low-level robot control sufficient for a range of common tasks, and that they can additionally detect failures and then re-plan trajectories accordingly. Videos, prompts, and code are available at: https://www.robot-learning.uk/language-models-trajectory-generators.**

***Index Terms*— AI-Based Methods, Big Data in Robotics and Automation, Deep Learning in Grasping and Manipulation**

I. Introduction

In recent years, Large Language Models (LLMs) have attracted significant attention and acclaim for their remarkable capabilities in reasoning about common, everyday tasks [1]. This widespread recognition has since led to efforts in the robotics community to adopt LLMs for high-level task planning [2]. However, for low-level control, existing proposals have relied on auxiliary components beyond the LLM, such as pre-trained skills, motion primitives, trajectory optimisers, and numerous language-based in-context examples (Fig. 5). Given the lack of exposure of LLMs to physical interaction data, it is often assumed that LLMs are incapable of low-level control [3], [4], [5].

However, until now, this assumption has not been thoroughly examined. In this paper, we now investigate if LLMs have sufficient understanding of low-level control to be adopted for **zero-shot dense trajectory generation for robot manipulators**, without the need for the aforementioned auxiliary components. We provide an LLM (GPT-4 [6]) with access to off-the-shelf object detection and segmentation models, and then require all remaining reasoning to be performed by the LLM itself. We also require that the same task-agnostic prompt is used for all tasks, such as "*open the bottle cap*" and "*wipe the plate with the sponge*", which we took from the recent literature. And through this investigation, we uncovered the underlying principles and strategies that empower LLMs to navigate the complexities of robot manipulation.

Consequently, our contributions are threefold: **(1)** We demonstrate, for the first time, that a pre-trained LLM, when provided with only an off-the-shelf object detection and segmentation model, can **guide zero-shot a robot manipulator by outputting a dense sequence of end-effector poses**, without the need for pre-trained skills, motion primitives, trajectory optimisers, or in-context examples. **(2)** We present several ablation studies which shed light on **what techniques and prompts lead to the emergence of these capabilities**. **(3)** We study how, by analysing the trajectory of objects

[1] Teyun Kwon, Norman Di Palo and Edward Johns are with the Robot Learning Lab at Imperial College London. {john.kwon20, n.di-palo20, e.johns}@imperial.ac.uk

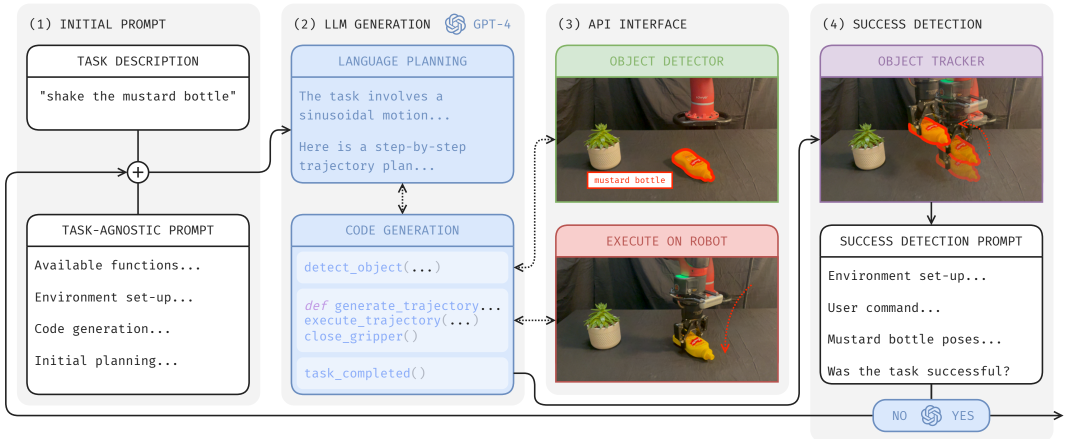

Fig. 2: An overview of the pipeline. **(1)** The main prompt along with the task instruction is provided to the LLM, from which it **(2)** generates high-level natural language reasoning steps before outputting Python code **(3)** to interface with a pre-trained object detection model and execute the generated trajectories on the robot. **(4)** After task execution, an off-the-shelf object tracking model is used to obtain 3-D bounding boxes of the previously detected objects over the duration of the task, which are then provided to the LLM as numerical values to detect whether the task was executed successfully or not.

across an image, **LLMs can also detect if a task has failed and subsequently re-plan an alternative trajectory**.

II. Problem Formulation

We investigate if an LLM (GPT-4 [6]) can predict a dense sequence of end-effector poses to solve a range of manipulation tasks. We now explain what the assumptions and constraints are in our investigation, followed by details of the tasks used for evaluation. Given this background, we then present our investigation and its results in Sec. III.

Assumptions and Constraints: We design a task-agnostic prompt to study the zero-shot control capabilities of LLMs, with the following assumptions: **(1)** no pre-existing motion primitives, policies or trajectory optimisers: the LLM should output the *full sequence of end-effector poses itself*; **(2)** no in-context examples: we study the ability of LLMs to reason about tasks given their *internal knowledge alone*, and no part of any task is explicitly mentioned in the prompt, either in the form of examples or instructions; **(3)** the LLM can query a pre-trained vision model to obtain information about the scene, but should *autonomously generate, parse and interpret the inputs and outputs*; **(4)** no additional pre-training or fine-tuning on robotics-specific data: we focus our research on *pre-trained and widely available models*, so that our work can easily be reproduced even with limited compute budget. Details of the real-world experimental setup are presented in Appendix B.

Task Selection: In pursuit of objectivity, we opt to benchmark our zero-shot LLM-guided robotic control against a challenging repertoire of everyday manipulation tasks. We **recreated 30 everyday manipulation tasks from recent robotics papers** published at leading conferences [5], [7], [8], [9], [3], often tackled by relying on hundreds of manual demonstrations. This serves as a representative benchmark of real-world challenges, mirroring the complexity and diversity of the tasks encountered in contemporary robotics research. We choose tasks which semantically cover the most representative tabletop robot behaviours expressed in these papers, and success criteria are human-evaluated and designed to mirror those proposed in the original papers. For each combination of task and method in the following experimental sections, we calculate the success rate over 5 trials, randomising the positions and orientations of the objects for each trial. The task description is provided in natural language to the LLM, after which no additional human feedback or intervention is allowed. The full list of tasks is shown in Fig. 3, and example task success and failure videos are available at https://www.robot-learning.uk/language-models-trajectory-generators.

III. Prompt Development

Full Prompt: The core motivation of our work is to investigate whether LLMs can inherently guide robots with minimal dependence on specialised external models and components, in order to provide effective and useful insights for the robotics community. Through this investigation, we designed a single task-agnostic prompt for a range of everyday manipulation tasks, which does not require any in-context examples or task-specific guidance. Fig. 2 illustrates the main information flow in our framework, showing how the task-agnostic prompt interfaces with the vision models and the robot.

Through our experiments outlined in this section, our final prompt formulation instructs the LLM to self-summarise and decompose the predicted plan into steps, before generating Python code which, when run by a standard Python interpreter, outputs a dense sequence of poses for the end-effector to follow; this pipeline resulted in the best performance across those we experimented with. We include details fundamental to all tasks, such as coordinate definitions, as

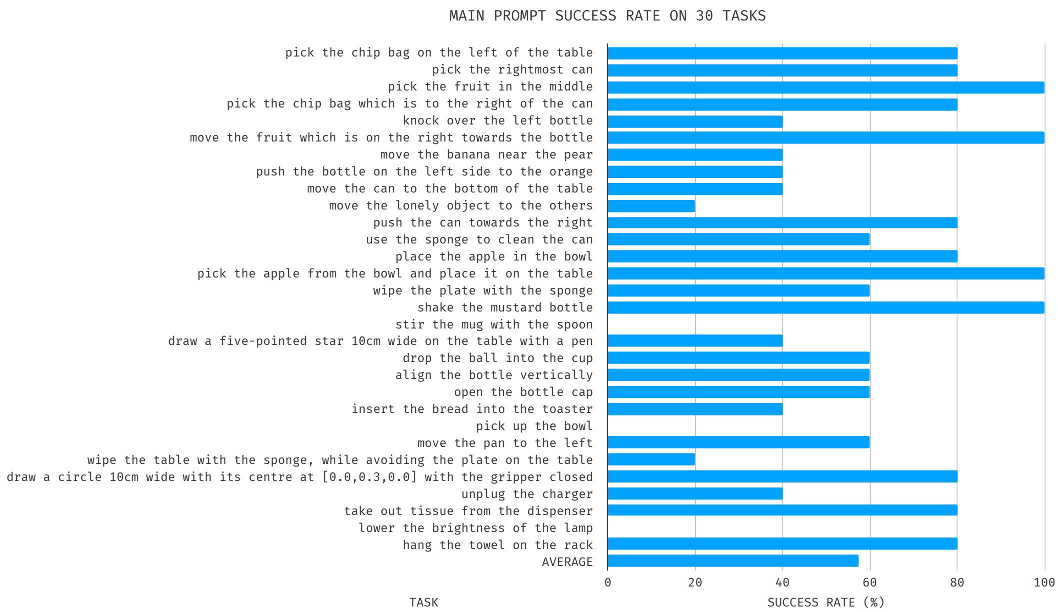

Fig. 3: Success rates of the best-performing prompt in our investigation on 30 manipulation tasks.

well as functions available for the LLM to call, such as `detect_object`, which returns the calculated 3-D bounding boxes of the queried objects directly to the LLM. We also include instructions which aim to improve the correctness and reliability of the generated trajectories, such as guidance on step-by-step reasoning, code generation, and collision avoidance. The full prompt is shown in Appendix D.

We investigated the LLM's ability to solve zero-shot a range of manipulation tasks, by evaluating the full prompt on the full set of tasks taken from the recent literature. These tasks and their success rates are presented in Fig. 3. Remarkably, our experiments reveal that LLMs, when equipped with an off-the-shelf vision model and no external motion primitives, policies, or trajectory optimisers, do indeed exhibit notable proficiency in executing the majority of these tasks, by directly predicting a dense sequence of end-effector poses. In the original papers from which these tasks are selected [8], [7], [9], solving these tasks required numerous human demonstrations. As such, these findings underscore the potential of LLMs as intuitive and versatile guides for robotic manipulation that minimise the need for human time and supervision. A sample LLM output is shown in Appendix F.

Prompt Ablations: During the design of this full prompt, we identified several challenges when using LLMs for low-level control, without access to other external dependencies. In this section, we now outline these challenges which motivated the final design of the prompt, and accompany them with results from ablation studies conducted across a diverse set of tasks (Fig. 4), where certain parts of the full prompt were removed. We choose a subset of the 30 original tasks for the ablation studies, which we list in Appendix E, that still capture the various manipulation challenges in the full set. The ablated components of the full prompt are shown in Appendix D.

(1) LLMs often require step-by-step reasoning to solve tasks. Prior work has shown that the reasoning capabilities of LLMs can be improved by asking them to break down the task in a step-by-step manner [10], [11], and adopting this strategy, we prompt the LLM **(1)** to break down the trajectory into a sequence of sub-trajectory steps, and **(2)** to include in the plan when to lower the gripper to make contact with an object. We find that, without including these step-by-step reasoning prompts, the LLM often omits key trajectory steps required to execute the task successfully, such as opening or closing the gripper, or aligning the gripper to be parallel to the graspable side of the object, which are not stated explicitly in the prompt. Indeed, the first three columns in Fig. 4 show that prompting the LLM to think step by step resulted in the highest performance increase.

(2) LLMs can be prone to write code which results in errors, both syntactically and semantically. While much improvement has been made in the domain of code generation by LLMs [12], [6], their outputs can still throw errors, as well as produce undesirable results when executed. In order to mitigate this, and inspired again by the power of LLMs performing an internal monologue with natural language reasoning, we prompt the LLM to document any functions it defines, with their expected inputs and outputs, and their data types. In addition, we include a prompt instructing the LLM to define reusable functions for common motions (for example, linear trajectory from one point to another), to prevent instances where, as a notable example, it would hard-code the height of the gripper inside a function definition, and reuse that function for another sub-trajectory step which should have been executed at a different height. Similarly, we prompt the LLM to name each sub-trajectory step variable with a number to relate it to each of the steps

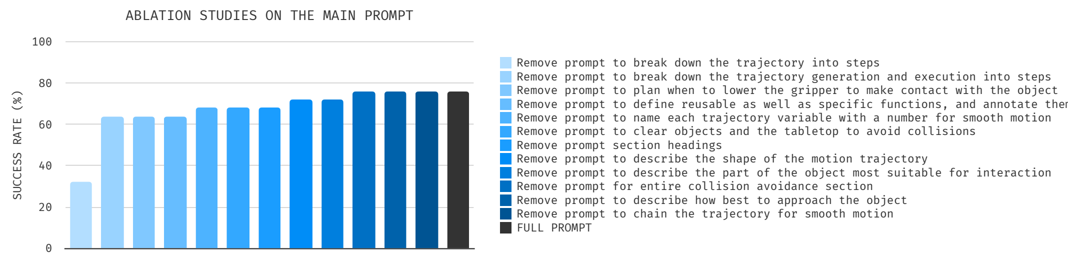

Fig. 4: We investigate the effect of removing parts of the main prompt on task success rates.

in the high-level trajectory plan, and to minimise the chance of omitting a sub-trajectory step. The effects of removing these prompt components are, again, noticeable (fourth and fifth columns in Fig. 4).

(3) LLMs are trained on limited grounded physical interaction data. Due to the scarcity of grounded physical interaction data in their training corpora [13], LLMs often fail to take into account possible collisions between the objects being manipulated. We therefore prompt the LLM to pay attention to the dimensions of the objects and to generate additional waypoints and sub-trajectories, which could help with avoiding collisions. We also include in the prompt a specific phrase which we noticed during our investigation was being used frequently by the LLM for its internal reasoning ("*clear objects and the tabletop*"). Our experiments show that, while removing this particular phrase from the collision avoidance prompt lowered performance (sixth column in Fig. 4), LLMs do possess some inherent understanding of possible collisions between different objects, as they performed well even after removing the entire collision avoidance prompt (tenth column in Fig. 4).

(4) LLMs often fail to reason about complex trajectory shapes. In a manner similar to the first challenge, we employ a two-step strategy, where initially, we explicitly ask the LLM to generate a textual description of the *shape of the motion trajectory* as internal reasoning (for example, shaking involves a sinusoidal motion), before outputting the actual sequence of poses required to execute this trajectory (in contrast to Challenge **(1)**, where we prompted the LLM to output a more detailed step-by-step trajectory plan). This has been shown to be beneficial in prior work [4], and indeed this result is also reflected in the eighth column in Fig. 4.

(5) LLMs often fail to reason about how to interact with objects. In our experiments, we found that LLMs often simplified and failed to reason about more intricate details of object interaction, such as realising that some objects require interaction with a specific part (for example, the rim of a bowl, or the handle of a pan). In order to enable the LLM to detect the most suitable object part to interact with, we prompt it to describe the object part in high-level natural language, and we see in the ninth column in Fig. 4 that this results in more tasks being executed successfully.

IV. Further Investigations

We conduct further ablation studies regarding the modality of the trajectory generation (whether to output the trajectory directly in numerical values as language tokens, or to generate Python code which, when executed, outputs the trajectory), the extent to which each output modality is executable by the robot, and the performance of the different available LLMs. The tasks used for these ablation studies are shown in Appendix E. We also detail the ability of LLMs to detect whether a task was executed successfully or not and subsequently re-plan the trajectory, the various sources of error which led to task execution failures when performing the 30 tasks with the main prompt, and provide a detailed comparison against Code as Policies [14], which is closest to our work. We present the full details of these investigations in Appendix C.

V. Conclusions

The primary contribution of this paper is an investigation into to what extent an LLM can successfully predict dense sequences of end-effector poses for a range of real-world manipulation tasks. To differentiate this from other previous works, we imposed constraints that the LLM must use a single task-agnostic prompt without any in-context examples, and has access to only off-the-shelf object detection and segmentation vision models, with no other auxiliary components. Our experiments encompassed 30 diverse tasks drawn from the recent literature, and we showed that GPT-4, together with the prompt we designed, can perform well on many of these tasks.

With today's LLMs and our zero-shot prompt, there are several types of tasks which are too challenging, such as those requiring high precision, complex trajectory shapes, and richer perception beyond just computing bounding boxes. As LLMs continue to improve, we may see improved abilities to solve these tasks, and recent advances in VLMs will help to better interface language and vision. But for now, this paper raises the assumed limit of the utility of LLMs for robotics, and we hope that the insights we provide will help those developing their own LLM-based robot controllers, and will encourage those leading the field in LLMs and VLMs to further align developments with robotics applications.

Acknowledgment

The authors wish to thank Kamil Dreczkowski, Georgios Papagiannis and Pietro Vitiello for their valuable discussion and feedback during the writing of the paper.

REFERENCES

[1] W. X. Zhao *et al.*, "A Survey of Large Language Models," *arXiv e-prints*, p. arXiv:2303.18223, Mar. 2023.

[2] L. Wang *et al.*, "A Survey on Large Language Model based Autonomous Agents," *arXiv e-prints*, p. arXiv:2308.11432, Aug. 2023.

[3] W. Huang *et al.*, "VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models," *arXiv e-prints*, p. arXiv:2307.05973, July 2023.

[4] W. Yu *et al.*, "Language to Rewards for Robotic Skill Synthesis," *arXiv e-prints*, p. arXiv:2306.08647, June 2023.

[5] M. Ahn *et al.*, "Do As I Can, Not As I Say: Grounding Language in Robotic Affordances," *arXiv e-prints*, p. arXiv:2204.01691, Apr. 2022.

[6] OpenAI, "GPT-4 Technical Report," *arXiv e-prints*, p. arXiv:2303.08774, Mar. 2023.

[7] T. Xiao *et al.*, "Robotic Skill Acquisition via Instruction Augmentation with Vision-Language Models," *arXiv e-prints*, p. arXiv:2211.11736, Nov. 2022.

[8] A. Brohan *et al.*, "RT-1: Robotics Transformer for Real-World Control at Scale," *arXiv e-prints*, p. arXiv:2212.06817, Dec. 2022.

[9] T. Yu *et al.*, "Scaling Robot Learning with Semantically Imagined Experience," *arXiv e-prints*, p. arXiv:2302.11550, Feb. 2023.

[10] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo *et al.*, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[11] T. Kojima *et al.*, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems*, S. Koyejo *et al.*, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 22 199–22 213. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf

[12] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," *arXiv e-prints*, p. arXiv:2107.03374, July 2021.

[13] J. Hoffmann *et al.*, "Training Compute-Optimal Large Language Models," *arXiv e-prints*, p. arXiv:2203.15556, Mar. 2022.

[14] J. Liang *et al.*, "Code as policies: Language model programs for embodied control," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, May 2023, pp. 9493–9500.

[15] S. Vemprala *et al.*, "Chatgpt for robotics: Design principles and model abilities," Microsoft, Tech. Rep. MSR-TR-2023-8, February 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/chatgpt-for-robotics-design-principles-and-model-abilities/

[16] A. Brohan *et al.*, "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control," *arXiv e-prints*, p. arXiv:2307.15818, July 2023.

[17] D. Driess *et al.*, "PaLM-e: An embodied multimodal language model," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause *et al.*, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 8469–8488. [Online]. Available: https://proceedings.mlr.press/v202/driess23a.html

[18] J.-B. Alayrac *et al.*, "Flamingo: a visual language model for few-shot learning," in *Advances in Neural Information Processing Systems*, S. Koyejo *et al.*, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 23 716–23 736. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf

[19] N. Di Palo and E. Johns, "Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics," *arXiv e-prints*, p. arXiv:2403.19578, Mar. 2024.

[20] S. Mirchandani *et al.*, "Large Language Models as General Pattern Machines," *arXiv e-prints*, p. arXiv:2307.04721, July 2023.

[21] A. Zeng *et al.*, "Socratic Models: Composing Zero-Shot Multimodal Reasoning with Language," *arXiv e-prints*, p. arXiv:2204.00598, Apr. 2022.

[22] W. Huang *et al.*, "Inner Monologue: Embodied Reasoning through Planning with Language Models," *arXiv e-prints*, p. arXiv:2207.05608, July 2022.

[23] N. Di Palo *et al.*, "Towards A Unified Agent with Foundation Models," *arXiv e-prints*, p. arXiv:2307.09668, July 2023.

[24] W. Shen *et al.*, "Distilled Feature Fields Enable Few-Shot Language-Guided Manipulation," *arXiv e-prints*, p. arXiv:2308.07931, July 2023.

[25] J. Kerr *et al.*, "LERF: Language Embedded Radiance Fields," *arXiv e-prints*, p. arXiv:2303.09553, Mar. 2023.

[26] A. Rashid *et al.*, "Language Embedded Radiance Fields for Zero-Shot Task-Oriented Grasping," *arXiv e-prints*, p. arXiv:2309.07970, Sept. 2023.

[27] N. Di Palo and E. Johns, "DINOBot: Robot Manipulation via Retrieval and Alignment with Vision Foundation Models," *arXiv e-prints*, p. arXiv:2402.13181, Feb. 2024.

[28] ——, "On the Effectiveness of Retrieval, Alignment, and Replay in Manipulation," *arXiv e-prints*, p. arXiv:2312.12345, Dec. 2023.

[29] I. Kapelyukh *et al.*, "DALL-E-Bot: Introducing Web-Scale Diffusion Models to Robotics," *arXiv e-prints*, p. arXiv:2210.02438, Oct. 2022.

[30] ——, "Dream2Real: Zero-Shot 3D Object Rearrangement with Vision-Language Models," *arXiv e-prints*, p. arXiv:2312.04533, Dec. 2023.

[31] L. Medeiros, "Langsam: Language segment-anything," https://github.com/luca-medeiros/lang-segment-anything, accessed: 2023-10-01.

[32] S. Liu *et al.*, "Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection," *arXiv e-prints*, p. arXiv:2303.05499, Mar. 2023.

[33] A. Kirillov *et al.*, "Segment Anything," *arXiv e-prints*, p. arXiv:2304.02643, Apr. 2023.

[34] H. K. Cheng and A. G. Schwing, "Xmem: Long-term video object segmentation with an atkinson-shiffrin memory model," in *Computer Vision – ECCV 2022*, S. Avidan *et al.*, Eds. Cham: Springer Nature Switzerland, 2022, pp. 640–658.

[35] H. Luo *et al.*, "WizardMath: Empowering Mathematical Reasoning for Large Language Models via Reinforced Evol-Instruct," *arXiv e-prints*, p. arXiv:2308.09583, Aug. 2023.

[36] Anthropic, "Model card and evaluations for claude models," https://www-files.anthropic.com/production/images/Model-Card-Claude-2.pdf, accessed: 2023-12-14.

[37] H. Touvron *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv e-prints*, p. arXiv:2307.09288, July 2023.

[38] Anthropic, "The claude 3 model family: Opus, sonnet, haiku," https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf, accessed: 2024-04-11.

[39] Gemini Team, "Gemini: A Family of Highly Capable Multimodal Models," *arXiv e-prints*, p. arXiv:2312.11805, Dec. 2023.

[40] W. Huang *et al.*, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri *et al.*, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 9118–9147. [Online]. Available: https://proceedings.mlr.press/v162/huang22a.html

APPENDIX

A. Related Work

METHOD \ REQUIRES	LLM	vision model	predefined primitives	in-context examples	external trajectory optimisers	robotics-specific training data
VoxPoser	●	●		●	●	
Code as Policies	●	●	●	●		
SayCan	●	●		●		●
Language to Rewards	●	●			●	
ChatGPT for Robotics	●	●	●			
RT-2	●	●				●
Our Investigation	●	●				

Fig. 5: A taxonomy of requirements of LLM-based zero-shot methods from the recent literature.

While prior works have made significant strides in leveraging LLMs for various aspects of robotic control [2], several limitations and dependencies on external modules persist. The core motivation of our work is to **investigate whether these limitations are inherent, or if LLMs can be deployed for low-level control**, going from language to a dense sequence of end-effector poses. In this section, we provide an overview of the relevant literature and highlight key distinctions between prior approaches and our research focus.

LLMs for Robotics: There have been a number of works which leverage the common-sense knowledge and instruction-following capabilities of LLMs, but they have relied on external components for the full low-level trajectory generation. Both Code as Policies [14] and ChatGPT for Robotics [15] rely on predefined motion primitives for robot control, and their focus is predominantly on high-level planning. In the case of SayCan [5], robotics-specific data is required to pre-train the skills. VoxPoser [3] and Language to Rewards [4] have explored the use of LLMs to generate high-reward regions for robot movement, but these methods still necessitate external trajectory optimisers to compute a trajectory, such as cost and reward functions used to evaluate randomly sampled trajectories along with Model Predictive Control (MPC) [3]. VoxPoser [3], Code as Policies [14], and SayCan [5] have also relied heavily on providing in-context examples to the LLM input. However, these methods can encounter challenges when extrapolating beyond the demonstrated tasks. A summary of these works and their required auxiliary components is shown in Fig. 5.

Out of the aforementioned works, Code as Policies [14] is closest to our work. However, instead of relying on predefined primitives and in-context examples, we focus our investigation on the generation of trajectories more complex than a sequence of linear interpolations and without any task-specific guidance, thus broadening the scope of applicability and adaptability in the real world and reducing the reliance on human expertise. We also show that these complex trajectories are generated from their internal knowledge of these tasks, instead of relying solely on the use of external libraries for code generation, and that this understanding is not only beneficial for trajectory generation, but also for task success detection as well, providing insight into the capabilities of LLMs to detect and recover from failures.

Foundation Models for Robotics: Recent works [16], [17] demonstrated that a Vision Language Model (VLM) [18] can be fine-tuned with a large robotics-related dataset of actions to enable zero-shot language-conditioned control. It has also been shown that such demonstrations can be provided directly to LLMs in the form of language tokens as in-context examples [19], [20]. Other works on foundation models include Socratic Models [21], which proposes to leverage a number of them via language-based exchange, and Inner Monologue [22] which focuses on environment feedback for success detection, scene description, and human interaction. VLMs and LLMs have also been used in reinforcement learning settings for their planning and success detection capabilities, especially in long-horizon tasks [23].

Language and Visual Grounding: Although not the main focus of our work, there have been numerous methods to ground the visual information for language-conditioned robot manipulation, such as embedding CLIP features into NeRFs [24], [25], [26], and keypoint extraction for object encoding [19] and demonstration retrieval [27], [28]. DALL-E-Bot [29] introduces web-scale image diffusion models to ground the target rearrangement scene, whereas Dream2Real [30] makes use of CLIP similarity scores to determine the most visually plausible target scene.

B. Real-World Experimental Setup

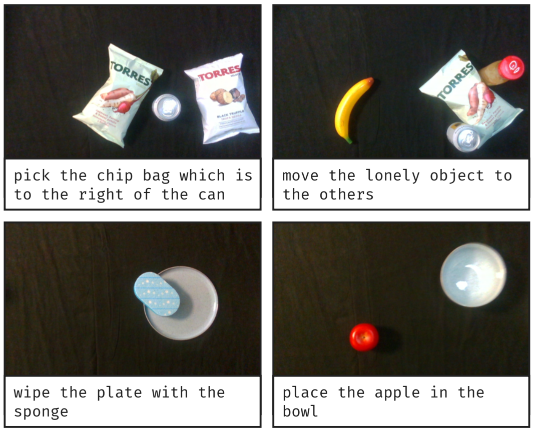

Fig. 6: Example wrist-camera observations received by the robot at the start of each task, and their corresponding task instructions.

We run our experiments on a Sawyer robot equipped with a 2F-85 Robotiq gripper. We use two Intel RealSense D435 RGB-D cameras, one mounted on the wrist of the robot, and the other fixed on a tripod, to observe the environment. The wrist-mounted camera captures a top-down view of the environment at the beginning of an episode (Fig. 6), which is used by a vision model if queried by the LLM. We utilise a pre-trained object detection model, LangSAM [31] (based on Grounding DINO [32] and Segment Anything [33]), and whenever the LLM calls `detect_object`, we automatically calculate 3-D bounding boxes of the queried objects from the segmentation maps returned by LangSAM using the camera calibration, and provide the bounding boxes to the LLM. The LLM then leverages this visual understanding of the environment to predict a sequence of 4-D end-effector poses (3 dimensions for position, 1 dimension for rotation about the vertical axis), as well as either `open_gripper` or `close_gripper` commands. This is then executed by the robot in an open loop, using a position controller to move sequentially between each pose, hence producing a full trajectory. After task execution, we use XMem [34] to track the segmentation maps over the entire duration of the task, which is then later used for detecting if the task was successful or not.

C. Additional Details to Further Investigations

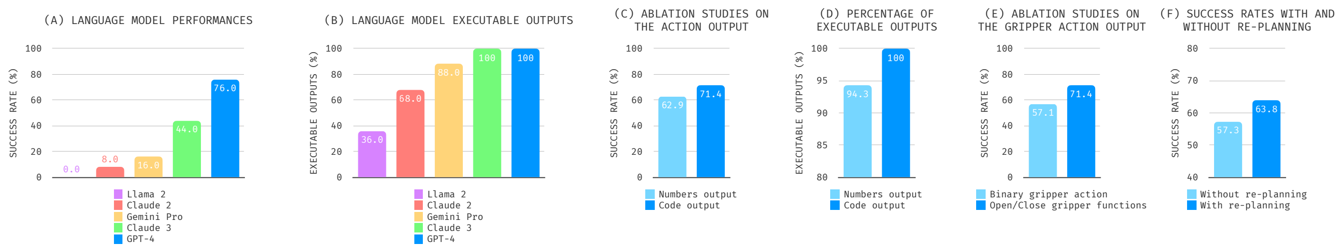

Fig. 7: **(A)** We compare the performance of different widely used LLMs. **(B)** We compare the abilities of these LLMs to generate outputs that are directly executable by the robot. **(C)** We compare different modes for the trajectory output. **(D)** We measure the percentage of control outputs from the LLM that are directly executable by the robot. **(E)** We compare different modes for controlling the gripper. **(F)** We demonstrate the ability of LLMs to detect failures and re-plan autonomously.

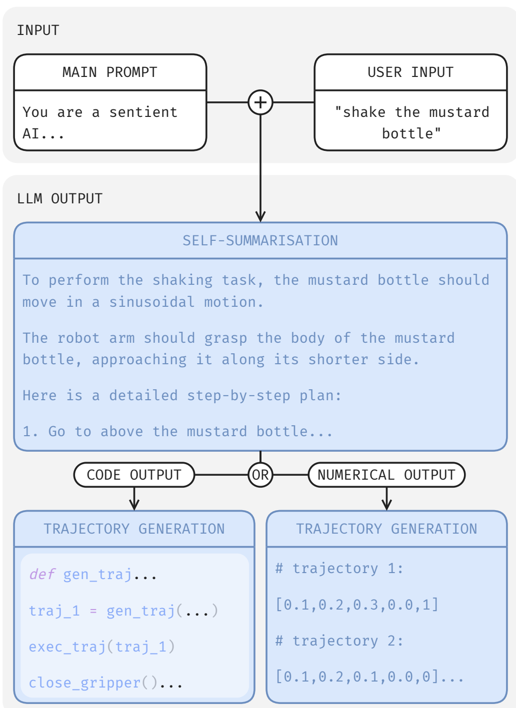

Fig. 8: Given the full main prompt and the user input command, the LLM first outputs a high-level natural language self-summarisation of the trajectory plan, before generating either code which computes and executes the trajectory, or the final trajectory directly as a list of numerical values.

(1) How should the final trajectory be represented? In this set of experiments, we explore the optimal way for the LLM to output the sequence of end-effector poses. Specifically, we conduct ablation studies to evaluate whether this should be represented as a list of *numerical values* or as *code for trajectory generation*. Fig. 8 shows the distinction between these two output modes.

The results, summarised in Fig. 7 C, offer valuable insights. Notably, our investigation shows that outputting code that generates the trajectory outperforms predicting the trajectory directly as an explicit list of numerical poses for the end-effector to follow, represented as language tokens (Fig. 8). In particular, we observe that representing trajectories as numerical values or as code yields similar performance for most tasks, with distinctions emerging in cases involving more intricate trajectories (for example, drawing a circle or a five-pointed star), where outputting code that generates such trajectories prevails (60% success rates for code output compared to 10% for numerical output). This suggests a *fundamental property of LLMs for control*: while not trained on physical interactions and trajectories, their **understanding of both code and mathematical / geometrical structures** [6], [35] **can bridge these two modes of thinking**. Once the overall trajectory shape has been identified by the LLM, while it can be challenging to follow it directly in numbers, it is proficient at generating code that itself can follow complex paths.

Additionally, we study whether directly generating a list of numerical poses, or code that then generates the poses itself, leads to executable outputs more often. Giving low-level control to the LLM poses the risk of the robot receiving wrongly formatted outputs that cannot be executed by the robot. Therefore, in this ablation, we investigate how often the output of the LLM is formatted such that it is executable by the robot. We include prompts instructing the LLM to follow a specific format for the trajectory generation (for the former, we require a list between the ⟨`trajectory`⟩ and ⟨`/trajectory`⟩ tags *without any Python functions*, and for the latter, we require any Python code to be between the ```` ```python ```` and ```` ``` ```` tags). Given the output of the LLM, if an error is thrown during automatic parsing according to this format, we provide the LLM with the error message and ask it to correct the output, for up to three times. Measuring the percentage of executable outputs (Fig. 7 D) demonstrates that outputting code results in 100% of executable trajectories, while direct numerical values cannot be parsed even after three self-corrections for some episodes.

(2) How should the LLM output the gripper action? We also investigate the optimal way of letting the LLM *control the gripper open or close action*: we compare using a binary variable $a \in \{0, 1\}$ or explicit functions `open_gripper`, `close_gripper`. Our results, in Fig. 7 E, demonstrate that the LLM achieves better performance when using explicit functions, while using a binary variable leads to more errors. A notable failure case stemmed from the LLM hard-coding

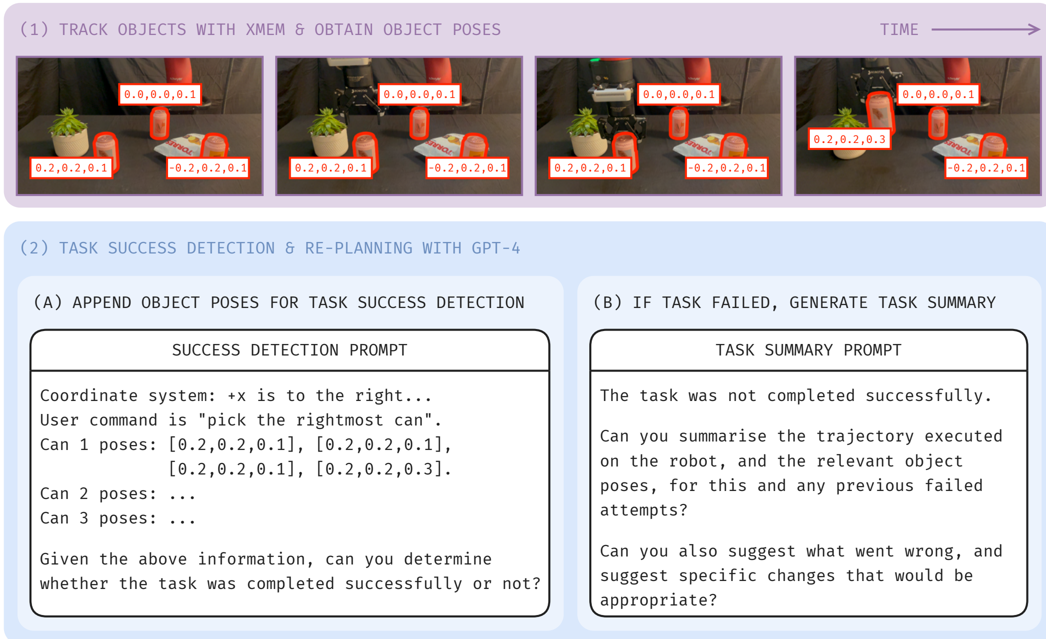

Fig. 9: Our experiments demonstrate that LLMs can interpret the trajectories of objects to detect successful and unsuccessful episodes.

the gripper state to be open in one of the functions it defined for itself, such that when the same function was then used to generate the object approach-and-lift sub-trajectory steps, the gripper failed to close and grasp the object. Having explicit functions to open and close the gripper, on the other hand, allowed a decoupling of these fundamental actions and enabled the correct functions to be called at any time during the overall trajectory generation plan.

(3) Which of the currently available LLMs perform best at following instructions outlined in the prompt? Next, we present results on ablation studies conducted to determine the best-performing LLM on the main prompt. We see in Fig. 7 B that out of the five most popular LLMs currently available for public use to date [6], [36], [37], [38], [39], and under the same constraints as the executable output study presented in Fig. 7 D, only GPT-4 and Claude 3 (Opus) were able to generate code which was able to be executed 100% of the time (regardless of whether the resulting code output completed the task successfully or not), whereas the outputs for Gemini 1.0 Pro, Claude 2 and Llama 2 (70B Chat) were only executable 88.0%, 68.0% and 36.0% of the time, respectively. The corresponding success rates for the five LLMs are shown in Fig. 7 A.

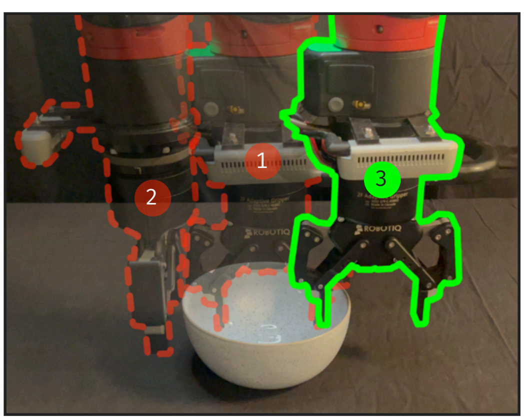

Fig. 10: **(1)** The LLM attempts to grasp the bowl at its centroid, recognises failure, and **(2)** proposes a new trajectory. **(3)** On its third attempt after re-planning again, it successfully grasps the bowl.

(4) Can LLMs recognise unsuccessful trajectories, and adapt their plan? We also delve into the ability of LLMs to recognise and respond to failures during task execution. Our experiments demonstrate that, by analysing the numerical trajectories of objects, LLMs can autonomously detect failure outcomes and initiate re-planning to rectify them. We therefore demonstrate that LLMs possess not only the ability to generate trajectories, but also to **discern whether they represent successful or unsuccessful episodes, given the tasks requested by the user**. Our proposed pipeline for task success detection and re-planning is shown in Fig. 9.

For each of the 5 trials of a task, when a failure is identified, the LLM modifies the original plan to tackle the possible issue. In Fig. 7 F, we demonstrate that this leads to a small improvement in performance, without the need for any human intervention. As a notable example, the LLM always fails at grasping a bowl on its first try (Fig. 3), attempting to grasp it by the centroid (Fig. 10). Through a sequence of two re-planning iterations, however, the LLM adapts its trajectory and then successfully grasps the bowl by its rim, leading to an increase from 0% to 20% in the overall task execution success rate.

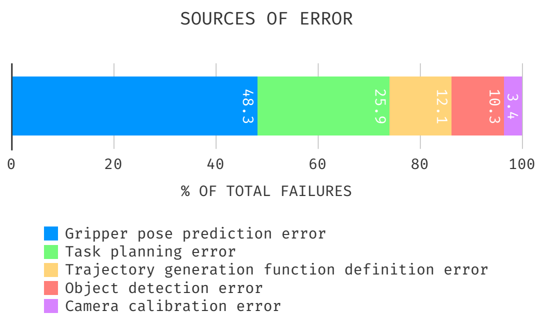

Fig. 11: A breakdown of the different error types which caused task execution failures with the main prompt across the 30 tasks.

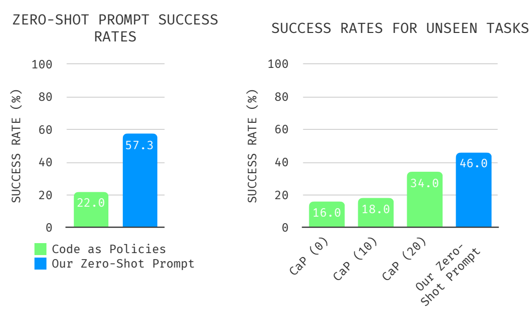

Fig. 12: **(Left)** We compare our zero-shot task performance across 30 tasks to that of Code as Policies (CaP). **(Right)** We demonstrate that our prompt is able to generalise to unseen tasks with higher performance than using CaP-style prompting with a varying number of in-context examples (in brackets).

(5) What were the main failure modes? Finally, we present the main failure modes of the main prompt on the 30 manipulation tasks. We group the sources of error into the following categories: (1) gripper pose prediction error, where the LLM was unable to predict accurate enough goal poses for the gripper, and this resulted in wrong parameterisations of trajectory generation functions (which the LLM defined for itself previously); (2) task planning error, where the LLM was unable to plan a correct high-level sequence of steps to complete the task successfully (for example, calling `close_gripper` before lowering the gripper to the object to grasp it); (3) trajectory generation function definition error, where the LLM coded wrongly the function which would be used to generate the dense sequence of end-effector poses (the function itself would be executable by a Python interpreter, but as opposed to Error **(1)** which was concerned with the parameterisation of such functions, the function itself was wrongly defined; for example, hard-coding the gripper to be open or closed, or its height within the function definition); (4) object detection error, where the wrong segmentation map was returned by the object detector based on the text queried by the LLM; (5) camera calibration error, where the LLM might have generated the correct trajectory for the bounding box it was provided with, but the bounding box itself was incorrectly calculated due to noisy camera data and this resulted in the task being unsuccessful. The full breakdown is shown in Fig. 11.

We can see that nearly half of the failure cases can be attributed to the more difficult task of predicting accurate gripper poses (48.3%), compared to high-level planning (25.9%), of which numerous works have already shown that LLMs are capable [5], [40]. Regarding the object detection errors, most of the errors were due to the object detector not being able to segment the object parts correctly when queried by the LLM, such as the rim of a bowl, or the slot of a toaster.

(6) How does our work compare against Code as Policies? We perform two further sets of experiments in simulation and with ground-truth object poses, comparing our zero-shot task-agnostic prompt against the prompt structure set out in Code as Policies (CaP) [14], which makes use of in-context examples and motion primitives. Firstly, we compare the performance of the original CaP prompt on the same 30 tasks, and find that while for the basic pick-and-place tasks, for which examples have already been provided, it has near-perfect success rates, it is unable to generalise at all to more complex tasks, resulting in a lower average success rate of 22.0%, compared to 57.3% for our prompt.

Secondly, we study the role of CaP-style in-context examples in enabling (or limiting) unseen task generalisation, and compare the success rates of CaP prompts with a varying number of additional manually engineered in-context examples (0, 10, and 20) on 10 unseen tasks. We find that with additional examples, the performance on unseen tasks increases, but even with 20 extra examples covering similar tasks to the unseen ones, the CaP prompts are unable to perform as well (highest 34.0%) as our zero-shot task-agnostic prompt (46.0%). These results are shown in Fig. 12.

D. Prompts and Ablations

```
MAIN PROMPT


You are a sentient AI that can control a robot arm by generating Python code which outputs a list of
trajectory points for the robot arm end-effector to follow to complete a given user command.
Each element in the trajectory list is an end-effector pose, and should be of length 4, comprising a 3D
position and a rotation value.

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. detect_object(object_or_object_part: str) -> None: This function will not return anything, but only print
the position, orientation, and dimensions of any object or object part in the environment. This information
will be printed for as many instances of the queried object or object part in the environment. If there are
multiple objects or object parts to detect, call one function for each object or object part, all before
executing any trajectories. The unit is in metres.
2. execute_trajectory(trajectory: list) -> None: This function will execute the list of trajectory points on
the robot arm end-effector, and will also not return anything.
3. open_gripper() -> None: This function will open the gripper on the robot arm, and will also not return
anything.
4. close_gripper() -> None: This function will close the gripper on the robot arm, and will also not return
anything.
5. task_completed() -> None: Call this function only when the task has been completed. This function will
also not return anything.
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 plan.

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.
The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the rotation value at 0, and
the gripper open.
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 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.
The gripper can only grasp objects along sides which are shorter than 0.08.
Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise
rotation. The rotation values should be in radians.

COLLISION AVOIDANCE:
If the task requires interaction with multiple objects:
1. Make sure to consider the object widths, lengths, and heights so that an object does not collide with
another object or with the tabletop, unless necessary.
2. It may help to generate additional trajectories and add specific waypoints (calculated from the given
object information) to clear objects and the tabletop and avoid collisions, if necessary.

VELOCITY CONTROL:
1. The default speed of the robot arm end-effector is 100 points per trajectory.
2. If you need to make the end-effector follow a particular trajectory more quickly, then generate fewer
points for the trajectory, and vice versa.

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default
speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required.
Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of
trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make
sure to include the orientation parameter.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function
to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which
part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you
obtain the printed outputs from the detect_object function calls.

INITIAL PLANNING 2:
Then, output Python code to decide which object to interact with, if there are multiple instances of the same
object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one
of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make
contact with the object, if necessary.
Finally, perform each of these steps one by one. Name each trajectory variable with the trajectory number.
Stop generation after each code block to wait for it to finish executing before continuing with your plan.

The user command is "[INSERT TASK]".
```

Fig. 13: The full main prompt.

```
MAIN PROMPT ABLATIONS (1)

You are a sentient AI that can control a robot arm by generating Python code which outputs a list of
trajectory points for the robot arm end-effector to follow to complete a given user command.
Each element in the trajectory list is an end-effector pose, and should be of length 4, comprising a 3D
position and a rotation value.

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. detect_object(object_or_object_part: str) -> None: This function will not return anything, but only print
the position, orientation, and dimensions of any object or object part in the environment. This information
will be printed for as many instances of the queried object or object part in the environment. If there are
multiple objects or object parts to detect, call one function for each object or object part, all before
executing any trajectories. The unit is in metres.
2. execute_trajectory(trajectory: list) -> None: This function will execute the list of trajectory points on
the robot arm end-effector, and will also not return anything.
3. open_gripper() -> None: This function will open the gripper on the robot arm, and will also not return
anything.
4. close_gripper() -> None: This function will close the gripper on the robot arm, and will also not return
anything.
5. task_completed() -> None: Call this function only when the task has been completed. This function will
also not return anything.
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 plan.

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.
The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the rotation value at 0, and
the gripper open.
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 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.
The gripper can only grasp objects along sides which are shorter than 0.08.
Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise
rotation. The rotation values should be in radians.

COLLISION AVOIDANCE:
If the task requires interaction with multiple objects:
1. Make sure to consider the object widths, lengths, and heights so that an object does not collide with
another object or with the tabletop, unless necessary.
2. It may help to generate additional trajectories and add specific waypoints (calculated from the given
object information) to clear objects and the tabletop and avoid collisions, if necessary.

VELOCITY CONTROL:
1. The default speed of the robot arm end-effector is 100 points per trajectory.
2. If you need to make the end-effector follow a particular trajectory more quickly, then generate fewer
points for the trajectory, and vice versa.

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default
speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required.
Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of
trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make
sure to include the orientation parameter.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function
to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which
part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you
obtain the printed outputs from the detect_object function calls.

INITIAL PLANNING 2:
Then, output Python code to decide which object to interact with, if there are multiple instances of the same
object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one
of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make
contact with the object, if necessary.
Finally, perform each of these steps one by one. Name each trajectory variable with the trajectory number.
Stop generation after each code block to wait for it to finish executing before continuing with your plan.

The user command is "[INSERT TASK]".
```

```
Remove prompt...

to break down the trajectory into steps
to break down the trajectory generation and execution into steps
to clear objects and the tabletop to avoid collisions
section headings
to describe the shape of the motion trajectory
to describe the part of the object most suitable for interaction
```

Fig. 14: The full prompt with the highlighted sections removed for the ablation studies on the main prompt.

```
MAIN PROMPT ABLATIONS (2)

...

COLLISION AVOIDANCE:
If the task requires interaction with multiple objects:
1. Make sure to consider the object widths, lengths, and heights so that an object does not collide with
another object or with the tabletop, unless necessary.
2. It may help to generate additional trajectories and add specific waypoints (calculated from the given
object information) to clear objects and the tabletop and avoid collisions, if necessary.

VELOCITY CONTROL:
1. The default speed of the robot arm end-effector is 100 points per trajectory.
2. If you need to make the end-effector follow a particular trajectory more quickly, then generate fewer
points for the trajectory, and vice versa.

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default
speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required.
Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of
trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make
sure to include the orientation parameter.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function
to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which
part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you
obtain the printed outputs from the detect_object function calls.

INITIAL PLANNING 2:
Then, output Python code to decide which object to interact with, if there are multiple instances of the same
object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one
of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make
contact with the object, if necessary.
Finally, perform each of these steps one by one. Name each trajectory variable with the trajectory number.
Stop generation after each code block to wait for it to finish executing before continuing with your plan.

The user command is "[INSERT TASK]".
```

```
Remove prompt...

to plan when to lower the gripper to make contact with the object
to define reusable as well as specific functions, and annotate them
to name each trajectory variable with a number for smooth motion
for entire collision avoidance section
to describe how best to approach the object
```

Fig. 15: The full prompt with the highlighted sections removed for the ablation studies on the main prompt (continued).

```
MAIN PROMPT ABLATIONS (3)

...

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default
speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required.
Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of
trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make
sure to include the orientation parameter.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function
to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which
part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you
obtain the printed outputs from the detect_object function calls.

INITIAL PLANNING 2:
Then, output Python code to decide which object to interact with, if there are multiple instances of the same
object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one
of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make
contact with the object, if necessary.
Finally, perform each of these steps one by one. Name each trajectory variable with the trajectory number.
Stop generation after each code block to wait for it to finish executing before continuing with your plan.

The user command is "[INSERT TASK]".
```

```
Remove prompt...

to chain the trajectory for smooth motion
```

Fig. 16: The full prompt with the highlighted sections removed for the ablation studies on the main prompt (continued).

ACTION OUTPUT ABLATION

You are a sentient AI that can control a robot arm by generating ~~Python code which outputs~~ a list of trajectory points for the robot arm end-effector to follow to complete a given user command.
Each element in the trajectory list is an end-effector pose, and should be of length ~~4~~5, comprising a 3D position~~ and a~~, rotation value, and gripper state.

AVAILABLE ~~FUNCTIONS~~ACTIONS:
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~~trajectory list yourself by making use of the available information, common sense, and general knowledge.
You are, however, able to ~~call~~perform any of the following ~~Python functions~~actions, if required, as often as you want:
1. ~~detect_object(object_or_object_part: str) -> None: This function will not return anything, but only print~~ <detect_object>object_or_object_part</detect_object>: This action will output the position, orientation, and dimensions of any object or object part in the environment. This information will be ~~printed~~output for as many instances of the queried object or object part in the environment. If there are multiple objects or object parts to detect, ~~call one function~~perform the action for each object or object part, all before ~~executing~~generating any trajectories. Make sure to stop generation after this action and wait for its output, before continuing with your plan and generating the trajectory list. The unit is in metres.
~~2. execute_trajectory(trajectory: list) -> None: This function will execute the list of trajectory points on the robot arm end-effector, and will also not return anything.~~
~~3. open_gripper() -> None: This function will open the gripper on the robot arm, and will also not return anything.~~
~~4. close_gripper() -> None: This function will close the gripper on the robot arm, and will also not return anything.~~
~~5. task_completed() -> None: Call this function only when the task has been completed. This function will also not return anything.~~
~~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 plan.~~
2. [task_completed]: Perform this action only when the task has been completed.

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.

The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the rotation value at 0, and the gripper open (1).
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 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.
The gripper can only grasp objects along sides which are shorter than 0.08.
Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise rotation. The rotation values should be in radians.

COLLISION AVOIDANCE:
If the task requires interaction with multiple objects:
1. Make sure to consider the object widths, lengths, and heights so that an object does not collide with another object or with the tabletop, unless necessary.
2. It may help to generate additional trajectories and add specific waypoints (calculated from the given object information) to clear objects and the tabletop and avoid collisions, if necessary.

VELOCITY CONTROL:
1. The default speed of the robot arm end-effector is ~~100 points per~~0.01 between each trajectory point.
2. If you need to make the end-effector follow a particular trajectory more quickly, then ~~generate fewer points for the~~increase the distance between each trajectory point, and vice versa.

~~CODE~~TRAJECTORY GENERATION:
When generating the ~~code for the~~ trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
~~2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default speed) should contain at least 100 points. Define general functions which can be reused for the different trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required. Output a step-by-step reasoning before generating the code.~~
~~3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory. Call the execute_trajectory function after each trajectory step.~~
~~4. When defining the functions, specify the required parameters, and document them clearly in the code. Make sure to include the orientation parameter.~~
~~5. If you want to print the calculated value of a variable to use later, make sure to use the print function to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory variables, since the output will be too long.~~
~~6. Mark any code clearly with the ```python and ``` tags.~~
2. The trajectory could be broken down into multiple steps. The distance between each trajectory point (at default speed) should be 0.01. Output a step-by-step reasoning before generating the trajectory list.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of trajectory 2 is the same as the end point of trajectory 1 and so on, to ensure a smooth overall trajectory.

INITIAL PLANNING 1:
If the task requires interaction with an object part (as opposed to the object as a whole), describe which part of the object would be most suitable for the gripper to interact with.
Then, detect the necessary objects in the environment. Stop generation after this step to wait until you obtain the ~~printed~~ outputs from the detect_object ~~function calls~~actions.

INITIAL PLANNING 2:
Then, output ~~Python code~~natural language reasoning to decide which object to interact with, if there are multiple instances of the same object.
Then, describe how best to approach the object (for example, approaching the midpoint of the object, or one of its edges, etc.), depending on the nature of the task, or the object dimensions, etc.
Then, output a detailed step-by-step plan for the trajectory, including when to lower the gripper to make contact with the object, if necessary.
Finally, perform each of these steps one by one.~~ Name each trajectory variable with the trajectory number. Stop generation after each code block to wait for it to finish executing before continuing with your plan.~~
Write out the final trajectory in full in between the <trajectory> and </trajectory> tags, as a Python list. Do not write any Python functions, and do not comment out parts of the trajectory, as these will not be run.

The user command is "[INSERT TASK]".

Parts highlighted in ~~red~~ are removed from the full main prompt.
Parts highlighted in green are added to the full main prompt.

Fig. 17: The full main prompt modified for evaluating the LLM's ability to generate trajectories directly in numbers as language tokens.

```
GRIPPER ACTION ABLATION

You are a sentient AI that can control a robot arm by generating Python code which outputs a list of
trajectory points for the robot arm end-effector to follow to complete a given user command.
Each element in the trajectory list is an end-effector pose, and should be of length 45, comprising a 3D
position and a, rotation value, and gripper state.

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. detect_object(object_or_object_part: str) -> None: This function will not return anything, but only print
the position, orientation, and dimensions of any object or object part in the environment. This information
will be printed for as many instances of the queried object or object part in the environment. If there are
multiple objects or object parts to detect, call one function for each object or object part, all before
executing any trajectories. The unit is in metres.
2. execute_trajectory(trajectory: list) -> None: This function will execute the list of trajectory points on
the robot arm end-effector, and will also not return anything.
3. open_gripper() -> None: This function will open the gripper on the robot arm, and will also not return
anything.
4. close_gripper() -> None: This function will close the gripper on the robot arm, and will also not return
anything.
5. task_completed() -> None: Call this function only when the task has been completed. This function will
also not return anything.
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 plan.

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.
The robot arm end-effector is currently positioned at [INSERT EE POSITION], with the rotation value at 0, and
the gripper open (1).

...

CODE GENERATION:
When generating the code for the trajectory, do the following:
1. Describe briefly the shape of the motion trajectory required to complete the task.
2. The trajectory could be broken down into multiple steps. In that case, each trajectory step (at default
speed) should contain at least 100 points. Define general functions which can be reused for the different
trajectory steps whenever possible, but make sure to define new functions whenever a new motion is required.
Output a step-by-step reasoning before generating the code.
3. If the trajectory is broken down into multiple steps, make sure to chain them such that the start point of
trajectory_2 is the same as the end point of trajectory_1 and so on, to ensure a smooth overall trajectory.
Call the execute_trajectory function after each trajectory step.
4. When defining the functions, specify the required parameters, and document them clearly in the code. Make
sure to include the orientation and gripper state parameters.
5. If you want to print the calculated value of a variable to use later, make sure to use the print function
to three decimal places, instead of simply writing the variable name. Do not print any of the trajectory
variables, since the output will be too long.
6. Mark any code clearly with the ```python and ``` tags.

...
```

```
Parts highlighted in red are removed from the full main prompt.
Parts highlighted in green are added to the full main prompt.
```

Fig. 18: The full main prompt modified for ablation studies on the gripper action output.

```
TASK SUCCESS DETECTION PROMPT


You are tasked with determining whether a user command was completed successfully or not, based on how the
positions and orientations of the relevant objects in the environment changed during the execution of the
task.

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.
The position values are in metres.

The objects can rotate about the z-axis, from -pi to pi radians.
Negative rotation values represent clockwise rotation, and positive rotation values represent anticlockwise
rotation. The rotation values are in radians.

The user command is "[INSERT TASK]".

1. Given the user command, describe how the object positions and orientations should have changed during the
execution of the task.
2. From the given positions and orientations of the relevant objects, output whether the task was completed
successfully or not.
3. If the task was completed successfully, output
```python
task_completed()
```.
4. If the task was not completed successfully, output
```python
task_failed()
```.
Do not define the task_completed and task_failed functions yourself.

The positions and orientations of the relevant objects in the environment are as follows:
```

Fig. 19: Task success detection prompt.

```
TASK FAILURE PROMPT (APPENDED TO THE MAIN PROMPT)


SUMMARY OF PREVIOUS FAILED ATTEMPTS:
[INSERT TASK SUMMARY]

PROBLEM RESOLUTION:
Can you suggest what was wrong with the plans for the trajectories, and suggest specific changes that would
be appropriate?
Then, replan and retry the task by continuing with INITIAL PLANNING 1.
```

Fig. 20: Task re-planning prompt, to be appended to the main prompt if the LLM detects that the task has failed and needs to be re-planned and retried.

```
TASK SUMMARY PROMPT


The task was not completed successfully, and it needs to be replanned and retried. Can you:
1. Summarise the trajectory executed on the robot, specifying key waypoint end-effector poses (positions,
orientations, and gripper states) along the trajectory, and describing their relevance with respect to the
given task.
2. Summarise the most recent positions, orientations, and dimensions of each of the relevant objects.
3. If applicable, include the same details for any given summaries of previous failed attempts as well.
```

Fig. 21: Task summary prompt, as part of task re-planning.

E. Tasks Selected for Ablation Studies

```
TASKS FOR THE MAIN PROMPT ABLATIONS

pick up the chip bag which is to the right of the can
place the apple in the bowl
shake the mustard bottle
open the bottle cap
move the pan to the left
```

Fig. 22: List of tasks selected for ablation studies of the main prompt in Sec. III and of the different LLMs in Sec. IV.

```
TASKS FOR THE ACTION OUTPUT ABLATIONS

pick up the chip bag which is to the right of the can
place the apple in the bowl
shake the mustard bottle
open the bottle cap
move the pan to the left
draw a five-pointed star 10cm wide on the table with a pen
draw a circle 10cm wide with its centre at [0.0,0.3,0.0] with the gripper closed
```

Fig. 23: List of tasks selected for ablation studies of the action output in Sec. IV.

F. Sample LLM Output

MAIN PROMPT

```
...

The user command is "draw a five-pointed star 10cm wide on the table with a pen".
```

GPT-4

````
INITIAL PLANNING 1:

To draw a five-pointed star, the robot arm needs to interact with a pen. The gripper should grasp the pen
at its body, which is usually the thickest part and provides a good grip. Let's detect the pen in the
environment.

```python
# Detect the pen in the environment
detect_object("pen")
```
Stop generation here and wait for the printed outputs from the detect_object function call.
````

PRINT OUTPUT

```
Print statement output:
Position of pen: [0.169, 0.234, 0.129]
Dimensions:
Width: 0.067
Length: 0.016
Height: 0.137
Orientation along shorter side (length): -1.352
Orientation along longer side (width): 0.219
```

Fig. 24: Sample LLM output on the "*draw a five-pointed star 10cm wide on the table with a pen*" task.

GPT-4

INITIAL PLANNING 2:

Since there is only one pen in the environment, we don't need to decide which object to interact with.

The best way to approach the pen is by its midpoint along the length (shorter side), as this will provide the most stable grip.

The detailed step-by-step plan for the trajectory is as follows:

1. Move the end-effector above the pen, aligning the gripper with the pen's orientation along the shorter side.
2. Lower the end-effector to the pen's height.
3. Close the gripper to grasp the pen.
4. Lift the pen slightly above the table.
5. Move the end-effector to the starting point of the star drawing.
6. Lower the pen to make contact with the table.
7. Draw the five-pointed star.
8. Lift the pen off the table.
9. Move the end-effector back to its initial position and release the pen.

Let's start with the first three steps of the plan.

```python
# Define the function to generate a straight-line trajectory
def straight_line_trajectory(start_point, end_point, num_points=100):
    """
    Generate a straight-line trajectory from start_point to end_point.

    Parameters:
    start_point (list): The start point of the trajectory. It is a list of four elements: [x, y, z,
rotation].
    end_point (list): The end point of the trajectory. It is a list of four elements: [x, y, z, rotation].
    num_points (int): The number of points in the trajectory. Default is 100.

    Returns:
    list: A list of points in the trajectory.
    """
    trajectory = []
    for i in range(num_points):
        point = [
            start_point[0] + i * (end_point[0] - start_point[0]) / num_points,
            start_point[1] + i * (end_point[1] - start_point[1]) / num_points,
            start_point[2] + i * (end_point[2] - start_point[2]) / num_points,
            start_point[3] + i * (end_point[3] - start_point[3]) / num_points
        ]
        trajectory.append(point)
    return trajectory

# Define the pen's position and orientation
pen_position = [0.169, 0.234, 0.129]
pen_orientation = -1.352

# Define the start and end points of the first trajectory
start_point_1 = [0.0, 0.0, 0.7, 0]
end_point_1 = [pen_position[0], pen_position[1], 0.7, pen_orientation]

# Generate the first trajectory
trajectory_1 = straight_line_trajectory(start_point_1, end_point_1)

# Execute the first trajectory
execute_trajectory(trajectory_1)

# Define the start and end points of the second trajectory
start_point_2 = end_point_1
end_point_2 = [pen_position[0], pen_position[1], pen_position[2], pen_orientation]

# Generate the second trajectory
trajectory_2 = straight_line_trajectory(start_point_2, end_point_2)

# Execute the second trajectory
execute_trajectory(trajectory_2)

# Close the gripper to grasp the pen
close_gripper()
```
Stop generation here and wait for the code to finish executing before continuing with the plan.

Fig. 25: Sample LLM output on the "*draw a five-pointed star 10cm wide on the table with a pen*" task (continued).

GPT-4

Great, now that the pen has been grasped, let's proceed with the next steps of the plan.

4. Lift the pen slightly above the table.
5. Move the end-effector to the starting point of the star drawing.
6. Lower the pen to make contact with the table.

The starting point of the star drawing will be at the center of the star. Let's assume the center of the star is at position [0.3, 0.3, 0.129] on the table.

```python
# Define the start and end points of the third trajectory
start_point_3 = end_point_2
end_point_3 = [pen_position[0], pen_position[1], 0.7, pen_orientation]

# Generate the third trajectory
trajectory_3 = straight_line_trajectory(start_point_3, end_point_3)

# Execute the third trajectory
execute_trajectory(trajectory_3)

# Define the start and end points of the fourth trajectory
start_point_4 = end_point_3
end_point_4 = [0.3, 0.3, 0.7, 0]

# Generate the fourth trajectory
trajectory_4 = straight_line_trajectory(start_point_4, end_point_4)

# Execute the fourth trajectory
execute_trajectory(trajectory_4)

# Define the start and end points of the fifth trajectory
start_point_5 = end_point_4
end_point_5 = [0.3, 0.3, 0.129, 0]

# Generate the fifth trajectory
trajectory_5 = straight_line_trajectory(start_point_5, end_point_5)

# Execute the fifth trajectory
execute_trajectory(trajectory_5)
```

Stop generation here and wait for the code to finish executing before continuing with the plan.

Fig. 26: Sample LLM output on the "*draw a five-pointed star 10cm wide on the table with a pen*" task (continued).

GPT-4

Now that the pen is in contact with the table at the starting point of the star, let's proceed with the next step of the plan.

7. Draw the five-pointed star.

The five-pointed star can be drawn by connecting five points in a certain order. The five points form a pentagon, and the distance from the center of the star to each point is half the width of the star. In this case, the width of the star is 10cm, so the distance is 5cm or 0.05m.

Let's calculate the positions of the five points. The points are evenly distributed around a circle, so the angle between each point is 2π/5 radians.

```python
import math

# Define the function to calculate the position of a point on a circle
def point_on_circle(center, radius, angle):
    """
    Calculate the position of a point on a circle.

    Parameters:
    center (list): The center of the circle. It is a list of two elements: [x, y].
    radius (float): The radius of the circle.
    angle (float): The angle of the point on the circle, in radians. The angle is measured
counterclockwise from the positive x-axis.

    Returns:
    list: The position of the point. It is a list of two elements: [x, y].
    """
    x = center[0] + radius * math.cos(angle)
    y = center[1] + radius * math.sin(angle)
    return [x, y]

# Define the center and radius of the star
star_center = [0.3, 0.3]
star_radius = 0.05

# Calculate the positions of the five points
points = []
for i in range(5):
    angle = 2 * math.pi * i / 5
    point = point_on_circle(star_center, star_radius, angle)
    points.append(point)

print("Positions of the five points:", points)
```

Stop generation here and wait for the printed output from the print function call.

Fig. 27: Sample LLM output on the "*draw a five-pointed star 10cm wide on the table with a pen*" task (continued).

PRINT OUTPUT

```
Print statement output:
Positions of the five points: [[0.35, 0.3], [0.3154508497187474, 0.34755282581475766],
[0.25954915028125264, 0.32938926261462365], [0.2595491502812526, 0.27061073738537633],
[0.3154508497187474, 0.2524471741852423]]
```

GPT-4

Now that we have the positions of the five points, let's draw the five-pointed star. The star is drawn by connecting the points in the order 1-3-5-2-4-1.

```python
# Define the function to generate a trajectory for drawing a line
def draw_line(start_point, end_point):
    """
    Generate a trajectory for drawing a line from start_point to end_point.

    Parameters:
    start_point (list): The start point of the line. It is a list of two elements: [x, y].
    end_point (list): The end point of the line. It is a list of two elements: [x, y].

    Returns:
    list: A list of points in the trajectory.
    """
    start_point_3d = [start_point[0], start_point[1], 0.129, 0]
    end_point_3d = [end_point[0], end_point[1], 0.129, 0]
    return straight_line_trajectory(start_point_3d, end_point_3d, num_points=50)

# Define the order of the points
order = [0, 2, 4, 1, 3, 0]

# Draw the star
for i in range(5):
    start_point = points[order[i]]
    end_point = points[order[i+1]]
    trajectory = draw_line(start_point, end_point)
    execute_trajectory(trajectory)
```

Stop generation here and wait for the code to finish executing before continuing with the plan.

GPT-4

Now that the star has been drawn, let's proceed with the final steps of the plan.

8. Lift the pen off the table.
9. Move the end-effector back to its initial position and release the pen.

```python
# Define the start and end points of the seventh trajectory
start_point_7 = [points[order[5]][0], points[order[5]][1], 0.129, 0]
end_point_7 = [points[order[5]][0], points[order[5]][1], 0.7, 0]

# Generate the seventh trajectory
trajectory_7 = straight_line_trajectory(start_point_7, end_point_7)

# Execute the seventh trajectory
execute_trajectory(trajectory_7)

# Define the start and end points of the eighth trajectory
start_point_8 = end_point_7
end_point_8 = [0.0, 0.0, 0.7, 0]

# Generate the eighth trajectory
trajectory_8 = straight_line_trajectory(start_point_8, end_point_8)

# Execute the eighth trajectory
execute_trajectory(trajectory_8)

# Open the gripper to release the pen
open_gripper()

# The task has been completed
task_completed()
```

Stop generation here and wait for the code to finish executing. The task has been completed.

Fig. 28: Sample LLM output on the "*draw a five-pointed star 10cm wide on the table with a pen*" task (continued).