

---

# Language Models as Zero-Shot Trajectory Generators

---

Teyun Kwon, Norman Di Palo, Edward Johns

The Robot Learning Lab  
Department of Computing  
Imperial College London

{john.kwon20, n.di-palo20, e.johns}@imperial.ac.uk

## Abstract

Large Language Models (LLMs) have recently shown promise as high-level planners for robots. However, it is often assumed that LLMs do not possess sufficient knowledge to be used directly for the low-level skills themselves. In this work, we now address this assumption thoroughly, and study how well a single task-agnostic prompt, without any in-context examples, can directly predict dense trajectories for 26 real-world language-based tasks, such as “open the bottle cap” and “wipe the plate with the sponge”. Videos are available at: <https://www.robot-learning.uk/language-models-trajectory-generators>.

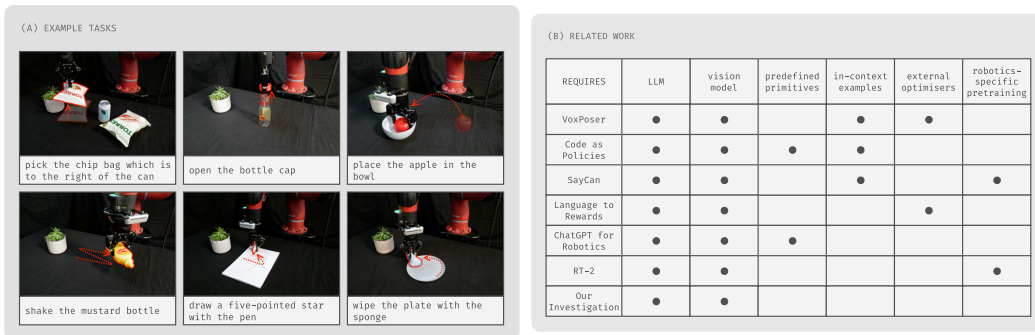


Figure 1: (A) A selection of the tasks which we use to study and evaluate the performance of a single, task-agnostic LLM prompt, without any in-context examples. (B) A taxonomy of requirements of LLM-based zero-shot methods from the recent literature. We investigate whether LLMs can predict trajectories for a range of tasks, assuming access to only vision models.

## 1 Introduction

In recent years, Large Language Models (LLMs) have attracted significant attention and acclaim for their remarkable capabilities in reasoning about common, everyday tasks [Brown et al., 2020, OpenAI, 2023, Touvron et al., 2023, Anil et al., 2023]. This widespread recognition has since led to efforts in the robotics community to adopt LLMs for high-level task planning [Wang et al., 2023]. 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. 1 B). Given the lack of exposure of LLMs to physical interaction data, it is often assumed that LLMs are incapable of low-level control [Huang et al., 2023, Yu et al., 2023a, Ahn et al., 2022].

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 as full-stack zero-shot dense trajectory generators for robot manipulators, without the need for the aforementioned auxiliary components. We provide an LLM with access to off-the-shelf object detection and segmentation models, and then require all remaining reasoning for trajectory generation to be performed by the LLM itself. We also require that the same task-agnostic prompt is used for all tasks, without any in-context examples.

Given these requirements, we studied whether a prompt could be designed to solve a range of tasks taken from the recent literature, such as “open the bottle cap” and “wipe the plate with the sponge”. And through this investigation, 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 trajectory of poses**, without the need for pre-trained skills, motion primitives, trajectory optimisers, or in-context examples.

## 2 Problem Formulation

**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 optimisers: the LLM should output the *full trajectory 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 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 be reproduced effortlessly even with limited compute budget. Details of the real-world experimental setup are presented in Appendix D.

**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 26 everyday manipulation tasks from recent robotics papers** published at leading conferences [Ahn et al., 2022, Xiao et al., 2022, Brohan et al., 2022, Yu et al., 2023b], often tackled by relying on hundreds of manual demonstrations. Success criteria are human-evaluated and designed to mirror those proposed in the original papers. We randomise the positions and orientations of the objects for each of the five test trials, provide the task description in natural language to the LLM, and then no additional human feedback or intervention is allowed. The full list of tasks is shown in Appendix B.

## 3 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 trajectory of poses for the end-effector to follow. We include details fundamental to all tasks, such as coordinate definitions, as well as functions available for the LLM to call, such as `detect_object`. 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 E.

We evaluated the full prompt on a set of diverse tasks from the recent literature, such as grasping, rearranging, drawing, wiping, and more. The task success rates, shown in Appendix B, reveal that LLMs, when equipped with an off-the-shelf vision model and no external motion primitives, policies, or optimisers, exhibit notable proficiency in executing the majority of these tasks zero-shot, by directly predicting a dense trajectory of end-effector poses.

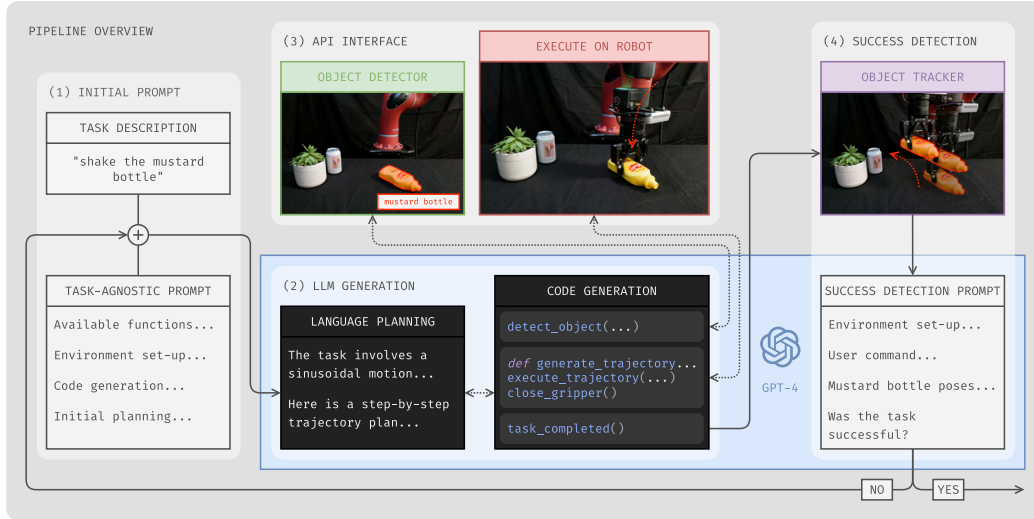


Figure 2: An overview of the pipeline. (1) The main prompt along with the task instruction is provided to the LLM, from which it (2) generates high-level natural language reasoning steps before outputting Python code (3) to interface with a pre-trained object detection model and execute the generated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

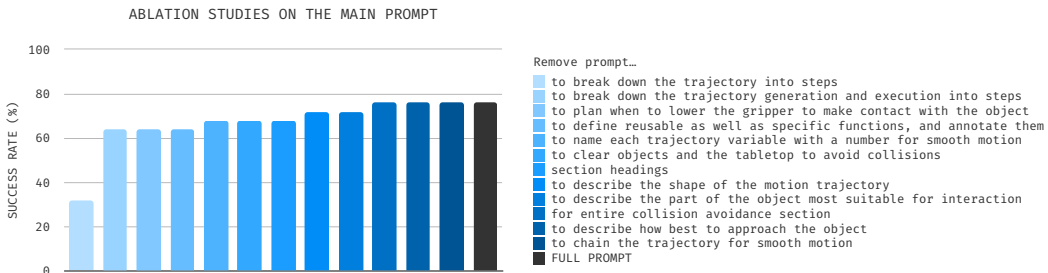


Figure 3: We investigate the effect of removing parts of the main prompt on task success rates.

**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. 3). The tasks used for these ablation studies are shown in Fig. 20 in Appendix F.

**(1) LLMs often require step-by-step reasoning to solve complex or long-horizon tasks.** Prior work has shown that the reasoning capabilities of LLMs can be improved by asking them to break down the task at hand in a step-by-step manner [Wei et al., 2022, Kojima et al., 2022], and adopting this strategy, we prompt the LLM to break down the trajectory into a sequence of sub-trajectory steps. We find that, without including these prompt components, the LLM often omits key trajectory steps required to execute the task successfully, such as lowering the gripper to reach and grasp the object, and opening and closing the gripper. Indeed, the first three columns in Fig. 3 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 improvements have been made in the domain of code generation by LLMs [Chen et al., 2021, OpenAI, 2023], 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 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 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. 3).

**(3) LLMs are trained on limited grounded physical interaction data.** Due to the scarcity of grounded physical interaction data in their training corpora [Hoffmann et al., 2022], 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, 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. 3), LLMs do possess some inherent understanding of possible collisions between different objects (tenth column in Fig. 3).

**(4) LLMs often fail to reason about complex trajectories.** In a manner similar to the first challenge, we employ a multi-step strategy, where initially, we 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 trajectory required to execute the task. This has been shown to be beneficial in prior work [Yu et al., 2023a], and indeed this result is also reflected in the eighth column in Fig. 3.

**(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. 3 that this results in more tasks being executed successfully.

## 4 Further Investigations

In this section, we conduct further ablation studies regarding **(1)** 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), **(2)** the extent to which each output modality is executable by the robot, and **(3)** the ability of LLMs to detect whether a task was executed successfully or not and subsequently re-plan the trajectory. We present the full details of these investigations in Appendix C. The tasks used for these ablation studies are shown in Fig. 21 in Appendix F.

## Acknowledgments

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

- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL [https://proceedings.neurips.cc/paper\\_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf).
- OpenAI. GPT-4 Technical Report. *arXiv e-prints*, art. arXiv:2303.08774, March 2023. doi: 10.48550/arXiv.2303.08774.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv e-prints*, art. arXiv:2307.09288, July 2023. doi: 10.48550/arXiv.2307.09288.
- Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. PaLM 2 Technical Report. *arXiv e-prints*, art. arXiv:2305.10403, May 2023. doi: 10.48550/arXiv.2305.10403.
- Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A Survey on Large Language Model based Autonomous Agents. *arXiv e-prints*, art. arXiv:2308.11432, August 2023. doi: 10.48550/arXiv.2308.11432.
- Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. *arXiv e-prints*, art. arXiv:2307.05973, July 2023. doi: 10.48550/arXiv.2307.05973.
- Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to Rewards for Robotic Skill Synthesis. *arXiv e-prints*, art. arXiv:2306.08647, June 2023a. doi: 10.48550/arXiv.2306.08647.
- Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. *arXiv e-prints*, art. arXiv:2204.01691, April 2022. doi: 10.48550/arXiv.2204.01691.
- Ted Xiao, Harris Chan, Pierre Sermanet, Ayzaan Wahid, Anthony Brohan, Karol Hausman, Sergey Levine, and Jonathan Tompson. Robotic Skill Acquisition via Instruction Augmentation with Vision-Language Models. *arXiv e-prints*, art. arXiv:2211.11736, November 2022. doi: 10.48550/arXiv.2211.11736.
- Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. RT-1: Robotics Transformer for Real-World Control at Scale. *arXiv e-prints*, art. arXiv:2212.06817, December 2022. doi: 10.48550/arXiv.2212.06817.
- Tianhe Yu, Ted Xiao, Austin Stone, Jonathan Tompson, Anthony Brohan, Su Wang, Jaspier Singh, Clayton Tan, Dee M, Jodilyn Peralta, et al. Scaling Robot Learning with Semantically Imagined Experience. *arXiv e-prints*, art. arXiv:2302.11550, February 2023b. doi: 10.48550/arXiv.2302.11550.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in*

- Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022. URL [https://proceedings.neurips.cc/paper\\_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf).
- Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc., 2022. URL [https://proceedings.neurips.cc/paper\\_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf).
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models Trained on Code. *arXiv e-prints*, art. arXiv:2107.03374, July 2021. doi: 10.48550/arXiv.2107.03374.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training Compute-Optimal Large Language Models. *arXiv e-prints*, art. arXiv:2203.15556, March 2022. doi: 10.48550/arXiv.2203.15556.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500, May 2023. doi: 10.1109/ICRA48891.2023.10160591.
- Sai Vemprala, Rogerio Bonatti, Arthur Buckner, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. Technical Report MSR-TR-2023-8, Microsoft, February 2023. URL <https://www.microsoft.com/en-us/research/publication/chatgpt-for-robotics-design-principles-and-model-abilities/>.
- J. Koenemann, A. Del Prete, Y. Tassa, E. Todorov, O. Stasse, M. Bennewitz, and N. Mansard. Whole-body model-predictive control applied to the hrp-2 humanoid. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3346–3351, Sep. 2015. doi: 10.1109/IROS.2015.7353843.
- Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012. doi: 10.1109/IROS.2012.6386109.
- Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. *arXiv e-prints*, art. arXiv:2307.15818, July 2023. doi: 10.48550/arXiv.2307.15818.
- Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. PaLM-e: An embodied multimodal language model. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 8469–8488. PMLR, 23–29 Jul 2023. URL <https://proceedings.mlr.press/v202/driess23a.html>.
- Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 23716–23736. Curran Associates, Inc., 2022. URL [https://proceedings.neurips.cc/paper\\_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf).

- Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. WizardMath: Empowering Mathematical Reasoning for Large Language Models via Reinforced Evol-Instruct. *arXiv e-prints*, art. arXiv:2308.09583, August 2023. doi: 10.48550/arXiv.2308.09583.
- Luca Medeiros. Langsam: Language segment-anything. <https://github.com/luca-medeiros/lang-segment-anything>, 2023. Accessed: 2023-10-01.
- Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, and Lei Zhang. Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection. *arXiv e-prints*, art. arXiv:2303.05499, March 2023. doi: 10.48550/arXiv.2303.05499.
- Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, et al. Segment Anything. *arXiv e-prints*, art. arXiv:2304.02643, April 2023. doi: 10.48550/arXiv.2304.02643.
- Ho Kei Cheng and Alexander G. Schwing. Xmem: Long-term video object segmentation with an atkinson-shiffrin memory model. In Shai Avidan, Gabriel Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner, editors, *Computer Vision – ECCV 2022*, pages 640–658, Cham, 2022. Springer Nature Switzerland. ISBN 978-3-031-19815-1.

## A Related Work

While previous works have made significant strides in leveraging LLMs for various aspects of robotic control [Wang et al., 2023], 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 to encompass the full control stack**, going from language to a dense trajectory 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.

**Predefined Motion Primitives** A subset of prior works, including Code as Policies [Liang et al., 2023] and ChatGPT for Robotics [Vemprala et al., 2023], have predominantly employed LLMs to address the high-level planning aspect of robotic control. These approaches often rely on predefined movement primitives or pre-trained skills (such as SayCan [Ahn et al., 2022]) to execute lower-level actions, thereby only partially solving the control stack. In contrast, our investigation aims to push these boundaries by demonstrating that LLMs can delve deeper into the control stack, predicting all lower-level actions for the robot autonomously, in the form of a dense trajectory of poses.

**External Trajectory Optimisers** VoxPoser [Huang et al., 2023] and Language to Rewards [Yu et al., 2023a] have explored the use of LLMs to shape optimal regions for robot movement, significantly contributing to trajectory planning. However, these methods still necessitate external optimisers, such as Model Predictive Control (MPC) [Koenemann et al., 2015, Todorov et al., 2012], to compute the precise trajectory of actions. Our research deviates from this paradigm by showcasing that LLMs are capable of autonomously shaping and generating their own trajectories. This includes the generation of trajectories as lists of end-effector positions and orientations computed as language tokens or the prediction of code snippets that can generate these trajectories, thus removing the reliance on external optimisers.

**Use of In-Context Examples** Previous approaches such as VoxPoser [Huang et al., 2023], Code as Policies [Liang et al., 2023], and SayCan [Ahn et al., 2022] have relied heavily on providing in-context examples to the LLM input. However, these methods can encounter challenges when extrapolating beyond the demonstrated tasks. In contrast, our research illustrates that, even by relying on their internal understanding alone, LLMs exhibit the capacity to comprehend and solve a diverse range of manipulation tasks, thus broadening the scope of applicability and adaptability in the real world and reducing the reliance on human expertise.

**Robotics-Specific Pre-Training and Fine-Tuning** Recently, Brohan et al. [2023], Driess et al. [2023] demonstrated that a Vision Language Model (VLM) [Alayrac et al., 2022] can be fine-tuned on a large robotics-related data set of actions to enable zero-shot language-conditioned control. However, both the VLM weights and the compute capacity to fine-tune them are unavailable to most research groups: therefore, we focus our investigation on widely available LLMs and vision models [OpenAI, 2023], and tackle many tasks from the recent literature not included in the work by Brohan et al. [2023].

In summary, while prior research has made notable strides in harnessing LLMs for robotics, often focusing on specific components of the control stack or relying on external modules, our investigation represents a departure from these paradigms. We explore the potential of LLMs to provide end-to-end solutions, encompassing the entire control stack from language comprehension to the prediction of dense robot trajectories. This approach not only expands the capabilities of LLM-guided robotics, but it also offers a promising avenue for enhancing human-robot interaction and task execution.



## B Evaluation Results on the Final Prompt Formulation

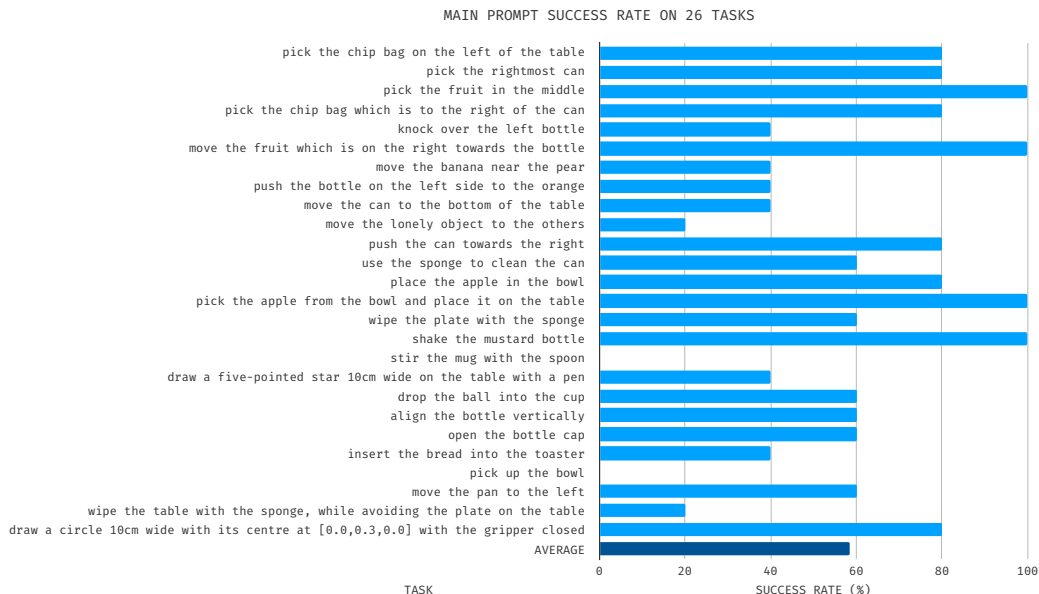


Figure 4: Success rates of the best-performing prompt in our investigation on 26 manipulation tasks.

## C Additional Details to Further Investigations

**(1) How should we guide the output of the LLM? And how should the final trajectory be represented?** In this set of experiments, we explore the optimal way to guide the LLM to output the final trajectories. Specifically, we conduct ablation studies to evaluate the role of two key factors: **(1)** whether the LLM outputs *a series of internal high-level natural language reasoning steps*, as explored in Sec. 3 and following prior works [Wei et al., 2022, Kojima et al., 2022, Brohan et al., 2023], or the *final trajectory directly*; and **(2)** whether the trajectory is presented as a list of *numerical values* or as *code for trajectory generation*.

The results, summarised in Fig. 5 A, offer valuable insights. Notably, our findings for **(1)** align with the recent literature [Wei et al., 2022, Kojima et al., 2022, Chen et al., 2021], highlighting the efficacy of an intermediate, internal reasoning in facilitating task success. Regarding **(2)**, our investigation shows that outputting code that generates the trajectory outperforms predicting the

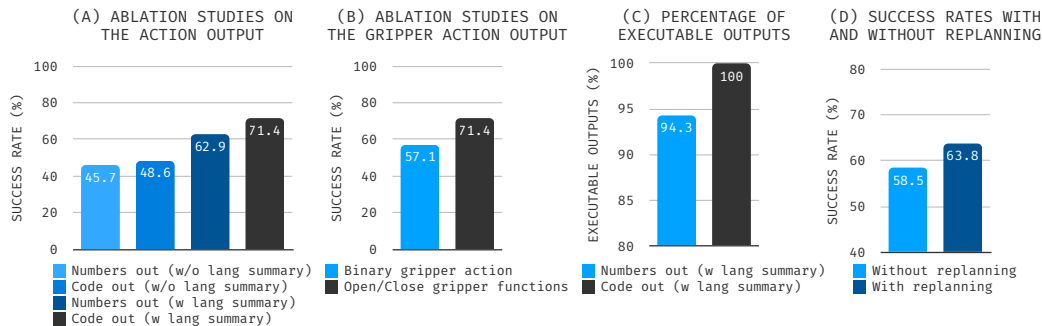


Figure 5: **(A)** We compare different output modalities and the use of internal language reasoning for planning. **(B)** We compare different modalities for controlling the gripper. **(C)** We measure the percentage of control outputs from the LLM that are directly executable by the robot. **(D)** We demonstrate the ability of LLMs to detect failures and re-plan autonomously.

trajectory directly as language tokens in the form of a list of numerical poses for the end-effector to follow. In particular, we observe that presenting 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 code, mathematical and geometrical structures** [OpenAI, 2023, Luo et al., 2023] **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 follows the desired paths.

Additionally, we 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. 5 B, 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 the gripper state to be open in one of the functions it defined for itself, and when the same function was 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 them to be called at any time during the overall trajectory generation plan.

**(2) How often is the output of the LLM formatted such that it is executable by the robot?** Giving full-stack control to the LLM poses the risk of receiving wrongly formatted outputs that cannot be executed by the robot. In this ablation, we study whether generating a list of numerical poses, or code that generates the desired trajectory, leads to executable outputs more often. If an error is thrown by our pipeline given the output of the LLM, 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. 5 C) 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.

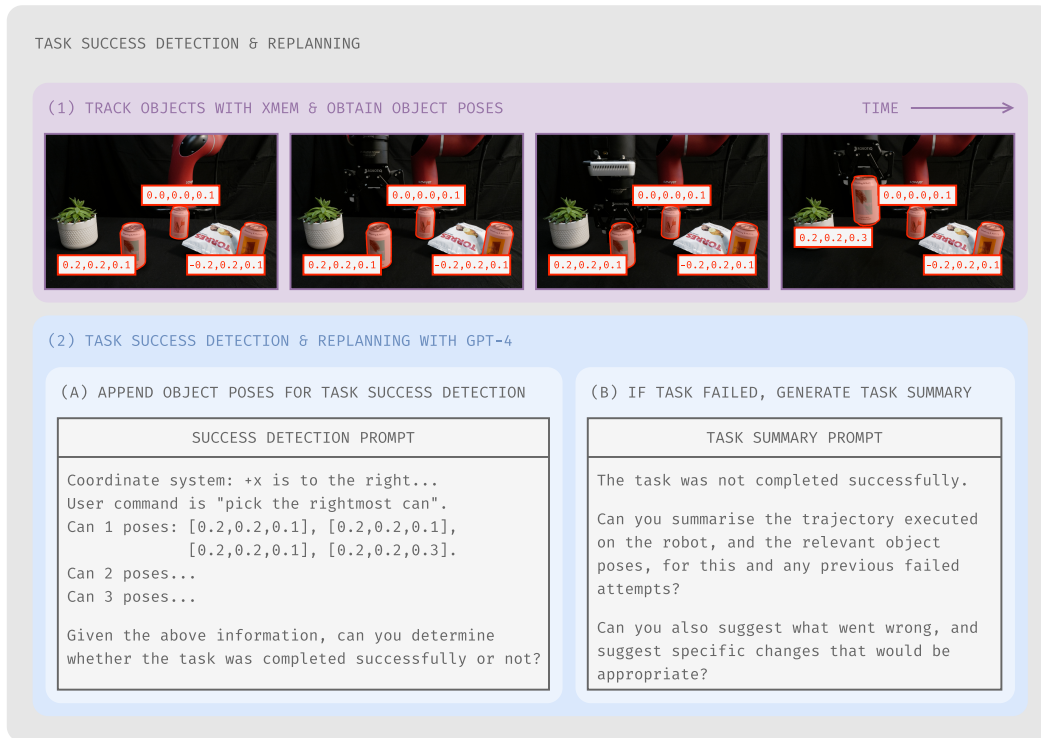


Figure 6: Our experiments demonstrate that LLMs can interpret the trajectories of objects to detect successful and unsuccessful episodes.

**(3) Can LLMs recognise unsuccessful trajectories, and adapt their plan?** Next, we delve into the ability of LLMs to recognise and respond to failures during task execution, as shown in Fig. 6. 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**.

When a failure is identified, the LLM modifies the original plan to tackle the possible issue. In Fig. 5 D, we demonstrate that this leads to improved performance on several tasks, without the need for any human intervention. As a notable example, the LLM always fails at grasping a bowl on its first try (Fig. 4), attempting to grasp it by the centroid (Fig. 7). 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.

## D Real-World Experimental Setup

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. 8), which is used by a vision model if queried by the LLM. We utilise a pre-trained object detection model, LangSAM [Medeiros, 2023] (based on Grounding DINO [Liu et al., 2023]) and Segment Anything [Kirillov et al., 2023]), and 3-D bounding boxes of the queried objects are calculated from the segmentation maps returned by LangSAM and provided to the LLM. The LLM then leverages this visual understanding of the environment to plan and then execute the trajectory in an open loop, and XMem [Cheng and Schwing, 2022] is used 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. We define a trajectory as a list of poses for the end-effector to follow. More concretely, the LLM should generate a list of 4-D poses  $x, y, z, \theta_z$  (three dimensions for position, one dimension for rotation), as well as either an `open_gripper` or a `close_gripper` command. We use GPT-4 [OpenAI, 2023] for the LLM.

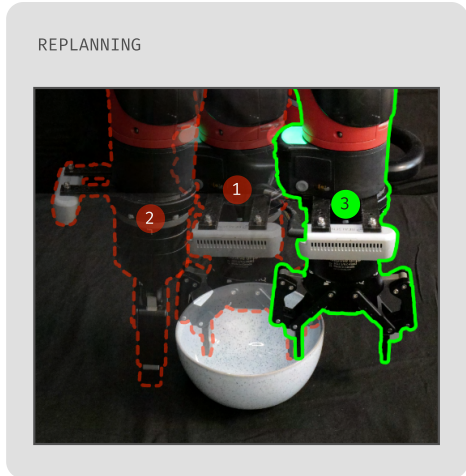


Figure 7: (1) The LLM attempts to grasp the bowl by the centroid, recognises failure, and (2) proposes a new plan. (3) On its third attempt, it successfully grasps the bowl.

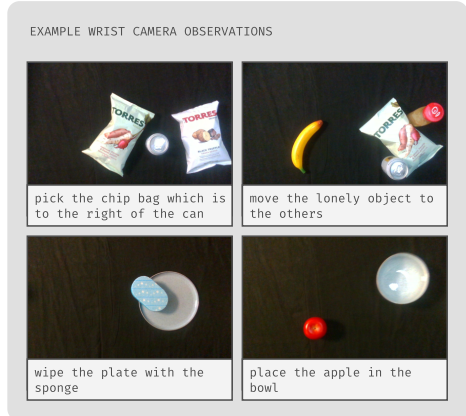


Figure 8: Example observations received by the robot at the start of the task.

## E 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  $\theta$ , 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]".

Figure 9: 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
```

Figure 10: 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

```

Figure 11: 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

```

Figure 12: The full prompt with the highlighted sections removed for the ablation studies on the main prompt (continued).

```

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 5, 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.
...

```

Figure 13: The full main prompt modified for ablation studies on the gripper action output.

```

ACTION OUTPUT ABLATIONS (Numbers out w/o lang summary)

You are a sentient AI that can control a robot arm by generating 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 5, comprising a 3D position, rotation value, and gripper state.

AVAILABLE 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 trajectory list yourself by making use of the available information, common sense, and general knowledge.
You are, however, able to perform any of the following actions, if required, as often as you want:
1. detect_object(object_or_object_parts/detect_objects): This action will output the position, orientation, and dimensions of any object or object part in the environment. This information will be 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, perform the action for each object or object part, all before 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. [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 0.01 between each trajectory point.
2. If you need to make the end-effector follow a particular trajectory more quickly, then increase the distance between each trajectory point, and vice versa.

TRAJECTORY GENERATION:
When generating the trajectory, do the following:
1. The trajectory could be broken down into multiple steps. The distance between each trajectory point (at default speed) should be 0.01.
2. 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:
Detect the necessary objects in the environment. Stop generation after this step to wait until you obtain the outputs from the detect_object actions.

INITIAL PLANNING 2:
Then, generate the trajectory list, including lowering the gripper to make contact with the object, if necessary.

The user command is "[INSERT TASK]".

```

Figure 14: The full prompt used for evaluating the LLM's ability to generate trajectories directly in numbers as language tokens, without generating high-level natural language reasoning steps.

```

ACTION OUTPUT ABLATIONS (Code out w/o lang summary)

...

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, and execute it on the robot.
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]".

```

Figure 15: The full main prompt modified for evaluating the LLM’s ability to generate trajectories directly in Python code, without generating high-level natural language reasoning steps.

```

ACTION OUTPUT ABLATIONS (Numbers out w lang summary)

You are a sentient AI that can control a robot arm by generating 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 5, comprising a 3D position, rotation value, and gripper state.

AVAILABLE 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 trajectory list yourself by making use of the available information, common sense, and general knowledge.
You are, however, able to perform any of the following actions, if required, as often as you want:
1. <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 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, perform the action for each object or object part, all before 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. [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 0.01 between each trajectory point.
2. If you need to make the end-effector follow a particular trajectory more quickly, then increase the distance between each trajectory point, and vice versa.

TRAJECTORY GENERATION:
When generating 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. 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 outputs from the detect_object actions.

INITIAL PLANNING 2:
Then, output 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.

The user command is "[INSERT TASK]".

```

Figure 16: The full prompt used for evaluating the LLM’s ability to generate trajectories directly in numbers as language tokens, after having generated high-level natural language reasoning steps.



#### 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:

Figure 17: Task success detection prompt.

#### 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.

Figure 18: Task summary prompt, as part of task re-planning.

#### 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.

Figure 19: 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.

## F Tasks 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
```

Figure 20: List of tasks selected for the main prompt ablation studies.

```
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
```

Figure 21: List of tasks selected for the action output ablation studies.

## G 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
```

Figure 22: 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.
```

Figure 23: 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.
```

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  $2\pi/5$  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.
```

Figure 24: 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.
```

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