How to Prompt Your Robot: A PromptBook for Manipulation Skills with Code as Policies

Montserrat Gonzalez Arenas, Ted Xiao, Sumeet Singh, Vidhi Jain, Allen Ren, Quan Vuong,
Jake Varley, Alexander Herzog, Isabel Leal, Sean Kirmani, Mario Prats, Dorsa Sadigh,
Vikas Sindhwani, Kanishka Rao, Jacky Liang, Andy Zeng

Google DeepMind

Abstract

Large Language Models (LLMs) have demonstrated the ability to perform semantic reasoning, planning and code writing for robotics tasks. However, most methods rely on pre-existing primitives (i.e. pick, open drawer), which heavily limits their scalability to new scenarios. Additionally, existing approaches like Code as Policies (CaP) rely on examples of robot code in the prompt to write code for new tasks, and assume that LLMs can infer task information, constraints, and API usage from examples alone. But examples can be costly, and too few or too many can bias the LLM in the wrong direction. Recent research has demonstrated prompting LLMs with APIs and documentation enables code writing for successful zero-shot tool use. However, documenting robotics tasks and naively providing full robot APIs presents a challenge to context-length limits in LLMs. In this work, we introduce PromptBook, a recipe that combines LLM prompting paradigms - examples, APIs, documentation and chain of thought, to generate code for planning a sorting task with higher success rate than previous works. We further demonstrate PromptBook enables LLMs to write code for new low-level manipulation primitives in a zero-shot manner: from picking diverse objects, opening/closing drawers, to whisking, and waving hello. We evaluate the new skills on a mobile manipulator with 83% success rate at picking, 50-71% at opening drawers and 100% at closing them. Notably, the LLM is able to infer gripper orientation for grasping a drawer handle (z-axis aligned) vs. a top-down grasp (x-axis aligned). Finally, we provide guidelines to leverage human feedback and LLMs to write PromptBook prompts.

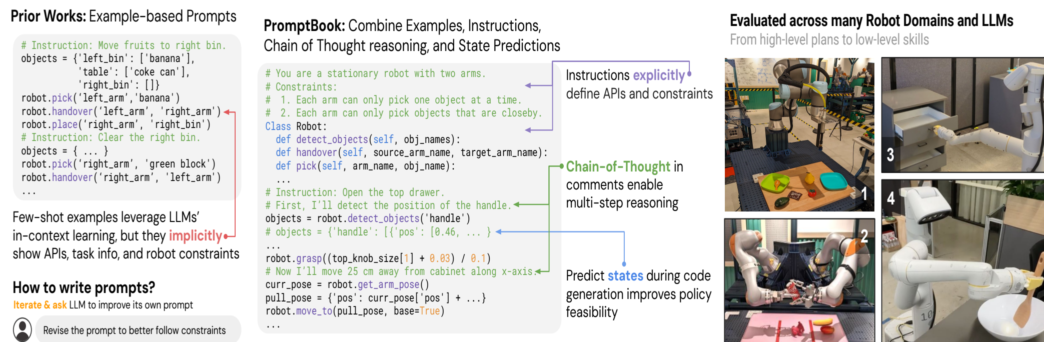

Figure 1: **PromptBook**, a recipe to combine CaP example-based prompts with documentation of APIs and constraints, Chain-of-Thought reasoning, and world state information. Experiments show that PromptBook has higher planning success rates across multiple robots, and interestingly give rise to new manipulation skills on-the-fly with LLMs zero-shot (e.g., picking, drawer opening, whisking).

NeurIPS 2023 Workshop on Robot Learning: Pretraining, Fine-Tuning, and Generalization with Large Scale Models, New Orleans, United States of America

1 Introduction and Related Work

Large language models (LLMs), through prompting and in-context-learning, exhibit a wide range of capabilities that are relevant to robotics tasks – from high-level planning [1, 6, 7, 23], logical reasoning [3, 8, 15, 20], to writing robot code [10, 14] and designing reward functions [5, 9, 22].

Example-based prompting is a promising paradigm for general in-context learning [2, 19], and has been effective in writing code for robotics applications [10]. [12] uses examples based prompting for the high-level task planning, followed by learned low-level policies. In contrast, our work presents code examples for both task and motion planning. Code as Policies (CaP) and other works [10, 14] use examples of language commands followed by corresponding policy code to prompt LLMs and write code for new tabletop tasks commands to autonomously generate code. However, this paradigm can be limiting as examples can be costly and some concepts are difficult to teach to LLMs through examples alone, such as robot constraints. Suppose there is a constraint that robot speed must not exceed 1 m/s. Multiple examples of calling `robot.set_velocity` must be shown to implicitly infer that total speed, that is, L2 norm of velocity, does not exceed 1. Instead of steering LLM behavior through *implicit* examples, we may prompt LLMs to follow *explicit* natural language instructions, that describe the objectives, constraints, and other information relevant for solving the task.

Instruction-based prompting has been explored in a number of prior work [13, 17, 18, 21, 22]. where the model is provided with a brief language description of the robot, its constraints, and accessible APIs, then expected to directly complete new robot code given a new task (often zero-shot). While these techniques provide powerful approaches for instruction-based prompting, they have not considered generating code to be executed directly on real robot settings.

In our work, we investigate different LLM prompting approaches across 3 robot platforms. We find that: (i) combining both instruction-based and example-based prompting yields the best of both worlds – performance improvements are observed across all language models and tasks, (ii) robot constraints are best explicitly specified via instruction-based prompting and perform better with instruction-tuned models, (iii) providing linguistic descriptions of environment states between lines of code in the prompt allows the LLM to write code considering the specific scene and keep track of internal variables, (iv) instruction-based prompting benefits from human feedback corrections, to which the LLM can be instructed to improve its own prompt.

We propose PromptBook (Figure 1), an LLM prompting recipe for improving LLMs' ability to convert natural language instructions to robot code that combines: instruction-based and example-based prompting, chain-of-thought, interleaved states and human feedback. PromptBook, leads to more robust planning performance as well as improved reasoning on geometric and embodiment constraints which gives rise to building new motion primitives – providing LLMs the capacity to generate high-level 3D trajectories for skills such as "open/close the drawer" or "stir the pot" which can be executed on-robot with an off-the-shelf IK-planner without any additional human intervention, data collection, or model training. These capabilities are not sufficient to replace specialized algorithms, but nevertheless offer a glimpse of the capacity of LLMs to compose motion primitives for low-level skills. While we provide results on few examples, our approach is extendable to other tasks without fine-tuning. We provide comprehensive details on the setup, results and analysis in the following sections.

2 The PromptBook Recipe

In this section, we will describe the 7 elements of the PromptBook recipe where the first one belongs to **Example-Based Prompting** and elements 2 through 5 are **Instruction-based Prompting**. Please see prompt examples and guidelines on how to apply PromptBook for specific robot task domains in the Appendix.

1) Examples. We can teach the LLM how to write robot code via packing a list of examples in prompt [10], where each example is a command-response pair. Examples implicitly show the LLM two types of information: how to ground robot commands (e.g., how to map spatial descriptions like "backwards" to code) and how to use first-party APIs (e.g., custom robot action functions not seen in the LLM's training set, such as `robot.set_velocity`).

Table 1: Prompting with both instructions and examples (instr. + ex.) yields stronger success rates (%) across robot settings and language models. See failure mode analysis in Appendix.

	Single Arm UR5			Bi-Arm Kuka2x		
Model	instr.	ex.	instr. + ex.	instr.	ex.	instr. + ex.
PaLM 2-L	42	83	**89**	71	72	**93**
Instruct-PaLM 2-L	72	**82**	80	66	82	**94**
PaLM 2-S* (Code)	47	78	**82**	5	50	**64**

2) High-Level Robot and Task Description. Directly specifying high-level robot embodiments and task information can give the LLM useful context when performing task planning. For example, e.g., we can specify that the robot is a bimanual stationary robot, with descriptions of important environment features and task information.

3) Robot API Documentation. Beyond high-level robot and task descriptions, Instruction-Based Prompts should also include low-level details about how to write domain-specific robot code. See Figure 1 for a simplified example for a single-arm robot with a mobile base.

4) Robot Policy Constraints. In addition to API documentation, we can also detail robot policy constraints that are not immediately obvious from the API themselves. See Figure 1 for a simplified example of such constraints in the prompt.

5) Code Guidelines. Finally, we can directly specify the desired policy code properties without relying on showing many implicit examples. Consider the requirement that the code written should strictly call the provided robot API and avoid API functions that do not exist.

6) Chain of Thought Policy Reasoning. Beyond adding instructions to the prompt, PromptBook makes two changes to the given examples to improve LLM planning performance. The first is Chain of Thought [20] (CoT), a popular prompting method that writes the step-by-step reasoning process of solving a task in the prompt. We can naturally incorporate CoT in code generation by formatting each thought step as a comment in the code. See a simplified example in Figure 1.

7) Interleaved State Predictions and Observations. Inspired by and analogous to CoT, we also include, in each example code output, explicit environment state predictions formatted as comments after each robot action. We can steer LLMs to predict and record current states explicitly, interleaved with the robot policy code, in its outputs. See Figure 1 for a simple example. At run time, we provide a new instruction and the initial state information, and the LLM will autoregressively generate the remaining sequence. Explicit state predictions encourage the LLM to utilize a simple transition model for more precise planning and to better obey the given constraints.

3 Experiments

Example vs. Instruction Prompting across LLMs for Sorting Task Planning. We evaluate planning success rates accross various language models using (i) example-based prompting, (ii) instruction-based prompting, and (iii) a combination of both. The tasks area collection of pick and place sorting tasks across 2 platforms: single arm (UR5) and bi-arm Kukas (Kuka2x). These are simple tasks, but they are sufficient to raise key challenges behind LLM-based planning. The distinction between a single arm and bi-arm setup makes allows measuring LLM's capacity to reason over reachability constraints – not only does the LLM need to reason that each arm can only reach the table or the bin nearest to it, but that moving objects from one bin to another requires taking an additional action in between (i.e., handover) (details in Appendix). See results in Table 1.

Interleaving State Predictions with Policy Code. We test two LLMs in a mobile robot trash sorting domain (similar to [4]). Here, the robot can move among three different trash bins (landfill, recycle, and compost), and it needs to sort the trash already placed in these bins to their correct bins (e.g., plastic bottles should go in recycle). However, the robot needs to be in front of the corresponding bin before placing to avoid reachability errors. See Table 2 for results. We show that without interleaved state predictions, LLMs struggle with reasoning about the bin location of the robot (state) which results in low success rate due to reachability errors.

Table 2: Prompting with interleaved robot state information improves task success rate (%) for the trash sorting task across two LLMs. Improvements are most substantial when prompting with instructions and examples.

Model	ex. only	instr. + ex.
GPT-4 w/o State	8	30
GPT-4 w/ State	17	74

Table 3: Instruction-based prompting benefits from iterative prompt improvement with higher success rates on the Bi-arm Kuka2x planning task. can substantially improve planning success.

Model	instr. only	instr. + feedback	instr. + ex. + feedback
Instruct-PaLM 2-L	66	80	**93**
GPT-4	10	**99**	**99**

Improving Instruction-Based Prompts with Human Feedback and LLM-aid. To evaluate the impact of iterative code improvement by human feedback, we evaluated two LLMs on the Bi-arm Kuka2x platform sorting task. Results in Table 3 show higher success rate with only 2-3 rounds of feedback and prompt iteration as described in 2.

Building Low Level Motion Primitives On-The-Fly. Bringing our findings from previous sections to a practical real world low level robot control setting, we evaluate whether a single expressive PromptBook can generate novel motion primitives on the fly just by changing the input task instruction. On a mobile manipulator, we build a PromptBook with a description of the robot, its constraints, robot APIs including `follow_arm_path()`, `detect_object()`, and `gripper()` functions, as well as an example of robot code for grasping an object on a countertop. This prompt can be queried to generate new motion primitives (code in Appendix) for multiple tasks, some of which are shown in Figure 2 in Appendix. The generated code exhibits "motion commonsense" knowledge from the LLM which are required to solve these low-level control tasks, including understanding of how a gripper should be oriented with respect to objects (e.g., vertically for a drawer handle, or horizontally for a knob). In quantitative evaluations of motion primitives generated by PromptBook on the fly, we find that policies can achieve reasonable success rates as detailed in Table 4.

4 Discussions and Future Work

Our work proposes PromptBook, a guide for creating and improving prompts for new robot task domains through human and LLM feedback. We demonstrate PromptBook across three robot domains: UR5, Bimanual arms, and mobile manipulator, improving LLM robot task planning performance and synthesizing novel motion primitives. While our work investigates the trade-offs between prompting methods, the space of LLM prompting strategies and models is vast. For more complicated tasks and systems, teaching complex concepts through examples can dominate the input context to the LLM, and so does providing large APIs and instructions. We can enhance PromptBook by leveraging LLMs with longer context lengths, or efficiently selecting prompt tokens.

Our strategy to synthesize novel motions rely on robot motion primitives and objection poses information obtained from vision models. Errors in pose estimations or in the motion primitives affect the success rates of the novel motions when executed on real robots. This limitation is not specific to our method; that is, any method that relies on object pose estimation will face similar challenges. In the future, it would be interesting to consider how LLM re-planning can autonomously compensate for failures of upstream models to improve success rate.

Table 4: Real robot execution success rate of LLM-generated motion primitives zero-shot for a mobile manipulator, evaluated across 50 trials.

		Top Drawer with Handle		Middle Drawer with Knob	
Model	Pick	Open	Close	Open	Close
GPT-4	83	71	100	50	100

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv:2204.01691*, 2022.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

[3] Antonia Creswell, Murray Shanahan, and Irina Higgins. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2205.09712*, 2022.

[4] Alexander Herzog, Kanishka Rao, Karol Hausman, Yao Lu, Paul Wohlhart, Mengyuan Yan, Jessica Lin, Montserrat Gonzalez Arenas, Ted Xiao, Daniel Kappler, et al. Deep rl at scale: Sorting waste in office buildings with a fleet of mobile manipulators. *arXiv preprint arXiv:2305.03270*, 2023.

[5] Hengyuan Hu and Dorsa Sadigh. Language instructed reinforcement learning for human-ai coordination. In *40th International Conference on Machine Learning (ICML)*, 2023.

[6] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv:2201.07207*, 2022.

[7] Wenlong Huang, Fei Xia, Dhruv Shah, Danny Driess, Andy Zeng, Yao Lu, Pete Florence, Igor Mordatch, Sergey Levine, Karol Hausman, et al. Grounded decoding: Guiding text generation with grounded models for robot control. *arXiv preprint arXiv:2303.00855*, 2023.

[8] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv:2205.11916*, 2022.

[9] Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. *arXiv preprint arXiv:2303.00001*, 2023.

[10] 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. IEEE, 2023.

[11] M Minderer, A Gritsenko, A Stone, M Neumann, D Weissenborn, A Dosovitskiy, A Mahendran, A Arnab, M Dehghani, Z Shen, et al. Simple open-vocabulary object detection with vision transformers. arxiv 2022. *arXiv preprint arXiv:2205.06230*.

[12] Priyam Parashar, Vidhi Jain, Xiaohan Zhang, Jay Vakil, Sam Powers, Yonatan Bisk, and Chris Paxton. SLAP: Spatial-language attention policies. In *7th Annual Conference on Robot Learning*, 2023.

[13] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. *arXiv preprint arXiv:2305.11014*, 2023.

[14] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE, 2023.

[15] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

[16] Jake Varley, Sumeet Singh, Deepali Jain, Krzysztof Choromanski, Andy Zeng, Somnath Basu Roy Chowdhury, Avinava Dubey, and Vikas Sindhwani. Embodied ai with two arms:zero-shot learning, safety and modularity. *arXiv preprint arXiv:2305.11014*, 2023.

[17] Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. *Microsoft Auton. Syst. Robot. Res*, 2:20, 2023.

[18] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

[19] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed Huai hsin Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[20] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv:2201.11903*, 2022.

[21] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

[22] 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 preprint arXiv:2306.08647*, 2023.

[23] Andy Zeng, Adrian Wong, Stefan Welker, Krzysztof Choromanski, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, Johnny Lee, Vincent Vanhoucke, et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv:2204.00598*, 2022.

Appendices

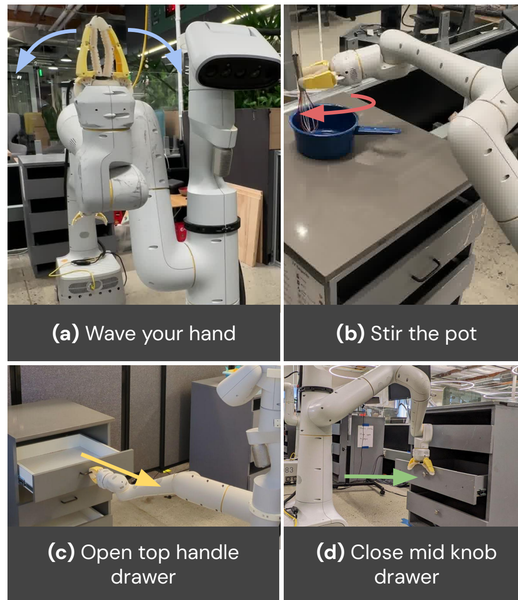

Figure 2: Examples of diverse motions generated with PromptBook. Notably, the same instruction-based prompt is re-used with only the specific low-level task instruction being swapped out for (a) waving, (b) stirring, (c) opening the top drawer by the handle, and (d) closing the middle drawer with a knob.

5 How to Build PromptBook Prompts

The final PromptBook prompt is assembled by combining the 8 prompt elements above. In this section, we provide a procedure that constructs and improve theses prompts, with the aid of LLMs. Specifically, we develop a method that can leverage LLM's retrospection capabilities that leverage language feedback to improve both its immediate outputs as well as the initial prompt. We provide our 3 step process as follows:

Step 1: Initial Prompt Draft. Given a new robot task domain, a robot engineer first drafts an initial prompt following the PromptBook recipe. This initial prompt may be suboptimal, either due to insufficient domain information, or style and presentation that is difficult for the LLM to understand.

Step 2: Human-in-the-Loop Code Improvement. With the initial prompt draft ready, the robot engineer then uses the prompted LLM to perform a series of validation tasks. In this stage, the robot engineer first gives the LLM the task command, the LLM then writes the policy code, then, if the policy fails, the human engineer provides error feedback to the LLM. After receiving the feedback, the LLM writes improved policy code, and this process is repeated until the task is solved. At the end of each trial, we obtain a sequence of (task, code, feedback) tuples.

Step 3: LLM-aided Prompt Improvement. While human-in-the-loop feedback can reduce errors for a specific task, it is desirable to modify the prompt so these errors are not repeated in the future. We do this by giving the LLM the original prompt and the history of (task, code, feedback) tuples, then asking the LLM "How would you modify the initial prompt to avoid making this mistake in the future while keeping existing constraints?" and "How would you add a general constraint to avoid making this mistake in the future?" These modifications are then incorporated in the initial prompt to improve performance on similar tasks.

6 Examples of Promptbook Elements

Examples.

For code-writing examples, we can format each command as a comment, and each response as the code block that immediately follows:

```
# if you see an orange, move backwards.
if detect_object("orange"):
    robot.set_velocity(x=-0.1, y=0, z=0)
```

High-level Robot and Task Description.

In the above below, we also tell the LLM that it is acceptable to make certain kinds of assumptions; eliciting this behavior is much more challenging with Example-Based Prompts.

```
# You are a stationary robot with a left arm and right arm placed on a table with a bin to the right and another
bin to the left. # Your task is to write code that will sort objects into the correct bins as defined from given
instructions. # Use your best world knowledge and make any necessary assumptions when selecting objects to sort.
```

Robot API Documentation

We first provide the robot perception and action APIs, written in the style of skeleton Python code. Then, we provide additional details on the given robot APIs by directly specifying them in language, which is possible because Instruction-Based Prompts are not limited to conveying information through examples. For instance, for a robot API that uses 6D poses, we can specify the pose formats and canonical directions as follows:

```
# Pose is a dict with 'position' and 'orientation' keys in robot frame.
# The 'position' value is a 3D array of [x, y, z]   coordinates in meters.
# The 'orientation' value is a 4D array quaternion in [w, x, y, z].
# In the robot frame, directions are defined as follows:
# Positive x / Negative x: Front and Back
# Positive y / Negative y: Left / Right
# Positive z / Negative z: Upward / Downward
```

Robot Policy Constraints

Below is a policy constraint prompt example for a bimanual robot setup, we may wish to inform the LLM on the different reachability constraints of workspace objects given the two arms, as well as additional preconditions of executing low-level pick place actions.

```
# Constraints:
# 1. The right arm can pick and place objects on the right bin and the table only. It cannot reach the left bin
to pick or place objects.
# 2. The left arm can place objects on the left bin and the table. It cannot reach the right bin to pick or
place objects.
# 3. You can handover objects from one arm to the other. You cannot do a handover if you don't have an object
in the passing arm or the receiving arm is not empty.
# 4. You can only pick one object at a time per arm.
```

Code Guidelines

We can communicate code guidelines as part of the instructions:

```
# Your response should be exclusively in the form of Python code and Python-formatted comments. Do not use
additional if statements or loops. Only write code that calls the provided robot API.
```

7 Example vs. Instruction Prompting Evaluation Protocol

7.1 Robot Platforms

- Single arm (UR5): consists of a UR5 equipped with a suction gripper overlooking a tabletop surface with objects that span kitchenware and plastic food items. A RealSense d435 camera is mounted on the wrist that captures an overhead view of the tabletop scene.
- Bi-arm Kukas (Kuka2x): a more challenging setting that consists of two Kuka IIWA 7 arms equipped with two-finger grippers overlooking a large surface area with a bin next to each arm (reachable only by that arm), as well as a shared tabletop zone that is reachable by both arms. Objects in this setting include plastic food items, wooden blocks, plush toys, and soda cans. A RealSense d435 is mounted above the shoulders of the bi-arm setup to capture an overhead view of the scene.

Table 5: Failure modes (% error, categorized by type, sum of columns per language model is total % error) across prompting methods and robot settings.

Model	Categories	Single Arm UR5			Bi-Arm Kuka2x		
		instr.	ex.	instr. + ex.	instr.	ex.	instr. + ex.
PaLM 2-L	Feasibility	1	4	0	9	16	4
	Semantic	57	13	11	20	12	3
	Syntax	0	0	0	0	0	0
Instruct-PaLM 2-L	Feasibility	0	0	0	12	6	0
	Semantic	28	18	20	21	12	6
	Syntax	0	0	0	1	0	0
PaLM 2-S (Code)*	Feasibility	6	1	0	16	28	13
	Semantic	47	21	18	79	22	23
	Syntax	0	0	0	0	0	0

7.2 Task Domains.

Both robots are tasked with 100 natural language instructions to sort multiple objects (randomly chosen and positioned) in the scene by their varying properties e.g., "Put the vegetables on the green plate." or "Move the soft objects to the right bin.". Given the language instructions and a description of the scene (dictionary of objects and poses) as input, the language model outputs code to call motion primitive APIs that sequence pick, place, or handover actions (bi-arm only) conditioned on a target location (or object name), and arm to use (bi-arm only). Both models use open-vocabulary object detectors (e.g., OWL-ViT [11]) to locate objects in the scene.

7.3 Evaluated LLMs.

We evaluate the different prompting methods across three LLMs: (i) in-context pre-trained vanilla PaLM 2-L (340B), (ii) instruction-tuned PaLM (Instruct-PaLM 2-L), and (iii) and a smaller code-writing language model PaLM 2-S* (24B) specifically fine-tuned on code-related tokens. We chose to use PaLM-2 for a side-by-side comparison of pre-trained LLMs with and without instruction-tuning, as well as a smaller pre-trained code-writing model (trained with the same infrastructure), which we expect should provide systems-level advantages including faster inference speeds (i.e., lower planning latency) at the cost of reduced performance.

7.4 Error Categorization

To better characterize failure modes, we additionally categorized LLM planning errors into 1 of 3 types:

- Feasibility: generated code does not respect robot constraints and either: (i) attempts to move a robot other than itself, (ii) pick up an object not there, (iii) pick up multiple object simultaneously when the gripper can only hold one, or (iv) does not respect reachability constraints (bi-arm only) in that each arm can only reach objects in its nearest bin.
- Syntax: code does not execute due to a syntax error.
- Semantic: code executes, but the task failed (i.e., objects not sorted correctly accordingly to given instructions).

Tab. 5 shows the ratio of planning errors categorized by types. Most errors are task planning errors (semantic) e.g., objects sorted incorrectly, or the task was incomplete. The next largest source of errors is reasoning over action feasibility. In particular, example-based prompting tends to struggle on reasoning over reachability constraints, but improves when the LLM is instruction-tuned. By contrast, instruction-based prompting performs similarly across models. There is a clear improvement when both instructions and examples are provided in the prompt, in which case both instruction-tuned and non-instruction-tuned models perform comparably – with non-instruction-tuned models still struggling more in reasoning on reachability.

7.5 Real robot execution.

We also directly run the robot policy code generated by the best performing LLMs and prompting methods on both platforms for all tasks. Instr. + ex. with PaLM 2-L on the single arm UR5 yields an average execution success rate of 83%, while instr. + ex. with Instruct-PaLM 2-L on the more challenging Bi-arm Kuka2x setup yields a success rate of 59%. Common execution failure modes on the UR5 setup include handling objects dynamics e.g., (i) slipping from gripper during picking (22% of errors), rolling away on contact (67% of errors), or colliding with another object during placing (11% of errors). On the other hand, for the Bi-arm Kuka2x setup, common failure modes include: (i) perception detection failures (47% of errors), (ii) objects dropping during handover (41% of errors), (iii) grasp failures (6% of errors), (iv) or other systems errors (6% of errors). See Varley et al. [16] for a more detailed analysis of the Bi-arm Kuka2x setup.

8 Final GPT-4 PromptBook Prompts

8.1 Interleaved State Prediction for Mobile Sorting Task

```
1  # You are a helpful robot with one arm and a mobile base. Your task is to sort objects into
       recycle, compost, and landfill bins according to their trash classification using a robot
        API.
2  # Given a list of object descriptions and their current bin placements, determine their
       correct trash classification (recycle, compost, or landfill).
3  # If the objects are not currently in their corresponding bin, move them to the right bin. Use
        your best judgement to decide the right trash classification for each object.
4  # Trash classification guidelines:
5
6  # Compost:
7  # Items primarily made of paper, cardboard or organic material. Products labeled as "
       compostable" belong exclusively to this category.
8  # Landfill:
9  # Items typically made of plastic or mixed materials, which are deformable or crumpled, i.e
       wrappers and bags, and might have contained snacks or dry food items.
10 # Recycle:
11 # Items primarily made of rigid plastic, aluminum, or glass. Most items in this category are
       bottles and cans.
12
13 # Constraints:
14
15 # 1. Your base starts in front of the compost bin.
16 # 2. Your cannot pick or place from a given bin if your are not currently in front of that bin
       .
17 # 3. Always track the robot's position in relation to the bins after each api call to ensure
       no unnecessary movements are made. Avoid commanding the robot to move to a bin it's
       already in front of, as this will result in an error.
18 # 4. Objects that are already in their respective correct bins should not be moved. Ensure
       that the robot does not pick up and replace items already in the appropriate bin.
19 # 5. You can only pick one object at a time.
20 # 6. Your response should be exclusively in the form of Python code and Python-formatted
       comments. Only output calls to the robot API provided. Do not use additional if
       statements or loops.
21
22 # Only python code below.
23
24 class Robot:
25   def __init__(self):
26     self.picked_object = None
27     sel.current_bin_position = 'compost'
28
29   def pick_object_from_bin(self, object_name, bin_name):
30     '''
31     :param object_name (string): Name of the object to be picked.
32     :param bin_name: One of ["compost", "recycle", "landfill"].
33     '''
34     pass
35
36   def place(self, bin_name):
37     '''
38     :param bin_name (string): One of ["compost", "recycle", "landfill"].
39     '''
40     pass
41
42   def move_base_to_bin(self, bin_name):
43     '''
44     :param bin_name (string): One of ["compost", "recycle", "landfill"].
```

```
45        '''
46        pass
47
48    robot = Robot()
49
50    # Instruction: Sort the objects into the right bin according to their trash classification.
51    # objects = {'compost': ['noosa yogurt plastic container'], 'landfill': [], 'recycle': ['taali water lily pops deformable plastic bag']}
52    # robot.current_bin_position = 'compost'
53    # robot.picked_object = None
54    robot.pick_object_from_bin('noosa␣yogurt␣plastic␣container', 'compost')
55    # objects = {'compost': [], 'landfill': [], 'recycle': ['taali water lily pops deformable plastic bag']}
56    # robot.current_bin_position = 'compost'
57    # robot.picked_object = 'noosa yogurt plastic container'
58    robot.move_base_to_bin('recycle')
59    # objects = {'compost': [], 'landfill': [], 'recycle': ['taali water lily pops deformable plastic bag']}
60    # robot.current_bin_position = 'recycle'
61    # robot.picked_object = 'noosa yogurt plastic container'
62    robot.place('recycle')
63    # objects = {'compost': [], 'landfill': ['hint cherry bottle'], 'recycle': ['taali water lily pops deformable plastic bag', 'noosa yogurt plastic container']}
64    # robot.current_bin_position = 'recycle'
65    # robot.picked_object = None
66    robot.pick_object_from_bin('taali␣water␣lily␣pops␣deformable␣bag', 'recycle')
67    # objects = {'compost': [], 'landfill': [], 'recycle': ['noosa yogurt plastic container']}
68    # robot.current_bin_position = 'recycle'
69    # robot.picked_object = 'taali water lily pops deformable plastic bag'
70    robot.move_base_to_bin('landfill')
71    # objects = {'compost': [], 'landfill': [], 'recycle': ['noosa yogurt plastic container']}
72    # robot.current_bin_position = 'landfill'
73    # robot.picked_object = 'taali water lily pops deformable plastic bag'
74    robot.place('landfill')
75    # objects = {'compost': [], 'landfill': ['taali water lily pops deformable plastic bag'], 'recycle': ['noosa yogurt plastic container']}
76    # robot.current_bin_position = 'landfill'
77    # robot.picked_object = None
78    # Done.
79
80    # Instruction: Sort the objects into the right bin according to their trash classification.
81    # objects = {'compost': ['paper coffee cup'], 'landfill': ['la croix lime can empty'], 'recycle': ['deep river chips deformable foiled plastic bag', 'hint cherry bottle']}
82    # robot.current_bin_position = 'compost'
83    # robot.picked_object = None
84    robot.move_base_to_bin('landfill')
85    # objects = {'compost': ['paper coffee cup'], 'landfill': ['la croix lime can empty'], 'recycle': ['deep river chips deformable foiled plastic bag', 'hint cherry bottle']}
86    # robot.current_bin_position = 'landfill'
87    # robot.picked_object = None
88    robot.pick_object_from_bin('la␣croix␣lime␣can␣empty', 'landfill')
89    # objects = {'compost': ['paper coffee cup'], 'landfill': [], 'recycle': ['deep river chips deformable foiled plastic bag', 'hint cherry bottle']}
90    # robot.current_bin_position = 'landfill'
91    # robot.picked_object = 'la croix lime can empty'
92    robot.move_base_to_bin('recycle')
93    # objects = {'compost': ['paper coffee cup'], 'landfill': [], 'recycle': ['deep river chips deformable foiled plastic bag', 'hint cherry bottle']}
94    # robot.current_bin_position = 'recycle'
95    # robot.picked_object = 'la croix lime can empty'
96    robot.place('recycle')
97    # objects = {'compost': ['paper coffee cup'], 'landfill': [], 'recycle': ['deep river chips deformable foiled plastic bag', 'hint cherry bottle', 'la croix lime can empty']}
98    # robot.current_bin_position = 'recycle'
99    # robot.picked_object = None
100   robot.pick_object_from_bin('deep␣river␣chips␣deformable␣foiled␣plastic␣bag', 'recycle')
101   # objects = {'compost': ['paper coffee cup'], 'landfill': [], 'recycle': ['hint cherry bottle', 'la croix lime can empty']}
102   # robot.current_bin_position = 'recycle'
103   # robot.picked_object = 'deep river chips deformable foiled plastic bag'
104   robot.move_base_to_bin('landfill')
105   # objects = {'compost': ['paper coffee cup'], 'landfill': [], 'recycle': ['hint cherry bottle', 'la croix lime can empty']}
106   # robot.current_bin_position = 'landfill'
107   # robot.picked_object = 'deep river chips deformable foiled plastic bag'
108   robot.place('landfill')
109   # objects = {'compost': ['paper coffee cup'], 'landfill': ['deep river chips deformable foiled plastic bag'], 'recycle': ['hint cherry bottle', 'la croix lime can empty']}
110   # robot.current_bin_position = 'landfill'
111   # robot.picked_object = None
112   # Done.
```

8.2 Human Feedback and LLM-aid for Kuka2x Sorting Task

```
# You are a robot with a left arm and right arm placed on a table with a bin to the right and another bin to the left.
# Your task is to write code sort objects into the right bin, left bin or table according to a given instruction, robot API and a dictionary specifying the objects available and their location.
# Use your best world knowledge and make any necessary assumptions to select the objects to sort. Consider typical color associations with popular brands.
# Constraints:
# 1. The right arm can pick and place objects on the right bin and the table only. It cannot reach the left bin to pick or place objects.
# 2. The left arm can place objects on the left bin and the table. It cannot reach the right bin to pick or place objects.
# 3. You can handover objects from one arm to the other. You cannot do a handover if you don't have an object in the passing arm or the receiving arm is not empty.
# 4. You can only pick one object at a time per arm.
# 5. Your response should exclusively be in the form of Python code and Python-formatted comments. Only output calls to the robot API provided. Do not use additional if statements or loops.

class Robot:

  def pick(self, arm_name, object_name):
    '''
    :param arm_name: One of ["left arm", "right arm"].
    :type arm_name: string.
    :param object_name: Name of the object to be picked.
    :type object_name: string.
    '''
    pass

  def place(self, arm_name, place_location):
    '''
    :param arm_name: One of ["left arm", "right arm"].
    :type arm_name: string.
    :parm place_location: One of ["right bin", "left bin", "table"].
    :type place_location: string.
    '''
    pass

  def handover(self, from_arm_name, to_arm_name):
    '''
    :param from_arm_name: Arm giving the object. One of ["left arm", "right arm"].
    :type from_arm_name: string.
    ::param to_arm_name: Arm receiving the object. One of ["left arm", "right arm"].
    :type to_arm_name: string.
    '''
    pass

robot = Robot()

# Instruction: Move the red objects to the table.
objects = {'left_bin': ['pink_plushie'], 'table': [], 'right_bin': ['red_mango', 'coke_can']}
robot.pick('right_arm', 'red_mango')
robot.place('right_arm', 'table')
robot.pick('right_arm', 'coke_can')
robot.place('right_arm', 'table')

# Instruction: Pick up the soft objects and place them on the right bin.
objects = {'left_bin': ['purple_plushie', 'yellow_plushie'], 'table': ['blue_block'], 'right_bin': []}
robot.pick('left_arm', 'purple_plushie')
robot.handover('left_arm', 'right_arm')
robot.place('right_arm', 'right_bin')
robot.pick('left_arm', 'yellow_plushie')
robot.handover('left_arm', 'right_arm')
robot.place('right_arm', 'right_bin')
```

8.3 Low-Level Manipulation Skills Generation

```
# You are a helpful robot with one arm and a mobile base.
# The gripper fingers are 10 cm long and the span between them when open is 15 cm.
# You are standing in front of a workspace where you will be given task instructions to perform writing python code using the robot api below.
```

```
4
5  # Poses are a dictionary with 'position' and 'orientation' keys in robot frame.
6  # The 'position' value is a 3D array indicating the [x, y, z]  coordinates in meters.
7  # The 'orientation' value is a 4D array indicating the [w, x, y, z] quaternion.
8  # Bounding boxes are a dictionary comprised by the 'centroid_pose'- which is a pose dictionary
       ,
9  # as well as 'size' representing the size of the [x, y, z] box edges in meters.
10 # Note: All poses and bounding boxes are in robot frame. This means they are relative to the
       current base position which is base_pose = {'position':[0,0,0], 'orientation':
       [1,0,0,0]}.
11 # If the base moves, all previous object poses are no longer valid; except the arm pose
       because it is attached to the base and it moves along with it. The arm pose will only
       change when the arm moves.
12 # All bouding boxes are z-aligned with no rotation.
13
14 # !!!!!!!!! IMPORTANT DIRECTIONAL INFORMATION !!!!!!!!!
15 # In the robot frame, front/back is along the x axis, left/right is along the y axis and up/
       down is along the z axis with following directions:
16 # Positive x: Forward/Front (away from the robot)
17 # Negative x: Backward/Back (towards the robot)
18 # Positive y: To the left of the robot.
19 # Negative y: To the right of the robot.
20 # Positive z: Up.
21 # Negative z: Down.
22 # Note: When comparing two coordinates, if one is greater than the other and you can only move
        the smaller one, then to increase the distance between the two you need to make the
       smaller one smaller by subtracting a delta.
23 # Conversely, to decrease the distance, you would add a delta to the smaller coordinate. On
       the other hand, if you can only move the larger one then adding a delta would increase
       the distance between the two and subtracting a delta would decrease it.
24
25 # The following Euler degree angles (roll, pitch, yaw) apply to the gripper orientations in
       robot frame:
26 # Pointing towards positive z (approaching from bottom):
27 # [0, 0, 0] # fingers aligned with y axis.
28 # [0, 0, -90] # fingers aligned with x axis.
29 # Pointing towards negative z (approaching from the top):
30 # [180, 0, 0] # fingers aligned with y axis.
31 # [180, 0, -90] # fingers aligned with x axis.
32 # Pointing towards positive x (approaching from front, away from the robot):
33 # [0, 90, 0] # fingers aligned with y axis.
34 # [-90, 0, -90] # fingers aligned with z axis.
35 # Pointing towards negative x (approaching from back side, towards the robot):
36 # [0, 0, 180] # fingers aligned with y axis.
37 # [0, -90, -90] # fingers aligned with z axis.
38 # Pointing towards positive y (approaching from right side):
39 # [-90, 90, 0] # fingers aligned with y axis.
40 # [-90, 0, 0] # fingers aligned with z axis.
41 # Pointing towards negative y (approaching from left side):
42 # [-90, 0, 90] # fingers aligned with y axis.
43 # [-90, 0, 180] # fingers aligned with z axis.
44 # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
45
46 # Ensure that the direction and orientation you want the gripper to move in follows the
       aforementioned definitions.
47
48 # When interacting with objects, consider their physical properties and how they are used or
       manipulated. Plan the trajectories of the arm accordingly to successfully complete the
       task motion: linear, circular, sinusoidal, elliptical, etc; along the corresponding x,y,z
        axis.
49
50 # Python code and comments only from here.
51
52 class RobotAPI(object):
53
54   def gripper_open(self, span=1.0):
55     '''
56     :param span: percentage of openness of the gripper. 1.0 is 17 cm.
57
58     '''
59     pass
60
61   def gripper_close(self):
62     pass
63
64   def orientation_quaternion_from_euler(self, roll, pitch, yaw):
65     '''Get quaternion from roll, pitch, yaw in radians.'''
66     pass
67
68   def get_arm_pose(self):
```

```
69      ''' returns current arm pose in robot frame. The arm pose indicates the point at the tip
            of the gripper fingers when closed. The size of the gripper is 12 cm.
70      '''
71
72    def detect_objects_in_scene(self, object_names):
73      ''' Detect objects in the scene and return list of bounding boxes.
74      :param object_names: List of object names.
75      '''
76
77    def print(self, objects):
78      pass
79
80    def follow_arm_trajectory(self, trajectory_poses, allow_base_moves):
81      '''
82      Execute an arm movement following the trajectory. This may move the base accordingly.
83      :param trajectory_poses: List of poses for the gripper to follow in robot frame.
84      :param allow_base_moves: Whether the robot is allowed to move the base while following
85      the arm trajectory. If False, the robot will follow the trajectory moving only the arm.
86      :param speed: One of "medium", "fast" or "slow". Default is "medium".
87      '''
88      pass
89
90  robot_api = RobotAPI()
91
92  # Instruction: Pick the bottle on the right.
93  # Scene objects: chips bag, water bottle, cabinet.
94  objects = robot_api.detect_objects_in_scene(['water␣bottle', 'chips␣bag', 'cabinet'])
95  robot_api.print(objects)
96  # I/O
97  # objects = {'cabinet': [{'centroid_pose': {'position': [2.07 , 0.54 , 0.20 ], 'orientation':
         [1.00 , 0.00 , 0.00 , 0.00 ]}, 'size': [3.01 , 3.06 , 0.40]}], 'water bottle': [{'
         centroid_pose': {'position': [0.44 , 0.24 , 0.45 ],'orientation': [0.97 , 0.00 , 0.00 ,
         0.23 ]}, 'size': [0.05 , 0.05 , 0.11 ]},{'centroid_pose': {'position': [0.34 , 0.14 ,
         0.45 ],'orientation': [0.97 , 0.00 , 0.00 , 0.23 ]}, 'size': [0.05 , 0.05 , 0.11 ]}], '
         chips bag': []}
98  # There are two bottles, and the right-most should be compared along the y-axis.
99  # First bottle is has y-value of 0.24, second has y-value of 0.14.
100 # The negative y direction corresponds to right, so the second bottle is the right-most.
101 right_bottle_position =  objects['water␣bottle'][1]['centroid_pose']['position]
102 right_bottle_orientation␣=␣␣objects['water bottle'][1]['centroid_pose']['orientation]
103 right_bottle_size = objects['water␣bottle'][1]['size']
104 # The bottle has a bounding box size of [0.05 , 0.05 , 0.11] in meters and is located on the
         top x-y plane of the cabinet.
105 # The gripper has a max span of 10 cm, the size of the bottle along the z-axis is 0.11 so it
         can only grasp the bottle with fingers aligned along the x axis and y axis.
106 # A bottom grasp is ruled out since the robot would collide with the cabinet.
107 # A back grasp is unfeasible since the robot is in front of the cabinet and cannot go around
         it to make a back grasp.
108 # A top grasp with fingers aligned with the x-axis or y-axis is feasible.
109 # A front grasp with fingers aligned with the y-axis and a side grasp with fingers aligned
         with the x-axis are feasible too.
110 # We choose the top grasp with fingers aligned with the y-axis for simplicity.
111 # Get quaternion corresponding to [180, 0, 0] roll,pitch and yaw for a top grasp with fingers
         aligned with the y-axis.
112 grasp_orientation_quaternion = robot_api.orientation_quaternion_from_euler(180, 0, 0)
113 # Calculate grasp position so object ends within gripper fingers.
114 grasp_pose = {'position': right_bottle_position, 'orientation': grasp_orientation_quaternion}
115 # The pregrasp pose is the pose right before the grasp.
116 # Since this is a top grasp, this means the gripper is pointing towards the negative z axis,
         so the pregrasp pose has a positive z delta over the grasp pose.
117 # Calculate pregrasp pose accounting for object size and gripper size (0.1 m).
118 pregrasp_pose = {'position': grasp_pose['position'] + [0,0, right_bottle_size[2]/2 + 0.1], '
         orientation': grasp_orientation_quaternion}
119 # Open the gripper according to the y axis size of the bottle plus a buffer of 2 cm.
120 robot_api.gripper_open(right_bottle_size[1] + 0.02)
121 robot_api.follow_arm_trajectory([pregrasp_pose, grasp_pose], allow_base_moves=True)
122 robot_api.gripper_close()
123 # bottle pose is not valid anymore since we might have moved the base so we use the current
         arm pose.
124 current_arm_pose = robot_api.get_arm_pose()
125 # The bottle is at the top of the cabinet. Picking means moving the object (bottle) away from
         their reference (cabinet) along the z axis.
126 # The cabinet is at z_cabinet = 0.2 and the bottle is at z_bottle = 0.45.
127 # When the object coordinate is lower than its reference, to increase distance you need to
         subtract a delta and to decrease distance  you need to add a delta.
128 # When the object coordinate is greater than its reference, to increase distance you need to
         add a delta and to decrease distance  you need to subtract a delta.
129 # Since z_bottle is greater than z_cabinet, it means the object coordinate is lower than its
         reference, so to increase the distance between the two we add a positive delta to
         z_bottle.
```

```
130 lift_arm_pose = {'position': current_arm_pose['position'] + [0,0, 0.25], 'orientation':
        current_arm_pose['orientation']}
131 # Allow for base moves for after grasp moves since arm could be in a difficult position to
        execute the lift.
132 robot_api.follow_arm_trajectory([lift_arm_pose], allow_base_moves=True)
133 # Done
134
135 # Instruction: Instruction to generate new skill here <------
```

```
# Instruction: Open the drawer.
# Scene objects: drawer handle, cabinet.
```

```
# Instruction: Close drawer.
# Scene objects: drawer handle, open drawer, cabinet top.
```

```
objects = ['7up can', 'apple', 'blue chip bag', 'coke can', 'green can', 'green chip bag', 'orange can', 'pepsi
can', 'redbull can', 'rxbar blueberry', 'water bottle'] # Instruction: Pick coke can.
# Scene objects: coke can.
```

```
# Instruction: Wave your gripper as if you were saying hello.
```

```
# Instruction: You are holding a spoon facing down inside a bowl with eggs. Whisk the eggs.
# Scene objects: bowl.
```