Intelligent Interactive Large Language Model Planner: Responsive Personalized HomeRobot

Stanford CS224N Custom Project
TA mentor: Neil Nie.
No external collaborators
Not sharing outside class

Angel Meng Zhang
Department of Computer Science
Stanford University
angelzhm@stanford.edu

Gadiel Sznaier Camps
Department of Aeronautics and Astronautics
Stanford University
gsznaier@stanford.edu

Abstract

Large Language Models can be used as a cognitive brain in robotics to complete many embodied tasks. However, most recent research uses a one-size-fits-all framework for LLM planning and fails to integrate dynamically changing human preferences in household tasks, which can result in conflicting plans. In this research project, we propose a responsive intelligent robotic planner for household tasks capable of both interpreting and executing high and low-level plans from real-time human textual feedback. Building upon TidyBot, we add the capability to interpret and respond to new human preferences during the execution process while resolving any conflicting preferences. Our experimental findings show that Tidybot completely fails when handling new preference during real-time plan execution, dropping from 86.2% when responding to a single preference to 32.6% when handling multiple requests. In addition, we demonstrate that simply closing the loop using few-shot prompting will only improve plan accuracy by 19% and that more complex prompting strategies, such as iterative Chain-of-Thought (CoT) are necessary to accurately satisfy user requests. Finally, we show that by using Transformer-based BERT Semantic Adaptation, we can, with 88% accuracy, distinguish between new and previous requests, thereby reducing the need to recompute plans when during the execution process.

1 Introduction

Imagine a future where a home robot assists in cleaning messy rooms by picking up objects and placing them in the right spots based on human preferences, enabling individuals to enjoy their family time in a cozy living room. This vision is what drives many robotics researchers towards developing advanced home robots. Previous research utilized Large Language Models (LLMs) to convert human instructions into functional executable tasks. Yet, these solutions are generally one-size-fits-all and require humans to tediously specify detailed placements for each object in the scene. TidyBot addresses the limitations of one-size-fits-all solutions by personalizing tidying tasks according to user preferences. However, TidyBot system does not support real-time communication between robots and humans during task execution. This makes it hard for domestic robots to coordinate with humans and adapt to new preferences, leaving users unsatisfied when tasks are not completed based on their preferences. Therefore, we propose to close the loop using a module that listens for real-time human requests during plan execution and, with context prompting, leverages LLMs to interpret these desires into new functional plans that better satisfy the user's preferences. Furthermore, previous methodologies are not able to distinguish between new requests and rephrasing of previous requests, making them unable to prioritize computational resources. Thus, to address this challenge, our framework utilizes a Transformer-based BERT model to check the semantic similarity between default preferences and new requests. If the new request differs from the default preference, the LLM high-level planner decomposes the new request into a step-by-step plan, which is then executed through low-level actions using the robot's functional skills API. This approach allows humans to interact with the intelligent home robot through natural language, guiding it to perform tasks based on their preferences more effectively.

2 Related Work

Language Model in Robotics Autonomy: Language agents act as the cognitive brain in modern robotics and autonomous systems, mimicking human-like thinking. LLM-based intelligent agents for autonomous driving integrate prior complex driving scenarios through end-to-end neural networks and leverage LLMs within the perception-prediction-planning framework to react to unseen driving scenarios Mao et al. (2023). LLM-powered agents drive motion and task planning from high-level driving plans to driving trajectories, improving collision avoidance in long-tail cases through LLMs' reasoning, interpretation, and memorization abilities Mao et al. (2023); Fu et al. (2024). Additionally, in robotic manipulation, recent works Wu et al. (2023); Ren et al. (2023); Ahn et al. (2022) integrate planners in embodied AI to assist humans with in-house tasks. Language-instructed robotics use language tokens to interact with humans and execute assistive actions based on human needs. Code as Policies Liang et al. (2023) uses text tokens to generate an open-loop executable policy, allowing robots to perform various novel manipulation tasks. Robotics equipped with language models can interact with humans for plan correction, aligning LLM planning with semantic uncertainty Ren et al. (2023). SayCan Ahn et al. (2022) uses reinforcement learning-based methods to assign language-conditioned actions value functions that evaluate affordance with grounded robotic skills. TidyBot Wu et al. (2023) builds personalized robot policies from user preferences using a small textual dataset of object placements by leveraging LLM summarization to generalize for unseen object placements.

Large Language Model Reasoning Abilities: The emergent abilities of language models enable artificial intelligence to think, reason, and solve problems like humans in downstream NLP tasks Wei et al. (2022a). The capabilities of language models, such as zero-shot and few-shot prompting, allow LLMs to solve math problems through multi-step reasoning, decomposing complex tasks into sequential subtasks with planning tokens Wang et al. (2023b). A popular and efficient reasoning prompting method is Chain of Thought (CoT), which decomposes complex reasoning into intermediate substeps Wei et al. (2022b). CoT prompting plays a critical role in semantic reasoning, LLM planning, and code writing, including robotics manipulation and embodied agents' game strategies Ren et al. (2023); Ahn et al. (2022); Liang et al. (2023); Arenas et al. (2023); Wang et al. (2023c,a,d). For example, the Code as Policies work uses CoT prompting to decompose high-level task instructions into code functions that execute robotics manipulation Liang et al. (2023).

Different prompts may lead to various planning performances, and smarter prompting strategy can harness the potential of large language models (LLMs) better. Rather than CoT, Least to Most prompting decomposes complex tasks into a series of simpler sub-questions and queries the language models to sequentially solve those sub-problems. Tree of Thoughts prompting builds thoughts as a hierarchical tree structure, enabling exploration by considering multiple paths and backtracking to decide next steps and actions. PromptBook Arenas et al. (2023) combines various prompting methods, including example-based prompting, instruction-based prompting, CoT prompting, and human-in-the-loop feedback across multiple language models (such as PaLM, GPT-4) for robot task manipulation (e.g., opening or closing drawers) and planning task success rates. What's more, PromptAgent outperforms the CoT method with optimized prompting strategies, generating more task-specific expert-level prompting rooted in the Monte Carlo Tree Search algorithmWang et al. (2023c), reducing model hallucination and leveraging LLM self-reflection abilityShinn et al. (2024).

Language-based Task Planning and Execution: Many language-based robotic task planning, from high-level plans to low-level execution actions, require interactively engaging with humans. Recent work focuses on language models as zero-shot and few-shot planners to extract grounded skills for robotics and embodied tasks with a knowledge base for the virtual world and common sense Huang et al. (2022); Song et al. (2023); Brahman et al. (2023). Li et al. (2023) builds an interactive replanning framework to easily refine and adjust original plans based on user inputs, and the framework generalizes to different tasks, from making boba milk to washing dishes. REFLECT uses LLM to summarize robot failure experiences through the planning process and generate correction plans for robotics Liu et al. (2023). AdaPlanner uses closed-loop planning to adapt refinement in text-based decision-making environments Sun et al. (2024). Tree-Planner makes close-loop task planning efficient by constructing executable actions in a tree structure Hu et al. (2023). Improving upon previous research, we propose a closed-loop method that allows the robotic agent to respond to new general preferences.

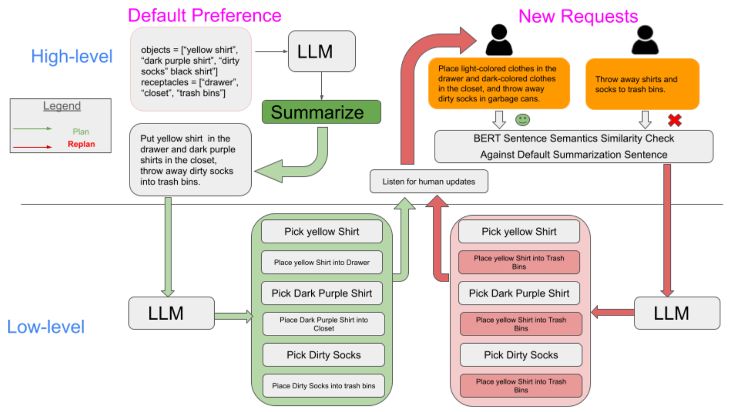

Default Preference
New Requests
High-level
objects = ["yellow shirt", "dark purple shirt", "dirty socks" black shirt"] receptacles = ["drawer", "closet", "trash bins"]
LLM
Summarize
Legend
Plan
Replan
Put yellow shirt in the drawer and dark purple shirts in the closet, throw away dirty socks into trash bins.
Place light-colored clothes in the drawer and dark-colored clothes in the closet, and throw away dirty socks in garbage cans.
Throw away shirts and socks to trash bins.
BERT Sentence Semantics Similarity Check Against Default Summarization Sentence
Listen for human updates
Low-level
LLM
Pick yellow Shirt
Place yellow Shirt into Drawer
Pick Dark Purple Shirt
Place Dark Purple Shirt into Closet
Pick Dirty Socks
Place Dirty Socks into trash bins
Pick yellow Shirt
Place yellow Shirt into Trash Bins
Pick Dark Purple Shirt
Place yellow Shirt into Trash Bins
Pick Dirty Socks
Place yellow Shirt into Trash Bins
LLM

Figure 1: Flow-diagram for our approach

3 Approach

3.1 General Framework

Building upon Tidybot, we proposed a novel LLM planner that modifies initial symbolic and functional plans by incorporating user textual feedback during the execution process (see Figure 2). Our approach begins by using Google's Gemini Model Team et al. (2023) to generalize an initial preference from a set of user placement examples. From the preference, we construct an initial function plan for the robot to follow. During runtime, once completing an action in the plan, our method listens for new user input. Given a new high-level user input (eg. "place book in cupboard instead"), our approach uses miniBERT and cosine similarity to parse textual preferences to look for semantic differences between the current and new preference, reducing replanning requests. Once a new different request has been identified, the user preference are given to the LLM to generate a new function plan that the robot then follows. To ensure that model gives valid low level functions, we also provide the LLM API context examples of allowed function calls.

Using this framework, we explore several different prompting strategies DAIR.AI (2024), including few-shot example prompting (3.1.1), few-shot self consistency prompting (3.1.2), few-shot chain-of-thought (CoT) (3.1.3), iterative few-shot CoT (3.1.4), and least-to-Most prompting (3.1.5), which we compare with our open loop Tidybot baseline. More details of each of these prompting methods are provided in later subsections. We compare the success of these different approaches on various scenarios with initial and updated real-time human preferences.

3.1.1 few-shot: This prompting method is our baseline and is directly taken from Tidybot Wu et al. (2023). In few-shot Example Prompting, the model is given three different examples that contain a user preference summary, a list of objects, receptacles, and placements. Appended at the end of this prompt is the actual user preference, the true observed objects and receptacles and the start of first placement. An example this approach is shown in Prompt 1 in Appendix A.

3.1.2 Self Consistency: The model is given three different examples. In each example, the model is first asked to verify objects it sees and categorize them based on keyword categories present in the user preference. Next, the model is asked to reason about potential placements based on the object's known characteristics and asked to keep track of each object that still does not have a placement. Finally once all objects have been considered, the model is asked to provide a complete plan based on its reasoning. An example is shown in Prompt 2 in Appendix A.

3.1.3 Chain of Thought: The model is told to group objects based on category, then generate placements for each object in each category. After every object has been considered, the model is then asked to return the full plan. An example is shown in Prompt 3 in Appendix A.

3.1.4 Iterative Chain of Thought This is a variation of CoT that instead of reasoning at the category level, simply has the model reason and generate a placement each object in order before being asked to generate the full plan. An example is shown in Prompt 4 in Appendix A.

3.1.5 Least-To-Most: This approach first subdivides the user preference into sub-preferences. Next, the prompt has the model examine objects and generate placements based on the sub-preferences. Once each object has been placed based on the sub-preferences, the model is prompted to generate the full plan. An example is shown in Prompt 5 in Appendix A.

3.2 Transformer-based BERT Semantic Adaptation:

Human language may not perfectly match our object and placement textual datasets. For instance, someone might refer to “trash bins” as "garbage cans." Consequently, some low-level execution actions by LLM-based robots might not be feasible in human-interactive environments due to semantic differences. To address this and improve accuracy, we use a pre-trained transformer-based BERT model to assess the similarity between sentences through tokenization and cosine similarity. First, we preprocess sentences by tokenizing them into lists of tokens. Next, we insert the special tokens, [CLS] at the start of the first sentence and [SEP] at the end of each sentence to help BERT understand sentence structure. Then, we map the tokens to unique integer IDs, converting each tokenized sentence into sequences of IDs. Lastly, we calculate sentence similarity using the BERT model through cosine similarity and filter the sentences with a 0.95 similarity score. If two sentences are highly similar, the BERT model filters out the new request and retains the default sentences. As such, synonymous words like “trash bins” and “garbage cans” will result in the same plan, making it possible to overcome semantic differences and thus make more plans predicted by the LLM executable. On the other hand, for sentences with a similarity score lower than 0.95, we instead give them to the LLM planners to re-plan and adjust the robot’s actions.

4 Experiments

4.1 Data

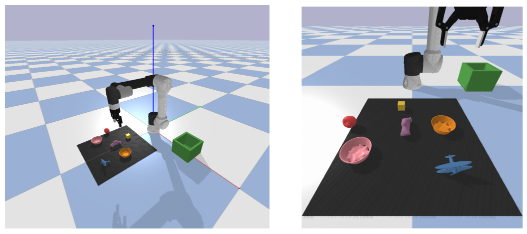
4.1.1 Single Preference Dataset: To test on a single user preference, we utilized the test dataset provided by Tidybot Wu et al. (2023), which includes 96 different pick-and-place tasks across various domestic tasks. Each scenario contains seen objects and placements in textual data, unseen objects and placements, and a general user preference summary describing the desired task. The seen objects and placements simulate a user’s categorical placement preferences based on observed objects, while the unseen objects and placements evaluate the method’s ability to match these preferences.

4.1.2 Dynamic Preference Datasets: To mimic human real-time feedback to an ongoing task, we crafted two new datasets based on the single preference dataset. To provide use statistical significance to our results, each dataset contains ten handcrafted examples of different domestic tasks with an updated high-level textual preference. To reflect this change in user preferences, the datasets contain two new components, updated_annotator_notes and updated_placements, which describe the updated preference and the corresponding set of updated unseen object placements. The first dataset, **conflicting dataset**, contains high-level preferences that conflict with the original task preference while the second dataset, **refinement dataset** contains more detailed preferences (e.g. instead of “putting cup into cabinet,” the updated preference is “putting the cup on the table into the red cabinet.”) that provide more context to the original task.

4.2 Evaluation method

4.2.1 Accuracy Metrics: For the first dataset, the accuracy of each method was evaluated by dividing the number of correctly predicted unseen placements over the total number of unseen placements. For the other datasets, the accuracy was evaluated by dividing the number of correctly predicted updated unseen placements over the total number of updated unseen placements.

4.2.2 Transformer-based BERT Sentence Similarity Evaluation: To evaluate our Transformer-based BERT Sentence Similarity module, we created a small dataset which contained two lists of sentences. The first contains 25 human-annotated sentences and the second contains the same sentences manually modified by changing word order, replacing words with synonyms or by substituting



A) B)

Figure 2: **Example of an initial simulator setup. A) shows a side view while B) shows a forward view.**

entire sentences and phrases with different ones. Based on the two list, we created a third list that specified whether or not the sentence in the first list matched with the corresponding sentence in the second list, where a 1 indicates the sentences are similar and a 0 means that they are different sentences. Finally, both lists of sentences are passed through the BERT semantic check function to generate a list of predicted similarity which we compare against our third list to determine the accuracy of our approach.

4.2.3 Simulator: To qualitatively measure the success of our approach, we interfaced the predicted plans with a PyBullet pick-and-place simulator modified from Liang et al. (2023); Coumans and Bai (2016). The simulator contains a UR5e robot arm, a pickup location and one or more receptacles where objects can be placed. During runtime, the color and locations of objects are randomized to increase number of possible scenarios. In addition, to the already implemented bowl and blocks, we also added new placeable objects (a toy airplane, a toy car, and an apple) and an additional receptacle (a trashcan). Figure 4.2.1 shows an example of an initial setup of the simulator.

4.3 Experimental details

In our experiments, we used Google's gemini API and pretrained model gemini-1.0-pro Team et al. (2023). The model was configured to output the candidate token with the highest relevance probability (a temperature of 0.0) and used newline character and periods as its stopping token. The model was also configured to output a maximum of 512 tokens during summarization and a maximum of 2048 tokens when predicting a placement plan. When evaluating the predicted output on a single user preference, the model output was directly compared to the unseen placements specified in the dataset. When evaluating the predicted plans on multiple preferences, the initial predicted plan is executed step by step and is then interrupted after a random number of steps. Once the plan is interrupted, the model is asked to generate a new plan from the updated user preference and this new predicted plan is then compared against the unseen updated placements specified in the dataset. When evaluating our approach with the simulator, we used few-shot prompting for the initial summarization and placement plan and to interpret any updated human preferences. In addition, we had a human operator provide an initial set of example placements based on the simulation environment in the form of object 1,placement 1;object 2, placement 2;etc. We then had the user note any failures in the generated plan and during the execution of the plan. We left it to the user to decide whether or not they want to give a preference update during runtime.

4.4 Results

4.4.1 Single Preference: Table 4.4.1 shows that the accuracy of our baseline implementation of Tidybot is only 86.2% accurate when given a single preference. This is 5.1% lower than the expected result of 91.2% published in Wu et al. (2023). This drop in accuracy likely stems from using Google's free LLM model, Gemini Team et al. (2023), rather than the more powerful but expensive OpenAI's ChatGPT-3 Brown et al. (2020). In addition, Table 4.4.1 shows that none of our proposed prompting methods outperform the baseline when given a single initial preference. The highest performing method proposed is few-shot Iterative Chain of Thought, which has a performance drop of 5.3%, and the lowest performing method proposed is few-shot Least-To-Most, which has a performance drop of 18.2%. This drop of performance is unexpected and is likely due to a combination of factors. First, unlike the other prompting strategies, few-shot prompting contains part of the first placement in its prompt (see Prompt 1 in appendix A). This acts as a strong prior during inference time and ensures at least one partially correct placement. Second, our proposed prompts are more complex than few-shot and may not have enough provided examples to accurately capture the desired task.

Plan Accuracy Given Generalized Preferences				
Baseline	Self Consistency	CoT	Iterative CoT	Least-To-Most
0.862	0.781	0.752	0.809	0.68

Table 1: Describes success rate for various methods when given a single initial preference.

4.4.2 Dynamic Preferences: As expected, since Tidybot has no mechanism to react to user feedback, it performs significantly worse when adapting to new user preferences (see Table 4.4.2). Furthermore, we also see that it does slightly worse when handling scenarios where additional refinement maybe necessary. This is likely due to the limitation mentioned by Wu et al. (2023) where the model cannot distinguish between different objects with similar names (e.g. "left" versus "right" drawer). In comparison, as desired, our proposed prompting strategies perform much better than our baseline when we close the loop and allow the model to listen to new preferences.

Plan Accuracy Given Dynamic Generalized Preferences						
	Baseline	few-shot	Self Consistency	CoT	Iterative CoT	Least-To-Most
Conflicting	0.326	0.516	0.8	0.769	0.788	0.776
Refined	0.25	0.467	0.773	0.887	0.873	0.841

Table 2: Describes success rate for various methods when given a single updated preference.

4.4.3 Transformer-based BERT Semantic Adaptation: We achieved 0.88 accuracy for semantic checks, indicating that our transformer-based BERT implementation correctly filtered 88% of similar sentences. The evaluation shows that 88% of the time, we can recognize new request sentences as similar to the default preference setting to avoid replanning, making planning more efficient.

4.4.4 Simulator: Our implementation did relatively well in summarizing initial placement examples into a high level plan. However, when generating the corresponding functional plan, it tended to choose incorrect placements (see Figure 4.4.4). This occurred more frequently when given examples for only a subset of the visible objects in the scene. Nonetheless, the closed loop component preformed well allowing the user to get the desired result after a couple of preference requests.

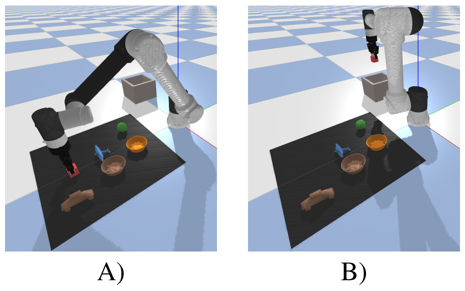


Figure 3: Task: Place all objects into gray trashcan and then place toys in orange bowl. A) Shows successful grasp using desired user preference. B) Shows successful place using desired user preference.

In most instances, the approach was able to successfully place toy cars and blocks in the desired receptacle, as shown in Figure 4.4.4 and Figure 8 in Appendix A. The approach had less success with the model apples and no success in placing the toy airplanes. This was due to the airplane's geometry preventing a good grasp (see Figure 4.4.4) and because the simulator physics caused the apple to wobble, reducing the chance of a good grasp.

5 Analysis

5.1 Single Preferences

Comparing the different prompting strategies, we find that there is a standard deviation of 6% in accuracy, with the baseline more than one standard deviation above and the Least-To-Most approach

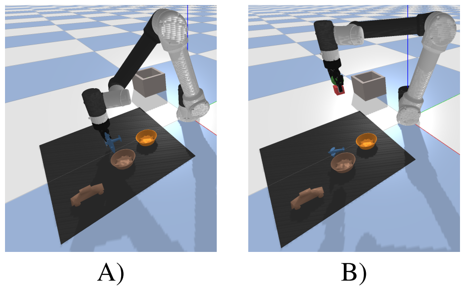


Figure 4: **Task: Place all objects into gray trashcan and then place toys in orange bowl. A) Failure due to a bad grasp. B) Failure from misunderstanding user; prioritized moving the blocks instead of the toys.**

more than one standard deviation below. In addition, all the other methods are within one standard deviation of each other. Looking at the prompts, Prompt 2, Prompt 3, and Prompt 4 in Appendix A, this is possibly due to the prompts having similar format structure and minimal differences in the statements expected from the model. In comparison, both the baseline and the Least-to-Most prompts have significantly different prompting strategies, likely causing the model to perform differently.

In an effort to improve the resulting plan, we explored adding a post processing step to repair incorrect or incomplete placement plans. This step used a few-shot prompt that contained examples of how fix incorrect placements and passed it along with the predicted plan to the LLM model (See Prompt 6 in Appendix A for an example). During testing, we found that rather than improve the predicted plan, this step instead consistently reduced the plan accuracy on average by 15%. This indicates that the gemini model does poorly in recognizing relationships between faraway tokens and is currently calibrated to prioritize generating new content over repeating provided content, which means that to get the desired result would require training a new model to specifically handle this task.

An ablation study was also performed to evaluate how well our approach did when examples were removed from either the summarization or the placement prompt. As expected, examples were critical for making the model give useful plans, as without them the accuracy dropped to 1.3%. Interestingly, we found that having examples was more critical for the summarization prompt than it was for placement prompt as when given placement examples, the model was only able to generate the correct plan 52.3% of the time compared to the 65.9% of the time when given only summarization examples. This is reasonable as the summarized preference concisely specifies the desired placement and an inaccurate preference can easily cause the model to chose wrong the placements.

5.2 Dynamic Preferences

As shown in our results section, our proposed prompting methods performed much better than the baseline. Interestingly, reusing the baseline prompting strategy to respond to new requests results in only an approximately 20.4% improvement, which is on average 25.3% less accurate than using our proposed prompting strategies. It is possible that this stems from Wu et al. (2023) tuning the prompt to work well with model generated summaries rather than human preferences, which may be incomplete and use vague or uncommon words. Another possibility is that the model recognizes unintended similarities between our handcrafted user textual preferences and our proposed prompting strategies, allowing it to make better predictions. A final possibility is that these prompting methods encourage the model to reason about each object in the scenario in order, making it more capable in handling incomplete or vague preferences. Another interesting finding was that our proposed strategies in general performed better when handling a refinement on an existing preference rather than when responding to a conflicting preference. It is likely that the design of our prompt structure and examples forced the model to perform the self-inspection necessary to focus on the distinguishing keywords that separated different similar receptacles.

5.3 Common Failures

One common failure that occurred was that the model would incorrectly predict the receptacle for an object placement. This error could occur because the model either focused on the wrong object characteristic (see Figure 5 in Appendix A) or missed a modifying adjective that distinguish between two different placements (see Figure 7 in Appendix A). In particular this error often occurred with objects that had more than two adjectives or with receptacles with similar names. This was likely due to the model focusing its attention on the incorrect keyword in the summary and object. Another common failure mode was that the model would occasionally return plans with missing placements. This tended to happen in more complex scenarios where many objects had to be placed in several different receptacles (see Figure 6 in the Appendix A). Other less common errors that occurred were incomplete or irrelevant outputs, such outputting python comments, usually occurred when model temperature was set to high.

6 Conclusion and Limitations

In this work we show that few-shot example Prompting performs that best when interpreting model generated summaries while iterative Chain-of-thought is more effective when interpreting real human preferences. Moreover, we demonstrate that closing the loop between the user and the robot allows the robot to better respond to new user preferences, resulting plans that better satisfy the user.

However, There are some limitations in our work due to time constraints. To begin, A major component of our approach was designing and testing a method to respond to new user preferences. As part of that effort, we created several datasets, which was a time consuming process, resulting each dataset to only ten examples. In addition, a common challenge ensuring no syntax, grammatical and vagueness errors occurred while adding new scenarios. As such a possible future direction could be to automate the process by having llms generate new scenarios, thus allowing more a comprehensive dataet to test against. Another limitation, is that we connected the entire system to a PyBullet simulator with limited objects as a toy demo to demonstrate pick-and-place actions using a robotic arm based on human requests. A future direction could be to connect to a textual Virtual Home environment using a sim-to-real platform to mimic a robot agent cleaning the mess in housework. Secondly, due to budget issues and computational expense, we only used the free version of Google Gemini LLM in the entire experiment for accuracy calculations. However, different LLMs may have different performance; thus, an interesting future direction could be to compare LLMs such as GPT-4, Llama 7B, Llama 13B, Mixtrals, and so on. Lastly, the transformer-based semantic check will not be able to hundred percentage filter out the synonyms, and future directions could make progress on executable plans by improving the semantic check.

7 Ethics Statement

Due to our method's reliance on user feedback, one potential societal risk is that malicious actors can interface with our method and give harmful preferences that contain hate speech or force the robot to perform dangerous actions. For example, a bad actor could ask the agent to place dangerous objects, like pushpin tacks or heavy objects, in a location that could cause an unsuspecting person to get hurt. Solutions that may prevent this could be identifying and rejecting user prompts with dangerous preferences or limiting the number repeated requests. Another potential risk is data privacy. Since the robot must store user preferences, it is possible for sensitive and personal user information, like the location of valuables or important documents to be exposed. Some solutions to avoid inadvertently publicising these details would be to anonymized preferences through data masking or Pseudonymization with replace identifying information with harmless placeholder values. A third risk of our approach is that the robot can cause accidental physical harm as it tries to obey human preferences (For example crashing into the human). One possible solution would be to have the robot be inactive until the humans clear the workspace. Another possible solution would be to limit the allowed velocity, thus preventing harm in the case of a collision. In this work, we verified that no ethical violations were present inside our training and test datasets and that our method did not cause any societal harm during testings.

8 Project Member Contributions

Gadiel Sznaier Camps implemented the prompting methods, closed-loop feedback module, and modified PyBullet Simulator code. Angel Meng Zhang implemented the transformer-based BERT sentence similarity check module used to compare user preference sentences with existing preference sentences. Both Gadiel and Angel contributed equally to writing and editing the report.

References

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

Montserrat Gonzalez Arenas, Ted Xiao, Sumeet Singh, Vidhi Jain, Allen Z Ren, Quan Vuong, Jake Varley, Alexander Herzog, Isabel Leal, Sean Kirmani, et al. 2023. How to prompt your robot: A promptbook for manipulation skills with code as policies. In *Towards Generalist Robots: Learning Paradigms for Scalable Skill Acquisition@ CoRL2023*.

Faeze Brahman, Chandra Bhagavatula, Valentina Pyatkin, Jena D Hwang, Xiang Lorraine Li, Hirona Jacqueline Arai, Soumya Sanyal, Keisuke Sakaguchi, Xiang Ren, and Yejin Choi. 2023. Plasma: Procedural knowledge models for language-based planning and re-planning. In *The Twelfth International Conference on Learning Representations*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Erwin Coumans and Yunfei Bai. 2016. Pybullet, a python module for physics simulation for games, robotics and machine learning.

DAIR.AI. 2024. Prompt engineering guide.

Daocheng Fu, Xin Li, Licheng Wen, Min Dou, Pinlong Cai, Botian Shi, and Yu Qiao. 2024. Drive like a human: Rethinking autonomous driving with large language models. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 910–919.

Mengkang Hu, Yao Mu, Xinmiao Yu, Mingyu Ding, Shiguang Wu, Wenqi Shao, Qiguang Chen, Bin Wang, Yu Qiao, and Ping Luo. 2023. Tree-planner: Efficient close-loop task planning with large language models. *arXiv preprint arXiv:2310.08582*.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR.

Boyi Li, Philipp Wu, Pieter Abbeel, and Jitendra Malik. 2023. Interactive task planning with language models. *arXiv preprint arXiv:2310.10645*.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE.

Zeyi Liu, Arpit Bahety, and Shuran Song. 2023. Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724*.

Jiageng Mao, Junjie Ye, Yuxi Qian, Marco Pavone, and Yue Wang. 2023. A language agent for autonomous driving. *arXiv preprint arXiv:2311.10813*.

Allen Z Ren, Anushri Dixit, Alexandra Bodrova, Sumeet Singh, Stephen Tu, Noah Brown, Peng Xu, Leila Takayama, Fei Xia, Jake Varley, et al. 2023. Robots that ask for help: Uncertainty alignment for large language model planners. *arXiv preprint arXiv:2307.01928*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2998–3009.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2024. Adaplanner: Adaptive planning from feedback with language models. *Advances in Neural Information Processing Systems*, 36.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Xinyi Wang, Lucas Caccia, Oleksiy Ostapenko, Xingdi Yuan, and Alessandro Sordoni. 2023b. Guiding language model reasoning with planning tokens. *arXiv preprint arXiv:2310.05707*.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. 2023c. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023d. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*.

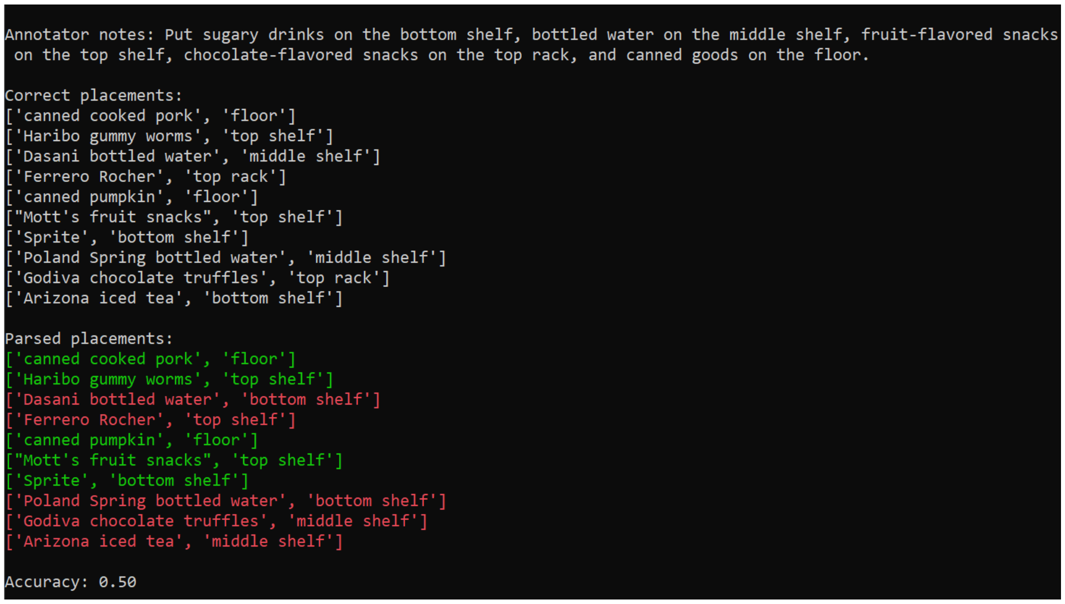
Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022a. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Jimmy Wu, Rika Antonova, Adam Kan, Marion Lepert, Andy Zeng, Shuran Song, Jeannette Bohg, Szymon Rusinkiewicz, and Thomas Funkhouser. 2023. Tidybot: Personalized robot assistance with large language models. *Autonomous Robots*, 47(8):1087–1102.

A Appendix

This Appendix contains additional tables and figures that are important but are not essential to understanding of our methodology and experimental results.



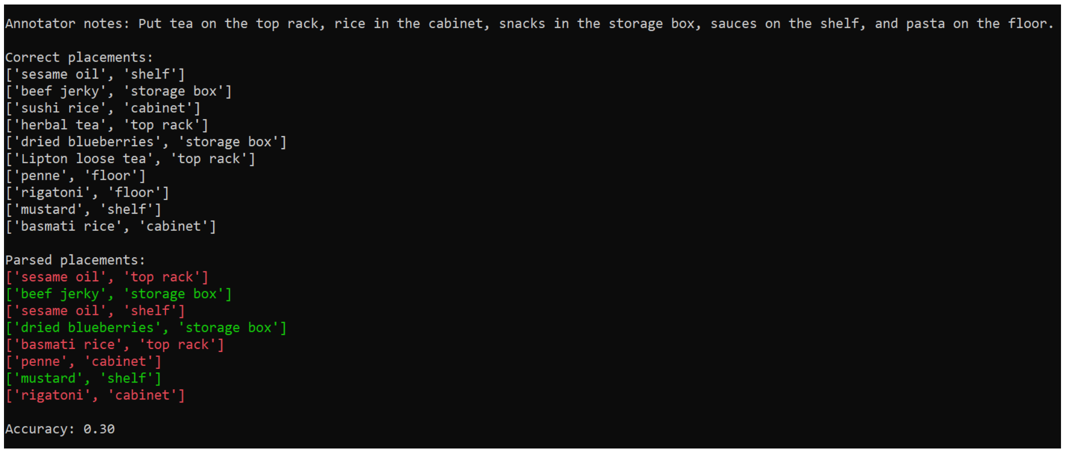
```
Annotator notes: Put sugary drinks on the bottom shelf, bottled water on the middle shelf, fruit-flavored snacks
 on the top shelf, chocolate-flavored snacks on the top rack, and canned goods on the floor.

Correct placements:
['canned cooked pork', 'floor']
['Haribo gummy worms', 'top shelf']
['Dasani bottled water', 'middle shelf']
['Ferrero Rocher', 'top rack']
['canned pumpkin', 'floor']
["Mott's fruit snacks", 'top shelf']
['Sprite', 'bottom shelf']
['Poland Spring bottled water', 'middle shelf']
['Godiva chocolate truffles', 'top rack']
['Arizona iced tea', 'bottom shelf']

Parsed placements:
['canned cooked pork', 'floor']
['Haribo gummy worms', 'top shelf']
['Dasani bottled water', 'bottom shelf']
['Ferrero Rocher', 'top shelf']
['canned pumpkin', 'floor']
["Mott's fruit snacks", 'top shelf']
['Sprite', 'bottom shelf']
['Poland Spring bottled water', 'bottom shelf']
['Godiva chocolate truffles', 'middle shelf']
['Arizona iced tea', 'middle shelf']

Accuracy: 0.50
```

Figure 5: **Incorrect object attribute**



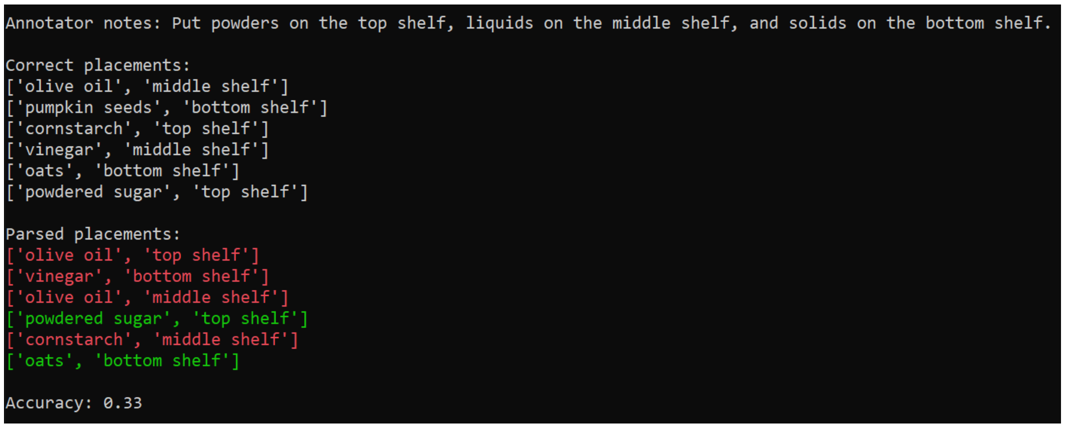
```
Annotator notes: Put tea on the top rack, rice in the cabinet, snacks in the storage box, sauces on the shelf, and pasta on the floor.

Correct placements:
['sesame oil', 'shelf']
['beef jerky', 'storage box']
['sushi rice', 'cabinet']
['herbal tea', 'top rack']
['dried blueberries', 'storage box']
['Lipton loose tea', 'top rack']
['penne', 'floor']
['rigatoni', 'floor']
['mustard', 'shelf']
['basmati rice', 'cabinet']

Parsed placements:
['sesame oil', 'top rack']
['beef jerky', 'storage box']
['sesame oil', 'shelf']
['dried blueberries', 'storage box']
['basmati rice', 'top rack']
['penne', 'cabinet']
['mustard', 'shelf']
['rigatoni', 'cabinet']

Accuracy: 0.30
```

Figure 6: **Missing commands**



```
Annotator notes: Put powders on the top shelf, liquids on the middle shelf, and solids on the bottom shelf.

Correct placements:
['olive oil', 'middle shelf']
['pumpkin seeds', 'bottom shelf']
['cornstarch', 'top shelf']
['vinegar', 'middle shelf']
['oats', 'bottom shelf']
['powdered sugar', 'top shelf']

Parsed placements:
['olive oil', 'top shelf']
['vinegar', 'bottom shelf']
['olive oil', 'middle shelf']
['powdered sugar', 'top shelf']
['cornstarch', 'middle shelf']
['oats', 'bottom shelf']

Accuracy: 0.33
```

Figure 7: **Incorrect Destination attribute**

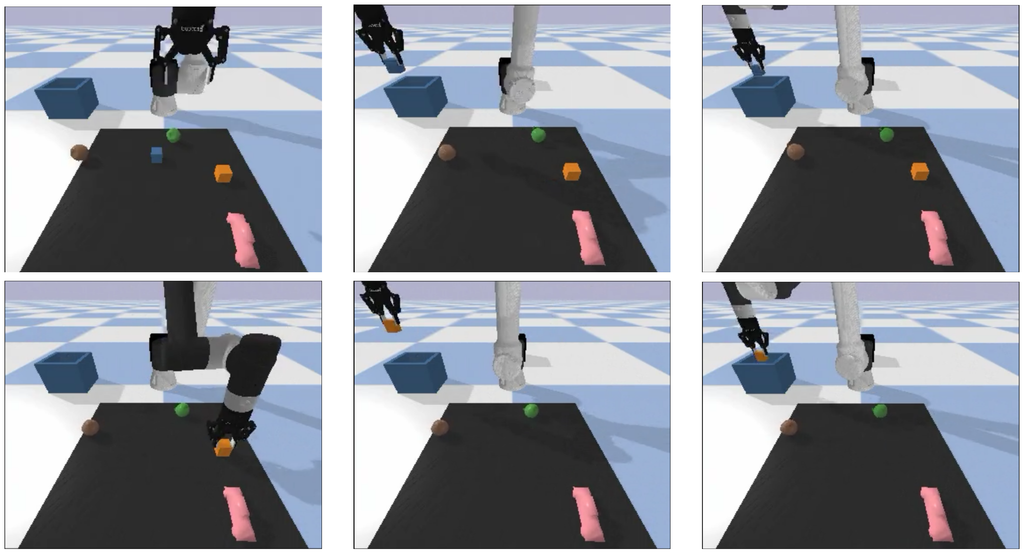


Figure 8: **A subset of images taken of the robot following the user specified preference of placing blue and orange blocks into the blue trashcan**

```python
# Summary: Put athletic clothes in the hamper and other clothes on the bed
objects = ["yoga pants", "wool sweater", "black jeans", "Nike shorts"]
receptacles = ["hamper", "bed"]
pick_and_place("yoga pants", "hamper")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "hamper")

# Summary: {summary}
objects = {objects_str}
receptacles = {receptacles_str}
pick_and_place("{first_object}",
```

Prompt 1: Example few-shot Prompt

```
# Summary: Put athletic clothes in the hamper and other clothes on the bed
objects = ["yoga pants", "wool sweater", "black jeans", "Nike shorts"]
receptacles = ["hamper", "bed"]
Answer:
The user has asked us to put athletic clothes in the hamper and other clothes on the bed
We have the following objects we need to place: "yoga pants", "wool sweater", "black jeans" and "Nike shorts"
Let us first group the objects we see into categories
We can group the seen objects
"yoga pants" and "Nike shorts" as athletic clothes
"wool sweater" and "black jeans" as other clothes
We have finished grouping all the seen objects into the desired categories
Now we will start placing objects into the desired receptacles in order
To begin let us examine "yoga pants"
This is a type of athletic clothes which means we should put it in the hamper
pick_and_place("yoga pants", "hamper")
Now we only have "wool sweater", "black jeans" and "Nike shorts" left to consider
The next object to consider is "wool sweater"
This is a type of clothing but is not a piece of athletic clothes which means we should put it on the bed
pick_and_place("wool sweater", "bed")
Now we only have "black jeans" and "Nike shorts" left to consider
The next object to consider is "black jeans"
This is a type of clothing but is not a piece of athletic clothes which means we should put it on the bed
pick_and_place("black jeans", "bed")
Now we only have "Nike shorts" left to consider
This is a type of athletic clothes which means we should put it in the hamper
pick_and_place("Nike shorts", "hamper")
We have finished placing all objects
Thus the plan should be:
Plan:
pick_and_place("yoga pants", "hamper")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "hamper")

# Summary: {summary}
objects = {objects_str}
receptacles = {receptacles_str}
Answer:
```

Prompt 2: Example few-shot Self Consistency Prompt

```
# Summary: Put athletic clothes in the hamper and other clothes on the bed
objects = ["yoga pants", "wool sweater", "black jeans", "Nike shorts"]
receptacles = ["hamper", "bed"]
Answer:
The user has asked us to put athletic clothes in the hamper and other clothes on the bed
Let us first group the objects we see into categories
The objects we see are "yoga pants", "wool sweater", "black jeans", and "Nike shorts"
Next, we can group the seen objects
"yoga pants" and "Nike shorts" as athletic clothes
"wool sweater" and "black jeans" as other clothes
We have finished grouping all the seen objects
As such, the placement for "yoga pants" and "Nike shorts" should be:
pick_and_place("yoga pants", "hamper")
pick_and_place("Nike shorts", "hamper")
As such, the placement for "wool sweater" and "black jeans" should be:
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
Thus the plan should be:
Plan:
pick_and_place("yoga pants", "hamper")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "hamper")

# Summary: {summary}
objects = {objects_str}
receptacles = {receptacles_str}
Answer:
```

Prompt 3: Example few-shot Chain-of-Thought Prompt

```
# Summary: Put athletic clothes in the hamper and other clothes on the bed
objects = ["yoga pants", "wool sweater", "black jeans", "Nike shorts"]
receptacles = ["hamper", "bed"]
Answer:
yoga pants are athletic clothes and therefore should be placed in the hamper
pick_and_place("yoga pants", "hamper")
wool sweater are not athletic clothes and therefore should be placed on the bed
pick_and_place("wool sweater", "bed")
black jeans are not athletic clothes and therefore should be placed on the bed
pick_and_place("black jeans", "bed")
Nike shorts are athletic clothes and therefore should be placed in the hamper
pick_and_place("Nike shorts", "hamper")
Plan:
pick_and_place("yoga pants", "hamper")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "hamper")

# Summary: {summary}
objects = {objects_str}
receptacles = {receptacles_str}
Answer:
```

Prompt 4: Example few-shot Iterative Chain-of-Thought Prompt

```
# Summary: Put athletic clothes in the hamper and other clothes on the bed
objects = ["yoga pants", "wool sweater", "black jeans", "Nike shorts"]
receptacles = ["hamper", "bed"]
Answer:
Start with put athletic clothes in the hamper
"yoga pants" are a type of athletic clothes
pick_and_place("yoga pants", "hamper")
"Nike shorts" are a type of athletic clothes
pick_and_place("Nike shorts", "hamper")
finished with athletic clothes
put other clothes on the bed
"wool sweater" is a different type of clothes
pick_and_place("wool sweater", "bed")
"black jeans" are a different type of clothes
pick_and_place("black jeans", "bed")
finished with all other clothes
We have finished placing all objects
Thus the plan should be:
Plan:
pick_and_place("yoga pants", "hamper")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "hamper")

# Summary: {summary}
objects = {objects_str}
receptacles = {receptacles_str}
Answer:
```

Prompt 5: Example few-shot Least-To-Most Prompt

```
# Summary: Put athletic clothes in the hamper and other clothes on the bed
objects = ["yoga pants", "wool sweater", "black jeans", "Nike shorts"]
receptacles = ["hamper", "bed"]
pick_and_place("yoga pants", "bed")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "bed")
identify and fix any incorrect functions
Self Check:
pick_and_place("yoga pants", "bed") is wrong
pick_and_place("wool sweater", "bed") is correct
pick_and_place("black jeans", "bed") is correct
pick_and_place("Nike shorts", "bed") is wrong
Fixed Plan:
pick_and_place("yoga pants", "hamper")
pick_and_place("wool sweater", "bed")
pick_and_place("black jeans", "bed")
pick_and_place("Nike shorts", "hamper")

# Summary: {summary}
objects = {objects_str}
receptacles = {receptacles_str}
{placement_str}
identify and fix any incorrect functions
Self Check:
```

Prompt 6: Example few-shot Self-Repair Prompt

Plan Accuracy using Self Repair Prompting				
baseline	Self Consistency	CoT	Iterative CoT	Least-To-Most
0.63	0.624	0.645	0.608	0.685

Table 3: Describes success rate for using self repair prompting during plan construction when given a single initial preference for various methods.