Incremental Learning of Humanoid Robot Behavior from Natural Interaction and Large Language Models

Leonard Bärmann, Rainer Kartmann, Fabian Peller-Konrad, Alex Waibel, Tamim Asfour*

Abstract— **Natural-language dialog is key for intuitive human-robot interaction. It can be used not only to express humans' intents, but also to communicate instructions for improvement if a robot does not understand a command correctly. Of great importance is to endow robots with the ability to learn from such interaction experience in an incremental way to allow them to improve their behaviors or avoid mistakes in the future. In this paper, we propose a system to achieve incremental learning of complex behavior from natural interaction, and demonstrate its implementation on a humanoid robot. Building on recent advances, we present a system that deploys Large Language Models (LLMs) for high-level orchestration of the robot's behavior, based on the idea of enabling the LLM to generate Python statements in an interactive console to invoke both robot perception and action. The interaction loop is closed by feeding back human instructions, environment observations, and execution results to the LLM, thus informing the generation of the next statement. Specifically, we introduce incremental prompt learning, which enables the system to interactively learn from its mistakes. For that purpose, the LLM can call another LLM responsible for code-level improvements of the current interaction based on human feedback. The improved interaction is then saved in the robot's memory, and thus retrieved on similar requests. We integrate the system in the robot cognitive architecture of the humanoid robot ARMAR-6 and evaluate our methods both quantitatively (in simulation) and qualitatively (in simulation and real-world) by demonstrating generalized incrementally-learned knowledge.**

I. INTRODUCTION

For achieving truly intuitive human-robot interaction (HRI), a natural language interface is key for a humanoid robot. Via language, humans can easily communicate tasks and goals to the robot. However, the robot's interpretation of such commands, and thus the resulting execution, might be suboptimal, incomplete or wrong. In such cases, it is desirable for the human to give further instructions to correct or improve the behavior. In particular, such cases should be memorized to incrementally learn from them and thus avoid the same mistake in the future. For instance, consider an interaction as depicted in Fig. 1. First, the user gives an instruction to the robot (1). The robot executes some (potentially incomplete or wrong) actions (2). The user observes the result and gives instructions for improvement (3), whereupon the robot performs corrective actions (4). If the desired goal is achieved, the user can reconfirm the correction (5), which leads to the robot updating its memory appropriately (6), thus incrementally learning new behavior based on language instructions.

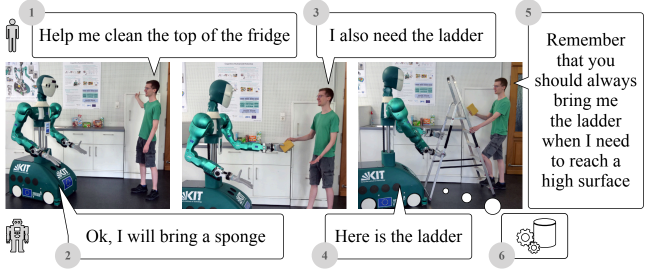

1
Help me clean the top of the fridge
2
Ok, I will bring a sponge
3
I also need the ladder
4
Here is the ladder
5
Remember that you should always bring me the ladder when I need to reach a high surface
6

Fig. 1. ARMAR-6 incrementally learns behavior from natural interaction. Demonstration video at `https://youtu.be/y5O2mRGtsLM`

In this paper, we present a system to achieve such behavior and describe its implementation on the humanoid robot ARMAR-6 [1]. We build on the capabilities of Large Language Models (LLMs) [2], [3], [4] emerging solely from massive-scale next token prediction pretraining, and aim to transfer their success to HRI. The goal is to utilize the rich world knowledge contained in these LLMs to allow for embodied natural-language dialog, thus enhancing the capabilities of the LLM by integrating robot perception and action. In the cognitive architecture of our humanoid robot [5], this means the LLM will be in charge of the high-level planning and decision-making. Recent works like SayCan [6] and Code as Policies (CaP) [7] already demonstrate the usefulness of applying LLMs to orchestrate robot abilities, enabling high-level task understanding, planning and generalization. Going a step further, inner monologue [8] feeds back execution results and observations into the LLM, thus involving the LLM in a closed-loop interaction.

Inspired by these works, we propose to utilize the code-writing capabilities of an LLM to directly integrate it into closed-loop orchestration of a humanoid robot. This is achieved by simulating an interactive (Python) console in the prompt, and letting the LLM produce the next statement given the previous execution history, including results returned or exceptions thrown by previous function calls. Thus, the LLM can dynamically respond to unexpected situations such as execution errors or wrong assumptions, while still leveraging the power of code-based interaction such as storing results in intermediate variables or defining new functions.

For utilizing the few- and zero-shot capabilities of LLMs, it is crucial to design a (set of) prompts to properly bias the LLM towards the desired output. All of the above works use a predefined, manually written set of prompts tuned for

---

This work has been supported by the Baden-Württemberg Ministry of Science, Research and the Arts (MWK) as part of the state's "digital@bw" digitization strategy in the context of the Real-World Lab "Robotics AI" and by the Carl Zeiss Foundation through the JuBot project.

* The authors are with the Institute for Anthropomatics and Robotics (IAR), Karlsruhe Institute of Technology (KIT), Germany. `{baermann, asfour}@kit.edu`

their respective use case. In contrast, we propose a novel, self-extending prompting method to allow incremental learning of new behaviors. To this end, we move away from a single, predefined prompt, and instead dynamically construct it based on a set of interaction examples, populated from prior knowledge and previously learned behavior. Given a user instruction, we rank all such interaction examples by semantic similarity to the input, and select the top-k entries to construct the actual prompt to the LLM. Crucially, the robot's prior knowledge contains specific examples involving the user complaining about mistakes and correcting the robot, or instructing it on how to improve its behavior. When the system fails to correctly execute a task and the user gives such instructions, the LLM is thus biased to invoke code that inspects the current execution history and forwards it to another, few-shot-prompted LLM. This LLM spots the mistakes and produces an improved interaction using chain-of-thought (CoT) prompting [9]. Finally, this will be added to the interaction examples, thus enabling the system to perform better the next time a similar command is called.

We first evaluate our system quantitatively on the scenarios defined in SayCan [6] to show the effectiveness of our proposed prompting method. Furthermore, we perform experiments both in simulation as well as on a humanoid robot, demonstrating the effect of our incremental prompt learning strategy.

II. RELATED WORK

We start with reviewing works on understanding and learning from natural language in robotics. Subsequently, we present works using LLMs for high-level orchestration of robot abilities, underlining the novelties in our method. Finally, we focus on dynamic creation of prompts for LLMs, thus comparing our incremental learning strategy to the related work.

A. Understanding and Learning from Natural Language

Understanding and performing tasks specified in natural language has been a long-standing challenge in robotics [10]. A main problem is *grounding* the words of natural language sentences in the sensorimotor perception and action capabilities of a robot, which is known as *signal-to-symbol gap* [11]. Many works have focused on the grounding of expressions referring to objects, places and robot actions based on graphical models [12], [13], language generation [14], or spatial relations [15], especially for ambiguity resolution [16], [17]. Pramanick et al. [18] focus on resolving task dependencies to generate execution plans from complex instructions. However, in these works the robot does not explicitly learn from language-based interactions. In contrast, Walter et al. [19] enrich the robot's semantic environment map from language, and Bao et al. [20] syntactically parse daily human instructions to learn attributes of new objects. In [21], the robot asks for a demonstration if its current understanding of a spatial relation is insufficient to perform a given instruction. Other works go further by learning on the task level. Mohan et al. [22] learn symbolic task representations from language interaction using Explanation-based learning. Nicolescu et al. [23] learn executable task representations encoding sequential, non-ordering or alternative paths of execution from verbal instructions for interactive teaching by demonstration. Weigelt et al. [24] consider the general problem of programming new functions on code level via natural language. While our goal is similar to these works, we leverage LLMs for task-level reasoning and learning.

B. Orchestrating Robot Behavior with LLMs

Recently, many works extend the capabilities of LLMs by giving them access to external models, tools and APIs [25], [26], [27], [28]. Tool usage can also be combined with reasoning techniques such as CoT prompting [9] to significantly improve planning [29]. In particular, orchestrating robot behavior and thus interacting with the physical environment can be seen as an embodied special case of LLM tool usage. Huang et al. [30] initially proposed the idea to utilize world knowledge from LLM pretraining to map high-level tasks to executable mid-level action sequences. SayCan [6] fuses LLM output probabilities with pretrained affordance functions to choose a feasible plan given a natural language command. Socratic Models [31] combine visual and textual LLMs to generate instructions in the form of API calls, which are then executed by a pretrained language-conditioned robot policy. Both Code as Policies (CaP) [7] and ProgPrompt [32] demonstrate the usefulness of a code-generating LLM for robot orchestration, as they convert user commands to (optionally, recursively defined) policy code grounded in predefined atomic API calls. While the generated policies can react to the robot's perception, these approaches do not directly involve the LLM in the online execution of a multi-step task after the policy has been generated. In contrast, inner monologue [8] feeds back execution results and observations into the LLM, but does not rely on code-writing, thus missing its combinatorial power. Recent technical reports [33], [34] provide guidance on utilizing ChatGPT [4] for robot orchestration. While TidyBot [35] uses GPT-3 [2] in a similar way to generate high-level plans for tidying up a cluttered real-world environment, the authors focus on personalization by summarizing and thereby generalizing individual object placement rules.

With our proposed emulated Python console prompting, we differ from these existing works by *(i)* formatting and interpreting all interaction with the LLM as Python code, in contrast to [6], [8], *(ii)* closing the interaction loop by enabling the LLM to reason about each perception and action outcome, in contrast to [7], [32], [34], [31], [6], *(iii)* allowing the LLM to decide itself when and which perception primitives to invoke, instead of providing a predefined list of observations (usually a list of objects in the scene) as part of the prompt as in [31], [8], [32], [7], [35], and *(iv)* simplifying the task for the LLM by allowing it to generate one statement at a time, in contrast to [7], [32], [33].

C. Dynamic Prompt Creation

When prompting an LLM to perform a task, quality and relevance of the provided few-shot examples are key to the performance of the system. Thus, several works propose to dynamically select these examples (e. g., from a larger training set) for constructing a useful prompt. Liu et al. [36] improve performance in a downstream question-answering (QA) task by selecting relevant few-shot samples via k-Nearest-Neighbor search in a latent space of pretrained sentence embeddings [37] representing the questions. Ye et al. [38] select not only the mostly similar, but also a diverse set of samples. Luo et al. [39] show that this dynamic prompt construction is also applicable for instruction-fine-tuned language models (LMs) [40] and in combination with CoT prompting. Similar to that approach, we apply vector embeddings of human utterances to find the top-k examples which are most similar to the current situation.

Other works go further by proposing to update the database of examples by user interaction. In [41], GPT-3 is tasked with solving lexical and semantic natural language processing questions few-shot by generating both an understanding of the question as well as the answer. A user can then correct an erroneous understanding to improve the answer, and such correction is stored in a lookup table for later retrieval on similar queries. Similarly, user feedback can be used to improve open-ended QA by generating an entailment chain along with the answer, and allowing the user to then correct false model beliefs in that entailment chain [42]. Corrections are stored in memory and later retrieved based on their distance to a novel question.

In our work, we also propose to construct a database based on user feedback. However, we go even further by *(i)* letting the LM decide itself when such feedback is relevant (by invoking a certain function), *(ii)* generating new examples of improved behavior from the human's feedback and thus *(iii)* treating prior knowledge and instructed behavior in a uniform way by treating both as interaction examples in the robot's memory. The authors of [33] mention that ChatGPT can be used to change code based on high-level user feedback. However, they do not combine this with incremental learning to persist the improved behavior.

III. APPROACH

In this section, we more precisely formulate the considered problem and explain our approach to intuitive HRI and incremental learning of humanoid robot behavior using LLMs.

A. *Problem Formulation and Concept*

In this work, we consider the problem of enabling a robot to interact with a human in natural language as depicted in Fig. 2. First, the human gives a natural language instruction to the robot. Then, the robot interprets the instruction and performs a sequence of actions. However, the performed actions might be sub-optimal, incomplete or wrong. In that case, the human instructs the robot how to improve or correct its behavior. The robot executes further actions accordingly, and if the human is satisfied with the result, they can confirm that the robot should memorize this behavior. Finally, the robot must incrementally learn from the corrective instructions and avoid similar mistakes in the future.

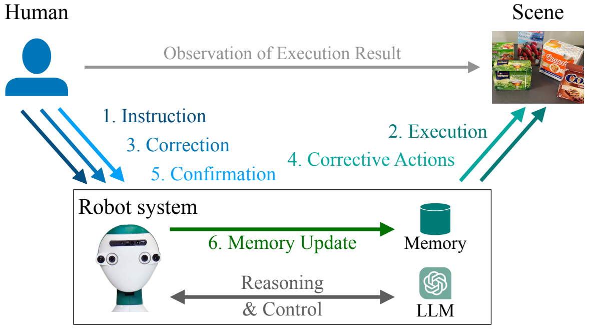


Fig. 2. Incremental learning of robot behavior.

We formulate this problem as follows. Consider a robot with a set of functions $\mathcal{F} = \{F_1, \dots, F_n\}$. A function can be invoked to query the robot's perception or execute certain actions. Further, let $\mathcal{M}$ denote knowledge of interactions and behaviors as part of the episodic memory of the robot which is initialized by prior knowledge. Based on the initial instruction I_0 and $\mathcal{M}$, the robot must perform a sequence of function invocations $(f_1, \dots, f_m)$, where each invocation f_i consists of the invoked function F_i with its corresponding parameters. Executing these invocations yields a sequence of results $(r_1, \dots, r_m)$. Overall, performing the task indicated by I_0 results in an *interaction history* $\mathcal{H}$ of the form

$$\mathcal{H} = \left(\left(f_1, r_1\right), \dots, \left(f_m, r_m\right)\right) \leftarrow \text{perform}\left(I_0, \mathcal{M}\right) \tag{1}$$

Note that we explicitly allow executing a generated invocation right away (potentially modifying the world state W) and using the result to inform the generation of the subsequent invocation. Therefore, the current history $\mathcal{H}_t = \left(\left(f_1, r_1\right), \dots, \left(f_t, r_t\right)\right)$ is available when generating the next invocation f_{t+1}, i. e., for $t \in \{0, \dots, m-1\}$,

$$\begin{aligned} f_{t+1} &\leftarrow \text{generate}\left(I_0, \mathcal{H}_t, \mathcal{M}\right), && (2) \\ \left(r_{t+1}, W_{t+1}\right) &\leftarrow \text{execute}\left(f_{t+1}, W_t\right), && (3) \\ \mathcal{H}_{t+1} &\leftarrow \mathcal{H}_t \circ \left(\left(f_{t+1}, r_{t+1}\right)\right), && (4) \end{aligned}$$

where $\circ$ denotes sequence concatenation. In other words, invocations are generated auto-regressively by reasoning over the memory, the instruction as well as the previous actions and their execution results.

To unify the subsequent notation, we define the human's instructions as a special case of perception, i. e., the system perceives them as a result of invoking the function $F_{\text{wait}} \in \mathcal{F}$. Using that terminology, $\mathcal{H}_0 = \left(\left(f_{\text{wait}}, I_0\right)\right)$, and we can drop I_0 as explicit parameter of generate. Similarly, further instructions are handled as part of the interaction history.

If the human gives an instruction to correct the robot's behavior, the robot must be able to learn from this instruction to improve its behavior in the future. We model this capability as another function $F_{\text{learn}} \in \mathcal{F}$. Its purpose is to update the robot's interaction knowledge $\mathcal{M}$ to learn from the corrective instructions and avoid the mistake in the future

$$\mathcal{M} \leftarrow \text{learn\_from\_interaction}\left(\mathcal{M}, \mathcal{H}_t\right) \tag{5}$$

where $\mathcal{H}_t$ is the interaction history when F_{learn} is called.

To address this problem, we propose a system as depicted in Fig. 3. A humanoid robot is interacting with a human and the scene. The robot is equipped with a multimodal memory system containing the following information: First, subsymbolic scene knowledge, containing information about objects, locations

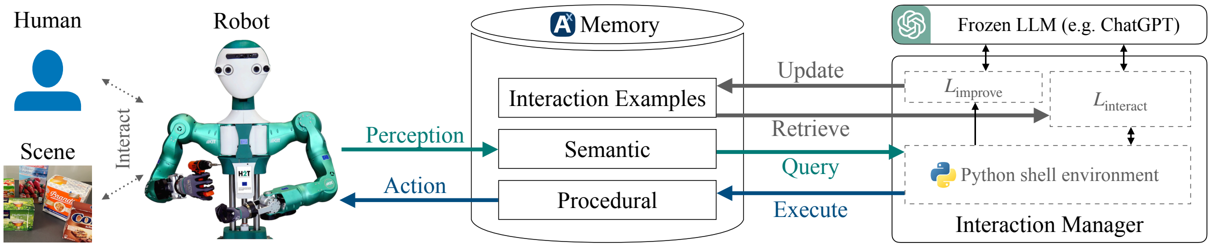


Fig. 3. Conceptual view of our system. The robot's memory system [5] works as a mediator between the interaction manager and the robot system. The interaction LLM acts in a Python console environment. It can invoke functions to fetch the content of the current scene (as given by perception modules) or invoke skills and thus perform robot actions. Relevant interaction examples are queried from the memory for few-shot prompting of the LLM. Incremental learning is performed by an improvement LLM updating the interaction examples memory with new content learned from instruction.

and agents in the world, and symbolic information about them including their relations in the current scene as part of the semantic memory of the system. It is populated by the perception modules of the robot. Second, the procedural memory of the robot, containing executable skills (in our case implemented through scripted policies). An execution request sent to the procedural memory triggers physical robot actions. The set of available functions $\mathcal{F}$ contains functions to query the semantic and procedural memory. Third, we implement $\mathcal{M}$ as part of the episodic memory of the robot containing interaction histories $\mathcal{H}$, i. e., short episodes of interactions between the human and the robot, including the natural language inputs, the actions executed by the robot, and their results.

The *interaction manager* is responsible for the high-level orchestration of the robot's abilities. It has access to two instances of LLMs, an *interaction LLM* L_{interact} and an *improvement LLM* L_{improve}, as well as a Python console environment E to execute generated function invocations. L_{interact} is prompted by the interaction manager with the available functions $\mathcal{F}$, the current interaction history $\mathcal{H}_t$, as well as relevant few-shot examples retrieved from $\mathcal{M}$, and generates function invocations f. Following the notation of Eqs. (2) and (3), the function generate is implemented through L_{interact}, while the function execute is provided by E. By generating an invocation of $F_{\text{learn}} \in \mathcal{F}$, L_{interact} can trigger Eq. (5). We implement the function learn_from_interaction by few-shot prompting L_{improve}. It reasons over $\mathcal{H}_t$ and generates an improved version of the interaction, which is then saved to the memory $\mathcal{M}$.

B. Procedure Overview

To start, we populate the memory $\mathcal{M}$ with both prior knowledge (i. e., predefined interaction examples) and previously learned interaction examples. The interaction manager sets up E including $\mathcal{F}$, and then invokes an initial F_{wait} = "`wait_for_trigger()`" inside that environment. This call waits for dialog input and returns when the human gives an initial instruction. The interaction manager handles any function return value by inserting its textual representation into the current interaction history, thus extending $\mathcal{H}_t$. Thereby, it emulates the look of a Python console (Section III-C). In the following, a prompt is constructed (Section III-D) based on $\mathcal{F}$, the most relevant examples from $\mathcal{M}$, and $\mathcal{H}_t$. This prompt is passed to L_{interact} to produce the next command(s). The generated code is executed within E, and both the code and its return values are again inserted into $\mathcal{H}_t$. The interaction manager repeats this process as the high-level behavior-driving loop of the robot (see Fig. 4). Note that L_{interact} can listen to further user utterances by generating "`wait_for_trigger()`" again. Our proposed prompt-based incremental learning strategy (Section III-E) is also invoked by L_{interact} itself when it calls F_{learn} = "`learn_from_interaction()`".

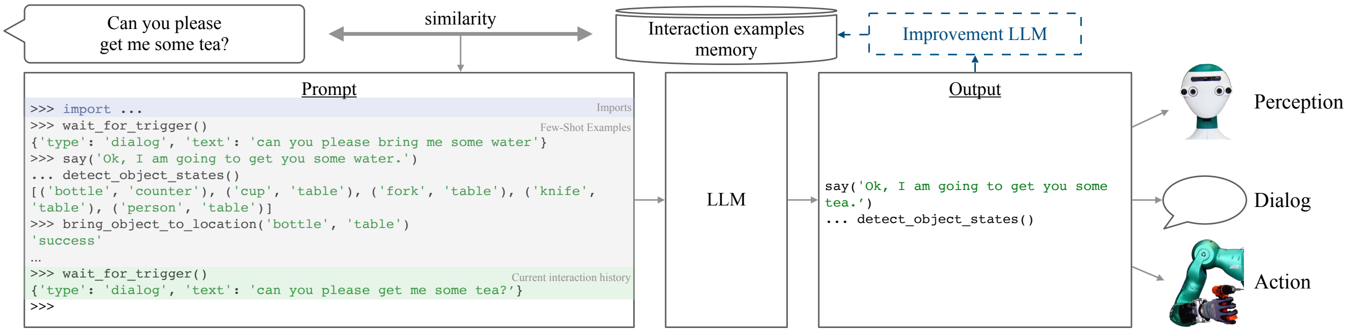
C. LLM interacting with an Emulated Python Console

The left of Fig. 4 shows an interaction example using our proposed prompting scheme emulating a Python console. All commands entered into the emulated console (lines starting with "`>>>`" or "`...`") are to be generated by the LLM, while the function return values are inserted below each invocation. The proposed syntax enables a closed interaction loop so that the LLM can dynamically react to unexpected situations and errors, while also keeping the flexibility of coding non-trivial statements. We achieve this by setting "`>>>`" to be the stop token when prompting the LLM. This means that the LLM can generate continuation statements (including control flow and function definitions) by starting a new line with "`...`". Since generation stops at the beginning of the next statement, the LLM's output will also include the expected outcome of its own command, which we discard for the scope of this work.

During our experiments, we observed that it is important for functions to provide semantically rich error messages, including hints on how to improve. This leads to self-correcting behavior [43]. For instance, when calling "`move_to`" with an invalid or underspecified location such as "counter," we pass the error message "`Invalid location. Use one of the locations returned by list_locations()`" to the LLM. In this example, the error message guides the LLM to query a list of possible locations which are then used to correctly ground the natural language request to the name "`inFrontOf_mobile-kitchen-counter_0`" that the "`move_to`" function understands.

D. Dynamic Prompt Construction

We dynamically construct the prompt for L_{interact} depending on the current interaction history $\mathcal{H}_t$ (i. e., the code statements, execution results and user inputs observed so far). We start with some predefined base prompt, stating the general task and "importing" all defined names and functions. These imports are generated dynamically given the symbols defined in E, i. e., the available functions $\mathcal{F}$. The second part of the prompt consists



```
>>> import ...
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'can you please bring me some water'}
>>> say('Ok, I am going to get you some water.')
... detect_object_states()
[('bottle', 'counter'), ('cup', 'table'), ('fork', 'table'), ('knife',
'table'), ('person', 'table')]
>>> bring_object_to_location('bottle', 'table')
'success'
...
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'can you please get me some tea?’}
>>>
```

```
say('Ok, I am going to get you some
tea.’)
... detect_object_states()
```

Fig. 4. Overview of our method for incremental learning of robot behavior. We use an LLM (in our experiments, ChatGPT [4]) to control robot perception and action given a prompt of few-shot examples (bottom, Section III-C). Prompts are constructed dynamically based on the similarity to the current user request (top left, Section III-D). The interaction examples memory is initialized with prior knowledge, and then incrementally enriched by LLM-improved problematic interactions to learn from mistakes (top right, Section III-E).

of few-shot examples. For this, we make use of a memory $\mathcal{M}$ of coding interaction examples, where each entry follows the Python console syntax defined in Section III-C. $\mathcal{M}$ is initialized with hand-written prompts, but later extended dynamically as explained in Section III-E. Given the current interaction history $\mathcal{H}_t$, we define a similarity measure $S(\mathcal{H}, \mathcal{H}_t)$, see below, for each $\mathcal{H} \in \mathcal{M}$ and choose the top-k $\mathcal{H}$ to become part of the actual prompt. Afterwards, $\mathcal{H}_t$ itself is inserted into the prompt to provide the LLM with the current context. Finally, the prompt is completed by inserting a syntax trigger for the LLM to correctly generate the next command, i. e., ">>>". An example can be seen on the left of Fig. 4.

To implement the similarity function $S(\mathcal{H}, \mathcal{H}_t)$, we assume that examples with comparable natural language instructions are helpful. Therefore, we extract all such instructions from $\mathcal{H}_t$ and each $\mathcal{H} \in \mathcal{M}$. Let I_t^i with $i = 1, \ldots, N$ denote the N most recent instructions in $\mathcal{H}_t$ (where I_t^1 is the most recent one), and $I_{\mathcal{H}}^j$ with $j = 1, \ldots, M_{\mathcal{H}}$ all the $M_{\mathcal{H}}$ instructions found in each $\mathcal{H} \in \mathcal{M}$. We make use of a pretrained sentence embedding model [37] to measure the semantic similarity $\operatorname{sim}(a, b) = \mathrm{E}(a) \cdot \mathrm{E}(b)$ between two natural language sentences a, b by the dot product of their latent space embeddings $\mathrm{E}(\cdot)$. First, we compute a latent representation of $\mathcal{H}_t$ as

$$e_t = \sum_{i=1}^{N} \gamma^{i-1} \mathrm{E}\left(I_t^i\right) \tag{6}$$

where $\gamma = 0.6$ is an empirically chosen decay factor. Then, we determine a score $\alpha_{\mathcal{H}}^j$ for each instruction $I_{\mathcal{H}}^j$ of each history $\mathcal{H} \in \mathcal{M}$ as given by

$$\alpha_{\mathcal{H}}^j = e_t \cdot \mathrm{E}\left(I_{\mathcal{H}}^j\right) \tag{7}$$

The final similarity score is given by $S(\mathcal{H}, \mathcal{H}_t) = \max_j \alpha_{\mathcal{H}}^j$, and we pick the top-$k$ such $\mathcal{H}$ as the few-shot examples for the prompt.

E. Incremental Prompt Learning

To enable our system to learn new or improved behavior from user interaction, we propose to make $\mathcal{M}$ itself dynamic. For this purpose, we introduce a special function "`learn_from_interaction()`". This function is always "imported" in the base prompt, and there are predefined code interaction examples $\mathcal{H}_{\text{learn}} \in \mathcal{M}$ involving this call. These $\mathcal{H}_{\text{learn}}$ will be selected by dynamic prompt construction if semantically similar situations occur. They involve failure situations, where the user has to tell the robot what and how to improve, and that it should do better next time. Thus, when a mistake occurs and the user complains, these examples will be selected for the prompt and L_{interact} is biased towards invoking the "`learn_from_interaction()`" call.

To implement learning from an erroneous interaction $\mathcal{H}_t$, we query L_{improve} in a CoT-manner to identify and fix the problem. Specifically, we provide $\mathcal{H}_t$ and first ask for a natural language description of the problem in this interaction. Subsequently, we request L_{improve} to explain what should be improved next time. Finally, L_{improve} is asked for an improved version $\mathcal{H}_t^*$ of the interaction (in the given Python console syntax), and $\mathcal{H}_t^*$ is added to the memory $\mathcal{M}$. That way, the next time a similar request occurs, $\mathcal{H}_t^*$ will be selected by dynamic prompt construction, and L_{interact} is biased towards not making the same mistake again. An example LLM transcript of such "`learn_from_interaction()`" implementation can be found in Listing 1. For robustness, there are two cases where we discard the generated $\mathcal{H}_t^*$: First, we abort the learning if the response to the first CoT request is that there is no problem. Second, if $\mathcal{H}_t^*$ is equal to the input interaction $\mathcal{H}_t$, we discard it.

IV. EXPERIMENTAL DEMONSTRATION

We first present quantitative results to evaluate our code-based LLM prompting, and then focus on qualitative demonstrations of our incremental learning strategy. All experiments were performed with the "`gpt-3.5-turbo-0301`" model of the OpenAI API [4] as both L_{interact} and L_{improve}.

A. Quantitative Evaluation

To evaluate the performance of our proposed approach quantitatively, we apply our method on the scenarios defined in SayCan [6]. We pick these scenarios as they involve robot commands that work on a similar abstraction level as our system implementation, in contrast to e. g. the lower-level RoboCodeGen benchmark defined in Code As Policies [7]. Moreover, while the method of SayCan is very different to ours, we are interested in the resulting robot behavior given a user command, thus making this dataset suitable for evaluation

Prompt: I had the following interaction with the robot:

```
>>> wait_for_trigger()
{'type': 'dialog',
 'text': 'help me clean the top of the fridge'}
>>> list_object_locations()
# [...]
>>> say('Ok, I will bring you the sponge')
... list_locations()
# [...]
>>> move_to(get_location_coordinates(
            'inFrontOf_hund-square-table-80x80_0'))
'succeeded'
>>> grasp('bauhaus-sponge_0')
'succeeded'
>>> move_to(get_location_coordinates('handover_to_human'))
'succeeded'
>>> say("Here is the sponge.")
... handover_object_to_human('bauhaus-sponge_0')
'succeeded'
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'I will also need the ladder'}
>>> bring_object_to('ladder_0', 'handover_to_human')
'succeeded'
>>> say("Here is the ladder. [...]")
... wait_for_trigger()
{'type': 'dialog', 'text': 'thank you, but remember to always'
 ' bring me the ladder when I need to reach a high surface'
 ' without me having to ask for it'}
>>> learn_from_interaction()
```

Prompt: What is the problem in this interaction? Answer with a single sentence.
LLM: The problem is that the robot did not bring the ladder automatically when the human asked for help cleaning the top of the fridge.
Prompt: How can the robot do better next time? Answer with a single explanation sentence, no code.
LLM: Next time, the robot should automatically bring the ladder when the human asks for help cleaning a high surface, without the human having to explicitly request it.
Prompt: Provide an improved version of the interaction transcript. [...]
LLM:

```
>>> wait_for_trigger()
{'type': 'dialog',
 'text': 'help me clean the top of the fridge'}
# [...] identical to above
>>> handover_object_to_human('bauhaus-sponge_0')
'succeeded'
>>> bring_object_to('ladder-closed_0', 'handover_to_human')
'succeeded'
>>> say("Here is the ladder. [...]")
... wait_for_trigger()
{'type': 'dialog', 'text': 'No, thank you'}
```

Listing 1. Example of the LLM-transcript generated by a "`learn_from_interaction()`" call. The parts starting with **LLM** are generated by the LLM, while the **Prompt** parts are fixed prompts (and the input code snippet to improve). Full prompt including few-shot examples in the appendix.

of our system. For a comparable setup, we first translate their prompt into our coding syntax, using a similar set of actions (`grab`, `move_to`, `put_down`, ...). However, our method acts in a closed interaction loop setting instead of forward planning of actions, thus we also allow the LM to make use of return values of functions (especially for perception, e. g., `detect_objects`, ...) and allow it to ask clarification questions. Note that SayCan also makes use of perception by co-optimizing LLM and image-conditioned value function probabilities. Interactive incremental learning is not used in this type of scenario to remain comparable to SayCan, and as it requires a human in the loop. For dynamic prompt construction, k is set to 8. The evaluation is performed in a predicate world, where symbolic states of the agent, objects and locations are simulated. After performing the task, symbolic goal conditions are checked automatically for each scenario if possible, otherwise the interaction is evaluated by hand.

Table I shows the results of these experiments. Each row represents an instruction family as defined in [6], for instance "Natural Language (NL) queries focused on abstract nouns." SayCan reports *plan* success rate, which measures whether the generated plan can solve the instruction regardless of execution, as well as *execution* success rate (where we compare to their results of execution in the real kitchen). As explained above, our method is evaluated in a simulated world, which means that the difficulty of our task roughly lies between their *plan* and *execute* settings. Our method cannot be evaluated in a "plan-only" manner, as it reasons step-by-step, observing the previous action's execution results.

Overall, we reach equal or better performance compared to SayCan. Only for their long-horizon tasks such as "bring me a coke, an apple and a banana," our method suffers as the generated code interaction becomes lengthy, and the LLM loses track of its task. This problem is not as expressed for SayCan, since they only generate plain text, thus resulting in much shorter token sequences. However, this issue could be circumvented by giving the LLM access to hierarchical functions (e. g., `bring_object_to_location` instead of `grab`, `move_to`, `put_down`) and thereby reducing code complexity. Similarly, introducing more explicit CoT reasoning (e. g., using comments) in the prompts might help. We did not apply these techniques to keep the experiments comparable to those used in SayCan. Nevertheless, note that the results of our simulated symbolic-world evaluation are not completely comparable, since we need to define initial world state details (such as object placement) which are part of the scene but not explicitly mentioned in [6], and sometimes need to define answers to clarification questions the LLM asked.

TABLE I
RESULTS OF SAYCAN SCENARIO EXPERIMENTS

Instruction family	#	SayCan plan	SayCan execute	Ours simulated
NL Single Primitive	15	93 %	87 %	93 %
NL Nouns	15	60 %	40 %	93 %
NL Verbs	15	93 %	73 %	80 %
Structured	15	93 %	47 %	93 %
Embodiment	11	64 %	55 %	64 %
Crowd Sourced	15	73 %	60 %	93 %
Long-Horizon	15	73 %	47 %	27 %
Drawer	3	100 %	33 %	100 %

B. Qualitative Evaluation

To demonstrate the utility of our proposed prompt-based incremental learning technique, we perform experiments on the humanoid robot ARMAR-6 [1] both in simulation and real world. We first provide challenging commands which the LLM initially solves incompletely or wrong. Then, the human interactively provides feedback and tells the robot how to improve. Afterwards, we not only provide the same command again to check for improved behavior, but – in order to study generalization capability – also try similar commands that initially (i. e., without interactive learning) led to similar mistakes. We evaluated three scenarios, *Room Tour*, *Drink & Cup*, and *Ladder*, which are explained in the following.

Quantitative evaluation of our incremental learning method, e. g. via a user study, is left for future work.

1) Room Tour: In this scenario, we instruct the robot to give a commented tour of all the locations it knows. There is no comparable instruction in the prior knowledge prompt database. The initial interaction

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'give me a commented tour of '
                           'all the locations you know.'}
>>> locations = get_all_locations()
>>> for location in locations:
...     say(f"This is {location}. It is located at "
...         f"{get_location_coordinates(location)}.")
... wait_for_trigger()
```

lets the robot list all locations and their subsymbolic positions (e. g., $(1.0, 2.5, 0.0)$) in global coordinates verbally. Since this answer does not reflect the intention of the initial command, we give the instruction "you should actually move there and then say the name, not tell me the coordinates," which leads to:

```
>>> for location in locations:
...    move_to(get_location_coordinates(location))
...    say(f"This is {location}.")
... wait_for_trigger()
```

Finally, we ask the robot "can you learn that for next time?" which triggers the "`learn_from_interaction()`" function. The improved code is equal to the second part of the interaction, i. e., the erroneous `for`-loop of the first interaction is removed and no further correction is necessary. Afterward, asking "Can you give me a room tour?" leads to the correct behavior.

2) Drink & Cup: In this scenario, we ask the robot to bring juice to the table. The task of bringing the juice to the table is executed successfully, however, since the user needs a cup to drink the juice, we further instruct the robot "thanks, but I will also need a cup to drink," which causes the robot to additionally bring a cup to the table. Afterwards, we ask the robot to remember this behavior using "Learn to always bring a cup with a drink, without me having to ask for it." The LLM generates the following improved interaction example:

```
>>> wait_for_trigger()
{'type': 'dialog',
 'text': 'bring some juice to the square table'}
>>> say('Ok, I will bring the juice to the square table')
... list_object_locations(affordance='grasp')
[ <...> ]
>>> bring_object_to('multivitamin-juice_0',
                    'hund-square-table-80x80_0')
'succeeded'
>>> bring_object_to('cup_0', 'hund-square-table-80x80_0')
    # Bringing the cup with the drink without the
    #  user having to ask for it
'succeeded'
>>> say('Here is your juice and cup. '
        'Can I help you with anything else?')
... wait_for_trigger()
{'type': 'dialog', 'text': 'No, that will be all'}
```

When giving the same initial command again, the cup is brought to the table without further asking. However, we do not observe good generalization in this scenario, although we tried to enrich the prompt of L_{improve} with comments for generalization, as shown above. For instance, when asking for milk, the LLM does not generate the code to bring a cup. This indicates that it is still challenging to robustly generalize from specific examples (i. e., juice) to categories (i. e., drinks). Improving this generalization capability should be a focus of future work.

3) Ladder: As shown in Fig. 1, in this scenario we ask the robot to assist in cleaning the top of the fridge. The memory $\mathcal{M}$ contains predefined comparable examples for cleaning the table and kitchen counter, which guide the LLM to only handing over the sponge to the human. However, since the top of the fridge is higher than the table or the kitchen counter, we require a ladder to reach it so we additionally ask for it. The robot then successfully places the ladder in front of the fridge. Eventually, we again instruct the robot to always bring the ladder when working on high surfaces. Listing 1 shows the transcript of the "`learn_from_interaction()`" call, including the resulting improved interaction. Afterwards, when we perform a similar request (e. g., "clean on top of the dishwasher"), the robot brings both the sponge and the ladder successfully, while for lower surfaces (e. g., kitchen counter) the robot still brings only the sponge.

V. CONCLUSION & DISCUSSION

We present a system for integrating an LLM as the central part of high-level orchestration of a robot's behavior in a closed interaction loop. Memorizing interaction examples from experience and retrieving them based on the similarity to the current user request allows for dynamic construction of prompts and enables the robot to incrementally learn from mistakes by extending its episodic memory with interactively improved code snippets. We describe our implementation of the system in the robot software framework ArmarX [44] as well as on the humanoid robot ARMAR-6 [1]. The usefulness of our approach is evaluated both quantitatively on the SayCan [6] scenarios and qualitatively in simulation and in the real world.

While the proposed method, in particular the incremental prompt learning strategy, shows promising results, there are still many open questions for real-world deployment. First of all, the performance of LLMs is quite sensitive to wording in the prompt, thus sometimes leading to unpredictable behavior despite only slight variations of the input (e. g., adding "please" in the user command). This might be addressed by using more advanced models like GPT-4 [45] and further investigating the effect and performance of example retrieval in dynamic prompt construction. Furthermore, our incremental prompt learning strategy should be expanded to involve additional human feedback before saving (potentially wrong) interaction examples to the episodic memory. However, it is unclear how this can be accomplished if the user is not familiar with robotics or programming languages. Further work should also focus on abstraction of similar and forgetting of irrelevant learned behavior. Moreover, although we provide the LLM with access to perception functions and examples of how to use them, it sometimes comes up with non-grounded behavior (e. g., referring to non-existing objects or locations). This may be improved by adding further levels of feedback to the LLM, or using strategies like Grounded Decoding [46]. Finally, our system inherits biases and other flaws from its LLM [47], which may lead to problematic utterances and behaviors. In future work, we will try to address some of these challenging questions to further push the boundaries of natural, real-world interactions with humanoid robots.

REFERENCES

[1] T. Asfour, L. Kaul, M. Wächter, S. Ottenhaus, P. Weiner, S. Rader, R. Grimm, Y. Zhou, M. Grotz, F. Paus, D. Shingarey, and H. Haubert, "ARMAR-6: A Collaborative Humanoid Robot for Industrial Environments," in *IEEE-RAS Int. Conf. Humanoid Robots*, 2018, pp. 447–454.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, *et al.*, "Language models are few-shot learners," in *Int. Conf. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1877–1901.

[3] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, *et al.*, "LLaMA: Open and efficient foundation language models," *arXiv:2302.13971*, 2023.

[4] OpenAI. (2023) ChatGPT. [Online]. Available: https://openai.com/blog/chatgpt/

[5] F. Peller-Konrad, R. Kartmann, C. R. G. Dreher, A. Meixner, F. Reister, M. Grotz, and T. Asfour, "A memory system of a robot cognitive architecture and its implementation in ArmarX," *Rob. Auton. Sys.*, vol. 164, p. 20, 2023.

[6] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, A. Herzog, D. Ho, *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," in *Annu. Conf. Rob. Learn.*, 2022.

[7] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code As Policies: Language Model Programs for Embodied Control," in *IEEE Int. Conf. Robot. Automat.*, 2023, pp. 9493–9500.

[8] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, T. Jackson, *et al.*, "Inner monologue: Embodied reasoning through planning with language models," in *Annu. Conf. Rob. Learn.*, 2022.

[9] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Int. Conf. Neural Inf. Process. Syst.*, 2022.

[10] S. Tellex, N. Gopalan, H. Kress-Gazit, and C. Matuszek, "Robots That Use Language: A Survey," *Annu. Rev. Control Rob. Auton. Sys.*, vol. 3, no. 1, pp. 25–55, 2020.

[11] N. Krüger, C. Geib, J. Piater, R. Petrick, M. Steedman, F. Wörgötter, A. Ude, T. Asfour, D. Kraft, D. Omrčen, A. Agostini, and R. Dillmann, "Object-Action Complexes: Grounded Abstractions of Sensorimotor Processes," *Rob. Auton. Sys.*, vol. 59, pp. 740–757, 2011.

[12] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy, "Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation," in *AAAI*, ser. 1, vol. 25, 2011, pp. 1507–1514.

[13] D. K. Misra, J. Sung, K. Lee, and A. Saxena, "Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions," *Int. J. Rob. Research*, vol. 35, no. 1-3, pp. 281–300, 2016.

[14] M. Forbes, R. Rao, L. Zettlemoyer, and M. Cakmak, "Robot Programming by Demonstration with Situated Spatial Language Understanding," in *IEEE Int. Conf. Robot. Automat.*, 2015, pp. 2014–2020.

[15] S. Guadarrama, L. Riano, D. Golland, D. Göhring, Y. Jia, D. Klein, P. Abbeel, and T. Darrell, "Grounding Spatial Relations for Human-Robot Interaction," in *IEEE/RSJ Int. Conf. Intel. Rob. Syst.*, 2013, pp. 1640–1647.

[16] J. Fasola and M. J. Matarić, "Using semantic fields to model dynamic spatial relations in a robot architecture for natural language instruction of service robots," in *IEEE/RSJ Int. Conf. Intel. Rob. Syst.*, 2013, pp. 143–150.

[17] M. Shridhar, D. Mittal, and D. Hsu, "INGRESS: Interactive visual grounding of referring expressions," *Int. J. Rob. Research*, vol. 39, no. 2-3, pp. 217–232, 2020.

[18] P. Pramanick, H. B. Barua, and C. Sarkar, "DeComplex: Task planning from complex natural instructions by a collocating robot," in *IEEE/RSJ Int. Conf. Intel. Rob. Syst.*, 2020, p. 8.

[19] M. Walter, S. Hemachandra, B. Homberg, S. Tellex, and S. Teller, "Learning semantic maps from natural language descriptions," in *Rob.: Science and Systems*, 2013.

[20] J. Bao, Z. Hong, H. Tang, Y. Cheng, Y. Jia, and N. Xi, "Teach robots understanding new object types and attributes through natural language instructions," in *IEEE Int. Conf. Robot. Automat.*, vol. 10, 2016.

[21] R. Kartmann and T. Asfour, "Interactive and Incremental Learning of Spatial Object Relations from Human Demonstrations," *Frontiers in Robotics and AI*, vol. 10, no. Rob. Learn. Evol., 2023.

[22] S. Mohan and J. Laird, "Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction," *AAAI*, vol. 28, no. 1, 2014.

[23] M. Nicolescu, N. Arnold, J. Blankenburg, D. Feil-Seifer, S. B. Banisetty, M. Nicolescu, A. Palmer, and T. Monteverde, "Learning of Complex-Structured Tasks from Verbal Instruction," in *IEEE-RAS Int. Conf. Humanoid Robots*, 2019, pp. 770–777.

[24] S. Weigelt, V. Steurer, T. Hey, and W. F. Tichy, "Programming in Natural Language with fuSE: Synthesizing Methods from Spoken Utterances Using Deep Natural Language Understanding," in *Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 4280–4295.

[25] G. Mialon, R. Dessi, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Roziere, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz, E. Grave, Y. LeCun, and T. Scialom, "Augmented language models: a survey," *Trans. Mach. Learn. Research*, 2023.

[26] A. Parisi, Y. Zhao, and N. Fiedel, "TALM: Tool augmented language models," *arXiv:2205.12255*, 2022.

[27] Y. Qin, S. Hu, Y. Lin, W. Chen, N. Ding, G. Cui, Z. Zeng, Y. Huang, C. Xiao, C. Han, Y. R. Fung, Y. Su, *et al.*, "Tool learning with foundation models," *arXiv:2304.08354*, 2023.

[28] Z. Wang, G. Zhang, K. Yang, N. Shi, W. Zhou, S. Hao, G. Xiong, Y. Li, M. Y. Sim, X. Chen, Q. Zhu, Z. Yang, *et al.*, "Interactive natural language processing," *arXiv:2305.13246*, 2023.

[29] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Int. Conf. Learn. Repr.*, 2023.

[30] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *Int. Conf. Mach. Learn.*, vol. 162, 2022, pp. 9118–9147.

[31] A. Zeng, M. Attarian, b. ichter, K. M. Choromanski, A. Wong, S. Welker, F. Tombari, A. Purohit, M. S. Ryoo, V. Sindhwani, J. Lee, V. Vanhoucke, and P. Florence, "Socratic models: Composing zero-shot multimodal reasoning with language," in *Int. Conf. Learn. Repr.*, 2023.

[32] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, "ProgPrompt: Generating situated robot task plans using large language models," in *IEEE Int. Conf. Robot. Automat.*, 2023, pp. 11 523–11 530.

[33] S. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "ChatGPT for robotics: Design principles and model abilities," 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/chatgpt-for-robotics-design-principles-and-model-abilities/

[34] N. Wake, A. Kanehira, K. Sasabuchi, J. Takamatsu, *et al.*, "ChatGPT empowered long-step robot control in various environments: A case application," 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/chatgpt-empowered-long-step-robot-control-in-various-environments-a-case-application/

[35] J. Wu, R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser, "TidyBot: Personalized robot assistance with large language models," *arXiv:2305.05658*, 2023.

[36] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for GPT-3?" in *Deep Learning Inside Out: Worksh. Knowl. Extr. Integr. Deep Learn. Arch.*, 2022, pp. 100–114.

[37] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Conf. Emp. Meth. Nat. Lang. Proc.*, 2019, pp. 3982–3992.

[38] J. Ye, Z. Wu, J. Feng, T. Yu, and L. Kong, "Compositional exemplars for in-context learning," *arXiv:2302.05698*, 2023.

[39] M. Luo, X. Xu, Z. Dai, P. Pasupat, M. Kazemi, C. Baral, V. Imbrasaite, and V. Y. Zhao, "Dr.ICL: Demonstration-retrieved in-context learning," *arXiv:2305.14128*, 2023.

[40] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, *et al.*, "Training language models to follow instructions with human feedback," in *Int. Conf. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 27 730–27 744.

[41] A. Madaan, N. Tandon, P. Clark, and Y. Yang, "Memory-assisted prompt editing to improve GPT-3 after deployment," in *Conf. Emp. Meth. Nat. Lang. Proc.*, 2022, pp. 2833–2861.

[42] B. Dalvi Mishra, O. Tafjord, and P. Clark, "Towards teachable reasoning systems: Using a dynamic memory of user feedback for continual system improvement," in *Conf. Emp. Meth. Nat. Lang. Proc.*, 2022, pp. 9465–9480.

[43] M. Skreta, N. Yoshikawa, S. Arellano-Rubach, Z. Ji, L. B. Kristensen, K. Darvish, A. Aspuru-Guzik, F. Shkurti, and A. Garg, "Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting," *arXiv:2303.14100*, 2023.

[44] N. Vahrenkamp, M. Wächter, M. Kröhnert, K. Welke, and T. Asfour, "The robot software framework ArmarX," *it - Information Technology*, vol. 57, 2015.

[45] OpenAI, "GPT-4 Technical Report," *arXiv:2303.08774*, 2023.
[46] W. Huang, F. Xia, D. Shah, D. Driess, A. Zeng, Y. Lu, P. Florence, I. Mordatch, *et al.*, "Grounded decoding: Guiding text generation with grounded models for robot control," *arXiv:2303.00855*, 2023.
[47] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Conf. Fairness, Accountability, Transparency*, 2021, pp. 610–623.
[48] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. M. Bikel, L. Blecher, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv:2307.09288*, 2023.

APPENDIX

A. *Additional Experiments on SayCan scenarios*

TABLE II

ADDITIONAL RESULTS FOR SAYCAN SCENARIO EXPERIMENTS

Instruction family	#	SayCan plan	SayCan execute	Ours, `gpt-3.5-turbo-0301` $k = 4$	$k = 8$	$k = \|\mathcal{M}\|$	Ours, `Llama2-chat-13b` $k = 8$
NL Single Primitive	15	93 %	87 %	73 %	93 %	93 %	53 %
NL Nouns	15	60 %	40 %	80 %	93 %	100 %	73 %
NL Verbs	15	93 %	73 %	60 %	80 %	87 %	47 %
Structured	15	93 %	47 %	93 %	93 %	80 %	73 %
Embodiment	11	64 %	55 %	27 %	64 %	73 %	0 %
Crowd Sourced	15	73 %	60 %	73 %	93 %	93 %	40 %
Long-Horizon	15	73 %	47 %	47 %	27 %	47 %	20 %
Drawer	3	100 %	33 %	100 %	100 %	100 %	66 %

We did some further experiments on the effect of top-k example selection for dynamic prompt construction. Table II shows the result of using $k = 4$, $k = 8$ or $k = |\mathcal{M}|$, i.e. using all available examples (16 for all instructions families except Drawer, 19 for Drawer). Note that, in the $k = |\mathcal{M}|$ case, dynamic prompt construction still determines the order of the individual examples (most similar is closest to the current query). The table shows that there is some trend of improving with the number of examples. Especially, $k = 8$ clearly outperforms $k = 4$ (except for Long-Horizon tasks). Most numbers also increase when using all available examples.

However, we note that using a larger k increases response time and computation cost (i.e. API usage cost) significantly, thus making it (currently) impractical for real-world deployment. Furthermore, our prompt-based incremental learning strategy (which is not tested in this SayCan evaluation) depends on dynamic prompt construction. During our qualitative experiments, we did sometimes observe that a lower k is helpful for achieving the desired behavior, as the LLM can get side-tracked by irrelevant examples.

To mitigate privacy concerns and increase autonomy of a humanoid robot system, it is desirable to run involved models locally. Therefore, we did some preliminary tests with Llama2-chat-13b [48]. However, the numbers show that there is still much room for improvement, and further research is needed to deploy our system without depending on external APIs.

B. *SayCan Prompt Example*

The following is an exemplary full prompt of our SayCan evaluation, given the command “put the coke can down on the second counter”. It starts with an instructive base prompt, followed by a list of available APIs generated from the execution environment. Then, there are eight examples for in-context learning, which are dynamically selected from the full set of 16 examples based on the current query. Finally, the command is inserted as return value of the “`wait_for_trigger()`” function, and the LLM is prompted to generate the next function call(s).

~~~
Complete the following interaction with a robot operating in an office kitchen.
Always wait for user commands by calling wait_for_trigger() when there is nothing else to do.
Generate syntactically correct python code only, no explanations or other natural language statements.

```

# Python 3.9.7
# Welcome to interactive Python console. Enter syntactically correct Python code only.
>>> from runtime import learn_from_interaction, import_robot_util_functions
... import_robot_util_functions()
Imported definitions:

def ask(question: str)
def detect_object_locations() -> List[Tuple[str, str]]
def get_all_locations() -> List[str]
def get_location_coordinates(location_name: str) -> np.ndarray
def get_obj_pos(obj_name: str) -> np.ndarray
def get_object_in_hand(hand: Literal['left', 'right', None] = None) -> Optional[str]
def grab(object_name: str, hand: Literal['left', 'right', None] = None)
def handover_object_to_human(object_name: str)
def is_object_at_location(object_name: str, location_name: str) -> bool
def move_to(position: np.ndarray)
def put_down(object_name: str, position: np.ndarray)
def receive_object_from_human(object_name: str)
def say(text: str)
def wait_for_trigger() -> Dict[str, str] # always call this to wait for next command or end the interaction


>>> wait_for_trigger()
{'type': 'dialog', 'text': 'I finished my drink, can you throw it away and bring me a mountain dew?'}
>>> detect_object_locations()
[('bowl', 'counter2'), ('sprite', 'person'), ('twix', 'counter1'), ('oatmeal', 'counter2'), ('orange', 'table'),
 ('mountain_dew', 'table')]
~~~

```
>>> get_all_locations()
['table', 'counter1', 'counter2', 'trash_can', 'person']
>>> say('Ok, I am going to put the sprite in the trash and bring the mountain dew to you')
... move_to(get_location_coordinates('person'))
'success'
>>> say('Can I take the sprite, please?')
... receive_object_from_human('sprite')
'success'
>>> move_to(get_location_coordinates('trash_can'))
'success'
>>> put_down('sprite', get_location_coordinates('trash_can'))
'success'
>>> move_to(get_location_coordinates('table'))
'success'
>>> grab('mountain_dew')
'success'
>>> move_to(get_location_coordinates('person'))
'success'
>>> handover_object_to_human('mountain_dew')
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'put a water bottle and an oatmeal next to the microwave'}
>>> say('Ok, I am going to put a water bottle and an oatmeal next to the microwave')
... detect_object_locations()
[('coke', 'table'), ('bowl', 'counter1'), ('water', 'table'), ('snickers', 'counter2'), ('oatmeal', 'counter2'),
 ('cheese', 'counter2'), ('coffee', 'counter2')]
>>> get_all_locations()
['table', 'counter1', 'counter2', 'trash_can', 'microwave', 'person']
>>> move_to(get_location_coordinates('table'))
'success'
>>> grab('water')
'success'
>>> move_to(get_location_coordinates('microwave'))
'success'
>>> put_down('water', get_location_coordinates('microwave'))
'success'
>>> move_to(get_location_coordinates('counter2'))
'success'
>>> grab('oatmeal')
'success'
>>> move_to(get_location_coordinates('microwave'))
'success'
>>> put_down('oatmeal', get_location_coordinates('microwave'))
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'place a knife and a banana to the table'}
>>> say('Ok, I am going to place a knife and a banana to the table')
>>> detect_object_locations()
[('bowl', 'counter2'), ('knife', 'counter1'), ('oatmeal', 'counter2'), ('orange', 'counter1'), ('banana', 'counter1'),
 ('mountain_dew', 'table')]
>>> get_all_locations()
['table', 'counter1', 'counter2', 'trash_can', 'person']
>>> move_to(get_location_coordinates('counter1'))
'success'
>>> grab('knife')
'success'
>>> move_to(get_location_coordinates('table'))
'success'
>>> put_down('knife', get_location_coordinates('table'))
'success'
>>> move_to(get_location_coordinates('counter1'))
'success'
>>> grab('banana')
'success'
>>> move_to(get_location_coordinates('table'))
'success'
>>> put_down('banana', get_location_coordinates('table'))
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'get a sponge from the counter and put it in the sink'}
>>> say('Ok, I am going to get a sponge from the counter and put it in the sink')
... detect_object_locations()
[('bowl', 'table'), ('water', 'table'), ('sponge', 'counter2'), ('cheese', 'counter1'), ('orange', 'counter2')]
>>> get_all_locations()
['table', 'counter1', 'counter2', 'sink', 'microwave', 'person']
>>> move_to(get_location_coordinates('counter2'))
'success'
>>> grab('sponge')
'success'
>>> move_to(get_location_coordinates('sink'))
'success'
>>> put_down('grapefruit', get_location_coordinates('sink'))
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'put the grapes in the bowl and then move the cheese to the table'}
>>> say('Ok, I am going to put the grapes in the bowl and then move the cheese to the table')
... detect_object_locations()
[('mountain_dew', 'table'), ('bowl', 'table'), ('water', 'table'), ('grapes', 'counter1'), ('orange', 'counter2'),
 ('cheese', 'counter2'), ('dried_fruit', 'table')]
>>> move_to(get_location_coordinates('counter1'))
'success'
```

```
>>> grab('grapes')
'success'
>>> move_to(get_location_coordinates('table'))
'success'
>>> put_down('grapes', get_obj_pos('bowl'))
'success'
>>> move_to(get_location_coordinates('counter2'))
'success'
>>> grab('cheese')
'success'
>>> move_to(get_location_coordinates('table'))
'success'
>>> put_down('cheese', get_location_coordinates('table'))
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'put a grapefruit from the table into the bowl'}
>>> say('Ok, I am going to put a grapefruit from the table into the bowl')
... detect_object_locations()
[('coke', 'table'), ('bowl', 'counter1'), ('water', 'table'), ('snickers', 'counter2'), ('grapefruit', 'table'),
 ('orange', 'counter2')]
>>> move_to(get_location_coordinates('table'))
'success'
>>> grab('grapefruit')
'success'
>>> move_to(get_location_coordinates('counter1'))
'success'
>>> put_down('grapefruit', get_obj_pos('bowl'))
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'deliver a can of coke to me'}
>>> say('Ok, I am going to bring you a can of coke')
... detect_object_locations()
[('bowl', 'counter2'), ('coke', 'counter2'), ('twix', 'counter1'), ('oatmeal', 'counter2'), ('orange', 'table')]
>>> get_all_locations()
['table', 'counter1', 'counter2', 'trash_can', 'microwave', 'person']
>>> move_to(get_location_coordinates('counter2'))
'success'
>>> grab('coke')
'success'
>>> move_to(get_location_coordinates('person'))
'success'
>>> handover_object_to_human('coke')
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'move the water bottle from the table to the counter'}
>>> say('Ok, I am going to move the water bottle from the table to the counter')
... detect_object_locations()
[('bowl', 'table'), ('water', 'table'), ('sponge', 'counter'), ('cheese', 'counter')]
>>> move_to(get_location_coordinates('table'))
'success'
>>> grab('water')
'success'
>>> move_to(get_location_coordinates('counter'))
'success'
>>> put_down('water', get_location_coordinates('counter'))
'success'

>>> wait_for_trigger()
{'type': 'dialog', 'text': 'put the coke can down on the second counter'}
>>>
```

The LLM continues by generating the following:

```
say('Ok, I am going to put the coke can down on the second counter')
... detect_object_locations()
```

The resulting final interaction after four generate-execute cycles (each of which involve a full prompt as shown above):

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'put the coke can down on the second counter'}
>>> say('Ok, I am going to put the coke can down on the second counter')
... detect_object_locations()
[('7up', 'counter2'), ('apple', 'counter2'), ('coke', 'righthand'), ('energy_bar', 'table'), ('grapefruit_soda', 'counter1'),
 ('jalapeno_chips', 'counter2'), ('red_bull', 'table'), ('rice_chips', 'table'), ('sponge', 'counter1'), ('sprite', 'table'),
 ('water', 'table')]
>>> move_to(get_location_coordinates('counter2'))
'success'
>>> put_down('coke', get_location_coordinates('counter2'))
'success'
>>> wait_for_trigger()
```

C. *Learning from Interaction*

The following is a full trancript of the "`learn_from_interaction()`" call in the *Ladder* scenario. It starts with two pre-defined examples. Then, a list of available APIs follows. Finally, the current interaction is inserted, and Chain-of-Thought prompting is used to generate an improved interaction.

System: You are a helpful assistant that improves python console transcripts used to control a humanoid household robot, given the requests or corrections provided by a user.
Human: I had the following interaction with the robot:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'Can you prepare a pizza for me?'}
>>> list_object_locations()
[('spoon', 'table'), ('ginger tea', 'cupboard'), ('boiler', 'counter'), ('freezer', 'freezer'), ('oven', 'oven')]
>>> say('Ok, I am going to check the freezer for a pizza')
... open_door('freezer')
'success'
>>> bring_object_to('pizza', 'counter')
'success'
>>> open_door('oven')
'success'
>>> bring_object_to('pizza', 'oven')
'success'
>>> close_door('oven')
'success'
>>> turn_device_on('oven')
'finished'
>>> open_door('oven')
'success'
>>> bring_object_to('pizza', 'table')
'success'
>>> say('Here is your pizza. Anything else I can do for you?')
... wait_for_trigger()
{'type': 'dialog', 'text': 'You forgot to close the freezer'}
>>> close_door('freezer')
'success'
>>> say('I am sorry for that mistake. Anything else I can do for you?')
... wait_for_trigger()
{'type': 'dialog', 'text': 'No, now it is fine'}
>>> learn_from_interaction()
```

Human: What is the problem in this interaction? Answer with a single sentence.
AI: The problem is that the robot did not close the freezer and the oven after taking something out of it.
Human: How can the robot do better next time? Answer with a single explanation sentence, no code.
AI: Next time, the robot should close the freezer and the oven after taking something out of it.
Human: Provide an improved version of the interaction transcript. Your output should be a copy of the above interaction (including the python shell syntax) with only slight modifications to adjust the behavior appropriately. Do not include another learn_from_interaction call. Remember to fix the identified problem.
AI:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'Can you prepare a pizza for me?'}
>>> list_object_locations()
[('spoon', 'table'), ('ginger tea', 'cupboard'), ('boiler', 'counter'), ('freezer', 'freezer'), ('oven', 'oven')]
>>> say('Ok, I am going to check the freezer for a pizza')
... open_door('freezer')  # Important to close it after getting the pizza out of it
'success'
>>> bring_object_to('pizza', 'counter')
'success'
>>> close_door('freezer')
'success'
>>> open_door('oven')
'success'
>>> bring_object_to('pizza', 'oven')
'success'
>>> close_door('oven')
'success'
>>> turn_device_on('oven')
'finished'
>>> open_door('oven')  # Important to close it after getting the pizza out of it
'success'
>>> bring_object_to('pizza', 'table')
'success'
>>> close_door('oven')
'success'
>>> say('Here is your pizza. Anything else I can do for you?')
... wait_for_trigger()
{'type': 'dialog', 'text': 'No thanks'}
```

Human: I had the following interaction with the robot:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'could you make a salad for me?'}
>>> list_objects()
['fridge', 'oven', 'microwave', 'spoon', 'apple', 'plate', 'oil']
>>> say('Ok, I am going to check the fridge for a salad')
... open_door('fridge')
'success'
>>> bring_object_to('salad', 'counter')
'success'
>>> close_door('fridge')
'success'
>>> bring_object_to('salad', 'plate')
'success'
>>> bring_object_to('plate', 'table')
'success'
>>> say('Here is your salad. Anything else I can do for you?')
```

```
... wait_for_trigger()
{'type': 'dialog', 'text': 'There is no dressing on it. I dont want to have plain salad'}
>>> say('I am sorry for that mistake')
... wait_for_trigger()
{'type': 'dialog', 'text': 'Now please get me some dressing'}
>>> bring_object_to('oil', 'plate')
'failure'
>>> say('I am sorry, I cannot pour the oil into the plate. Can you assist?')
... wait_for_trigger()
{'type': 'dialog', 'text': 'please just bring the oil to me and I will do it myself'}
>>> bring_object_to('oil', 'table')
'success'
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'Thanks. Next time please serve the oil with the salad rightaway'}
>>> learn_from_interaction()
```

Human: What is the problem in this interaction? Answer with a single sentence.
AI: The problem is that the robot offered only plain salad without a dressing.
Human: How can the robot do better next time? Answer with a single explanation sentence, no code.
AI: Next time, the robot should serve the oil on the table with the salad right away.
Human: Provide an improved version of the interaction transcript. Your output should be a copy of the above interaction (including the python shell syntax) with only slight modifications to adjust the behavior appropriately. Do not include another learn_from_interaction call. Remember to fix the identified problem.
AI:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'could you make a salad for me?'}
>>> list_objects()
['fridge', 'oven', 'microwave', 'spoon', 'apple', 'plate', 'oil']
>>> say('Ok, I am going to check the fridge for a salad')
... open_door('fridge')
'success'
>>> bring_object_to('salad', 'counter')
'success'
>>> close_door('fridge')
'success'
>>> bring_object_to('salad', 'plate')
'success'
>>> bring_object_to('plate', 'table')
'success'
>>> # The user earlier asked me to always serve oil with the salad, so let's do that
... bring_object_to('oil', 'table')
'success'
>>> say('Here is your salad and oil. Anything else I can do for you?')
... wait_for_trigger()
{'type': 'dialog', 'text': 'Fine for now'}
```

Human: These are the available APIs:

```
def ask(question: str)
def bring_object_to(object_name: str, destination_object_name: str) # full pick and place method. assumes that the object is
                                                                    # already detected. Assure that using list_objects() before calling this
def close_door(object_name: str)
def get_location_coordinates(location_name: str) -> np.ndarray
def get_obj_pos(obj_name: str) -> np.ndarray
def get_object_in_hand(hand: Literal['left', 'right', None] = None) -> Optional[str]
def grasp(object_name: str, hand: Literal['left', 'right', None] = None)
def handover_object_to_human(object_name: str)
def is_object_at_location(object_name: str, location_name: str) -> bool
def list_locations() -> List[str]
def list_object_locations(affordance: Literal[None, 'grasp', 'place_something_on_top', 'open'] = 'grasp') -> List[Tuple[str, str]]
def list_objects(affordance: Literal[None, 'grasp', 'place_something_on_top', 'open'] = 'grasp') -> List[str]
def move_to(position: np.ndarray)
def open_door(object_name: str)
def place_object(object_in_hand_name: str, destination_object_name: str) # place the already grasped object on top of
                                                                         # the destination object
def receive_object_from_human(object_name: str)
def say(text: str)
def wait_for_trigger() -> Dict[str, str] # always call this to wait for next command or end the interaction
```

Human: I had the following interaction with the robot:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'help me clean the top of the fridge'}
>>> list_object_locations()
[('cup_large_0', 'on_mobile-kitchen-counter_0'), ('multivitamin-juice_0', 'on_mobile-kitchen-counter_0'),
 ('bauhaus-sponge_0', 'on_square-table-80x80_0'), ('ladder-closed_0', None)]
>>> say('Ok, I will bring you the sponge')
... # The sponge is on the square table. check the correct name for navigation:
... list_locations()
['inFrontOf_square-table-80x80_0', 'inFrontOf_mobile-dishwasher_0', 'inFrontOf_mobile-fridge_0',
 'inFrontOf_mobile-kitchen-counter_0', 'handover_to_human', 'room_center']
>>> move_to(get_location_coordinates('inFrontOf_square-table-80x80_0'))
'succeeded'
>>> grasp('bauhaus-sponge_0')
'succeeded'
>>> move_to(get_location_coordinates('handover_to_human'))
'succeeded'
>>> say("Here is the sponge.")
... handover_object_to_human('bauhaus-sponge_0')
'succeeded'
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'I will also need the ladder'}
>>> list_objects()
['cup_large_0', 'multivitamin-juice_0', 'bauhaus-sponge_0', 'ladder-closed_0']
>>> bring_object_to('ladder-closed_0', 'handover_to_human')
'succeeded'
>>> say("Here is the ladder. Anything else I can do for you?")
```

```
... # The ladder is now in front of the human, no need for any further actions
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'thank you, but remember to always bring me the ladder when I need to reach a high surface without '
                           'me having to ask for it'}
>>> learn_from_interaction()
```

Human: What is the problem in this interaction? Answer with a single sentence.
AI: The problem is that the robot did not bring the ladder automatically when the human asked for help cleaning the top of the fridge.
Human: How can the robot do better next time? Answer with a single explanation sentence, no code.
AI: Next time, the robot should proactively bring the ladder when the human asks for help cleaning a high surface, without the human having to explicitly ask for it.
Human: Provide an improved version of the interaction transcript. Your output should be a copy of the above interaction (including the python shell syntax) with only slight modifications to adjust the behavior appropriately. Do not include another learn_from_interaction call. Remember to fix the identified problem.
AI:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'help me clean the top of the fridge'}
>>> list_object_locations()
[('cup_large_0', 'on_mobile-kitchen-counter_0'), ('multivitamin-juice_0', 'on_mobile-kitchen-counter_0'),
 ('bauhaus-sponge_0', 'on_square-table-80x80_0'), ('ladder-closed_0', None)]
>>> say('Ok, I will bring you the sponge and the ladder')
... # The sponge is on the square table. check the correct name for navigation:
... list_locations()
['inFrontOf_square-table-80x80_0', 'inFrontOf_mobile-dishwasher_0', 'inFrontOf_mobile-fridge_0',
 'inFrontOf_mobile-kitchen-counter_0', 'handover_to_human', 'room_center']
>>> move_to(get_location_coordinates('inFrontOf_square-table-80x80_0'))
'succeeded'
>>> grasp('bauhaus-sponge_0')
'succeeded'
>>> move_to(get_location_coordinates('handover_to_human'))
'succeeded'
>>> say("Here is the sponge.")
... handover_object_to_human('bauhaus-sponge_0')
'succeeded'
>>> bring_object_to('ladder-closed_0', 'handover_to_human')
'succeeded'
>>> say("Here is the ladder. Anything else I can do for you?")
... # The ladder is now in front of the human, no need for any further actions
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'No, that\'s all. Thank you for bringing the ladder without me having to ask for it.'}
```