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

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

Institute for Anthropomatics and Robotics (IAR), Karlsruhe Institute of Technology (KIT), Germany

Correspondence*:
Leonard Bärmann
baermann@kit.edu

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 let robots 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 such incremental learning of complex high-level behavior from natural interaction, and demonstrate its implementation on a humanoid robot. Our system 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. Human instructions, environment observations, and execution results are fed back to the LLM, thus informing the generation of the next statement. Since an LLM can misunderstand (potentially ambiguous) user instructions, we introduce incremental learning from interaction, which enables the system to 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. Subsequently, we store the improved interaction in the robot's memory so that it can later be retrieved on semantically 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.

Keywords: Incremental Learning, Human-Robot Interaction, Cognitive Modeling, Knowledge Representation for Robots, Humanoid Robots, Large Language Models

Content: ≈ 7400 Words, 5 Figures, 2 Tables, 1 Listing

1 INTRODUCTION

Humans can easily communicate tasks and goals to a robot via language. Such natural language interface is key for achieving truly intuitive human-robot interaction (HRI). However, the robot's interpretation of such commands, and thus the resulting execution, might be sub-optimal, incomplete or wrong. In such cases, it is desirable for the human to give further instructions to correct or improve the robot's behavior.

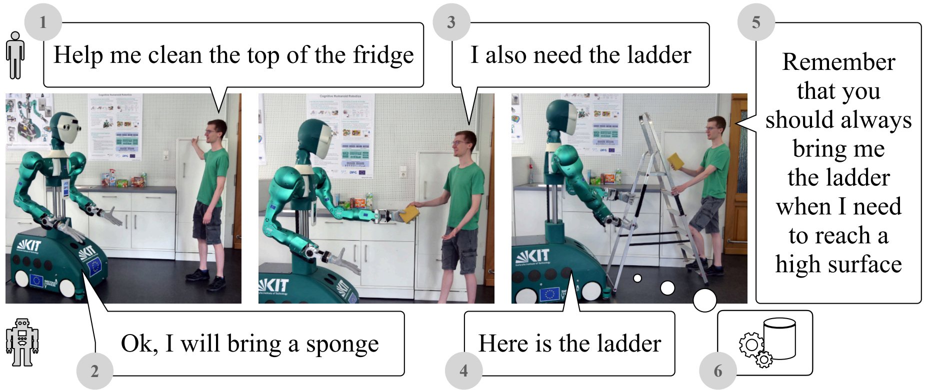

Figure 1. ARMAR-6 incrementally learns behavior from natural interaction. Demonstration videos at `https://lbaermann.github.io/interactive-incremental-robot-behavior-learning/`

Furthermore, the robot should memorize the improvement strategy given by the human to incrementally learn from them and thus avoid the same mistake in the future. For instance, consider the interaction depicted in Fig. 1. First, the user instructs the robot to help him cleaning the top of the fridge (1). The robot then executes several actions to hand over a sponge to the human (2). The user observes this insufficient result and gives instructions for improvement ("I also need a ladder") (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.

In this paper, we present a system to achieve such behavior and describe its implementation on the humanoid robot ARMAR-6 (Asfour et al., 2018). We build on the capabilities of Large Language Models (LLMs) (Brown et al., 2020; Touvron et al., 2023; OpenAI, 2023a,b) emerging 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 LLMs 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 (Peller-Konrad et al., 2023), this means the LLM will be in charge of the high-level planning and decision-making. Recent works like SayCan (Ahn et al., 2022) and Code as Policies (CaP) (Liang et al., 2023) 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 (Huang et al., 2022b) 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 LLMs 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 their respective use case. However, no LLM or prompting scheme will always interpret each user instruction correctly, especially since natural language can be ambiguous and correct execution might depend on user preferences. Therefore, we propose a novel, self-extending prompting method to allow incremental learning of new and adaptation of existing high-level behaviors. To this end, our system dynamically constructs prompts based on a set of interaction examples, populated from the robot's 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. Therefore, when the system fails to correctly execute a task and the user gives such corrective instructions, the LLM is biased to invoke code that inspects the current execution history and forwards it to another, few-shot-prompted LLM. This LLM can inspect the complete interaction including all user inputs, performed actions and observed results, represented as the transcript of an interactive Python console. It then spots the mistakes and produces an improved interaction using chain-of-thought (CoT) prompting (Wei et al., 2022). Finally, the improved transcript will be added to the interaction examples, thus enabling the system to perform better the next time a similar task is requested.

Our method is explained in detail in Section 3. We evaluate our system quantitatively on the scenarios defined in CaP (Liang et al., 2023) to show the effectiveness of our proposed approach in Section 4. Furthermore, Section 5 demonstrates the capabilities of incremental learning from natural-language interaction on a real-world humanoid robot. Our code can be found at `https://github.com/lbaermann/interactive-incremental-robot-behavior-learning`.

2 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.Finally, we focus on dynamic creation of prompts for LLMs.

2.1 Understanding and Learning from Natural Language

Understanding and performing tasks specified in natural language has been a long-standing challenge in robotics (Tellex et al., 2020). Of great challenge is *grounding* the words of natural language sentences in the robot's perception and action, which is known as *signal-to-symbol gap* (Krüger et al., 2011). Many works have focused on the grounding of expressions referring to objects, places and robot actions based on graphical models (Tellex et al., 2011; Misra et al., 2016), language generation (Forbes et al., 2015), or spatial relations (Guadarrama et al., 2013), especially for ambiguity resolution (Fasola and Matarić, 2013; Shridhar et al., 2020). Pramanick et al. (2020) 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. (2013) enrich the robot's semantic environment map from language, and Bao et al. (2016) syntactically parse daily human instructions to learn attributes of new objects. In Kartmann and Asfour (2023), 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 and Laird (2014) learn symbolic task representations from language interaction using Explanation-based learning. Nicolescu et al. (2019) learn executable task representations encoding sequential, non-ordering or alternative paths of execution from verbal instructions for interactive teaching by demonstration. Weigelt et al. (2020) 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.

2.2 Code-Generation and Interaction with LLMs

Generating code from natural language specifications is a large area of active research. For instance, LLMs tuned specifically on code (Chen et al., 2021; Nijkamp et al., 2023) perform well in common code-generation benchmarks. Madaan et al. (2022b) show that code-based models have more structured representations, thus aiding structured (e.g. graph-based) tasks. Training code-LLMs can also benefit from using an interpreter in the optimization loop (Le et al., 2022; Haluptzok et al., 2023). We refer the reader to recent surveys (Zheng et al., 2024; Ahmed et al., 2023; Dehaerne et al., 2022; Wang and Chen, 2023) for a more in-depth discussion.

Another recent trend is to use LLMs in an interactive, chat-style format. This became popular through OpenAI's models (OpenAI, 2023a,b) and is typically powered by finetuning on alignment data using reinforcement learning from human feedback (Ouyang et al., 2022). In a code-based setting, such interaction can, for instance, assist software development (Lahiri et al., 2023; Google, 2023). Further, many recent works utilize interactive coding strategies to deploy LLMs as agents (Yang et al., 2024). For instance, Voyager (Wang et al., 2024a) iteratively learns to master the game of Minecraft by letting an LLM code functions, and InterCode (Yang et al., 2023) connects an LLM to a Bash shell to solve file system task, similar to our use of an interactive Python console. Recent benchmarks (Liu et al., 2024; Wang et al., 2024b) will further catalyze this development. We deploy such interactive coding strategy to real-world humanoid robotics, and enrich it with incremental learning from natural interactions.

2.3 Orchestrating Robot Behavior with LLMs

Recently, many works extend the capabilities of LLMs by giving them access to external models, tools and APIs (Mialon et al., 2023; Parisi et al., 2022; Qin et al., 2023; Wang et al., 2023). Tool usage can also be combined with reasoning techniques such as CoT prompting (Wei et al., 2022) to significantly improve planning (Yao et al., 2023). 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. (2022a) initially proposed the idea to utilize world knowledge from LLM pretraining to map high-level tasks to executable mid-level action sequences. SayCan (Ahn et al., 2022) fuses LLM output probabilities with pretrained affordance functions to choose a feasible plan given a natural language command. Socratic Models (Zeng et al., 2023) 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) (Liang et al., 2023) and ProgPrompt (Singh et al., 2023) 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 (Huang et al., 2022b) feeds back execution results and observations into the LLM, but does not rely on code-writing, thus missing its combinatorial power. KnowNo (Ren et al., 2023) iteratively asks the LLM for a set of possible next steps, determines the LLM's confidence in each possibility using its output token distribution in a multiple-choice setup, and then uses conformal prediction to decide whether the system is sure how to proceed or should ask the user for help. AutoGPT+P (Birr et al., 2024) combines an LLM with a symbolic planner. Recent technical reports (Vemprala et al., 2023; Wake et al., 2023) provide guidance on utilizing ChatGPT (OpenAI, 2023a) for robot orchestration. While TidyBot (Wu et al., 2023) uses GPT-3 (Brown et al., 2020) 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.

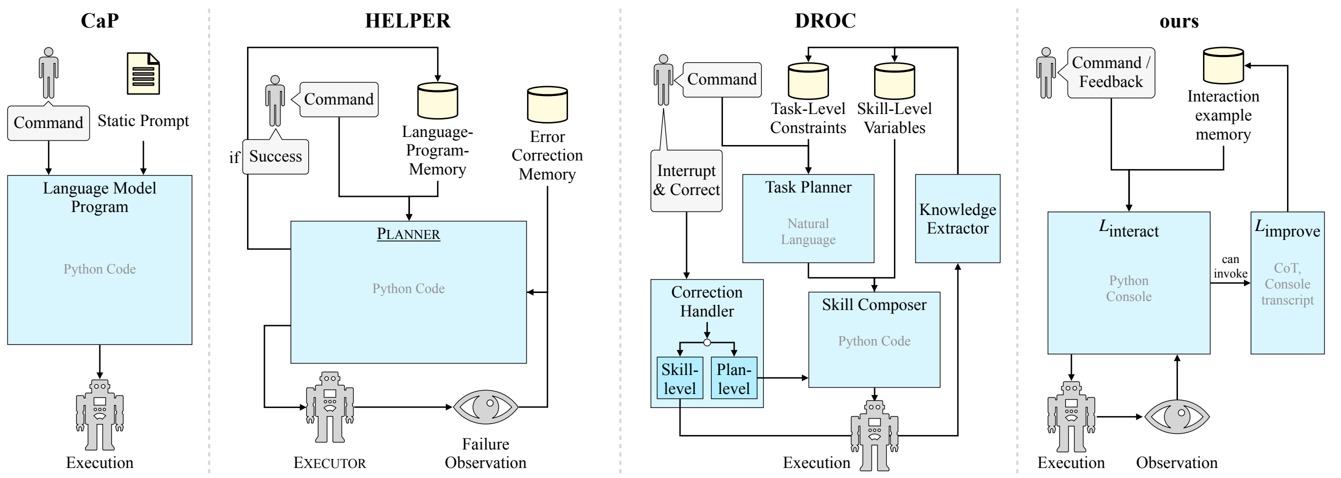

Figure 2. Comparison of Code as Policies (Liang et al., 2023), HELPER (Sarch et al., 2023), DROC (Zha et al., 2023) and our method, focusing on information flow from user input, observations, prompts, memories to LLM modules to robot execution, and how the methods learn from user interactions. Building on the interactive Python console prompting scheme, our method realizes incremental learning from natural interaction in a conceptually simple way.

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 (Ahn et al., 2022; Huang et al., 2022b), *(ii)* closing the interaction loop by enabling the LLM to reason about each perception and action outcome, in contrast to (Liang et al., 2023; Singh et al., 2023; Wake et al., 2023; Zeng et al., 2023; Ahn et al., 2022), *(iii)* allowing the LLM to decide 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 (Zeng et al., 2023; Huang et al., 2022b; Singh et al., 2023; Liang et al., 2023; Wu et al., 2023), and *(iv)* simplifying the task for the LLM by allowing it to generate one statement at a time, in contrast to (Liang et al., 2023; Singh et al., 2023; Vemprala et al., 2023).

2.4 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. (2022) 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 (Reimers and Gurevych, 2019) representing the questions. Ye et al. (2023) select not only the most similar, but also a diverse set of samples. Luo et al. (2023) show that this dynamic prompt construction is also applicable for instruction-fine-tuned language models (LMs) (Ouyang et al., 2022) and in combination with CoT prompting. Song et al. (2023) use top-k retrieval for instructing an LLM to plan robotic tasks. 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 Madaan et al. (2022a), 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 (Dalvi Mishra et al., 2022). Corrections are stored in memory and later retrieved based on their distance to a novel question.

In our work, we also propose to store corrective user feedback as interaction examples in the robot's memory. However, we go even further by *(i)* letting the LLM decide 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 (Vemprala et al., 2023) 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.

Closest to our approach are the concurrent works DROC (Zha et al., 2023) and HELPER (Sarch et al., 2023), shown in Fig. 2. Similar to our learning from interaction, DROC (Zha et al., 2023) distills knowledge from problematic interactions and retrieves it later when solving new tasks. While the goal and problem setting are similar, we differ by formulating the complete interaction in code, instead of separating task-level and skill-level into natural-language- and code-level interaction, respectively, and also generalizing incremental learning as code manipulation, instead of explicitly memorizing task-level natural language constraints and skill-level variable assignments separately. HELPER (Sarch et al., 2023) retrieves few-shot examples for the LLM's prompt from a language-program memory similar to our interaction examples memory, and learns personalized robot behavior by extending the memory. In contrast to our approach, they add examples only from successful episodes, and they have separate mechanisms for normal behavior and error correction. We focus on learning from feedback in erroneous or suboptimal episodes, and we treat initial and follow-up instructions uniformly using the proposed Python console prompting.

3 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.

3.1 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. 3. 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

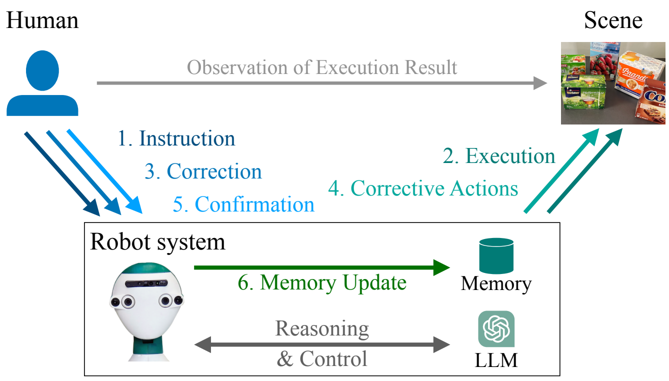

Figure 3. Incremental learning of robot behavior from interaction

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.

We formulate this problem as follows. Consider a robot with a set of functions $\mathcal{F} = \{F_1, \ldots, F_n\}$. 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 $(f_1, \ldots, f_m)$, where each invocation f_i consists of the invoked function F_i with its corresponding parameters. Executing these invocations yields a sequence of results $(r_1, \ldots, r_m)$. Overall, performing the task indicated by I_0 results in an *interaction history* $\mathcal{H}$ of the form

$$\mathcal{H} = ((f_1, r_1), \ldots, (f_m, r_m)) \leftarrow \mathrm{perform}\,(I_0, \mathcal{M}) \tag{1}$$

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 $\mathcal{H}_t = ((f_1, r_1), \ldots, (f_t, r_t))$ is available when generating the next invocation f_{t+1}, i. e., for $t \in \{0, \ldots, m-1\}$,

$$f_{t+1} \leftarrow \mathrm{generate}\,(I_0, \mathcal{H}_t, \mathcal{M})\,, \tag{2}$$

$$(r_{t+1}, W_{t+1}) \leftarrow \mathrm{execute}\,(f_{t+1}, W_t)\,, \tag{3}$$

$$\mathcal{H}_{t+1} \leftarrow \mathcal{H}_t \circ ((f_{t+1}, r_{t+1}))\,, \tag{4}$$

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, $\mathcal{H}_0 = ((f_{\text{wait}}, I_0))$, and we can drop I_0 as explicit parameter of $\mathrm{generate}$. 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

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

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. 4. A humanoid robot is interacting with a human and the scene. The robot is equipped with a multimodal memory system containing the following information about the current scene: First, semantic knowledge about objects, locations, agents and their relations in the world. Second, additional subsymbolic knowledge about the current scene. Third, executable skills (in our case implemented through scripted policies) as part of the robots procedural memory. An execution request sent to the procedural memory triggers physical robot actions. The set of available functions $\mathcal{F}$ contains methods to query knowledge from the semantic memory and to trigger actions from the procedural memory. Finally, as part of the robots episodic memory, $\mathcal{M}$ contains interaction

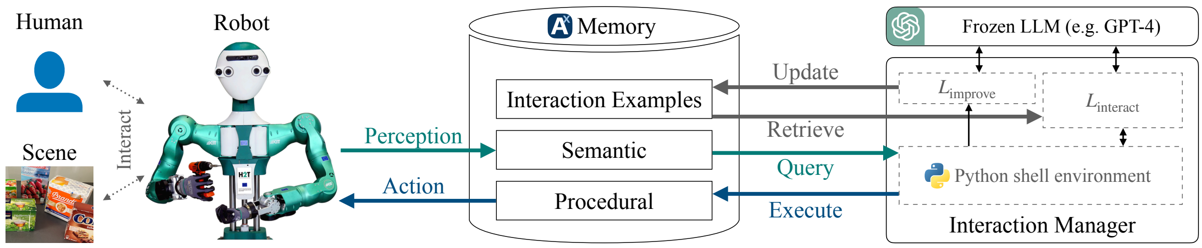

Figure 4. Conceptual view of our system. Here, the robot's memory system (Peller-Konrad et al., 2023) works as a mediator between the interaction manager and the robots low-level system components, such as controllers, sensors and drivers. The interaction LLM acts in a Python console environment. It can invoke functions to fetch the content of the current scene (as given by perception modules and stored in the memory) 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.

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. Utilizing E, we uniformly represent all $\mathcal{H} \in \mathcal{M}$ as well as $\mathcal{H}_t$ as a textual Python console transcript, i. e., a sequence of function invocations f_i represented as Python statement and return values r_i converted to text using Python's "`repr`" function. 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 $\mathcal{M}$.

3.2 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 3.3). In the following, a prompt is constructed (Section 3.4) 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. 5). Note that L_{interact} can listen to further user utterances by generating "`wait_for_trigger()`" again. Our proposed prompt-based incremental learning strategy (Section 3.5) is also invoked by L_{interact} itself when it calls F_{learn} = "`learn_from_interaction()`".

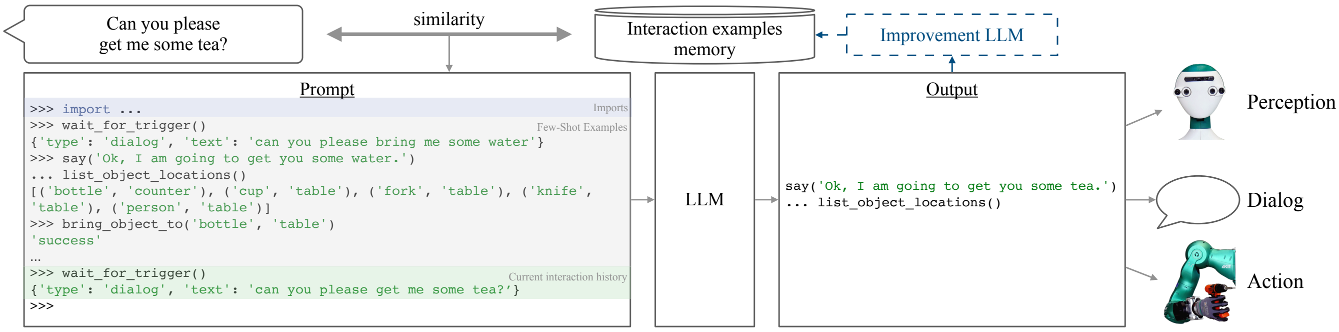

Figure 5. Overview of our method for incremental learning of robot behavior. We use an LLM (in our experiments, GPT-4 (OpenAI, 2023b)) to control robot perception and action given a prompt of few-shot examples (bottom, Section 3.3). Prompts are constructed dynamically based on the similarity to the current user request (top left, Section 3.4). 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 3.5).

3.3 LLM interacting with an Emulated Python Console

The left of Fig. 5 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 (Skreta et al., 2023). 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.

Analogously to Code as Policies (Liang et al., 2023), we dynamically generate non-existing functions the LLM tries to use. Specifically, when L_{interact} generates code that refers to an undefined function, we invoke another LLM L_{fgen} that is prompted to define the function, given the line of code that is using it as context. For L_{fgen}, we exactly follow the method of Liang et al. (2023), including recursive function generation. The generated function is then inserted into the emulated Python console *before* the statement that referred to the undefined function, and then that statement is executed. The purpose of inserting the function definition into the execution history is that it is thereby accessible to user feedback and can be improved upon by incremental learning.

3.4 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 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 3.3. $\mathcal{M}$ is initialized with hand-written prompts, but later extended dynamically as explained in Section 3.5. 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. 5.

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 $\mathcal{H} \in \mathcal{M}$. In our specific Python-console-based representation, this means that we search for function calls that trigger user interaction ("`ask`", "`wait_for_trigger`"), and extract their respective return values. Let I_t^i with $i = 1, \dots, N$ 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 $j = 1, \dots, M_{\mathcal{H}}$ all the $M_{\mathcal{H}}$ instructions found in each $\mathcal{H} \in \mathcal{M}$. We make use of a pretrained sentence embedding model (Reimers and Gurevych, 2019) to measure the semantic similarity $\text{sim}\,(a, b) = \text{E}\,(a) \cdot \text{E}\,(b)$ between two natural language sentences a, b by the dot product of their latent space embeddings $\text{E}\,(\cdot)$. First, we compute a latent representation of $\mathcal{H}_t$ as

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

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

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

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.

3.5 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 F_{learn} = "`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 F_{learn}.

Prompt: I had the following interaction with the robot:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'I want to have some juice'}
>>> list_object_locations(affordance='grasp')
# ...
>>> move_to(get_location_coordinates('kitchen-counter_0'))
'success'
>>> grasp('multivitamin-juice_0')
'success'
>>> move_to(get_location_coordinates('handover_to_human'))
'success'
>>> say("Here is the multivitamin juice.")
>>> handover_object_to_human('multivitamin-juice_0')
'success'
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'can you also bring me a cup?'}
>>> say('Ok, I will bring you a cup')
# ... [analogous to the above]
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'Thanks, but next time, '
 'can you do this both at once? You have two hands'}
>>> learn_from_interaction()
```

Prompt: What is the problem in this interaction? Answer with a single sentence.
LLM: The problem is that the robot is not utilizing both of its hands to bring the juice and the cup at the same time, resulting in two separate trips.
Prompt: How can the robot do better next time? Answer with a single explanation sentence, no code.
LLM: Next time, the robot should use both of its hands to grasp the juice and the cup at the same time, reducing the number of trips and increasing efficiency.
Prompt: Provide an improved version of the interaction transcript. [...]
LLM:

```
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'I want to have some juice'}
# ... [LLM lists objects & locations and moves to counter]
>>> grasp('multivitamin-juice_0', 'left')  # with left hand
'success'
>>> grasp('cup_large_0', 'right')  # with right hand
'success'
>>> move_to(get_location_coordinates('handover_to_human'))
'success'
>>> say("Here is the multivitamin juice and the cup.")
>>> handover_object_to_human('multivitamin-juice_0')
'success'
>>> handover_object_to_human('cup_large_0')
'success'
>>> wait_for_trigger()
{'type': 'dialog', 'text': 'Thanks, that was much quicker'}
```

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 **??**

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 F_{learn} implementation can be found in Listing 1. For robustness, there are three cases where we discard the generated $\mathcal{H}_t^*$: First, we ignore the call to F_{learn} if it does not follow immediately after a user utterance, since we only want to learn from explicit human feedback. Second, we abort the learning if the response to the first CoT request is that there is no problem. Third, if $\mathcal{H}_t^*$ is equal to the input interaction $\mathcal{H}_t$, we discard it.

4 SIMULATED EVALUATION

4.1 Experimental Setup

To quantitatively assess the performance of our method, we utilize the evaluation protocol from Code as Policies (Liang et al., 2023), involving a simulated tabletop environment with a UR5e arm and Robotiq 2F85 gripper manipulating a set of blocks and bowls of ten different colors. We use their seven seen and six unseen instructions (SI/UI), where each instruction is a task with placeholders that are filled with attributes (e.g. "pick up the <*block*> and place it on the <*corner*>"). The set of possible attribute values is also split into seen and unseen attributes (SA/UA). For more details, refer to Liang et al. (2023).

As our focus is on incremental learning from natural-language interaction, our methodology involves human supervision as follows: We first set up a randomly generated scene and pass the instruction to the evaluated system. The system generates some code that utilizes the same API as in Liang et al. (2023). Specifically, there are "perception" functions (utilizing the ground-truth simulation state) to query all object names and positions, and convert normalized to absolute coordinates, as well as one "action" function to move an object to another object or position. For details, see **??** or Liang et al. (2023). During code execution, the human observes the robot's actions by watching the simulation rendering. Each run can result in success (goal reached), failure (goal not reached), error (system threw unhandled exception), or timeout (e.g. system got stuck in a loop). The latter two lead to immediate termination of the experiment. In contrast, when the system yields control normally (after code execution for CaP and on F_{wait} for our method), the resulting world state is checked using scripted ground-truth evaluation functions, leading to either success or failure outcome. The human is then presented with this outcome and has the option to provide feedback or improvement instructions to the robot, which are again passed to the system. The success detection is performed every time the system yields control, and the sequence of states and user interactions is recorded. Note that we allow user feedback even when already in success state, as the execution might still have been suboptimal and the human may want to provide feedback to learn from for next time. Details and example interactions can be found in **??**.

Every task is repeated ten times using randomly generated scenes, and each run is performed in sequence, i.e., the interaction memory is not reset between runs in order to allow for incremental learning. To assess the results, we compute the following metrics from the execution traces:

- s is the turnout success rate, i.e. the percentage of runs that ended in success state (optionally after user interaction that clarifies the goal or helps the system)
- i is the initial success rate, i.e. the percentage of runs that yielded a successful state on the first system return, i.e. where no user interaction was required to reach success
- n counts the number of user interactions that were required until the success state was first reached. For runs that count into the initial success category, $n = 0$, while for non-successful runs, n is undefined. When aggregating n, we average only over the runs that ended successfully.

4.2 Baselines & Methods

CaP: We utilize the prompts provided by Liang et al. (2023). This is equivalent to our system without incremental learning and without the interactive console formatting. Specifically, we note that CaP has no way of feeding back coding errors to the system, i.e. it fails immediately if the generated code is syntactically invalid or throws an exception.

HELPER: We adapt the code and prompts provided by Sarch et al. (2023) to the simulated tabletop evaluation scenario & API. For few-shot example retrieval, we set $k = 16$ for a fair comparison. Specifically, we feed back execution errors to the *Self-Reflection & Correction* prompt, and user feedback is passed as a new command to the PLANNER. HELPER's few-shot memory is expanded with successful trials. Further details can be found in **??**.

Dynamic CaP: To make CaP a more competitive baseline, we add a simple form of learning and top-k retrieval and call this method *Dynamic CaP*. Similar to HELPER and our method, Dynamic Cap uses a memory of few-shot samples and stores code transcripts of successful episodes as new samples therein. On every request, we fill the prompt with the top-k similar examples retrieved from the memory. Further implementation details can be found in **??**.

ours: This is our full system with incremental learning and a value of $k = 16$ for few-shot sample retrieval. We split and translated the 16 samples from the CaP prompts into our interactive console syntax to initialize the memory of interaction examples. Furthermore, there are two very short samples that demonstrate when to call F_{learn}.

ours w/o learning: This is our system, but without incremental learning. $k = 16$ means that all samples are used, as the interaction examples memory is static.

ours w/o retrieval: This is our system with incremental learning but a very high value of $k = 64$ for few-shot sample retrieval, which effectively is a system that does not use retrieval. Note that the prompt construction is still dynamic as the order of the samples is determined by the similarity to the current request (cf. Section 3.4).

Furthermore, we compare the differently capable LLMs `gpt-3.5-turbo-0301` and `gpt-4-0613` of the OpenAI API (OpenAI, 2023a,b). For $L_{improve}$, we always use `gpt-4`. We note that the original CaP numbers (Liang et al., 2023) were reported with the `codex` model (Chen et al., 2021) that is no longer available. We reproduce their experiments with the newer models but did not perform further prompt tuning, therefore our success rates for CaP are lower than those reported in (Liang et al., 2023). Specifically, `gpt-3.5` sometimes generates natural language responses instead of code, which causes CaP to fail with a SyntaxError.

4.3 Results

Table 1 and 2 present the aggregated results of our experiments, while further details can be found in **??**. From these results, we draw the following main insights:

Interactive feedback helps to achieve success. For all methods, s is notably above i, which means that $L_{interact}$ effectively uses human feedback to improve its behavior. This effect is less stressed for CaP with `gpt-3.5`, as it often immediately fails with an error, thus not allowing for further interaction.

Incremental learning reduces necessity of corrective interactions. For many tasks, i is notably higher and n lower when comparing systems with learning to systems without learning, indicating that the feedback from earlier (failed) attempts is effectively utilized to improve following executions of the same task. This effect is also confirmed by **????** in the appendix. While for `gpt-4` on seen instructions, performance is already on a high level and corrections are rarely necessary, the numbers strongly support that incremental learning reduces interactions for unseen instructions, as well as for `gpt-3.5` on all instructions. Thus, our method for incremental learning is especially useful for “hard” tasks with respect to the predefined examples and general capabilities of the used model.

Incremental learning improves in-task success rate. Our systems with incremental learning also have higher s than those without learning. The reason is that our incremental learning method reflects on the erroneous behavior and generates a new sample for in-context learning that demonstrates the desired behavior. With such nearly identical demonstration, the generalization to new situations is much better, thus causing fewer errors that cannot be corrected through interaction.

Incremental learning generalizes to new tasks. Qualitatively, we observed several cases where a correction for one task is useful for another task as well. For instance, `gpt-3.5` initially interprets “the corner” as some position like $(0.1, 0.9)$. When instructing to “put it right into the corner without any margin”, the behavior of using full numbers, e.g. $(0, 1)$, transfers to subsequent different tasks that also involve corners. Quantitatively, this effect is entangled with the previous points in higher s and i, especially for the later unseen tasks. For a further investigation, see **??**.

Demonstration retrieval improves performance. For both LLMs, our system with retrieval outperforms the system that always uses all samples. This is especially true for `gpt-3.5`, as the system without retrieval accumulated to many interaction examples in its memory in the final experiments, thus leading to

		ours						HELPER		Dyn. CaP		CaP	
		full		w/o retrieval		w/o learning							
	Test	s	i	s	i	s	i	s	i	s	i	s	i
GPT-4	SA SI	100	97.5	97.5	90.0	98.8	90.0	97.5	87.5	88.8	86.2	85.0	71.2
	UA SI	100	92.5	98.8	95.0	98.8	92.5	100	93.8	97.5	93.8	96.2	81.2
	UA UI	93.3	85.0	91.7	81.7	91.7	78.3	91.7	81.7	63.3	46.7	53.3	35.0
GPT-3.5	SA SI	95.0	87.5	93.8	82.5	85.0	43.8	93.8	77.5	57.5	55.0	53.8	52.5
	UA SI	97.5	86.2	96.3	88.8	80.0	45.0	87.5	71.2	65.0	57.5	60.0	58.8
	UA UI	85.0	70.0	56.7	51.7	66.7	43.3	80.0	50.0	46.7	36.7	16.7	15.0

Table 1. Evaluation results on simulated tabletop tasks: success rate s and initial success rate i

		ours			HELPER	Dyn. CaP	CaP
	Test	**full**	w/o retrieval	w/o learning			
GPT-4	SA SI	0.04	0.12	0.37	0.21	0.06	0.26
	UA SI	0.14	0.12	0.1	0.1	0.07	0.35
	UA UI	0.16	0.18	0.55	0.22	0.62	0.74
GPT-3.5	SA SI	0.14	0.25	1.09	0.31	0.16	0.02
	UA SI	0.33	0.15	0.95	0.38	0.23	0.06
	UA UI	0.28	0.19	1.29	0.68	0.48	0.07

Table 2. Evaluation results on simulated tabletop tasks: average number of interactions until success n

immediate failure due to exceeding the LLMs token limit. While this is not the case for `gpt-4` with its much larger context length, the performance of the system with retrieval is still better. We hypothesize that this is due to too many irrelevant samples distracting the LLM.

Better LLMs lead to better performance. This can be clearly seen when comparing the numbers for `gpt-4` and `gpt-3.5`. Nonetheless, we emphasize that `gpt-3.5`'s performance as L_{interact} is still reasonably well, while it is faster and a factor of ten times cheaper. Specifically, the total cost to perform the experiments in Table 1 was \$ 245.6 for `gpt-4` vs. \$ 19.8 for `gpt-3.5` (which includes the use of `gpt-4` for L_{improve}). Our method of incremental learning can thus be seen as a knowledge distillation method, with `gpt-4` as the expensive teacher model L_{improve} generating task-specific new prompts for the cheaper `gpt-3.5` to improve its future behavior as L_{interact}.

Comparison with HELPER and Dynamic CaP. As a key difference to our method, HELPER learns from successful trials by storing them as an example, while our method only inspects erroneous experiences and then stores improved versions thereof. The experimental results show that this strategy is more effective, leading to higher s, i and lower n. Furthermore, HELPER cannot see its own previously-generated code when responding to errors or feedback, in contrast to our method, which utilizes the interactive Python console prompting for this purpose. Thus, HELPER cannot handle feedback such as “slightly more to the left” effectively.

Dynamic CaP improves performance over plain CaP, but cannot compete with HELPER or our method. This confirms that our method of interactive Python console prompting is more effective than producing all code to solve the task at once. Furthermore, we can observe that learning from successful trials helps with seen instructions by reinforcing correct behavior, but does not transfer to unseen instructions. Note that this observation also applies to HELPER, but mainly to i since HELPER can better respond to execution errors and user feedback than CaP. We conclude that our proposed method to learn from erroneous interactions is more effective than reinforcing successful behavior only.

Further results. ?? presents two additional experiments: First, we investigate the effect of k by setting $k = 4$ (instead of 16), showing that lower k comes with a higher n and lower i, as potentially relevant demonstrations might not be retrieved, thus requiring another user interaction. Second, we change the behavior of F_{learn} to just save the current interaction in $\mathcal{M}$, skipping L_{improve}. This hurts performance, as the erroneous behavior from previous trials is often repeated, despite the prompt containing improvement instructions from earlier interactions.

5 REAL-WORLD DEMONSTRATION

To demonstrate the utility of our proposed prompt-based incremental learning technique, we perform experiments on the real-world humanoid robot ARMAR-6 (Asfour et al., 2018). 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 – also try similar commands that initially (i. e., before learning) led to similar mistakes. Details on the implementation of these experiments, especially on the API exposed to the LLM, can be found in **??**. The system is connected to a memory-centric cognitive robot architecture where the memory mediates between high-level components and low-level abilities (see Fig. 4). Specifically, the API provided to the LLM allows querying the robot's memory with functions to list all objects and location names (opt. with a given affordance), query subsymbolic coordinates of objects or locations, or retrieve state information about specific objects. The robot's memory is filled beforehand by the robot's perception and cognition components. In our experiments, we use a mixture of predefined prior knowledge (e.g., about static objects in the scene) and online perception (e.g. object pose-detection, self-localization). Further, the API allows to invoke registered skills, behaviors and movements of the robot, such as grasping, navigation, object placement, or handing objects to a human. However, we do not focus on scenarios where the involved skills themselves fail, but rather on high-level semantic problems. Please refer to **??** for further details.

We present three scenarios: *Improving Plans* to demonstrate complex improvement of suboptimal or unintended performance, *Learning User Preferences* to show how to adapt to non-generic task constraints, and *Adapting Low-Level Parameters* to demonstrate that our system can learn from vague user instructions. Demonstration videos can be found at `https://lbaermann.github.io/interactive-incremental-robot-behavior-learning/`.

5.1 Improving Plans

In this scenario, we tell the robot that we want juice. The prior knowledge contains some similar interaction examples, picking up a single object and handing it over to the human. Thus, the task of bringing the juice is executed successfully. However, since the user needs a cup to drink, we further instruct the robot "can you also bring me a cup?", which causes the robot to additionally hand over a cup. Afterwards, we ask the robot to improve this behavior using "Thanks, but next time, can you do this both at once? You have two hands". L_{improve} generates an improved interaction example as shown on the right of Listing 1 (simplified, cf. **??**).

Afterwards, when giving the same initial command again, the robot uses bimanual behavior to hand over both juice and cup. Furthermore, the learned bimanuality generalizes to "can you bring something to drink to the table?", which does not use handover, but places both objects on the table. Unfortunately, a further test with "can I have some milk, please?" shows the unimanual behavior again, so we again have to ask for a cup and trigger incremental learning. In the next session, we ask "hey, can you serve some drink?", which correctly generalizes the behavior to use both hands to pick up a different drink and cup, but misinterprets “serve” as doing a handover instead of putting it on the table. However, we can successfully trigger learning again by teaching “when I say serve, I mean that you should put it on the table”, and subsequent requests do behave as intended.

We conclude that our interactive, incremental learning system can flexibly generate complex behavior from concise improvement instructions. However, it is still challenging to robustly generalize from a single instruction to all cases a human might have intended, as shown by the milk example, where a second correction was necessary for successful generalization. Improving this generalization capability should be a focus of future work.

5.2 Learning User Preferences

As shown in Fig. 1, in this scenario we ask the robot to assist with 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 (`gpt-4` did, in contrast to `gpt-3.5`, proactively ask whether it should also bring the ladder). The robot then successfully places the ladder in front of the fridge. Eventually, we instruct the robot to always bring the ladder when working on high surfaces. The generated improved interaction example correctly brings the ladder after the sponge, without further request (details in **??**). 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. The behavior also transfers to different tasks than cleaning, e.g. the robot brings the cereals and the ladder on “can you get me the cereals, I want to put it in the topmost shelf”, while it does not bring the ladder when tasked with “I want to put the cereals into the shelf”.

In summary, this example demonstrates that our method can be used to learn task constraints or preferences that a user specifies, and this knowledge can be generalized to similar situations.

5.3 Adapting Low-Level Parameters

In this scenario, we ask the robot to bring some object from the table to the workbench (details in **??**). Subsequently, we say “remember that the route from the table to the bench is safe, you can go faster”. F_{learn} correctly generates a sample that adapts the numeric speed factor of the `move_to` function on that route. However, if we test the same task afterwards, L_{interact} still uses the default speed. Annoyed by that, we shout “you forgot that I told you to go faster from the table to the workbench. When moving on that route, you should go faster!”, triggering another learning process, generating another correct sample, including an explicit comment:

```
…

>>> grasp('sponge_0')
'success'
>>> # The user earlier asked me to move faster from the
    # table to the workbench, so let's do that
... move_to(get_location_coordinates('workbench_0'),
            speed_factor=2.0)
'success'
>>> place_object('sponge_0', 'workbench_0')

…
```

Proceeding requests now behave correctly and increase the speed from the table to the workbench. However, an adversarial test shows that L_{interact} does now dangerously use increased speed from another location to the workbench, too, while routes to different places still correctly use the default speed.

To conclude, our system can successfully learn to adapt low-level API parameters as requested by a user, but ensuring the LLM applies learned knowledge in the intended context only is not fully solved yet.

6 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 (Vahrenkamp et al., 2015) as well as on the humanoid robot ARMAR-6 (Asfour et al., 2018). The usefulness of our approach is evaluated both quantitatively on the tasks from Code as Policies (Ahn et al., 2022) and qualitatively on a humanoid robot 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 solved with more advanced models in the future, as we did observe this issue much more often with GPT-3.5 than with GPT-4. Investigating the effect and performance of example retrieval in dynamic prompt construction might also contribute to improving robustness. 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, this is challenging to accomplish if the user is not familiar with robotics or programming languages. One possible approach would be to verbalize the improved interaction example using an LLM, present it to the user, and ask for confirmation. Similarly, the improved code could first be executed in a simulation environment to check its validity before saving it in the memory of interaction examples. Both approaches have some open challenges, such as ensuring correctness of the verbalization or accuracy of the simulation, as there will be a large sim-to-real gap for the type of behaviors considered in our paper. To rigorously evaluate our incremental learning method in the real world, future work may want to incorporate a user study with non-technical participants. Further work should also focus on abstraction of similar and forgetting of irrelevant learned behavior. While our system is limited by the APIs exposed to the LLM, it could be combined with complementary approaches (Parakh et al., 2023) to support learning of new low-level skills, which would then be exposed through new functions added to the API. Furthermore, designing an API that enables robust yet flexible interactions is a challenge that should be considered in future work. In particular, providing the LLM access to subsymbolic parameters (such as positions to navigate to) enables fine-grained user corrections ("move a little more to the left"), but can significantly harden the task for the LLM and entails many more failure cases. 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 (Huang et al., 2023). Finally, our system inherits biases and other flaws from its LLM (Bender et al., 2021), 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.

CONFLICT OF INTEREST STATEMENT

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

AUTHOR CONTRIBUTIONS

LB developed the methods and their implementation and performed the evaluation experiments. LB, RK and FP implemented and performed the real-world experiments. The entire work was conceptualized by LB, TA and AW and supervised by TA and AW. JN made important suggestions for the experimental methodology and reviewed the manuscript. The initial draft of the manuscript was written by LB and revised jointly by LB, RK, FP and TA. All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

FUNDING

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.

REFERENCES

Ahmed, A., Azab, S., and Abdelhamid, Y. (2023). Source-code generation using deep learning: A survey. In *Progr. Art. Intel.* (Springer Nature Switzerland), vol. 14116, 467–482. doi:10.1007/978-3-031-49011-8_37

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., et al. (2022). Do as i can, not as i say: Grounding language in robotic affordances. In *Annu. Conf. Rob. Learn.*

Asfour, T., Kaul, L., Wächter, M., Ottenhaus, S., Weiner, P., Rader, S., et al. (2018). ARMAR-6: A Collaborative Humanoid Robot for Industrial Environments. In *IEEE-RAS Int. Conf. Humanoid Robots*. 447–454

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

Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In *Conf. Fairness, Accountability, Transparency*. 610–623

Birr, T., Pohl, C., Younes, A., and Asfour, T. (2024). Autogpt+p: Affordance-based task planning with large language models. *arXiv:2402.10778*

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., et al. (2020). Language models are few-shot learners. In *Int. Conf. Neural Inf. Process. Syst.* vol. 33, 1877–1901

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. *arXiv:2107.03374*

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

Dehaerne, E., Dey, B., Halder, S., De Gendt, S., and Meert, W. (2022). Code generation using machine learning: A systematic review. *IEEE Access* 10, 82434–82455. doi:10.1109/ACCESS.2022.3196347

Fasola, J. and Matarić, M. J. (2013). 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.* 143–150

Forbes, M., Rao, R., Zettlemoyer, L., and Cakmak, M. (2015). Robot Programming by Demonstration with Situated Spatial Language Understanding. In *IEEE Int. Conf. Robot. Automat.* 2014–2020

Google (2023). Code-chat (Google VertexAI). *Online.* `https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/code-chat`

Guadarrama, S., Riano, L., Golland, D., Göhring, D., Jia, Y., Klein, D., et al. (2013). Grounding Spatial Relations for Human-Robot Interaction. In *IEEE/RSJ Int. Conf. Intel. Rob. Syst.* 1640–1647

Haluptzok, P., Bowers, M., and Kalai, A. T. (2023). Language models can teach themselves to program better. In *Int. Conf. Learn. Repr.*

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. (2022a). Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *Int. Conf. Mach. Learn.* vol. 162, 9118–9147

Huang, W., Xia, F., Shah, D., Driess, D., Zeng, A., Lu, Y., et al. (2023). Grounded decoding: Guiding text generation with grounded models for robot control. *arXiv:2303.00855*

Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., et al. (2022b). Inner monologue: Embodied reasoning through planning with language models. In *Annu. Conf. Rob. Learn.*

Kartmann, R. and Asfour, T. (2023). Interactive and Incremental Learning of Spatial Object Relations from Human Demonstrations. *Frontiers in Robotics and AI* 10

Krüger, N., Geib, C., Piater, J., Petrick, R., Steedman, M., Wörgötter, F., et al. (2011). Object-Action Complexes: Grounded Abstractions of Sensorimotor Processes. *Rob. Auton. Sys.* 59, 740–757

Lahiri, S. K., Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Musuvathi, M., et al. (2023). Interactive code generation via test-driven user-intent formalization. *arXiv:2208.05950*

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. (2022). CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Int. Conf. Neural Inf. Process. Syst.* vol. 35, 21314–21328

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., et al. (2023). Code As Policies: Language Model Programs for Embodied Control. In *IEEE Int. Conf. Robot. Automat.* 9493–9500

Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., and Chen, W. (2022). What makes good in-context examples for GPT-3? In *Deep Learning Inside Out: Worksh. Knowl. Extr. Integr. Deep Learn. Arch.* 100–114. doi:10.18653/v1/2022.deelio-1.10

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., et al. (2024). AgentBench: Evaluating LLMs as agents. In *Int. Conf. Learn. Repr.*

Luo, M., Xu, X., Dai, Z., Pasupat, P., Kazemi, M., Baral, C., et al. (2023). Dr.ICL: Demonstration-retrieved in-context learning. *arXiv:2305.14128*

Madaan, A., Tandon, N., Clark, P., and Yang, Y. (2022a). Memory-assisted prompt editing to improve GPT-3 after deployment. In *Conf. Emp. Meth. Nat. Lang. Proc.* 2833–2861

Madaan, A., Zhou, S., Alon, U., Yang, Y., and Neubig, G. (2022b). Language models of code are few-shot commonsense learners. In *Conf. Emp. Meth. Nat. Lang. Proc.* 1384–1403. doi:10.18653/v1/2022.emnlp-main.90

Mialon, G., Dessi, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., et al. (2023). Augmented language models: a survey. *Trans. Mach. Learn. Research*

Misra, D. K., Sung, J., Lee, K., and Saxena, A. (2016). Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions. *Int. J. Rob. Research* 35, 281–300

Mohan, S. and Laird, J. (2014). Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. *AAAI* 28

Nicolescu, M., Arnold, N., Blankenburg, J., Feil-Seifer, D., Banisetty, S. B., Nicolescu, M., et al. (2019). Learning of Complex-Structured Tasks from Verbal Instruction. In *IEEE-RAS Int. Conf. Humanoid Robots*. 770–777

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., et al. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. In *Int. Conf. Learn. Repr.*

OpenAI (2023a). ChatGPT. *Online.* `https://openai.com/blog/chatgpt/`

OpenAI (2023b). GPT-4 Technical Report. *arXiv:2303.08774*

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., et al. (2022). Training language models to follow instructions with human feedback. In *Int. Conf. Neural Inf. Process. Syst.* vol. 35, 27730–27744

Parakh, M., Fong, A., Simeonov, A., Gupta, A., Chen, T., and Agrawal, P. (2023). Lifelong robot learning with human assisted language planners. In *Work. Learn. Eff. Abstr. Plan., CoRL*

Parisi, A., Zhao, Y., and Fiedel, N. (2022). TALM: Tool augmented language models. *arXiv:2205.12255*

Peller-Konrad, F., Kartmann, R., Dreher, C. R. G., Meixner, A., Reister, F., Grotz, M., et al. (2023). A memory system of a robot cognitive architecture and its implementation in ArmarX. *Rob. Auton. Sys.* 164, 20

Pramanick, P., Barua, H. B., and Sarkar, C. (2020). DeComplex: Task planning from complex natural instructions by a collocating robot. In *IEEE/RSJ Int. Conf. Intel. Rob. Syst.* 8

Qin, Y., Hu, S., Lin, Y., Chen, W., Ding, N., Cui, G., et al. (2023). Tool learning with foundation models. *arXiv:2304.08354*

Reimers, N. and Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Conf. Emp. Meth. Nat. Lang. Proc.* 3982–3992. doi:10.18653/v1/D19-1410

Ren, A. Z., Dixit, A., Bodrova, A., Singh, S., Tu, S., Brown, N., et al. (2023). Robots that ask for help: Uncertainty alignment for large language model planners. In *Annu. Conf. Rob. Learn.*

Sarch, G., Wu, Y., Tarr, M., and Fragkiadaki, K. (2023). Open-ended instructable embodied agents with memory-augmented large language models. In *Conf. Emp. Meth. Nat. Lang. Proc.* 3468–3500

Shridhar, M., Mittal, D., and Hsu, D. (2020). INGRESS: Interactive visual grounding of referring expressions. *Int. J. Rob. Research* 39, 217–232

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., et al. (2023). ProgPrompt: Generating situated robot task plans using large language models. In *IEEE Int. Conf. Robot. Automat.* 11523–11530

Skreta, M., Yoshikawa, N., Arellano-Rubach, S., Ji, Z., Kristensen, L. B., Darvish, K., et al. (2023). Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv:2303.14100*

Song, C. H., Wu, J., Washington, C., Sadler, B. M., Chao, W.-L., and Su, Y. (2023). LLM-planner: Few-shot grounded planning for embodied agents with large language models. In *Int. Conf. Comp. Vis.* 2998–3009

Tellex, S., Gopalan, N., Kress-Gazit, H., and Matuszek, C. (2020). Robots That Use Language: A Survey. *Annu. Rev. Control Rob. Auton. Sys.* 3, 25–55

Tellex, S., Kollar, T., Dickerson, S., Walter, M. R., Banerjee, A. G., Teller, S., et al. (2011). Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation. In *AAAI*. vol. 25 of *1*, 1507–1514

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., et al. (2023). LLaMA: Open and efficient foundation language models. *arXiv:2302.13971*

Vahrenkamp, N., Wächter, M., Kröhnert, M., Welke, K., and Asfour, T. (2015). The robot software framework ArmarX. *it - Information Technology* 57

Vemprala, S., Bonatti, R., Bucker, A., and Kapoor, A. (2023). ChatGPT for robotics: Design principles and model abilities. *Online.* `https://www.microsoft.com/en-us/research/publication/chatgpt-for-robotics-design-principles-and-model-abilities/`

Wake, N., Kanehira, A., Sasabuchi, K., Takamatsu, J., and Ikeuchi, K. (2023). ChatGPT empowered long-step robot control in various environments: A case application. *Online.* `https://www.microsoft.com/en-us/research/publication/chatgpt-empowered-long-step-robot-control-in-various-environments-a-case-application/`

Walter, M., Hemachandra, S., Homberg, B., Tellex, S., and Teller, S. (2013). Learning semantic maps from natural language descriptions. In *Rob.: Science and Systems*

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., et al. (2024a). Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Research*

Wang, J. and Chen, Y. (2023). A review on code generation with LLMs: Application and evaluation. In *Int. Conf. Med. Art. Intel.* 284–289. doi:10.1109/MedAI59581.2023.00044

Wang, X., Wang, Z., Liu, J., Chen, Y., Yuan, L., Peng, H., et al. (2024b). MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *Int. Conf. Learn. Repr.*

Wang, Z., Zhang, G., Yang, K., Shi, N., Zhou, W., Hao, S., et al. (2023). Interactive natural language processing. *arXiv:2305.13246*

Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Int. Conf. Neural Inf. Process. Syst.*

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

Wu, J., Antonova, R., Kan, A., Lepert, M., Zeng, A., Song, S., et al. (2023). TidyBot: Personalized robot assistance with large language models. *arXiv:2305.05658*

Yang, J., Prabhakar, A., Narasimhan, K., and Yao, S. (2023). InterCode: Standardizing and benchmarking interactive coding with execution feedback. In *Int. Conf. Neural Inf. Process. Syst.* vol. 36, 23826–23854

Yang, K., Liu, J., Wu, J., Yang, C., Fung, Y., Li, S., et al. (2024). If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. In *ICLR 2024 Workshop on LLM Agents*

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., et al. (2023). ReAct: Synergizing reasoning and acting in language models. In *Int. Conf. Learn. Repr.*

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

Zeng, A., Attarian, M., ichter, b., Choromanski, K. M., Wong, A., Welker, S., et al. (2023). Socratic models: Composing zero-shot multimodal reasoning with language. In *Int. Conf. Learn. Repr.*

Zha, L., Cui, Y., Lin, L.-H., Kwon, M., Arenas, M. G., Zeng, A., et al. (2023). Distilling and retrieving generalizable knowledge for robot manipulation via language corrections. In *Work. Lang. Robot Learn., CoRL*

Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., et al. (2024). A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv:2311.10372*