SCHOOL OF COMPUTATION,
INFORMATICS AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition and Intelligence

Foundation Models for Robotics

Max Fest

SCHOOL OF COMPUTATION,
INFORMATICS AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition and Intelligence

Foundation Models for Robotics

Foundation Models für die Robotik

Author:	Max Fest
Supervisor:	Prof. Alois Knoll
Advisor:	Dr. Zhenshan Bing
Submission Date:	March 31, 2024

I confirm that this master's thesis in robotics, cognition and intelligence is my own work and I have documented all sources and material used.

Munich, March 31, 2024 Max Fest

Acknowledgments

Thanks Zhenshan, for being an important part of my academic journey.

I want to thank my family for their ongoing and unwavering support, and Vio, for putting up with me through all of it. I love all of you.

Abstract

Foundation Models encapsulate broad world knowledge that was previously difficult to integrate into robot learning, enabling a new wave of approaches aimed at grounding this knowledge in embodied action. Many of these approaches rely on a Large Language Model to translate human intentions expressed in natural language into robot policies – building on a fixed set of perception and control primitives for the agent to call. We propose to extend these approaches by enabling humans to *teach the agent new skills*, by iteratively refining behaviours with corrective feedback. Humans naturally understand the physical world and many of the domains in which we expect robots to operate, and natural language provides a powerful interface for us to both communicate this understanding, and to instil human preferences. We argue that prior works have not sufficiently placed humans at the centre of robot learning, and demonstrate that skills provide an intuitive mechanism for doing so. This opens the door to fast, language-based adaptation to new environments.

Kurzfassung

Foundation Models bündeln umfangreiches Weltwissen, das zuvor nur schwer in das Roboterlernen integrierbar war, und ermöglichen eine neue Welle von Ansätzen, die darauf abzielen, dieses Wissen in verkörperte Handlung zu überführen. Viele dieser Ansätze stützen sich auf Large Language Models, um menschliche Absichten, die in natürlicher Sprache ausgedrückt werden, in Roboterstrategien zu übersetzen – basierend auf einem festen Satz von Wahrnehmungs- und Steuerungsgrundfunktionen, auf die der Agent zugreifen kann. Wir schlagen vor, diese Ansätze zu erweitern, indem Menschen dem Agenten neue Fähigkeiten beibringen, durch iterative Verfeinerung von Verhaltensweisen mittels korrigierendem Feedback. Menschen verstehen die physische Welt sowie viele der Domänen, in denen Roboter eingesetzt werden sollen, auf natürliche Weise, und natürliche Sprache stellt eine leistungsfähige Schnittstelle dar, um dieses Verständnis zu vermitteln und menschliche Präferenzen einzuprägen. Wir argumentieren, dass bisherige Arbeiten den Menschen nicht ausreichend ins Zentrum des Roboterlernens gestellt haben, und zeigen, dass Fähigkeiten einen intuitiven Mechanismus bieten, um dies zu erreichen. Dadurch wird eine schnelle, sprachbasierte Anpassung an neue Umgebungen möglich.

Contents

1. Introduction

The fundamental challenge of robotics is to align robot behaviour with human intentions and expectations. We want robots that do what we want, how we want it, across a large variety of tasks, environments, and robot embodiments, and we want them to do it reliably.

This is an open challenge. While we have successfully deployed robots in very controlled environments, like the ones found in manufacturing, warehouse logistics, or even agriculture, we are still a long way from the popular dream of a "household" robot, that readily handles a variety of chores like folding the laundry, or emptying the dishwasher.

With the rise of deep learning in the early 2010's, data-driven approaches took on a dominant role in robotics research. Deep Reinforcement Learning emerged as a technique that promised to learn skills purely from trial-and-error interaction with a simulated environment. However, with a few notable exceptions, successes in simulation have largely failed to materialise in the real world [83].

A parallel class of methods have attempted to make it possible for humans to "teach" robots by providing demonstrations of a desired behaviour. This limits the behaviours the robot can learn to ones that humans can effectively demonstrate, which is not as constraining as one might think, since there are many "mundane" tasks that we would like robots simply to take over from humans. We may recall Moravec's paradox in observing that these tasks continue to elude robots.

Until recently, we expected robots to operate in a human world, with no understanding of this world, and limited ways for us to imbue them with the necessary understanding. This may have changed with the advent of Foundation Models, large models pretrained on internet-scale data. Not only do they contain some of the basic world understanding previous data-driven methods lacked completely, but they enable a natural-language based interaction with computers. We can now talk to our robots.

A large and rapidly growing body of research is exploring all the ways that these Foundation Models may be used within the context of robotics. In this thesis, we focus on a subset of this research that enables non-expert humans to interact with robots using natural language, by using Large Language Models (like GPT-4 or LLaMa) to generate python code that controls the robot. Humans are almost universally capable of evaluating robotic behaviour, particularly in the many simple task domains in which

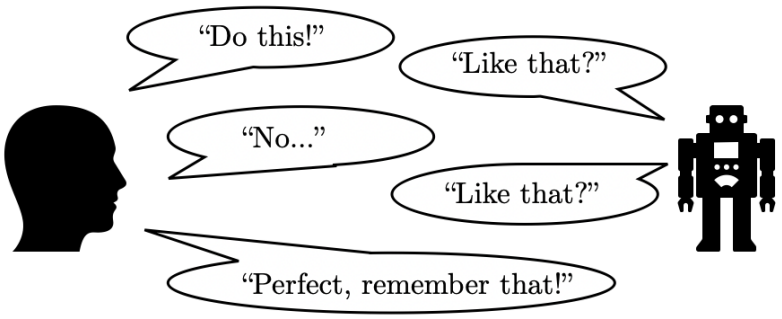


Figure 1.1.: Foundation Models enable a natural language interaction with robots.

we want to deploy them.

While prior works have demonstrated that we can use Large Language Models to elicit [47] [103] and correct [48] [109] [4] robot behaviours from natural language inputs, relatively few have imbued them with the ability to adapt their behaviour in a lasting way in response. One solution to this relies on Retrieval-Augmented Generation to provide agents with a *memory* [89] [111] [109] [80]. Building on these works, we propose a framework for allowing users to *interactively teach robots new skills*, based on natural language interaction.

We argue that effectively aligning robot behaviours with human intentions necessarily requires humans in the loop – and that one of the key advantages of applying Foundation Models to robotics is that they shorten this feedback loop. The method outlined in this thesis demonstrates one approach for leveraging this potential. Looking ahead, we hope this could lead to intuitive end-user programming and the crowdsourced generation of human-verified robotic behaviours.

2. Preliminaries

In this chapter, we introduce the notation we will lean on in the remainder of this thesis, as well as framing robot learning as an alignment i.e. communication problem. We begin by introducing the reward-free Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, \mathcal{P})$:

$\mathcal{S}$ – *State space*: the full representation of agent and environment state, i.e. all the information the agent has at its disposal to inform its decisions, including sensors, cameras, proprioception, etc...

$\mathcal{A}$ – *Action space*: the base set of actions the agent can take, which may involve low-level joint manipulations or more temporally abstracted behaviors.

$\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ – *Transition function*: probabilistically maps an action in a given state to the subsequent state, under the current environment dynamics. These dynamics are typically unknown, though we assume we can sample from them via interaction with the environment.

In practice, the state s is always an imperfect and incomplete representation of the *true* state space, an assumption modelled explicitly in the Partially Observable MDP [78], but omitted here for notational clarity.

At every time-step t, given a state s_t, the agent chooses an action a_t, bringing it to the state $s_{t+1} \sim \mathcal{P}(s_t, a_t)$ (see Figure 2.1). We may term a sequence of such interactions

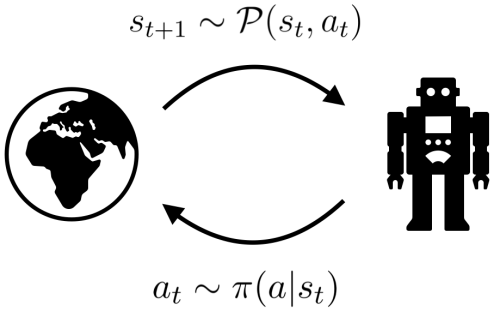


Figure 2.1.: The agent-environment interaction loop

$\tau = (s_0, s_1, ..., s_T) \in \mathcal{S}^T$ a *trajectory*.

We may view then define a *behaviour* $b \in \mathcal{S}^T$ we intend to elicit as a desirable or semantically meaningful trajectory, or an equivalence class $[b]$ of such trajectories, and the set of all behaviours we want to elicit as $\mathcal{B} \subsetneq \mathcal{S}^T$. For example, in a quadruped robot, walking, running, or hopping would each qualify as distinct behaviours. In a robotic arm, a behaviour might be grasping an avocado without squashing it, or flipping a pancake.

The goal then is to learn a *policy* $\pi : \mathcal{S} \times \mathcal{A}$, that chooses actions to achieve the desired behaviours. Correspondingly, we may define *skills* as the policies that elicit these behaviours, and skill-learning as the goal. The skill reliably enacts the behaviours.

Definition 1. A *skill* $z \in \mathcal{Z}$ is a policy $\pi_z(a|s)$ which induces a specific behaviour b.

We refer to $\mathcal{Z}$ as the *skill space*. The skill space is an extension of the action space $\mathcal{A}$ to enable temporally extended behaviours. Importantly, a skill $z \in \mathcal{Z}$ may build on other skills. Further, we define *tasks*:

Definition 2. A *task* is specified by a set of initial states $\mathcal{I} \subset \mathcal{S}$ and a task-success function $\beta : \mathcal{S}^T \to \{False, True\}$, for some time horizon T.

We use tasks both to learn and to test behaviours. The set of initial states $\mathcal{I}$ includes all variations in environment setup for a specific task. We define β to determine task success based on the entire trajectory to highlight that we may also be interested in *how* the agent achieves a final state. In practice, task-success has often been determined purely on the achievement of a final state (see Section 2.2.2). Our definitions of skills and tasks are in close analogy with options in hierarchical reinforcement learning (see Section 2.2.2).

The distinction between tasks and skills is a subtle one, and not typically made explicitly. We make this distinction to emphasise that we are interested in teaching robust, reusable and repeatable behaviours (skills), while a task is tied to a specific environment setup and task description. The agent may *use* skills to solve a task.

Appropriately choosing tasks to elicit and test the desired behaviours is in itself a challenge, requiring significant human effort. Moreover, task success is only a proxy for successfully learning a skill. In this thesis, we focus particularly on the interface provided to humans in aligning robot behaviours.

2.1. The Human-Agent-Environment loop

Aligning robot behaviours with human expectations induces another feedback loop, visualised in Figure 2.2. What this feedback loop looks like in reality depends heavily on

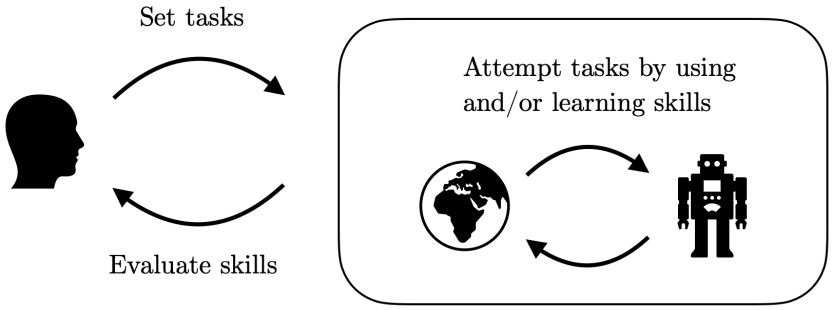


Figure 2.2.: The human-agent-environment interaction loop

the exact problem statement, i.e. the environment $(\mathcal{S}, \mathcal{A}, \mathcal{P})$, and the set of behaviours $\mathcal{B}$ we want to learn.

For example, if the environment dynamics $\mathcal{P}$ are near deterministic and known, and the environment is fully observable (i.e. $\mathcal{S}$ is a complete representation of the environment), the robot is sufficiently simple ($dim(\mathcal{A})$ is small) and we only want to learn a single repetitive behaviour $|\mathcal{B}| = 1$, opting to hard-code the behaviour is a reasonable choice, which ensures full interpretability and modifiability. This ensures a tight feedback loop between human and agent. We expect the agent to solve a single task.

On the other hand, if the environment dynamics are unknown and stochastic, the robot needs to map complex percepts to complex actuations, and we expect it to learn a wide variety of behaviours, learning-based approaches become more appealing. However, the typical training times associated with learning-based approaches naturally extend the time between a human setting a task and evaluating its results, with some experiments requiring days of training on expensive hardware.

A fundamental consideration in this feedback loop is the interface provided to humans in this process. What modes of communication exist for the human to adapt the robots behaviour?

For example, in Reinforcement Learning (Section 2.2), humans communicate a task by setting a *reward function*, following intuitions from behaviourist psychology. Good behaviours are rewarded, bad behaviours are punished. However, the actual learning process typically also involves extensive *environment shaping* [64], setting up simulators, tasks, and/or modifying the state and action spaces. Learning more complex and more diverse behaviours may take a long time, creating slow human-agent-environment feedback loops. Moreover, while we explicitly model the tasks, we typically expect the agents to decipher and learn the necessary skills on its own.

Under the umbrella of *Learning from demonstrations* (Section 2.4) fall all the methods in which humans are enabled to "show" robots what they want from them, by providing demonstrations, or providing interfaces to directly control them (Sections 2.4.1 and 2.4.2). While this enables robots to learn new behaviours more quickly, it requires further "communication channels" [79] to allow robots to generalise these behaviours, i.e. for humans to communicate to them what they are actually doing.

Natural Language is perhaps the most powerful and intuitive such communication channel, as observed in a number of recent works [79] [56] [8] [51].

Foundation Models have enabled a fundamentally new interaction with robots. Prior works [47] [103] have demonstrated that we can issue a natural language instruction (i.e. description of a task), and get a behavioural response within seconds, which we may then respond back to, etc... This not only creates extremely tight human-agent-environment feedback loops, but can also take advantage of the inherent structure of natural language to organise the behaviours that are being created.

2.2. Reinforcement learning

Reinforcement Learning (RL) [81] has emerged as one of the key methods for learning individual robotic skills [32]. Central to reinforcement learning is the *reward function* $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R} : \mathcal{R}(s, a)$. The agent then maximises the cumulative reward $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, traditionally given some discount factor γ to model the uncertainty about achieving future rewards.

This brings us to the usual MDP formulation: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. By effectively specifying this reward function, the agent is supposed to learn robust behaviours directly through trial-and-error interaction with the environment. This is appealing, because it allows the engineer to specify a high-level goal and hope that the agent picks up the necessary low-level skills in trying to achieve this goal.

We assume that a single reward function r maps to a single behaviour b, though it is possible to formulate complicated composite reward functions that cover multiple (see Figure 2.3). While the engineer may hope that constituent lower-level skills are learned, these are not explicitly modelled in this setup.

2.2.1. Reward shaping

The reward function becomes the central bottleneck to informing the robot of the behaviour we intend to elicit. This is a challenging problem, and coming up with effective rewards is a research discipline in itself, typically referred to as *reward shaping* or *reward engineering*. Effective reward design typically involves a tight feedback loop through the engineer [18], in which the engineer designs a function, observes the

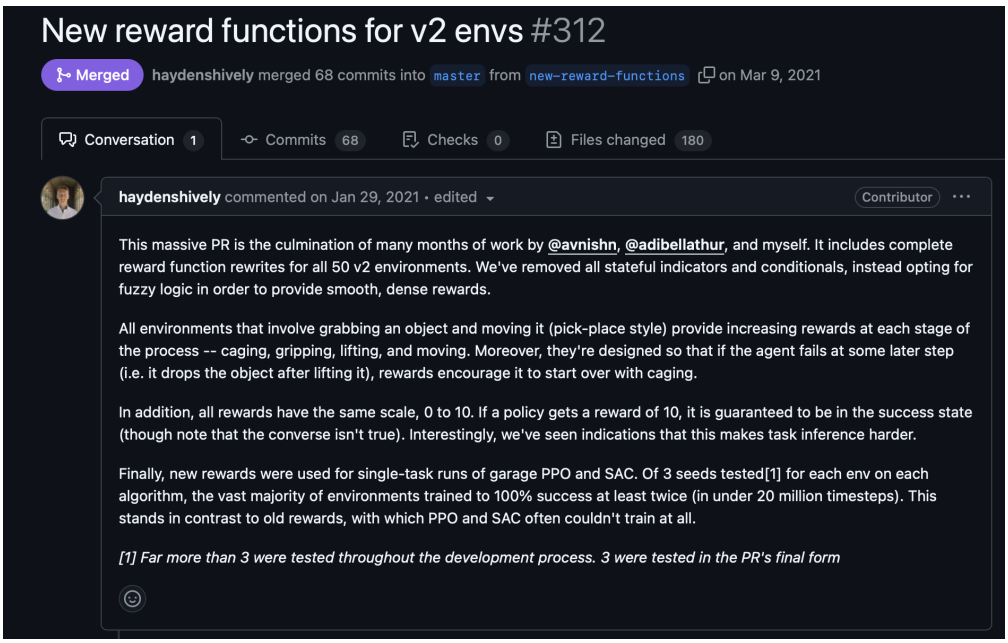


Figure 2.3.: A github pull request, demonstrating the inherent challenges of reward shaping, in the popular Multi-task benchmark environment Metaworld. https://github.com/Farama-Foundation/Metaworld/pull/312

resulting behaviour, and then adapts the behaviour accordingly. While a well-designed reward may improve sample efficiency and asymptotic performance of an RL algorithm, it is difficult to predict when it produces unintended local optima (termed *reward hacking* [76]).

The long training times inherent to RL make this kind of reward tuning a very slow process, and the tightness of the reward bottleneck as a communication medium makes it difficult to integrate human knowledge. Although to a human it is often plain to see what is going wrong, it is hard to take advantage of this knowledge.

Consider the following example: to solve a multi-step task, the task engineers may introduce rewards that are conditionally triggered when an agent achieves a subtask [102] (see 2.3).

2.2.2. Extensions

There are many extensions to this basic RL setup, which are worth mentioning as general principles to inspire the learning of diverse behaviours.

Goal-conditioned RL

Multi-task or **goal-conditioned RL** (GCRL) introduces a goal variable g to explicitly model the learning of multiple behaviours with a single goal-conditioned policy $\pi(a|s,g)$ [71], which map closely to our definition of behaviours. The hope is that the agent can exploit the shared structure between tasks, and learn multiple tasks more quickly than learning each from scratch, though the lower-level behaviours that should make this possible are not modelled directly, as in plain RL.

Imitation Learning

Imitation learning (IL) aims to circumvent the reward bottleneck by allowing the agent to learn directly from expert demonstrations $\tau = (s_0, a_1, ..., a_T, s_T)$ (see Section 2.4). *Behavior Cloning* methods aim to directly learn a policy π that produces the expert trajectory, but are brittle to deviations from this trajectory [106]. *Inverse Reinforcement Learning* aims to first infer the reward function that produced the expert trajectories, and then optimises this inferred reward function [106].

Hierarchical RL

Hierarchical Reinforcement Learning (HRL) [82] introduces the notion of temporal abstraction, i.e. of explicitly modelling *skills*, as in Definition 1 and learning a high-level controller to choose among the more temporally extended actions (termed options). HRL has long been hypothesised as key to tackling some of RL's most fundamental problems, centred around alleviating the curse of dimensionality [5], though its successes have remained largely hypothetical [59].

Curriculum Learning

Curriculum learning [6] proposes to start learning with easy tasks, and to gradually increase their difficulty, inspired by learning in humans. In the RL context, this typically involves the definition of task curricula by humans [60]. However, this relies on human understandings of task difficulty, which may not necessarily align well with difficulty for the RL agent.

Natural-Language conditioned RL/IL

We may also use a natural language description l of the task to condition the policy $\pi(a|s,l)$. Language is a natural medium for encoding abstractions, for generalisation, and most importantly: for communication. Language may be key to moving past *Tabula Rasa* RL, i.e. blank-slate learning [50]. See [112] for a more recent survey.

RL from Human Feedback

Reinforcement Learning from Human Feedback (RLHF) attempts to leverage non-expert human inputs, recognising that they are usually capable of effectively evaluating and providing feedback for learning agents. The reward function is dynamically defined and refined with humans in the loop, ideally allowing it to model complex human preferences [37].

2.2.3. Challenges

Aside from the reward bottleneck, RL faces some fundamental challenges arising from its general formulation. Perhaps most importantly, it requires a large amount of environment interaction, taking hours or days to train on expensive hardware.

Since RL is so data-hungry, it is usually trained in simulation environments, which can never completely model real-world environments. This makes the deployment of policies trained in simulation to real-world robots (Sim-to-real transfer) challenging, and is a significant contributor to the limited success of DRL on real robots [32].

Moreover, the policy resulting from this training is difficult to interpret, or to extend. Attempting to adapt a trained agent to a new task may cause it to no longer be able to solve prior tasks, a long known problem with neural networks termed *catastrophic forgetting* [19]. Solutions to this problem fall under the header of *Continual Learning*. Because the learned policy is difficult to adapt, much RL training is *Tabula Rasa*, meaning many complex functions must be relearned from scratch when training a new agent.

In our eyes, another significant weakness of DRL is that its general formulation belies all the human engineering required to make it work for a specific task, which we may refer to overall as environment shaping [64]. For example, parts of the state space $\mathcal{S}$ may be strategically selected and modified to speed up learning, and the action space $\mathcal{A}$ is typically altered to some higher-level representation than joint-level torques. For example, in manipulation environments, the end-effector is typically directly controlled in cartesian space. Due to the difficulty in introducing non-expert human feedback, the training burden rests entirely on experienced engineers.

2.3. Task-and-Motion Planning

Task-and-Motion Planning (TAMP) approaches focus specifically on very long-horizon scenarios in real-world, unstructured environments, like homes, hospitals or hotels [22]. As the name indicates, TAMP methods split this problem into two layers: high-level task planning, and low-level motion primitives. The task plan sequences, coordinates

and parameterises low-level primitives, which include both perception and control modules.

While this is reminiscent of Hierarchical Reinforcement Learning, TAMP methods express task plans in a symbolic language like PDDL [1] or Logic-Geometric Programming [85]. TAMP methods then use classical planning algorithms, like Probabilistic Roadmap Method (PRM) or Rapidly exploring Random Trees (RRT) to find optimal paths. The upshot of this is that we can compute guarantees, and that symbolic languages are interpretable. The drawback is that this tends to be difficult in dynamic environments, and that this inevitably also faces the curse of dimensionality as the number of primitives and parameters increases.

Moreover, these approaches require a manually defined domain model. This represents an attempt at providing an intuitive interface for leveraging human knowledge. For instance, the engineer may define the objects that are relevant to the agent, and the possible interactions with these objects. This is challenging and time-consuming, even for an engineer, and limits the possible instructions to those that can be expressed in the chosen symbolic language [79].

TAMP methods fit into contexts with *factorised perception and control*, unlike the end-to-end methods discussed in the previous section. Because of this, TAMP methods are well integrated with Robot Operating System (ROS) [53], an open-source library designed for the effective programming of robots that is popular in industrial applications.

2.4. Learning from demonstrations

Between hardcoded rule-based approaches and trial-and-error based learning, there is a spectrum of robot "teaching" approaches, wherein humans instil knowledge in more intuitive ways. These approaches may be referred to under the umbrella term of *Learning from Demonstrations* (LfD), or Programming from Demonstrations. This is appealing, because it allows application domain experts (non-expert in robot programming) to "program" the robot, rather than this burden falling on the robotics engineer. End-users often work in close contact with the deployed robots and are deeply familiar with the environment in which the robots operate, and as such are well-equipped to identify faults, corner cases and continually ensure reliable operation.

While the bulk of LfD methods fall into the category of Imitation Learning methods, LfD may be viewed as a superset [68]. There is a wide variety of approaches within LfD, including *kinesthetic teaching* methods, which allow human operators to physically move the robotic arm through the desired trajectories $(s_0, ..., s_T)$ – learning from *passive observation* of humans performing a task (like picking up an object) - and teleoperation, in which humans are provided some interface to control the robot.

2.4.1. Teleoperation

Teleoperation methods require some form of UI for a user to input trajectories, such as a joystick, haptic devices, VR interfaces, or even human-controlled grasp devices with cameras mounted to them [77] [3]. Importantly, this allows users to operate a real or simulated robot remotely, with the latter opening the door to crowdsourced collection of demonstrations [54]. The main drawback to these kinds of approaches, is that their success depends heavily on the provided UI, and that the development of this UI incurs costs and is not always straightforward. However, teleoperation has been successfully applied also to more complex systems, including robotic hands, humanoids, or underwater robots [68].

2.4.2. End-user Programming

End-user programming interfaces allow non-expert humans to "program" robots, by providing a simple UI for doing so, in the same way that modern computer software allows non-technical users to work with computers. While LfD methods typically try to eliminate programming completely, End-user programming aims to make it "simple enough", though the wide variety in end-user expertise in general programming and robotics makes this a challenging problem. Approaches range from simplified APIs (e.g. KUKA Robot Language or Universal Robots URScript) to fully-fledged GUIs [2].

2.5. Foundation Models

Foundation Models are large machine learning models, trained in an unsupervised manner on very large datasets, that can then be adapted for downstream, specific tasks [9], often significantly outperforming their specialist predecessors. Although the earliest and most well-known examples of Foundation Models are Large Language Models (LLM) like the first GPT [67] or BERT [16], both based on the *Transformer* architecture [86], Foundation Models have since been trained for a variety of domains and modalities, and with a variety of techniques. Other notable examples include the Vision-Language model (VLM) CLIP [66], which generates textual descriptions of images, RT-2 [10] for natural language conditioned robotic control, Gato [69], purporting to be a general-purpose agent foundation model, and GPT-4o [31], a multi-modal model mapping text, image, sound or video to text, image or sound. The global research community is still struggling to get a grasp of the capabilities and limitations of these models.

In this thesis, we are primarily interested in LLMs, especially as applied to the problem of code generation. We describe the interface with which we may can tune them to a

specific task and provide an overview of the emerging literature on Foundation Model Agents. For completeness, we briefly discuss the robotics-native Vision-Language-Action (VLA) Foundation models.

2.5.1. Interface

Foundation Models are named as such because they are viewed to be *incomplete*, completed by adaptation to a specific task [9]. This adaptation can be done in a number of ways, discussed in the following.

In-context learning

In-context learning describes all approaches that involve formulating "better" inputs to the FM, sometimes also dubbed prompt engineering. In other words, it refers to all approaches that improve the outputs of a model without altering the model itself.

For instance, we can provide the model with examples of the kind of responses we might expect, a technique referred to as *Few-shot learning* [11]. This may be viewed more broadly as providing the FM with a clearer specification of what we "want" from it.

Chain-of-thought [96] prompting involves prompting the FM to produce a specific sequence of output tokens that is supposed to make successful responses more likely, often reframed as prompting the FM to "reason", or to "think". For instance, when solving math questions, it makes sense to output intermediate calculation steps, each of which must be correlated with one another in some way, rather than attempting to output the answer without these intermediate steps.

A number of techniques somehow chain and/or aggregate multiple calls to the FM, to both take advantage of and also compensate for the probabilistic nature of FM outputs. For instance, if we expect the LLM to answer an arithmetic question, we may prompt it to do so multiple times, and select the most frequent answer [93]. Taking this idea further, we can impose "reasoning" on the LLM by getting it to progressively generate *trees* [100] or *graphs* [7] of "thoughts", embedding LLM calls into a search structure managed by the model itself, and taking a step towards *Language Agents* (see Section 2.5.2).

Retrieval-augmented Generation

Retrieval-augmented generation (RAG) is a technique to provide LLMs with query-related information from a knowledge base [45], typically to combat *hallucinations*, i.e. the generation of plausible but non-factual content.

In its simplest form, the database contains documents d_i, indexed by a vector embedding $E(d_i)$, generated by a text-encoding model E like OpenAI's text-embedding-3 or SBERT [70]. Given some query, we can then retrieve documents that are semantically similar to this query, where semantic similarity is typically measured by cosine similarity $sim(d_1, d_2) = E(d_1)\dot{E}(d_2)$.

In the context of question-answering (QA), the query might be the question, for which the answer might be contained in one of the documents. Typically multiple documents are retrieved, and the LLM is left to choose what information it uses in its response.

In this way, RAG offers an approach to extend the capabilities of an LLM without altering its weights. It allows developers to effectively separate factual knowledge from the knowledge contained in the training parameters of an LLM. With an effective retrieval pipeline, the role of the LLM is reduced to translator, or interpreter. Importantly, applied to code generation, this could enable an LLM to effectively incorporate unfamiliar APIs.

However, this example also highlights a weakness of RAG, as we are measuring the similarity between prompt and response - a more effective pipeline would embed each document with a hypothetical prompt it is the response to [20]. Ineffective retrieval, and thus attaching irrelevant information to the prompt, both makes inference more costly, and may increase the likelihood of false answers [72].

Tool use

Another common paradigm within LLM applications is tool use, referring broadly to the use of external functions. RAG may be viewed as a database access tool. Other examples of tools include a code interpreter, which the model may invoke to check code it generates for runtime errors, or a function to move a robotic end effector. Tools may represent workflow components the model is technically capable of, but that we wish to be reliable and tested, or it may represent components the LLM is simply incapable of. Tools can further enhance the robustness, interpretability, and domain expertise of models [65]. Extending this notion, the agent may also call other agents, with different roles or capabilities [107], discussed further in Section 2.5.2.

Finetuning

We can finetune a Foundation Model by post-training it on a smaller, domain-specific dataset. Similar to few-shot prompting, it steers the model towards the part of the problem space we are interested in, rather than serving as a viable method for knowledge injection [62]. For instance, if we wish to use the model for text classification, we may

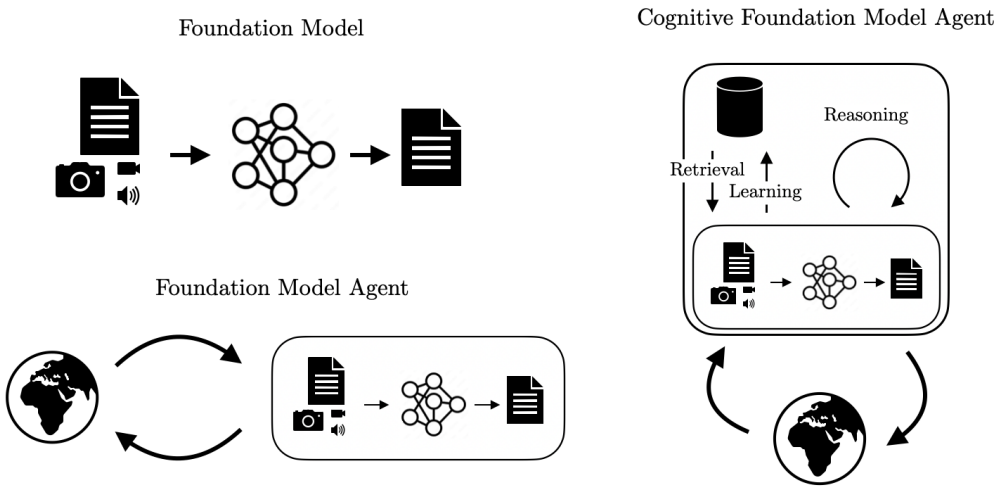


Figure 2.4.: Cognitive Foundation Model Agents, adapted from [80].

finetune it on labelled data, indicating both the specific task, and how we would like the task to be solved.

2.5.2. Foundation Model Agents

An intuitive next step is to provide FMs with tools to perceive and act in an environment, creating Agents. We may describe the sum all tools and structures built around Foundation Models as their *cognitive architecture* (following [80]), visualised in Figure 2.4.

Planning

Planning may be understood synonymously with *reasoning*, and involves any FM interaction that happens between perceiving and acting. When combined with a set of control primitives, this may be viewed analogously to task-planning in TAMP. In the context of language agents, this has been extended to employ common search algorithms like Monte-Carlo Tree Search (MCTS) [24]. When integrated with closed-loop feedback from the environment, numerous works have demonstrated the benefits of leveraging LLMs as planners [30]. On the other hand, FMs do not contain perfect world models. As with any model-based approach, if planning is not sufficiently interlaced with grounding feedback, it tends to diverge from reality [35].

Feedback and Grounding

Foundation Models are capable of interpreting complex state descriptions from the environment. For instance, in VOYAGER [89], a minecraft agent is expected to interpret a full textual description of the environment, including its inventory at every timestep. In Reflexion [73], the authors show that language agents can respond to scalar reward signals, dubbing this capability a form of "verbal reinforcement learning". VLMs are capable of interpreting and acting on visual inputs [27].

Importantly, the language-first nature of these models provides an intuitive and powerful communication mechanism for humans to interact with these agents and provide feedback [29]. While this capability is made note of in most works, few make this the central thesis of their method.

Memory and Learning

As described previously, we may imbue agents with an external memory. Importantly, this also introduces a mechanism for *learning*, by writing to this memory for future retrieval. For example, the agent may store validated code-pieces for later use [89], user corrections [109], or insights gleaned from experience interacting with a specific environment [111], discussed further in Section 3.3.1.

Multi-agent collaboration

The notion of introducing multiple agents is often somewhat of a misnomer, as they typically collaborate towards achieving a single goal, and may as such be viewed as parts of a single agent. However, it encapsulates the idea that we can introduce distinct in-context learning and RAG setups within a single problem-solving setup, and this may improve overall performance. This is a common design pattern in recent works, for instance introducing modules to self-reflect [73], generate multiple plans and select the best one [95], or generate task curricula [89].

2.5.3. Vision-Language Action models

Vision-Language-Action (VLA) models are robotics-native Foundation Models, representing attempts to translate the success of Foundation Models to this domain. Examples include RT-2 [10], Octo [84], and OpenVLA [39]. VLA are end-to-end models, which directly translate robot visual observations along with natural language task inputs to low-level robotic actions. The hope is that we may then experience similar "emergent" capabilities and generalisation in robotics, that we saw in text generation, and that the internal "common sense" of existing Foundation Models translate to

robotic manipulation. Moreover, as with LLMs, finetuning on domain-specific data (i.e. demonstrations) would then ideally allow an intuitive interface for adaptation.

These approaches are fundamentally limited by the lack of existing large-scale robotics datasets, with the largest being the Open-X Embodiment dataset [61], at roughly 1 million robot manipulation trajectories across multiple emobodiments. Due to their large size, they also demand for novel solutions to improve inference speeds.

3. Background

In this chapter, we provide a more focused overview of how LLM and VLM models have been applied in the context of robotics. We focus specifically on the effective use of skills, and on their acquisition.

First, we extend the definitions in Chapter 2. We are interested specifically in works that directly use an LLM f to map a natural language instruction l to a robot control code c, i.e. $f(l) = c$, and term f a *Language Model Program* [47]. Multiple representations of c have been explored in the literature, most notably python code, PDDL, or reward function code to train a parametric policy via RL, or for use with fast control-based methods. We use c to represent the policy to highlight that, in this thesis, we are interested in methods that translate natural language to python control code. As in Chapter 2, there is a direct mapping from the policy c to a behaviour $b = (s_0, ..., s_T)$, and as such we may by extension view f as a mapping from natural language instruction l to behaviour b (Figure 3.1).

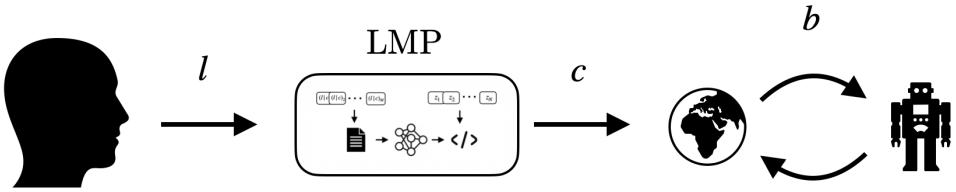


Figure 3.1.: Language Model Programs allow us to generate robot behaviours b from natural language instructions l, by rolling out the generated policy code c in the environment.

Such works typically rely on In-Context Learning (section 2.5.1) to adapt general-purpose models for their specific task, via two primary mechanisms: providing few-shot examples $\mathcal{E} = \{(l_1, c_1), ..., (l_M, c_M)\}$ that demonstrate the desired mapping from instruction to policy code, and a set of **skills** $\mathcal{Z} = \{z_1, ..., z_N\}$ that the agent may call to produce c, which correspond with tools in the broader literature on foundation model agents. This is visualised in Figure 3.2.

Both the number of few-shot examples $|\mathcal{E}| = M$ and the number of skills $|\mathcal{Z}| = N$ are

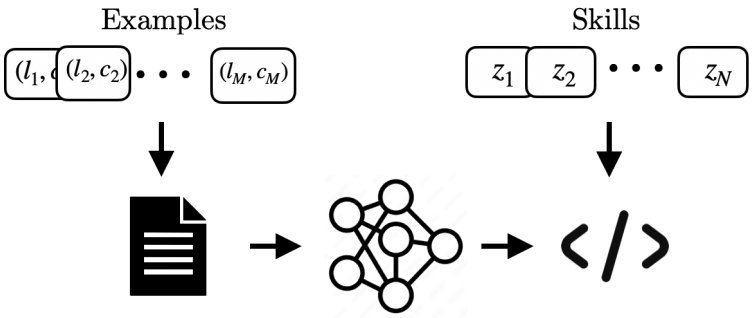


Figure 3.2.: A Language Model Program is a Foundation Model tuned for a specific task via In-Context Learning (see section 2.5.1), in this case by providing a fixed set of skills (tools) (z_i), and few-shot examples $((l_i, c_i))$ demonstrating their use.

typically fixed at the outset. The hope is that the examples are sufficiently representative of the overall set of language instructions $\mathcal{L}$ and desired behaviours $\mathcal{B}$, such that the agent can *generalise* to unseen instruction-behaviour pairs (l', c'). This presumes that the instructions l' accurately reflect the humans intention, and that the LLM is capable of correctly interpolate l' from the given examples $\mathcal{E}$, and thus to produce the desired behavioural response c', i.e. b'. As we will discuss in Chapter 4, these are strong limitations.

In practice, this means the abilities of the agent are defined at the outset. There is no mechanism for *learning* (see Section 2.5.2), so the space of behaviours $\mathcal{B}$ that the agent can execute is fixed, and dependent on the effective choice of examples $\mathcal{E}$ and skills $\mathcal{Z}$, because the prompt is fixed across generations. Moreover, the mapping from natural language prompt l to behaviour b is encoded in f, with no way to refine this mapping other than rewriting the prompts.

In the following, we begin by discussing Code-as-Policies, as a representative work in generating robot code from natural language instructions, and use this as a springboard to discuss some of the fundamental benefits and limitations of this general approach, outlined in the paper. We then provide some examples of how Language Model Programs have otherwise been used in robotics. Finally, we discuss insights from the broader space of Language Model Agents (see Section 2.5.2), intended to enable the agent to *learn* from experience, either by independent interaction with the environment, or from human feedback. This is enabled by building a *cognitive architecture*, to allow for *dynamic prompt composition* based on the current instruction l.

```
objs = ['red block', 'blue bowl', 'blue block', 'red bowl']
# the left most block.
block_names = ['red block', 'blue block']
block_positions = np.array([get_pos(name) for name in block_names])
left_block_name = block_names[np.argmin(block_positions[:, 0])]
ret_val = left_block_name
```

Figure 3.3.: An example output from the `parse_obj` Language Model Program.

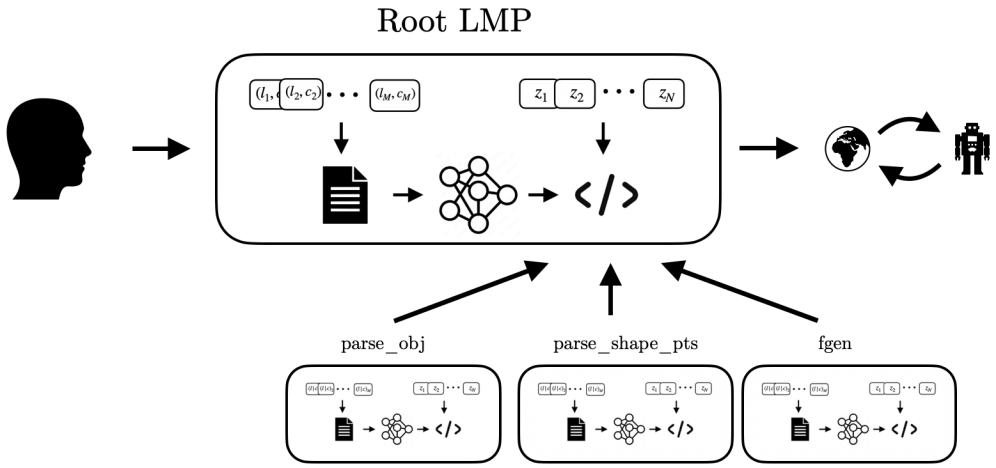


Figure 3.4.: Code-as-Policies demonstrates that we can compose Language Model Programs hierarchically, with each LMP fulfilling a distinct function.

3.1. Code-as-Policies

Code-as-Policies (CaP) [47] first convincingly demonstrated the usefulness of using an LLM generating python code as the orchestrator for a robotic agent, coordinating different perception and control APIs to generate behaviours from natural language.

As in the previous discussion, each LMP maps a natural language instruction to python code, and is few-shot prompted to fulfil a specific role in the behaviour-generating pipeline. For instance, the `parse_obj` LMP maps an open-vocabulary natural language description of an object to a specific objects representation in the python environment (see Figure3.3).

The authors also define other LMPs, for example for converting a natural language description of a shape to a corresponding set of waypoints, or for generating functions in the policy code c that are undefined in the code environments. Importantly, the authors show that in this way LMPs can be composed hierarchically (see Figure 3.4).

These LMPs can be defined in a use-case specific way. Each LMP is defined by its

own set of few-shot examples $\mathcal{E}_L MP$ and skills $\mathcal{Z}_L MP$, thus tailoring each to specific function in the robot pipeline. A number of other works follow the same basic structure (discussed in Section 3.2), providing a simple way to modularize and integrate many different programs.

3.1.1. Benefits

Representing policies as python code has a number of advantages, discussed briefly here. Moreover, it allows us to leverage results from the much broader space of Language agents that use code as their language of expression.

Simple Integration of Perception and Control modules

In CaP, the authors use open-vocabulary object detection models out of the box. Many later works in this space follow-suit (see Section 3.2). Each of these functions can be treated as an LMP, and thus easily be integrated with the CaP framework. This large diversity of complementary approaches enables a new modular approach to imbuing robots with the necessary skills to adapt to a given environments affordances.

Moreover, there is a large number of third-party python libraries that augment an agents capabilities in a straightforward way, that it is likely capable of using somewhat effectively due to their presence in the data used for pretraining the initial model. The authors of CaP incorporate numpy and shapely packages for geometric reasoning, though we can imagine this opens the door to a large number of existing libraries, for example related to Computer Vision or Motion Planning.

Interpretability

Code is the standard mode of communication between humans and computers, and particularly python is already a very popular choice for research. We can read code, and we can edit it. Moreover, code comes with a vast ecosystem for debugging and error-handling, and agents may use stack traces to fix bugs autonomously [13].

Code enhances reasoning capabilities of LLMs

A key result in CaP is that planning with code is more effective than generating Natural Language task plans. One reason is that we can naturally reason about spatial-geometric relationships using code. But a more important reason is that code is broadly a more effective language for expressing plans [92], and for tool use [99].

It addresses some of the core weaknesses of end-to-end approaches

Many of the core challenges of parametric approaches can be addressed by representing policies as code, including but not limited to:

Sim-to-real transfer – This problem is localised to the particular control and perception primitives, and can be tackled on a case-by-case basis.

Catastrophic Forgetting – New skills can be added without inadvertently overwriting old ones.

Cross-embodiment – By implementing the low-level API used by CaP on different robot embodiments, the method can be made somewhat neutral to the choice of embodiment.

Training Time - CaP systems can be deployed without any further model training.

3.1.2. Limitations

This common setup, presented in CaP and replicated in numerous later papers, in which a root LMP can call on a fixed set of primitives/LMPs, has some limitations.

In this setup, the prompts for each LMP are hand-engineered with few-shot examples. Since the LLM is dealing with an unfamiliar API, it needs these few-shot examples to understand the functions, their parameters, and their usage. As the authors note, this produces a setup in which "only a handful of named primitive parameters can be adjusted without oversaturating the prompt".

This problem can be illustrated easily in CaP. We have a small number of low-level LMP's (e.g. `parse_obj`) , each with a specific task, and the associated few-shot examples. The main LMP calls these LMP's, as well as a number of other primitives, so the few-shot examples for the main LMP need to give examples of different parameterisations for each of these, as well as demonstrate their different usage contexts. As the number of functions that the main LMP can call grows, so do the number of examples needed to allow it to function properly. In a similar vein, the authors note that CaP doesn't handle commands well that operate at a different level of abstraction than the provided examples. For example, the authors state that CaP couldn't "build a house" in the table-top domain, because they included no examples of complex structures.

3.2. Language Model Programs for Robotics

Language Model programs have since been used to fulfil a wide variety of roles in the context of robotics. There are a large number of surveys describing the broader

integration of Foundation Models for robotics [40] [90] [87] [112] [26]. We provide some examples to give an overview here.

A popular pipeline in recent works has been to first use Segment-Anything [41], a Foundation Model that allows prompt-based image segmentation, followed by the VLM CLIP [66] to label these segmentation masks, though there are a variety of Foundation Models to choose from to fill either role.

If we have access to point-cloud data, we may also use an object-agnostic grasp pose generator [104] coupled with a VLM to select the best grasp pose [17]. Another recent paper shows how we might use an LLMs capability to infer properties of an object to be grasped, like its mass, surface friction and deformability to further inform grasps [98]. When learning from human grasps, we could employ common hand-detection models [58] [88].

VoxPoser [28] uses a VLM to generate affordance maps (i.e. a heatmap that shows where in the environment it is "good" to move the end-effector, and where it is "bad"), and then uses a motion planner to guide a robotic arm to trace out collision-free trajectories towards goals specified purely in natural language.

A number of works also explore using VLMs as cheap behaviour critics, whether by evaluating the final frame [110], multiple frames from different viewpoints [17] or an entire trajectory [21], while also being capable of providing feedback as to what might have gone wrong.

Another common application of Foundation Models in Robotics is in autonomous planning, i.e. in generating task decompositions for long-horizon plans, which was previously the domain of Task-and-Motion planning. Some works use the LLM directly as a translator to PDDL [49], allowing it to take advantage of the existing infrastructure around it. More commonly, plans and task decompositions are expressed directly as hierarchies of Natural Language, or as code, as in CaP. ProgPrompt [75] attempts to provide a structured approach for prompting LLMs in this context.

3.2.1. Foundation Models for Reinforcement Learning

A subset of methods has also focused on leveraging Foundation Models directly to improve the sample-efficiency and performance of Reinforcement Learning. For a review, see [12].

LLMs may be employed in a number of places, but the most obvious one is to translate Natural Language to reward code [103]. Other works have used policies generated by CaP as exploration guides while optimising a parameterised policy [101]. Similarly, ScalingUp [23] uses an elaborate CaP setup to autonomously solve tasks, along with a verifier to classify trajectories according to their success. These successful trajectories are then used to train a Diffusion Policy [14].

RL-VLM-f[94] uses a VLM to give preferences over pairs of an agents image observations to learn a reward function. In EUREKA [52], the authors propose to leverage fast simulators to perform autonomous reward *evolution*, directly employing the LLM as an RL engineer, proposing a reward function, training the agent, and then revising the reward function based on the results, finding that the agent can outperform human expert reward engineers.

GenSim [91] shows that we can improve the generalisation of RL agents by generating task setup variations using coding LLMs, again alleviating the burden on the RL engineer.

The advantage of these approaches is that we retain the flexibility of end-to-end approaches, while extracting relevant knowledge into fast policies that no longer require Foundation Models in the loop.

Following a slightly different route, a small number of recent works directly address the fixed skill library discussed at the beginning of the chapter. The authors of League++ [46] propose to extend the skill library by using the LLM as reward engineer for DRL-based skill learning, along with another LLM to translate natural language prompts into TAMP plans. Two other recent works [63][55] propose to learn new skills via Imitation Learning, which they demonstrate both on a real robot and in a simulated teloperation environment. The operator provides 5-10 demonstrations of the task, from which the agent may learn via Neural Descriptor Fields [74]. These skills are then made available to the CaP-planner.

3.3. System 2 Learning

We now turn our attention to a specific subclass of these approaches that leverage non-parametric adaptation, which we refer to as *System 2 Learning*, following the popular idea from cognitive psychology [34]. In this analogy, System 1 refers to the fast, black-box processing enabled by neural networks, while System 2 refers to the slower processing enabled by symbolic languages. As discussed in Section 2.5.2, System 2 learning in Foundation Model agents is enabled by writing useful knowledge to long-term memory, which can then later be retrieved in a task-related manner. As such, it may be viewed as a method for dynamically managing the context window of the Foundation Model.

While this idea has been explored in the broader domain of autonomous Language Agents, it has received less attention within the Robotics domain. We distinguish between two crucial subproblems: learning from experience, and learning from human interaction.

3.3.1. Learning from Experience

In order for an agent to autonomously learn from experience, it must be able to interpret the experience in a way that it can leverage for downstream tasks.

VOYAGER [89] demonstrates this in the popular videogame Minecraft. Initialised with a small set of control and perception primitives, it interacts with the environment autonomously, and learns new skills in the process, slowly increasing the level of temporal abstraction at which it can act. This process relies on a curriculum-generating LMP to propose tasks of appropriate difficulty, and on a self-verification LMP that determines whether a behaviour was successful. If successful, the behaviour is stored to the skill library, and may be retrieved and used for downstream tasks.

The authors find that this growing skill library, and the accompanying introduction of temporal abstraction are key to the continual learning capability of their agent. Without the skill library, learning plateaus eventually, when the programs the agent would need to synthesise to solve a task become too long.

In a similar way, in ExpeL [111], the authors introduce an LMP for extracting natural language *insights* (e.g. "you can't do this when...", or "when this happens, do this...") from environment interaction in different domains. These insights are then similarly retrieved in a task-specific manner. The success of this method is demonstrated (among others) in the long-horizon embodied reasoning benchmark Alfworld.

3.3.2. Learning from Human Interaction

Distilling and Retrieving Generalisable Knowledge from Online Corrections (DROC) [109] shows how the ideas in the previous section might translate to robotics. Particularly, they use the same principles to more effectively leverage human interaction. Rather than generating insights from autonomous interaction with the environment, these insights are generated from user feedback, and appended to prompts in a task-related manner.

TidyBot [97] similarly tackles the problem of *personalising* robots, testing this ability on the problem of picking up objects and restoring them to the locations preferred by the user. Users provide preferences, and the LLM is used to summarise these preferences, and convert them into situated robot control code.

PromptBook [4] provides another principled approach for adapting to feedback, by allowing the LLM to rewrite prompts based on user feedback. Similar to DROC, the authors provide an LMP with a history of (task, code, feedback) tuples, but then ask it how it would revise a specific LMP *prompt* to integrate this feedback for downstream tasks.

ShowTell [58] proposes a novel method for Programming by Demonstrations, using a

VLM and LLM to interpret a human demonstration with accompanying language narration, and generating modular robot code that imitates the demonstration, demonstrating that low-level behaviours (i.e. skills) can be synthesised using a similar approach to CaP.

In a somewhat different vein, another recent work [43] directly attempts to incorporate a large number of skills, in the form of well-tested control blocks, via Retrieval-Augmented Generation. We mention it here because it demonstrates that we *can* take advantage of a large skill library, should it already exist, though no learning takes place in this work.

4. Motivation

Foundation Models have enabled new methods for interacting with robots, and imbuing them with more capabilities. In particular, it has become more straightforward to develop robots in a modular, environment-dependent fashion, by providing them with an appropriate base set of skills, and then using an LLM to effectively orchestrate these skill calls. However, as of yet, few works integrate mechanisms for learning from human feedback into their agents. We view this as an important direction for research, since robotics is a domain in which humans are exceptionally capable of providing corrective feedback to improve the agents performance, and since natural language provides a very intuitive interface for communicating these corrections. Moreover, we want to take advantage of how quickly these algorithms produce behaviours from natural language instructions, to enable an *interactive behaviour creation experience*.

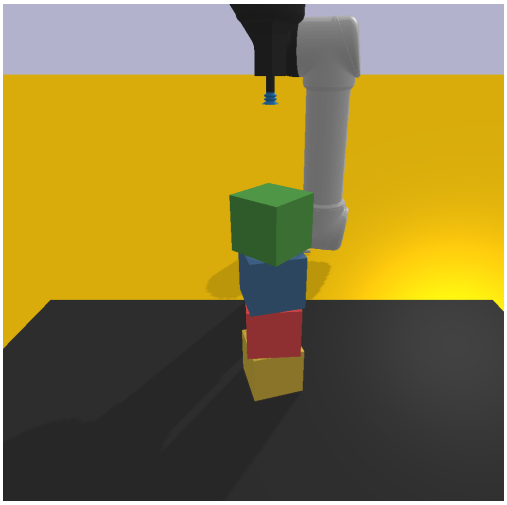
4.1. Problem statement

The focus of this thesis can be illustrated by a simple example. A simple version of Code-as-Policies in the block manipulation domain is capable of stacking blocks. However, since this behaviour is entirely generated by the LLM, there is some noise to it, and without further prompt manipulation, this doesn't always align with our expectations (Figure 4.1a). Moreover, the agent isn't always successful in stacking the blocks, since sometimes it tries to place a block where it collides with another block (Figure 4.1b).

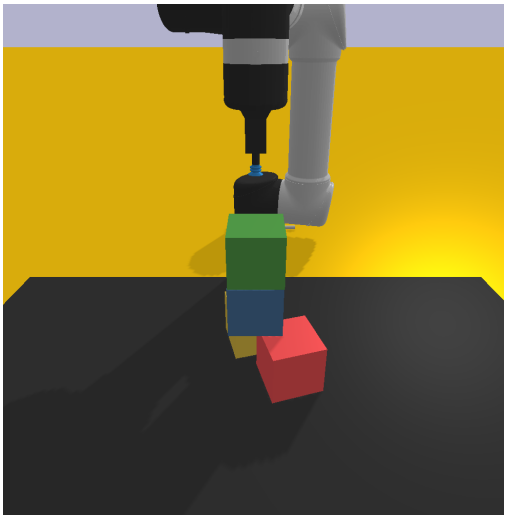
These problems are straightforward for humans to identify, and to communicate to the agent with an LLM as translator. However, to the best of our knowledge, there are only few works actively trying to leverage this information source in this context.

Moreover, as discussed in Section 3.1.2, the capabilities of the agent are limited from the outset by the set of few-shot examples and tools provided to it, and there is no clear mechanism for extending these capabilities, or for online adaptation. As long as we operate with fixed prompts, the number of primitives the agent can handle is limited.

Another fundamental limitation of these approaches is the inherent ambiguity of natural language. While a human issuing a command has an imagination of the behaviour they wish to elicit, it is likely underspecified or imprecise. As these behaviours become



(a) The blocks are not stacked corner-to-corner. We can augment the prompt, but how do we ensure it just stacks the blocks the intended way, every time?



(b) Stacking fails because another block is in the way.

Figure 4.1.: Responses to the prompt "stack the blocks", demonstrating both the inherent ambiguity in language, and failure modes that are easy for humans to correct.

more complex, this problem becomes more pronounced. Even in the simple example in Figure 4.1a, the agent produced a valid response - it was just not the desired one.

4.2. Skill learning via natural language interaction

Code-writing LLMs can generate and revise behaviours according to user inputs on the order of seconds, creating tight human-agent-environment feedback loops (see Section 2.1). The user can prescribe a behaviour, and see the result within seconds, enabling an entirely new interactive behaviour creation experience. However, there needs to be an explicit mechanism for taking advantage of this interaction, absent in prior work.

Ultimately, the goal of this interaction is to learn a *shared language* with the robot: a library of mappings from a natural language instruction l to an expected behavioural response $b \in \mathcal{B}$. We posit that the only reliable way to learn this shared language is by letting users define the *meaning* of their prompts, i.e., not hoping that the LLM can infer it correctly based on its "common sense". In other words, we want to enable humans to *teach* the agent what we mean, visualised in Figure 4.2.

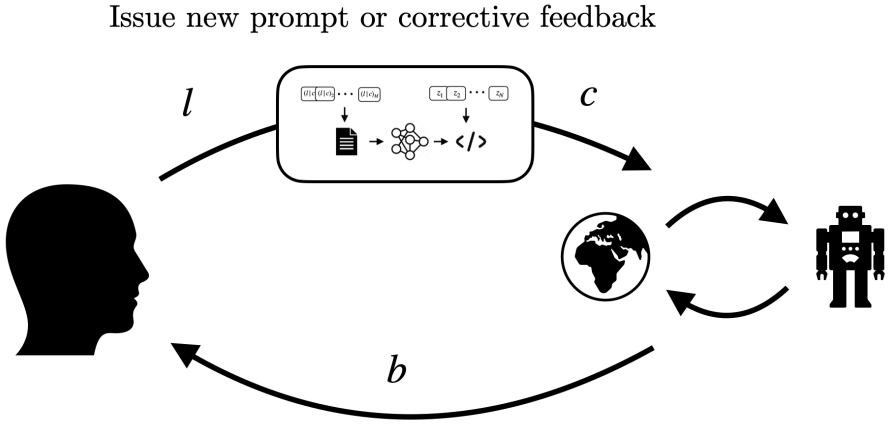


Figure 4.2.: LMPs allow humans to issue instructions via natural language, evaluate behaviour and provide corrections on very short time-scales.

While this is already possible in the basic CaP framework, there is no framework for learning from these corrections. For example, if the instruction l is to "stack blocks", and the user has to correct the agent once before the desired behaviour b is evoked, the

agent will do it wrong again the next time (or not, by chance). We would like the agent to *remember*.

This can be accomplished in a conceptually simple way. We note that few-shot examples are our way to steer the LLM: they provide examples both of the (l, b) mapping, and which skills $z \in \mathcal{Z}$ are relevant for a specific task l. We may then store the successful pairs (l, c), and retrieve these when the user issues a *similar* prompt l' (see Section 2.5.1).

However, this alone doesn't provide a mechanism for *ensuring* that a learned behaviour mapping (l, b) is executed the same way the next time the same language instruction is uttered: the agent may simply ignore parts of the examples that are important to us, and LLMs can be very sensitive to the exact composition of the input prompt – there may be *unwanted behavioural variance*. Moreover, as behaviours become more temporally extended, and the resulting code strings become longer, the agent becomes less capable of generating them [89]. In the same vein, it becomes more difficult to respond to corrective feedback when applied to longer code strings.

We propose to circumvent this by learning *skills*, i.e. reliable and reusable behaviours that were validated by human feedback. In the context of code-writing agents, this means simply that we want to learn python functions, modelling the assumption that the code hidden behind the function header has been sufficiently verified by human interaction, and that we don't want it to change in the current problem-solving context. We identify many advantages of this approach, discussed in Chapter 6.

Skills effectively model the assumption that different parts of the problem-space require different knowledge. For example, to move a 6-DOF end-effector from one position to another, we might want to employ motion planning control primitives, whereas to grasp an object we might use a specific grasp module. Code-as-Policies addresses this problem by composing Language Model Programs hierarchically. We view our solution as complementary, in that we view skills as the most intuitive mechanism for reliably injecting user knowledge into the system. We view it as a step towards embodiment-agnostic, intuitive end-user programming and human-verified behaviour data collection via teleoperation, particularly when coupled with high-fidelity simulators.

While there are some works exploring online *skill* learning in this context, to the best of our knowledge there are none that focus on doing so purely from natural language interaction, nor that focus specifically on learning a shared language between agent and user.

5. Method

Our aim is to create an interactive behaviour creation experience: the user proposes a skill in natural language, along with some tasks that verify that this skill is successful, the agent produces a behaviour, and the user provides corrective feedback until this behaviour is aligned with the users expectations. Once this is the case, the behaviour is memorised and stored for later use. This allows us to build a growing library of verified behaviours, which can reliably be invoked by natural language instructions.

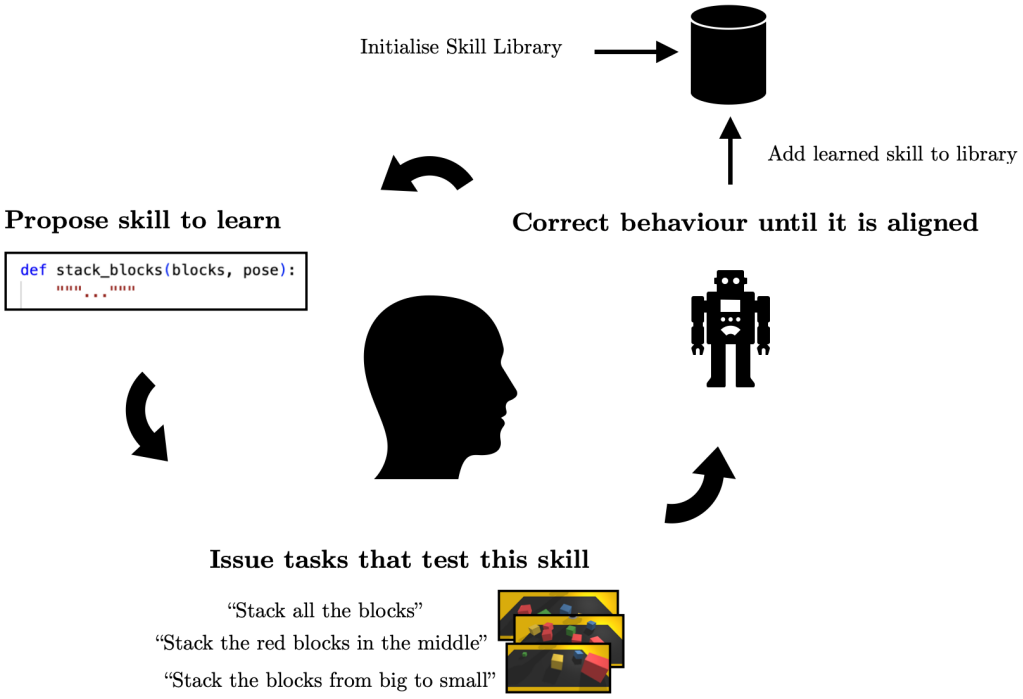


Figure 5.1.: An interactive behaviour creation experience. The user is in control of every step of the process.

As described in Section 3, we build on a function $f(l) = c$ that maps natural language inputs l to python robot policy code, and thus a behaviour b, and that does so on the order of seconds, rather than hours or days. As discussed in the previous chapter,

we aim to learn *skills*, represented as python functions, and to do so explicitly, with human guidance. We then want to enable the agent to effectively use these skills for downstream tasks, to respond to user instructions and corrections in a predictable manner.

In this thesis, we choose a very deliberate, user-centric approach to learning skills, in which the user specifies both a general description of what the skill is supposed to do, and then proposes a small number of semantically different tasks that test this skill. Figure 5.2 gives an example of this. Only if the skill can be used successfully to solve all tasks proposed to test it, is it accepted and committed to memory, and is made available for downstream tasks.

Similar to previous works, the agent is initiated with a fixed set of skills $z \in \mathcal{Z}_0$, along with examples $e = (l, c) \in \mathcal{E}_0$ that demonstrate their use, that enable basic environment interaction. The user then interacts with the agent on a trial-and error basis, proposing different behaviours, providing feedback to refine them, and then either discarding them or committing them to memory. In this way, the agent progressively becomes more and more capable, building a growing repertoire of verified behaviours in a manner that is tightly integrated with human intentions.

Ultimately, as described in Chapter 2, we expect our agent to be capable of a specific set of behaviours $b \in \mathcal{B}$, and we want to be able to elicit each of these with a known set of natural language instructions $l \in \mathcal{L}$.

We outline this method by further elaborating the specific role of *tasks* and *skills*, then we discuss the basic components that are required to make this work, and the toy environment in which we test our approach.

5.1. Tasks and Skills

While a task represents a specific instantiation of a desired behaviour, skills are one layer of abstraction removed, and represent a reusable behaviour, with defined parameters of generalisation.

In the following, a task (l, s_0) is given by a natural language description l and an initial environment setup s_0. This is the primary interface with which the user interacts with the agent. When a task is solved successfully (with user feedback), we append the solution code c and achieved final state s_T to this tuple (l, s_0, c, s_T).

Skills z represent the reusable behaviours that enable task solutions. In order to learn a skill z, we specify tasks $\{(l_i, (s_0)_i)\}_i = \{1, ..., \mathcal{E}_z\}$ that the agent should be able to solve successfully with this skill, including variations in both instruction l and initial environment configuration s_0. This relationship is visualised in figure 5.2.

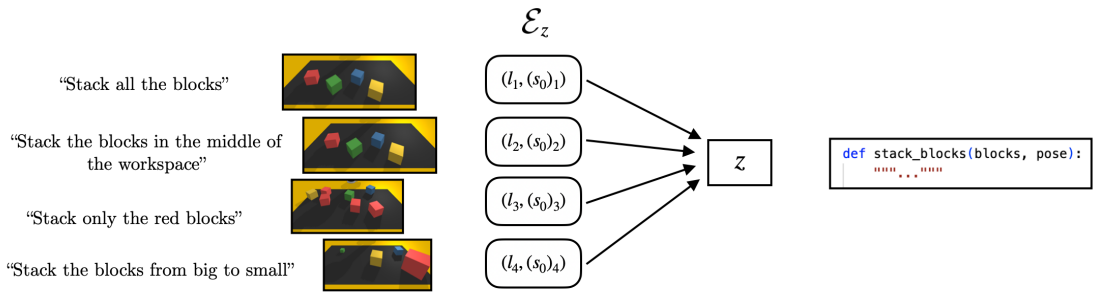


Figure 5.2.: Tasks and skills. While learning, each task the user proposes minimally tests a specific skill.

The solved tasks $\{(l, s_0, c, s_T)\}$ specified while learning a skill play an important role: they serve as the few-shot examples (l, c) for downstream tasks, demonstrating the successful use of the skill, and the different natural language instructions that elicit the particular variations in how this skill is used. As such, these tasks should be selected deliberately, minimising redundancy. For example, if we are learning the skill of picking up a book, it should not be necessary to first move around other obstacles. Instead, each task might correspond to a meaningfully different position in which we would find a book, for instance lying flat on the table, or standing in a book-shelf. The concrete tasks that are chosen should depend entirely on the environment in which the robot operates, and what is expected of it in this environment.

After learning a skill successfully, this leaves us with a set of (l, c) pairs, i.e. pairs of language instructions l and code-strings c, where c calls the skill z (as well as other previously learned skills) to solve the task l. When learned successfully, the skill z can be added to the skill library $\mathcal{Z}$, and the corresponding examples (l, c) can be added to the example library $\mathcal{E}$, which represents the set of all instructions we have *taught* the agent. We use $\mathcal{E}_z \subset \mathcal{E}$ to represent the (l, c) pairs used while learning the skill z.

While the resulting pairs of (l, c) show the agent how to use the skill, the skill itself represents the single-source-of-truth that encodes the concrete reusable behaviour the agent is trying to learn, and encapsulates the parts of the behaviour that the user does not want the agent to vary in downstream tasks. In other words, a skill is a python function, and the task-specific code calls this function and sets its arguments appropriately. This distinction is visualised in Figure 5.3.

```python
# Task: stack only the red blocks
blocks = get_objects()
red_blocks = [block for block in blocks if get_object_color(block) == "red"]
middle_of_workspace = Workspace.middle
stack_blocks(blocks, middle_of_workspace)
```

```python
def stack_blocks(blocks, pose):
    """Stacks the blocks at the given pose, ensuring all blocks are rotated the same way.
    Blocks are placed in the order in which they are given, from first to last."""
    for block in blocks:
        cur_block_pose = get_object_pose(block)
        put_first_on_second(cur_block_pose, pose)
```

Figure 5.3.: An example of the task-skill separation. On the left is the task-specific code, demonstrating how to correctly use the *learned* skill (on the right).

5.2. Skill Learning

The core mechanism we want to introduce in this thesis is that of learning skills. As described, this requires two things: the current skill z^* that is being learned, and a concrete task (l, s_0). The skill is given by a python function, which may or may not already be implemented, which we refer to as c_z^*. To learn the skill, we prompt the code-writing agent f to jointly output both the task-specific code c and the skill-code c_z^*, though in principle these could also be generated separately. This makes the code-writing problem slightly more difficult, by imposing a bottleneck through the function header, but it enables us to encapsulate the code we do not want the agent to change in downstream tasks.

Figure 5.4 and Algorithm 1 provide an overview of skill learning in our method.

The main challenge is that the skill code c_z may be modified while solving each task, while we expect it to still solve prior tasks $(l, s_0, c, s_T) \in \mathcal{E}_z$. We handle this by providing a simple mechanism to allow the user to verify whether the updated skill code still successfully solves prior tasks: we run the task-specific code for each prior task $(l, c) \in \mathcal{E}_z$ with the updated function code c_z. If it doesn't, the user continues to provide feedback, or alternatively gives up, revises the skill, chooses a new skill, etc... This step could be augmented with a VLM-based verifier (e.g. [21] [110]), that serves as a first check.

In practice, this requires some ingenuity on the part of the end-user, by defining sensible task curricula, and not overloading a single skill. The more we expect a skill to generalise, i.e. the more tasks we expect it to solve, and the more distinct these tasks are, the more difficult this becomes.

Due to the quick response times, we view this as a manageable weakness, in line with our aim of shifting trial-and-error from the agent to the user, who can draw on a wealth of world knowledge to effectively interpret and guide the robot behaviours.

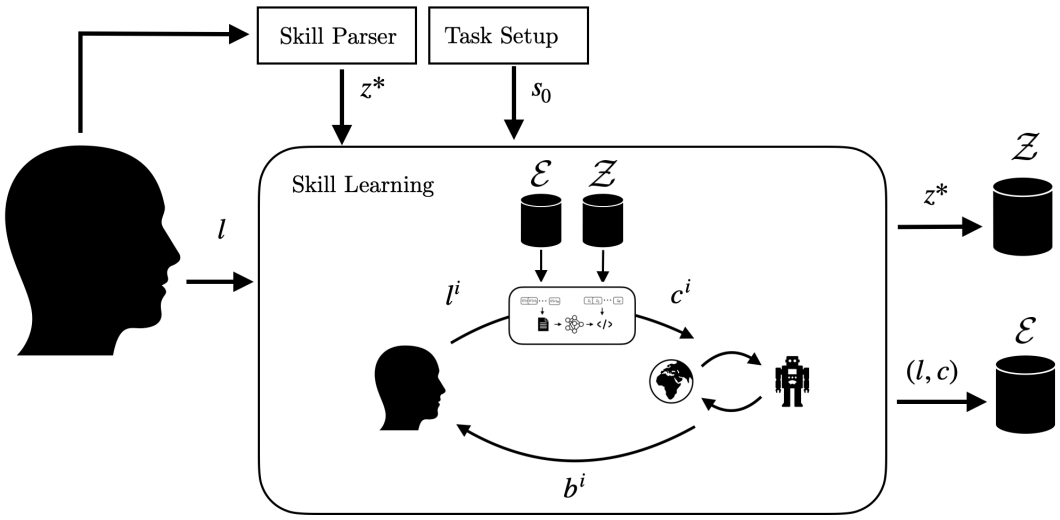


Figure 5.4.: The system diagram for a single iteration of skill learning within our approach. The user specifies skill, initial state and instruction (z^*, l, s_0), and then iteratively refines the agents response by observing the resulting behaviour b^i and providing a correction l^i. Once b^i is aligned with the instruction l, the updated skill z^* and example (l, c) are added to the respective libraries $\mathcal{Z}$ and $\mathcal{E}$.

Algorithm 1 Learning a Skill

Initialize Skill Library $\mathcal{Z}_0$ and Examples $\mathcal{E}_0$
$z^* \leftarrow$ SkillParser(skill_description) ▷ Choose current skill to learn
while True **do**
    $l \leftarrow$ task_description ▷ Provide task instruction
    $s_0 \leftarrow$ TaskSetup(initial_state_description) ▷ Set up the environment
    correction $\leftarrow \varnothing$ ▷ Set initial correction
    $c \leftarrow \varnothing$ ▷ Set initial task-specific code
    while True **do**
        examples $\leftarrow (l'_i, c'_i)_{i=1,\ldots,K} \in \mathcal{E}$ ▷ Retrieval based on l and correction
        $c, c_{z^*} \leftarrow$ Code-as-Policies(l, c, c_{z^*}, correction, examples)
        $b \leftarrow$ Rollout(c) ▷ Roll out policy code
        if b is aligned with l **then**
            $\mathcal{E} = \mathcal{E} \cup (l, c)$ ▷ Add example to library
            break
        end if
        update correction based on b
    end while
end while
Update z^* in $\mathcal{Z}$

5.3. Managing the Context Window

Key to our approach is dynamic management of the context window, to effectively leverage the *in-context learning* abilities of the LLM. As the number of few-shot examples grows, appending all of them to the prompt becomes untenable. Moreover, most of these examples are likely not relevant to the current task: given the task "fold the cloth in half", knowledge related to opening a drawer is likely not relevant, and may oversaturate the prompt [72].

We use two vector databases to store the few-shot examples (l, c) and the skills z, indexed by the embedding of their natural language instruction and their docstring respectively. We use ChromaDB for the vector database, and OpenAI's text-embeddings-3-model to compute the embeddings. When presented with an instruction l', we retrieve the $K = 10$ examples (l, c) with the highest cosine-similarity between l and l', and append them to the prompt. Since we also store the skills in a vector database, we may similarly retrieve these and add their function headers to the prompt.

It is worth noting that we can cripple the agent if we don't perform retrieval of the examples carefully, since the agent has access only to the knowledge we pass it – without examples of basic functions, the agent can not call them.

On the other hand, when an agent has previously solved a similar task, and we successfully retrieve this example, solving this task becomes trivial. It is also through the examples that we steer the agent towards using the learned skills, and thus towards reliably enacting the desired behaviour.

5.3.1. Hints

Our proposed framework enables a simple but powerful mechanism for steering the agent, in line with the previously discussed idea of *learning a shared language* with the agent: hints. The user can draw on this library of known behaviours, by deliberately triggering the retrieval process. For example, the user might specify that a specific previously learned behaviour could help in solving the current task.

This is particularly relevant during skill learning, since the agent is faced with an unfamiliar instruction, and the agent needs to infer what the necessary substeps are (as in LLM-based planning). This becomes more challenging as the number of skills $|\mathcal{Z}|$ and learned behaviours $|\mathcal{E}|$ grows. Most of these skills and behaviours are likely to be irrelevant, and hints provide a mechanism for the user to guide the agent towards the correct specific subset.

When a sub-behaviour has not been learned, (i.e. is not being correctly produced by CaP, and can not be fixed by providing a hint), and the skill that is currently being learned fails because of this, this presents a cue to the user to pause learning of the

current behaviour, and to learn the sub-behaviour first. Again, we rely on the user to correctly interpret this, and to have an understanding of what behaviours the agent is capable of.

5.4. Auxiliary Functions

We require two more functions to make this method work, and to enable an end-to-end interactive behaviour creation experience: **skill parsing**, and **task setup**. We model each as an individual Language Model Program, though realistically both would be better complemented or even replaced by a GUI.

Skill parsing (figure 5.5) refers to the process of inferring which skill the user currently wants to learn. When the user declares the intention to learn a new skill, this involves generating an appropriate function header, parsing appropriate parameters for this function, and laying out the basic expectations of what this function should accomplish. The skill parser also interacts with the skill library, in case the user wants to revise a previously learned skill, and to prevent learning of redundant skills.

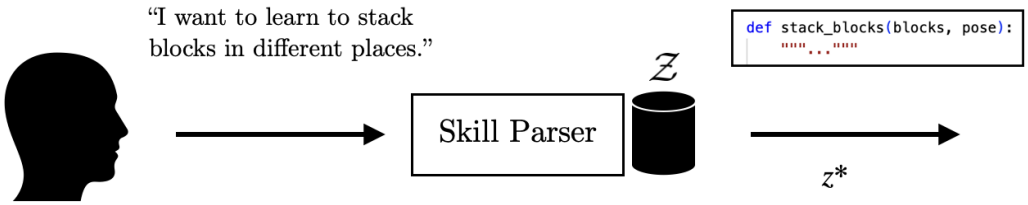


Figure 5.5.: The skill parser converts a natural language description of a skill into the corresponding skill z^*.

The task setup (Figure 5.6) LMP aims to reduce the friction of setting up the environment, and allows a natural-language based environment setup. In our experiments, this typically just means adding a specific number of blocks, with specific colors and sizes to the environment.

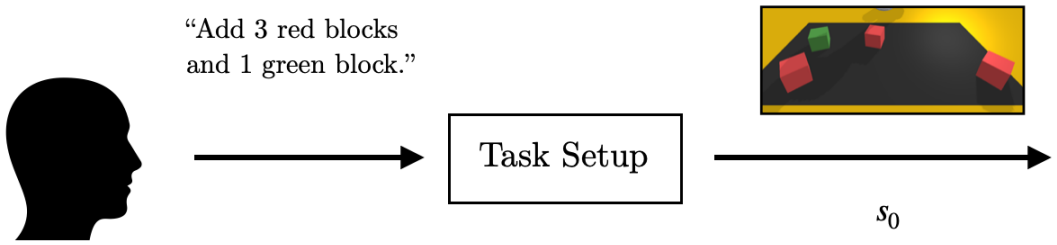


Figure 5.6.: Task setup involves getting the environment into an appropriate initial configuration s_0.

In practice, this module can benefit from the agent itself becoming more capable: as the agent reaches more diverse states s, we can name and store them in a vector database, and use them to initialise further tasks. The simplest instantiation of the task-setup module would be purely GUI-based, where the user drags the relevant objects into the workspace and then issues a task.

5.5. Toy Environment

We test this method in a simple, block-based environment, building on the Ravens benchmark [108] and modified for Code-as-Policies, set in pybullet (Figure 5.7). This environment allows us to test whether we can generate long-horizon behaviours interactively in a natural-language based manner, whether we can effectively instil user knowledge, and whether we can reliably evoke specific behaviours with learned commands.

We make similar assumptions to Code-as-Policies, which relies on ground-truth, oracle knowledge of what objects are in the scene, as well as their positions, orientations, and sizes. We consider this assumption only to be weakly limiting (especially in this block-based environment) as these primitives can be replaced with perception pipelines for the real world (see Sections 3.1 and 3.2).

Table 5.1 lists the basic primitives made available to the agent, prior to user interaction. They comprise the perception primitives, for detecting an objects pose, size, and color, as well as a single high-level control primitive for picking up a block, and placing it at a specified pose.

We provide few-shot examples only to demonstrate the correct use of the API's, and some showing how they can be coordinated together, to provide the simplest and most general API setup.

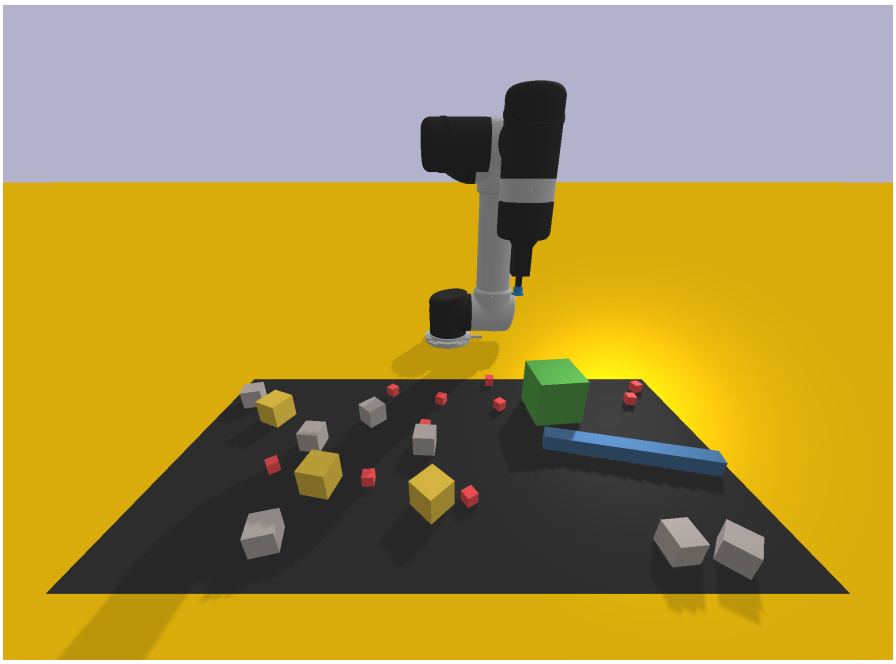


Figure 5.7.: The Toy Environment, with blocks of different sizes and colors, and a 6-DOF robotic arm with a suction gripper.

Name	Function
get_objects	This function gets all objects in the environment. The agent can retrieve specific properties of these objects with the functions below.
get_object_color	Returns the color of the block.
get_object_size	Returns the size of the block.
get_object_pose	Returns the pose of the block, given as a 3-dimensional position vector, and a 4 dimensional quaternion rotation.
get_bbox	Returns the axis-aligned bounding box of an object, to simplify collision queries.
put_first_on_second	The main pick-and-place primitive. It picks up an object at the specified Pose, lifts it vertically to a specified height, moves along the x-y plane to a point directly above the place Pose, then moves it down until it detects contact.
move_end_effector_to	Moves the end effector the specified position, and suction gripper rotation.

Table 5.1.: List of the core-primitives for our agent to build on

As in CaP, we could also decide to include examples of more complex behaviours, like placing blocks relative to one another, or using third party libraries to ease specific computations. We opted not to do this, modelling the assumption that you can't predict in advance every behaviour you need to learn to effectively operate in an environment.

We share our code, as well as prompts and the initial set of examples in Appendix A.

6. Experiments

In this section we describe experiments conducted to validate our approach, but primarily to demonstrate its potential. In the following, we first demonstrate how we learn a single skill with this method, then demonstrate how skills like this can be composed to enable temporally extended behaviours, show how *hints* can be used, along with a qualitative discussion of the benefits our approach offers.

We note here that this discussion is necessarily qualitative at this early stage of integrating Foundation Models with robotics, due to the novel problem-spaces Foundation Models have enabled, and the lack of an existing infrastructure surrounding these ideas. A more thorough validation would require tailored benchmarks and experiments with non-expert users, which we did not have the resources to build/conduct within the scope of this thesis.

6.1. Learning a Skill

In the limited block-manipulation environment, one skill that will be useful for many downstream tasks, and that should be executable reliably, is placing blocks *next* to one another, generalising across different block dimensions, rotations, and directions (visualised in Figure 6.1). Since it involves the precise coordination of multiple perception primitives, this is non-trivial, and our baseline CaP-agent fails it most of the time.

Figure 6.1 shows a simple user-defined curriculum to progressively learn the skill of placing one block next to another block, and doing so in a predictable way across different block sizes, rotations, and directions. By proposing constrained tasks, correcting the agent on each of them, and ensuring that the resulting behaviour aligns with the users expectations, the resulting function encapsulates all of these expectations.

In practice, curricula like the one in Figure 6.1 would likely rarely be as linear. The user might try to build on the skill `move_block_next_to_reference`, only to notice later that it doesn't always behave exactly the way the user expects, and may then return to learning this skill.

For each "test", the user sets up the environment with a prompt (e.g. "add 4 blue blocks and 1 red block"), gives the task (e.g. "put one blue block on each side of the red block"), and then provides corrections to guide the LLM's outputs. Figure 6.2 provides

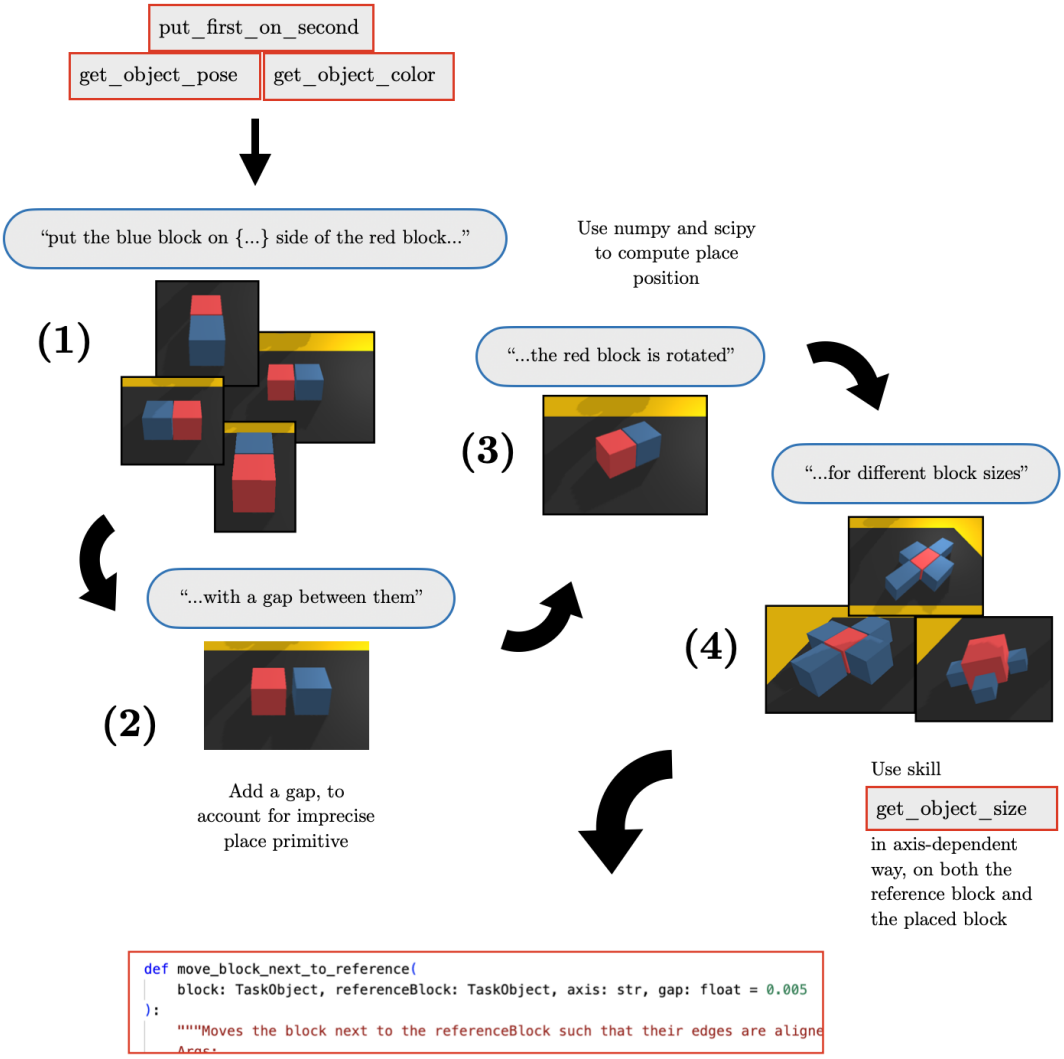


Figure 6.1.: A basic task curriculum to learn the skill of "placing one block next to another block", encapsulating the user's expected behaviour across different axes of variance, elicited via repeated interaction and corrections by the user.

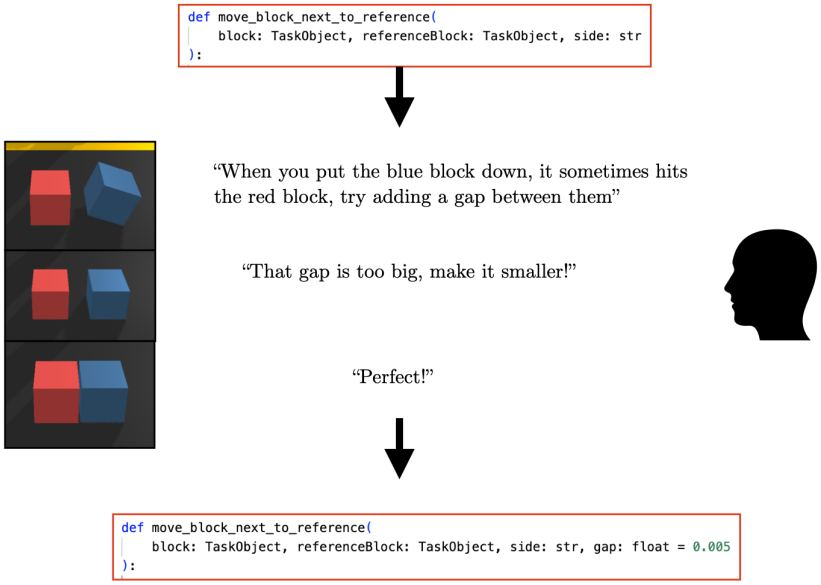


Figure 6.2.: Correcting a faulty behaviour caused by imprecision in the pick-and-place primitive. This fault is simple for a non-expert to identify and correct. The user preference for the appropriate "safe" gap is encoded in the default gap parameter value.

a single instance of this, to show how this curriculum progressively forces the LLM to generate more general code.

For instance, every 10 runs or so, the agent places the block in such a way that it lands on the edge of the reference block, and rolls over, probably because of an imprecision in the pick-and-place primitive. Correcting this fault requires reasoning over multiple runs, interpreting the full trajectory to determine where the fault arises, and then suggesting a sensible correction, all of which would be difficult for any LLM-based autonomous agent to make independently, but are trivial for humans to observe and point out.

We can also justify our use of skills with this simple example, by comparing with a baseline that generates flat function codes for each task. By introducing the skill, we are initially increasing the difficulty of the problem faced by the code-writing agent, because it needs to infer which codes are task-specific, and which are skill-related (i.e. what is flat function code, and what is skill code). However, few-shot examples play a crucial role in steering the LLM towards the correct solution. For instance, if the agent has solved a task $t = (l, s_0)$ previously, we could just retrieve the flat solution code

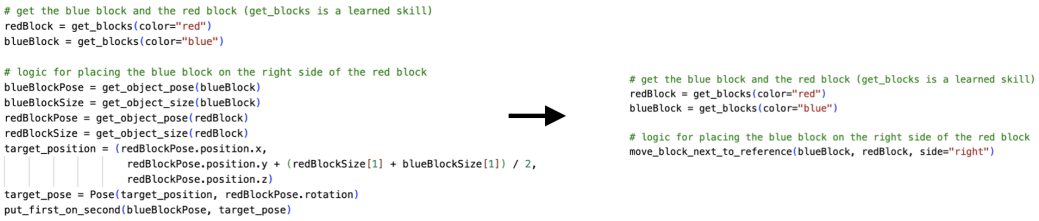


Figure 6.3.: On the left side we generate flat solution codes, on the right we encapsulate the skill logic in a function. When the skill code is rewritten, the flat solution code is implicitly updated too.

$c \in (l, s_0, c, s_T)$. In Figure 6.1, this would mean we would store the codes in step (1) as valid examples for the prompt "put the blue block on the ... side of the red block", **even though these contain failure modes identified later**.

We want to keep these examples (l, s_0, c, s_T) because they contain valuable information about the mapping from a task (l, s_0) to the desired final state s_T. Our approach makes this possible, by abstracting away the logic for `move_block_next_to_reference` in a skill, such that the solution codes that call this function from step (1) remain valid (see Figure 6.3).

Moreover, this ensures that user feedback like the gap parameter, learned later, is integrated every time the user tries to place a block next to another block.

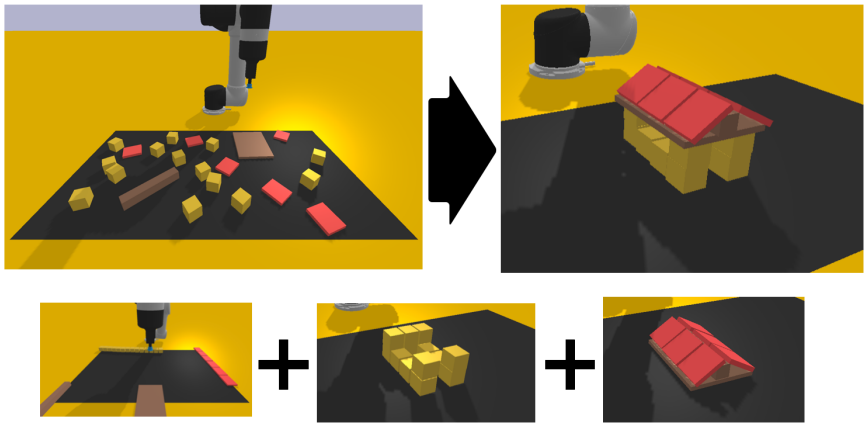
6.2. Learning a Long-Horizon Behaviour

In this section we describe an extended user interaction to enable the long-horizon behaviour of *building a house*, from a set of blocks with different dimensions. This requires allocating the different block types to different parts of the house, and accurately placing them.

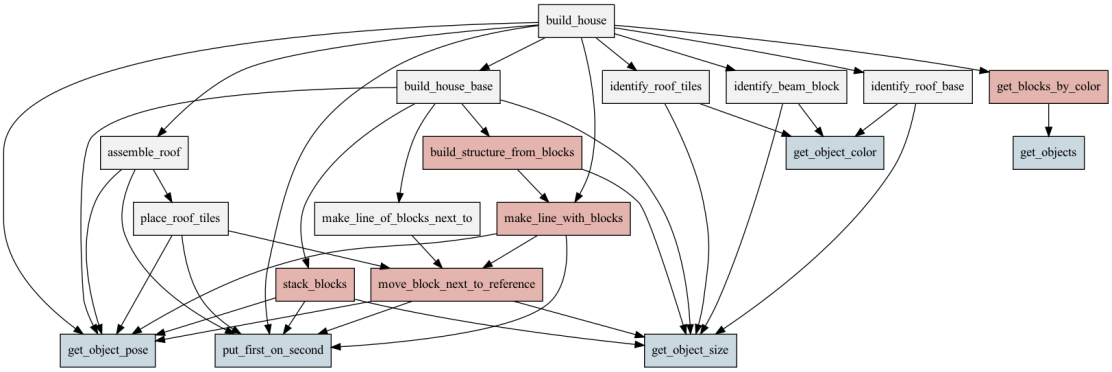
In the CaP paper, the authors state that: "In the tabletop domain, it would be difficult for LMPs to 'build a house with the blocks', since there are no Examples on building complex 3D structures." We believe that even if there were such examples, CaP would not be capable of this, unless those examples were specifically of building houses, and at a sufficiently high level of semantic abstraction to leverage LLM "common sense". We show that CaP is capable of this, provided the right cognitive architecture, and extensive human interaction.

This is made possible by allowing the user to decompose this problem into subtasks, and interactively correcting and refining CaP code generations for each subtask. Figure

6.4 shows the key sub-behaviours for achieving this, as well as the resulting skill-tree.



(a) Initial and final configurations (s_0, s_T), as well as important checkpoints.



(b) The corresponding skill tree. Core primitives are marked in blue, skills learned independently from the task of building the house in red, and skills learned specifically with the aim of building a house in gray.

Figure 6.4.: Example of a long-horizon behaviour "build a house", enabled by extended user interaction.

Only a subset of the skills needed to build the house were actually learned in the context of building the house, while others were learned previously to achieve specific behaviours not included in the initial set of primitives.

6.3. Hints

In this section we demonstrate how we can apply a hint, in order to make sure prior knowledge is incorporated into a new behaviour.

When we try to learn the skill of "lining up blocks next to each other", the agent initially tries to come up with solutions for the task that call the core set of primitives, similar to the ones initially proposed when learning to place blocks next to each other, leading to the same issues. We can incorporate the knowledge encapsulated in `move_block_next_to_reference` by applying the hint "we learned to place blocks next to each other, use that" (Figure 6.5).

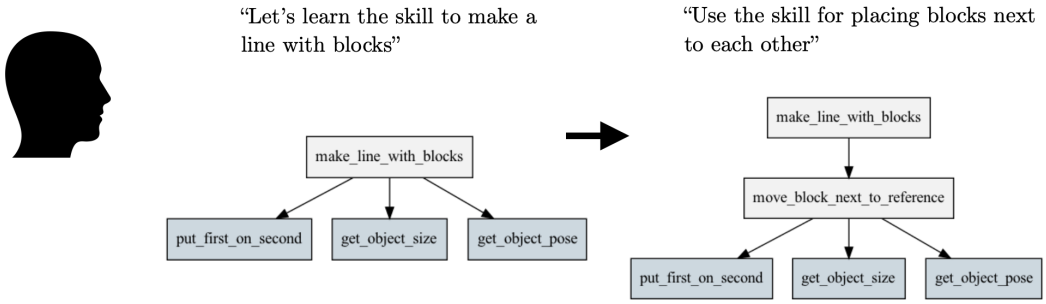


Figure 6.5.: Example of applying a *skill hint*. The skill for making a line with blocks inherits the knowledge instilled while learning how to place blocks next to each other.

In a similar way, we could provide context for the agent by telling it "skill agent is learning is like skill agent has already learned", thus inserting the actual skill code into the agent prompt, though we leave this for future work.

6.4. Advantages of learned skills

A simple alternative approach to our chosen method of learning *skills* (i.e. python functions) is to simply generate flat solution codes and to store and retrieve those. The problem statements are very similar, but learning skills has some clear advantages, discussed in the following.

6.4.1. Encapsulation

Functions explicitly model the assumption that we don't want the agent to alter this behaviour when it uses it. When the agent calls a skill, the concrete implementation is hidden behind this function call. Attaching flat function codes would allow the agent more flexibility, however this also increases the chance for errors. Assuming we've sufficiently tested a specific learned behaviour for alignment with our expectations, this flexibility has no advantage.

This is important, because LLM outputs can be very sensitive to inputs. This is particularly relevant in our scenario of dynamic prompt composition, where it is not just the task text that is varying.

By generating shorter example codes using skills (in which the agent mostly has to correctly set the arguments a skill call), we reduce the number of parameters the agent can vary, thus reducing noise. A larger chunk of the agents behavioural response to any task prompt becomes deterministic. Figures 6.3 and 6.6 provide examples of this.

6.4.2. Interpretability

By introducing these functions, we introduce a single source of truth for the robot behaviours, which would not exist if we stored flat solution codes. This allows us to pinpoint and correct the source of errors more easily, both on the user-end, and on the end of an engineer capable of interpreting the code. This is important particularly in real-world scenarios, where the primitives the agent builds on are not oracle observations, and may also produce invalid responses.

Beyond introducing a single source of truth, the temporal abstraction also leads to more semantically meaningful code. Figure 6.6 demonstrates this.

```python
def make_smiley_face():
    """
    Arranges blocks in the workspace to form a basic smiley face pattern in the middle of the workspace.
    The smiley face consists of eyes, a mouth, and a circle around it.
    Note:
    This function builds the smiley face in a predefined formation in the middle of the workspace,
    ensuring no parameter variance or additional configuration is required.
    """
    # Collect objects and get middle of workspace (omitted for brevity)
    # ...
    # Place the eyes and mouth in a triangular pattern
    place_smiley_face_features(cylinders[0], cylinders[1], mouth_block, smiley_center_pose)
    # Define a smaller radius for the surrounding circle
    radius = 0.15
    # Use remaining blocks (cubic) to form a circular pattern around the smiley face
    arrange_blocks_in_circle(blocks, smiley_center_pose, radius)
```

Figure 6.6.: Skill-based code is more interpretable than flat code, and restricts the parameters that can be varied by the code-writing agent. The corresponding flat code was too long to be included here.

Another advantage is that we can *talk* to the agent about what it is capable of, by interacting with the skill library $\mathcal{Z}$ and example library $\mathcal{E}$. While the actual language instructions that the agent has correctly handled in the past may provide a more intuitive interface, skills represent concrete generalisations of behaviours, leading to a much smaller and more manageable space of options. We leave this for future work.

6.4.3. Defined axes of generalisation

Functions allow us to define the desired axes of generalisation, along which we test and evaluate the LLM, as demonstrated in the block placing example in section 6.1. We believe it makes sense to specify the desired axes of generalisation in reusable behaviours. Figure 6.7 shows another example of how this feature can be used, generalising over the pose and dimensions of a block structure.

```
def build_structure_from_blocks(
    blocks: list[TaskObject],
    dimensions: tuple[int, int, int],
    pose: Pose,
):
```

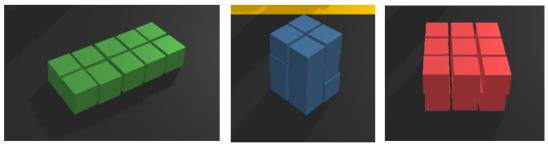


Figure 6.7.: Functions parameters allow us to define axes along which we expect a skill to generalise. This is important to maintain the semantic meaningfulness of skills, and to ensure skills are reusable.

For example, one could imagine a robot that is supposed to scoop ice-cream, with arguments for the number of scoops and the flavour it's supposed to scoop, given the necessary constituent skills e.g. for navigating to the bucket, scooping, placing the scoop in the cup, and handing the cup to the customer.

6.4.4. Targeted and deliberate integration of preferences

An important consideration for robot behaviour in everyday, general scenarios, is the integration of user preferences. Prior works [109] [97] have approached this problem by directly storing these preferences, and retrieving them in a task-related manner. In our method, these preferences can be integrated in a deterministic manner, and attached to specific behaviours i.e. skills. Figure 6.2 provides one example of this, with the gap parameter, which was tuned by user interaction. Another somewhat contrived example for this is provided in figure 6.8.

```python
def is_big_red_block(block: TaskObject) -> bool:
    """Function to verify whether a block is 'big red block'.
    A big red block has side lengths longer than 5cm"""
    return get_object_color(block) == "red" and min(get_object_size(block)) >= 0.05

def pick_and_place_big_red_block(
        block: TaskObject,
        place_pose: Pose):
    """
    Function to pick and place a 'big red block'.
    Big red blocks should always be picked up on one end of the block.
    """
```

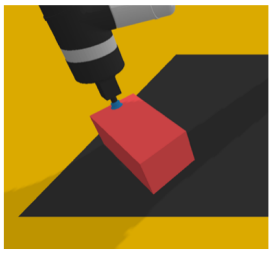


Figure 6.8.: Assume we want a CaP-based agent to always pick up big red blocks on one side of the block, rather than in the center. Our method introduces a mechanism for iteratively refining how to do this, and then reusing this behaviour only when desired.

6.4.5. Preconditions

Functions also allow us to explicitly model preconditions, similar to the ones used in Task-and-Motion Planning methods, for example by introducing error handling (figure 6.9). These error messages can be traced back to a specific method in the call stack, and other functions can catch these errors as appropriate. Alternatively, the error can simply be relayed to the user, to let them determine how to respond.

```python
def pick_and_place_big_red_block(
        block: TaskObject,
        place_pose: Pose):
    """..."""

    if not is_big_red_block(block):
        raise Exception("The block isn't a big red block.")
```

```python
def build_structure_from_blocks(
    blocks: list[TaskObject],
    dimensions: tuple[int, int, int],
    pose: Pose,
):
    """..."""

    if len(blocks) < dimensions[0] * dimensions[1] * dimensions[2]:
        say("There's not enough blocks!")
        return
```

Figure 6.9.: Examples of how we could add preconditions to skills.

6.4.6. Continual Learning

Prior works have pointed out that prompts quickly oversaturate when attaching too many few-shot examples [47], and that this constrains the number of primitives that can be made available to the agent. By retrieving these examples in a task-specific manner, we alleviate this restriction, at least in theory allowing an effectively unlimited number of reliable mappings from natural language instruction to behaviour. This is the quality we referred to previously as *learning a shared language*. Figure 6.10 gives some examples of instructions we *taught* our agent in the blocks domain, inspired by LohoRavens [110].

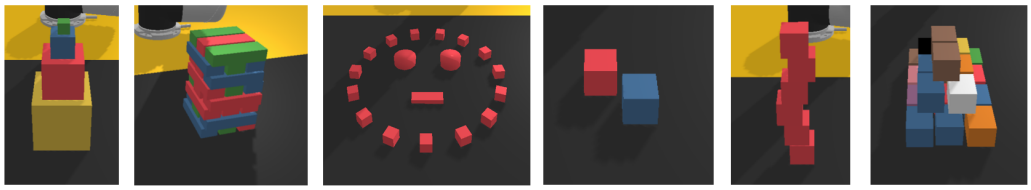


Figure 6.10.: Different learned behaviours, which are now reliably executable with concise prompts, since we taught the agent what we mean by them. From left to right: "stack the blocks (from biggest to smallest)", "make a smiley face", "build a jenga tower", "place (blue block) diagonally to the (front-right) of (red block)", "build a zig-zag tower", "build a block pyramid".

Our method also enables a simple integration of new API methods, again, importantly without requiring any prompt engineering. Figure 6.11 provides an example of this.

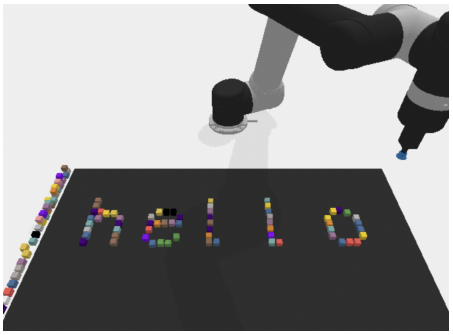


Figure 6.11.: The API can be extended without any prompt engineering. The engineer provides a new skill, and the user can test its applicability via trial-and-error interaction in the environment.

6.5. Challenges

There are many challenges with this setup, that in its current state hurt its applicability. Significant tuning would have to be done to enable effective black-box interaction with non-expert users, though we believe this thesis demonstrates a step in that direction, and can in principle already be tested in this context. As the method gets stronger, by integrating with the many other complementary methods within the Foundation Model Agents literature (see Section 7.3.1), the LLM becomes a more capable translator, and we believe the amount of human trial-and-error required to learn new behaviours will decrease.

In the following we describe the various failure modes and challenges of our method, and the context surrounding it.

6.5.1. Limited ability to respond to feedback

Although the LLM enables natural language interaction, feedback can't always be integrated effectively. We tried many things to enhance this ability (including introducing skills), but overall this ability is quite brittle, and it is often not clear what exactly is going wrong. One solution to this, is to define many skills, such that the LLM generates very short codes, and that makes it more capable of incorporating feedback, but this makes the entire process somewhat more laborious.

6.5.2. Writing code

We view the strength of our approach in relegating the LLM to the role of translator, translating language instructions into executable python code. The more complex the instruction, and the more it is removed from the existing skills that it is building on, the more difficult this problem becomes, and the more room for error there is. While syntax errors are relatively rare, and mostly related to calling functions for which it doesn't have sufficient few-shot examples in the current prompt, we describe some other common issues in the following.

Code Rewriting Loops

When learning a skill over multiple tasks, we adapt the same code multiple times. We prompt the LLM to take this into account, but of course it often ignores this. While we can check if the updated skill code still successfully solves the prior tasks on which it was trained (not all tasks that use it), there is currently no better mechanism than just telling the LLM "now you're not solving {task} anymore, {this is what's going wrong}", leading to code rewriting loops. This is primarily an issue when the skill being learned requires more complex logic. This can be circumvented somewhat by learning more constrained skills, however we may also argue that such functions could simply be filled in by an engineer capable of doing so.

Updating and Setting Skill Parameters

The Skill Parser is supposed to extract the relevant parameters for the skill proposed by the user, and as such serves a crucial function. However, it will sometimes infer strange function arguments, which can be corrected with user interaction, but work would need to be done to make this feasible with non-expert users.

Moreover, it is difficult to change the function parameters after having learned a skill unless we also provide default values, since otherwise all examples and skills that call this skill will have to be rewritten too. Although in our toy environments and tasks, and under the supervision of a developer, this was not too difficult to avoid, it would require specific mechanisms to handle this.

Invalid or Unnecessary Assumptions

Another problem we encountered was that the agent made invalid or unnecessary assumptions. Some could be teased out by continued interaction. Some assumptions are related specifically to code design choices, that the agent then needs to reliably follow. For instance, many skills get passed a list of blocks to use - in some functions, the agent chooses to remove these blocks from the input list. Even if the agent makes a note of this, this can lead to errors that are difficult to interpret. We might observe over time that the mistake always happens when a specific skill is called, so the user could just try relearning it, or an engineer could be involved. Alternatively, introducing mechanisms for the agent to explain its reasoning could enable a kind of natural language debugging, which might be possible if skills are kept sufficiently concise. In a similar but less harmful way, LLMs will write reasonable but ultimately meaningless code (e.g. `center_of_workspace.translate((0, 0, 0))`).

Separation of Task and Skill Code

Related to this, another difficulty we introduce with our separation between task and skill code, is that it is not always obvious where this separation lies. In some cases, task-specific code, like the location at which a block should be placed or a structure should be built, was included in the function code, in spite of being a parameter. Again, this can be teased out by sufficiently testing the skill with different tasks, but it increases the difficulty of the problem faced by the agent. We could try to address this by introducing another reasoning step, focused purely on correctly establishing this separation, or simply by generating both codes separately.

Poor Design Choices

A problem we encountered specifically in our implementation, was that we introduced custom types, which turned out to hamper the performance of our agent, since it made it more difficult for the LLM to effectively leverage pretraining knowledge about third-party libraries like numpy. For example, we introduced custom types for `Point3D` and `Pose`, with the intention of allowing parameter type annotations, but these would occasionally cause errors when the LLM interacted with them incorrectly.

6.5.3. Retrieval

Dynamic prompt composition also comes with some drawbacks. The agent simply doesn't know about functions that don't get included in the prompt, which can severely cripple the agent. With hints (Section 6.3), we have provided a simple and effective mechanism to remedy this, though this requires an understanding of the agents capabilities, or simply more trial-and-error. Alternatively we could employ top-down planning approaches, as discussed later in Section 7.3.

We note that this is primarily an issue during skill learning, since the agent is dealing with unfamiliar prompts. At test-time, we only expect the agent to be able to respond to instructions similar to ones it was previously trained on. The natural language capabilities of LLMs ensure that familiar tasks can mostly be retrieved successfully, though it is unclear whether this will remain the case if we expect the agent to perform a much larger number of behaviours.

6.5.4. User experience

Our method is, at its core, a natural language interface for robot programming. However, the interface we created is still fairly laborious in its interaction, and there are many simple friction factors that could have been reduced to create a more seamless experience. For example, if we are learning a skill, and then realise we need to learn another subskill first, in our current setup this involves breaking off learning of the current skill, initiating learning of the subskill, along with the corresponding task setup, then returning to the skill we were initially learning and applying a skill hint to indicate that the subskill should be used. This is primarily a UI issue, which could be fixed simply by introducing a mechanism for it. Many functions in our method would be complemented well with a GUI, discussed further in Section 7.3. As it stands, this is a significant limitation in truly leveraging human input effectively.

6.5.5. Environment

While we present a broad idea, we only test it in a very constrained environment, and one not necessarily well suited for leveraging human intuition. The Ravens benchmark [108] was developed for planar pick-and-place, meaning end-effector motions were either vertical (along the z-axis), or along the x-y plane at a specified height over the workspace, where the exact path the end-effector takes to go from point A to point B doesn't matter much.

We were incapable of eliciting some more interesting and conceptually simple behaviours, since the controller used in this environment is based on a simple way-point-based wrapper around pybullets built-in Inverse Kinematics (IK). For example, with a

slightly modified API (that enables picking up an object without dropping it), it would have been straightforward to demonstrate a robot playing jenga, but the IK-based solution was incapable of generating straight-line end-effector motions in cartesian space. Similarly, we were unable to reliably rotate the end-effector when the axis was not purely the z-axis, although this too could have enabled interesting behaviours. As such, we were limited mostly to planar pick-and-place tasks, like the ones demonstrated in the initial Transporter networks paper. This demonstrates how the effective API-design determines the downstream behaviours that can be learned.

More sophisticated controllers, learning-based methods, or motion-planning algorithms could have made this possible, but this remained outside of the scope of this thesis.

Beyond these weaknesses, we are of course also limited by operating in a purely block-based environment. It leaves much open with regard to demonstrating the potential of our method. Implementing such a robot API effectively requires some expertise, distinct from the knowledge required to build the layer of Foundation Model based factorised perception-and-control algorithms you could build on top of such an API.

7. Discussion

We demonstrated that it is possible to learn new *skills* via user interaction, and described the advantages that this promises. However, to really motivate this idea, much work is left to be done. In the following, we provide a brief discussion of our work, followed by suggestions for extending it.

7.1. Novelty

The primary novelty of this thesis, is that we propose a method that allows Foundation Model agents to *learn* from user interaction, tailored to the context of robotics, where we have argued it is particularly relevant. This provides a step into the direction of robots that are endowed with relatively basic capabilities, but that can easily be adapted to the affordances and demands of specific environments. We view python as a reasonable choice for this high-level functionality, particularly for rapid prototyping.

In our method, we reduce the burden on the LLM as an agent, and treat it rather as a translator, that slowly enables the generation of personalised robotic behaviour via a learned shared language between robot and human. The robot progressively becomes capable of responding to natural language prompts the way the human expects, through iterative and interactive refinement.

We propose skills as an effective mechanism for instilling this knowledge, and demonstrate how we could leverage modern robotics simulators to learn such skills. Our method for dynamic prompt composition based on Retrieval-augmented Generation allows for the integration of a larger number of distinct primitives and behaviours. A core strength of methods that build on code generation to produce robotic behaviour is modularity, and we provide a controlled mechanism for organising this modularity.

Moreover, similar to VOYAGER [89], we demonstrate that classical ideas from Reinforcement Learning, like curriculum learning and hierarchical learning have an important role to play in the context of LLM-based code generation, in that we may employ them to slowly steer the LLM to the desired outputs.

The most interesting marker of how successful a method is, is what it makes possible. We are confident that our approach extends the capabilities of Code-as-Policies by leveraging human interaction, and that this difference would become more pronounced

as we add more interesting control and perception primitives, tested in more interesting environments.

7.2. Limitations

The primary limitation of this thesis is that we were unable to provide a truly convincing demonstration of our approach. We were able to demonstrate that it is possible, and justified why this is useful in principle, but we were unable to provide convincing examples of this.

This was in part due to the restricted set of examples and skills we started out with, with which we unnecessarily hampered our agent. In retrospect, we should have introduced more functions (including LMPs) from the get-go, and simply demonstrated how you can recombine them, sequence them, and refine them to generate novel behaviours, that we can evoke reliably, and tested what happens when the number of behaviours gets large. While we showed that we can grow the space of behaviours from a small starting set, we believe that the results would have demonstrated the power of our approach more meaningfully if we had expanded this starting set.

Moreover, we were limited to a mostly qualitative evaluation of our results. This is a general consequence of the nascence of this area of research, and the wide variety of methods, each suited to their specific problem statement. There was a large number of possible alternative design choices we could have made in implementing our method, many of which we weeded out by trial-and-error. For example, we considered a variant in which we generate flat, task-specific codes while learning a skill, and later use an LLM to separate the task- and skill-specific logic, but found this to be unsuccessful. We also tried an approach similar to CodeChain [44], in which we prompt the LLM to write modular code to solve each task, and then later try to cluster the generated functions to filter out the reusable behaviours. Each of these would have allowed for a more intuitive user-interaction, removing the step of explicitly specifying the skill we want to learn. We found neither of them to work, though this may simply be because we didn't *try hard enough*. We discuss this further in Section 7.2.1.

Another key limitation of our approach is that the UI is currently quite unwieldy. It is possible to elicit the desired behaviours via human-in-the-loop trial-and-error, but it is often not a pleasant experience, though this could likely also be improved with a richer set of primitives. Ultimately, the primary way to improve this experience is to provide more specific and powerful user interactions, coupled with an expressive and understandable GUI, as discussed in Sections 7.3.1 and 7.3.4.

7.2.1. Evaluation

How could we evaluate a method like this? Prior works (e.g. DROC [109], LMPC [48]) similar to ours have focused on the number of corrections required to get to the desired results. Effectively assessing this would require studies with non-expert users, and likely a GUI (see Section 7.3.4) to report in a meaningful way, as well as standardised and well-tuned implementations of baseline algorithms (e.g. with the same set of primitives).

In DROC, the authors report results collected on a real-world robot, meaning that to replicate their results, researchers would have to set up similar manually created task-sets, then interact with the system over an extended period, providing corrections. This makes replicating results cumbersome.

Similarly, in Language2Reward [103], the authors compare with Code-as-Policies in a continuous control setting (e.g. to control a quadruped), and thus need to choose a viable set of primitives for CaP. This choice determines what CaP is capable of, and a more extensive set of primitives might have allowed it to perform better, making this a weak comparison. Moreover, although their chosen reward-function based API can generate more diverse behaviours, these behaviours don't fulfil basic expectations (like motion smoothness, or only using specific actuators when they are actually needed).

ChatbotArena [15] offers some inspiration, directly computing performance scores based on human preferences. A simple way to translate this to the robotics domain would be to record prompt-behaviour pairs, and have humans evaluate those. We believe that this kind of human-centric evaluation will become the norm as the space of behaviours that robots are capable of grows.

7.3. Future Work

7.3.1. More advanced cognitive architectures

We proposed a specific cognitive architecture to enable user-centric skill learning in Foundation Model agents, taking advantage of robotics simulators. The research area that studies such Foundation Model agents is growing explosively, and is being explored in a wide variety of contexts. We view our approach as complementary to many of these methods.

For instance, a particularly common application of LLMs is to act as *Planners*, to decompose a task into its subtasks in natural language. While we technically do this by expressing the plan in code, the skills and examples that are available to the agent need to be retrieved dynamically. For example, during skill learning, the agent could autonomously propose relevant subtasks, and thus autonomously retrieve examples

accordingly, possibly alleviating the need for user-provided *hints*, and more quickly providing better responses.

When evaluating whether the updated skill code still solves prior tasks as expected, semantic verifiers which use a VLM [21] [110] to validate task success could perform the initial check. In our blocks environment, we experimented with position-based equality checks, similar to the ones used in the original Ravens benchmark environment, however even here these were often too strict.

An important function similarly omitted in this thesis, is a simple mechanism for translating plans expressed as code back into natural language. This would provide a stronger basis for a non-expert user to provide corrections, and would be simple to implement, since the LLM already documents its reasoning steps in comments in the code.

Moreover, skills are somewhat overloaded in our method right now, serving as the universal mechanism with which a user can break a task apart into substeps. Often, this will lead to overly fragmented skills, where we would prefer not to semantically separate subtasks, because we will not use them separately later on. However, as the number of subtasks we squeeze into a single prompt increases, the agent becomes less capable of responding to it, meaning the only choice in our current implementation is to separate them into skills. This is a weakness of how we currently implement our method, and could be corrected simply by adding more features to the UI. For instance, a powerful addition would be to allow the user to learn a skill subtask by subtask, focusing LLM code generation on smaller windows of the code.

In a similar way, the user should be able to step backwards if the LLM response got worse in response to feedback. Since LLM responses are noisy, another useful function would be to simultaneously generate multiple responses to the feedback, allowing the user to choose the best one (similar to the evolutionary approach adopted in EUREKA [52]).

7.3.2. Richer primitives

A simple but important modification to our method would be to replace the oracle perception modules with something that would actually be executable in the real-world. For example, a common pipeline for understanding a scene visually is to use an open-world segmentation model, followed by VLM labelling of segmentation masks, and augmented by point-cloud data from a depth camera to provide accurate positional sensing.

Similarly, it would be crucial to enable more robust and expressive robot motion primitives. This could be achieved for example by integrating more powerful control methods (e.g. Operational Space Control [38] or Model Predictive Control [42]). MINK

[105] offers a differential inverse kinematics controller tightly integrated with the powerful physics engine Mujoco, similarly capable of performing more interesting motions. Alternatively, we could rely on traditional trajectory planning approaches.

Integrating with richer control methods also poses an avenue for interacting with richer embodiments, as demonstrated in Language-to-Reward [103]. Building on a fast implementation of Model Predictive Control (also Mujoco) [25] and providing a simple robot API based on reward functions (e.g. `set_torso_height(0.5)`), the authors demonstrate quadruped control from natural language instructions. Although the authors compare with Code-as-Policies, we view these approaches as complementary. Some problems are more suited to symbolic logic, and some to optimal control. Importantly, they do exclude each other when we are restricted to relatively small APIs, as is the case with hand-engineered and fixed prompts.

Following a slightly different direction, we can also introduce Language Model Programs for specific subproblems, as put forth in Code-as-Policies and discussed in Section 3.2, and integrating and coordinating many of these provides an interesting direction. These could be added as primitives, or they could be learned in the same way that we learned skills (explored in PromptBook [4]): instead of a skill being represented as a python function, we could represent it as an LMP - the user defines its purpose, iteratively adds preferences to it, and collects examples, and leaves it up to the LLM to produce the correct program code given the current instruction. The tradeoff is that each LMP introduces further latency.

7.3.3. Tailored Environments

To truly test the capabilities of methods like the one presented in this thesis, there is a need for rich environments, tailored to the context of Foundation Model agents, in which agents build on an existing set of perception and control primitives. Such an environment would ideally come with predefined and standardised sets of primitives for different embodiments, along with a straightforward mechanism for setting up tasks. We briefly mention some existing benchmark environments that would likely have made for more interesting experiments.

RLBench [33] provides a large set of more semantically meaningful tasks, and in VoxPoser [28] the authors set up a Code-as-Policies API tailored to their method. Robosuite [113] directly integrates with more powerful controllers, and provides an interface for teleoperation I/O devices for user interaction, like a keyboard or space mouse, which could be used to learn low-level skills (demonstrated in [55] [63]). Most relevant is Orbit [57], for its inherent focus on modularity, on supporting a large variety of robot learning frameworks (RL, TAMP, LfD), and providing a GUI for scene generation.

7.3.4. User interface

While we presented a text-based interface, many aspects of our approach would be better suited for a GUI. For instance, while text-based retrieval of skills when applying *skill hints* (Section 6.3) is likely a good choice anyway, coupling this with a GUI, and visualising skill trees generated during a specific behaviour would further enhance interpretability. We may imagine a simple drag-and-drop based interface for building behaviours, allowing users to attach feedback to specific parts of this behaviour, with the LLM acting primarily as the glue that combines the skills into executable code that reflects the users intentions.

For every skill, we define a number of skill tests, that verify that this skill behaves as we expected, and represent what the skill does in a visual and semantically meaningful way. As we try to solve new tasks, a simultaneous visualisation of how the updated skill-code performs on the prior tests would also be helpful.

Moreover, the chat-based interaction is quite unwieldy, and simply adding buttons would be helpful. For instance, at every chat-turn, the user has to decide whether the task was solved successfully, whether to provide feedback, whether to re-run the behaviour to verify whether it was successful, to simply try to generate the solution code again, or whether to give up on the current skill. Another viable function would be to explicitly attach a precondition (Section 6.4.5), and there are likely many more such functions we could identify. If we were unable to learn a skill, it would be useful to keep track of this, either to try again later, or to notify an engineer to take a look.

Similarly, as mentioned previously, we setup tasks via a simple LMP to add objects to the environment, though imbuing this LMP with a stronger API would lead to a more seamless experience. In principle, this LMP could learn to follow user instructions in a similar manner that the agent can. For example, we trained our agent to build a jenga tower. By storing this configuration, we enable interesting downstream tasks (inspired by MetaWorld) like put-block-in-fixture, or remove-block-from-fixture (i.e. play jenga), without having to rebuild the tower every time, without having had to write task setup code ourselves. Again, presenting those configurations in a GUI would make this interaction more seamless.

There are some existing examples of GUIs for chat-based behaviour creation with Large Language Models [55] [36] [103], visualised in Figure 7.1. We view our work as complementary to each of these, in that we proposed a cognitive architecture to empower the chat-based interaction, rather than on the creation of a scalable end-user UI.

We believe such a UI to complement the fundamentally natural language interaction is crucial to further reduce the friction in the human-robot interaction.

Alternatively, of course, reality presents the simplest environment for allowing a

person to set up different tasks - you simply place the objects, and start learning. On the other hand, this precludes the simple resets we take advantage of heavily while iterating through robot code, as well as replay of prior solved tasks $\mathcal{E}_z$. Ultimately, we think that our method is better suited for learning in simulation, with the promise of simpler real-world deployment.

7.3.5. Finetuning

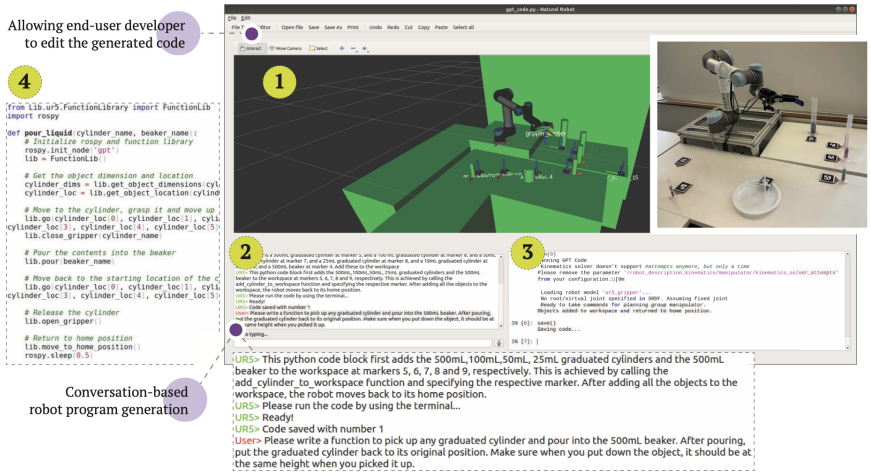
Finetuning represents a simple mechanism for refining the agents responses to user prompts. In this thesis, we relied on the GPT-4 API, which provides a strong general-purpose model, though it is possible that we could improve performance by finetuning the agent on code that follows specific style guidelines, uses useful third-party APIs, or even on successful user-generated codes. We think this provides a promising avenue, since we aimed to force the LLM into a constrained problem space, in which it generates code that represents user intent as concisely as possible. ProgPrompt [75] provided suggestions for achieving something similar via In-Context Learning.

In Language Model Predictive Control [48], the authors propose a finetuning strategy directly in the context of better responding to feedback from non-expert users. They demonstrate that finetuning can be used to enable the LLM to respond more effectively to the user at each interaction turn, i.e. learning how to respond better to natural language feedback. Interestingly, the authors also point out that some users were simply *better* at interacting with their system, i.e. more capable of providing the right instructions to elicit the desired behaviours.

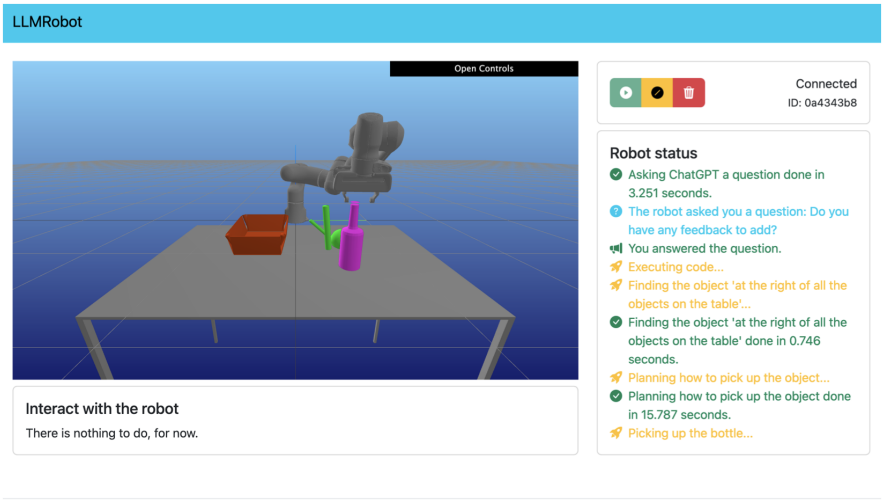
7.3.6. Experiments with non-expert users

A central hypothesis of this thesis is that non-expert humans are capable of correcting robot behaviour, inspired by prior works [109] [103] [48] [97]. We did not get to actually testing this hypothesis ourselves, and this would be a worthwhile future goal.

Since Foundation Models are relatively recent, only a relatively small body of research has focused on this problem, and much remains to be learned and explored about how humans would interact with robots, given the opportunity to talk to them.



(a) Alchemist [36] UI to enable natural-language based robot program authoring.



(b) HaLP 2.0 [55] UI for lifelong robot learning powered by non-expert users.

Figure 7.1.: Two examples of GUI's to enable robot programming by non-expert users, each based on using LLMs for code generation.

8. Conclusion

Foundation Models have driven a fundamental shift in robotics research. Recent approaches, based on a simple, fixed set of core perception and control functions orchestrated by an LLM have enabled robots to respond to natural language instructions, without any further training, accomplishing a wide variety of tasks that would have previously been largely unthinkable without heavily tailored solutions.

While these models encode an abundance of world knowledge that was previously elusive to robotics methods, they were trained without embodiments (with the exception of VLA models), and as such lack an understanding of fundamental concepts for interacting in the physical world. Humans are exceptionally proficient at operating in the physical world, particularly in many of the simple domains we expect robots to operate. With Large Language Models serving as the translator between human and robot, there is now a cheap, universal, and powerful channel for us to communicate this knowledge.

In this thesis, we have proposed a novel method for tightly integrating humans into the robot training pipeline, taking advantage of our ability to both effectively evaluate erroneous behaviours, and to provide corrective feedback. While previous methods have demonstrated that this is possible, actually *learning* from this interaction in a robotics context is as of yet an under-explored research topic.

We proposed a conceptually simple approach for learning based on Retrieval-Augmented Generation, allowing a human operator to teach the robot *skills*. We have justified this approach, demonstrated its merits, provided insight into its weaknesses and limitations, and discussed a variety of directions for future work that we deem promising.

A. Code

We share our code at `https://github.com/maxf98/cap_options`.

We use ChromaDB for the vector database, OpenAI text-embeddings-3-small for embeddings, and GPT-4o for all LLM calls. You will need to add an OpenAI API-key to the environment to interact with the agent.

A.1. Initial set of Examples

Listing A.1: Initial set of few-shot examples the agent receives

```
#TASK: put one block on top of another block
objects = get_objects()
blocks = [block for block in objects if block.objectType == "block"]
pose0 = get_object_pose(blocks[0])
pose1 = get_object_pose(blocks[1])
put_first_on_second(pose0, pose1)

#TASK: put the red block in the middle of the workspace
objects = get_objects()
red_block = next(block for block in objects if block.objectType == "block" and block.color == "red")
middle_pose = Pose(position=Workspace.middle, rotation=Rotation.identity())
put_first_on_second(red_block, middle_pose)

#TASK: rotate the blue block by 45 degrees
objects = get_objects()
red_block = next(block for block in objects if block.objectType == "block" and block.color == "blue")
red_block_pose = get_object_pose(red_block)
rotated_pose = Pose(red_block_pose.position, red_block_pose.rotation * Rotation.from_euler('z', 90, degrees=True))
put_first_on_second(red_block_pose, rotated_pose)

#TASK: move the smallest block 10cm to the left
objects = get_objects()
smallest_block = min(objects, key=lambda x: x.size[0])
smallest_block_pose = get_object_pose(smallest_block)
translated_pose = Pose(smallest_block_pose.position.translate(Point3D(0, -0.1, 0)), smallest_block_pose.rotation)
put_first_on_second(smallest_block_pose, translated_pose)

#TASK: move the end effector to the middle of the workspace
move_end_effector_to(Pose(Workspace.middle))
```

A.2. Prompts

We use python format strings for our prompts.

Listing A.2: Main Actor prompts

```
actor_system_prompt = f"""
You write python code to control a robotic arm in a simulated environment, building on an existing API.

You will be given:
- a task for the robotic agent to solve
- api functions you may use to solve the task
- if available, examples of codes that solve prior similar tasks

You are supposed to write flat code to solve the task, i.e. do not write any functions.
DO NOT make any imports.

Adhere to the following basic types:
{get_core_types_text()}
"""

def actor_prompt(task, few_shot_examples: list[TaskExample], api: list[Skill]):
    return f"""
    {get_few_shot_examples_string(few_shot_examples)}

    {get_skill_string(api)}
    The task is: {task}

    Write flat code to solve the task.
    """

def actor_iteration_prompt(feedback, examples: list[TaskExample] = []):
    return f"""
    Rewrite the previous code to integrate the feedback: {feedback}.
    {get_few_shot_examples_string(examples)}
    Only make changes that take into account this feedback.
    """
```

Listing A.3: Skill Learning prompts

```
actor_skill_learning_system_prompt = f"""
You write python code to control a robotic arm in a simulated environment, building on an existing API.
We are trying to learn skills, and are using different tasks to test and effectively learn a specific skill.

You will be given:
- a task for the robotic agent to solve
- the skill you are supposed to use to solve the task

You are supposed to complete the function, as well as flat, task-specific code, as follows:

def given_function(...) -> ...:
    \"\"\" ... \"\"\"
    <function code>

<task-specific code>

For example:
-------------------------------------------------
IN:
task: "put the red block on the green block"
skill:
def put_block_on_other_block(block: TaskObject, otherBlock: TaskObject):
    \"\"\" places the block on top of otherBlock \"\"\"
    pass

OUT:
def put_block_on_other_block(block: TaskObject, otherBlock: TaskObject):
    \"\"\" places the block on top of otherBlock \"\"\"
    put_first_on_second(get_object_pose(block), get_object_pose(otherBlock))

red_block = get_block(color="red")
green_block = get_block(color="green")
put_block_on_other_block(red_block, green_block)
-------------------------------------------------

If the new task requires you to rewrite the function header, you may do so, for example to add arguments, or to update the
      docstring with important usage information.
You should try to preserve the previous functionality though, since the function might have previously been used to solve other
      tasks, which should remain solvable after changes.
```

```
DO NOT make any imports.
DO NOT write any functions other than the given one.

Adhere to the following basic types:
{get_core_types_text()}

"""


def skill_learning_prompt(
    task,
    few_shot_examples: list[TaskExample],
    skill: Skill,
    other_useful_skills: list[Skill],
):
    return f"""
    The task is: {task}
    The function you are supposed to implement is:

    {str(skill)}

    ----------------------------------------------------------------
    {get_few_shot_examples_string(few_shot_examples)}

    The following skills may be useful in your implementation:
    {"\n\n".join([skill.description for skill in other_useful_skills])}
    ----------------------------------------------------------------
    Implement the function and solve the task, while trying to ensure that prior tasks remain solvable.
    """
```

Listing A.4: Task Setup prompts

```
task_setup_api_string = """
def add_block(
        self,
        env: Environment,
        color=None,
        size: tuple[float, float, float] = (0.04, 0.04, 0.04),
        pose: Pose=None
    ):
\"\"\" adds a block of a given size and color to the environment
If the pose is left unspecified, a random collision-free pose is selected
 \"\"\"


def add_zone(
        self,
        env: Environment,
        color: str,
        scale: float = 1,
        pose: Pose = None
    ):
\"\"\" adds a zone of a given size and color to the environment
If the pose is left unspecified, a random pose in the workspace is selected
 \"\"\"


def add_cylinder(self, env: Environment, color: str = "red", scale: float = 0.5):
\"\"\" adds a cylinder of a given scale and color to the environment \"\"\"
"""

task_setup_system_prompt = f"""
You are writing python code to setup a simulated environment, translating user instructions into executable code, based on an
      existing API.

You should adhere to the following types:
{get_core_types_text()}

You may use the following API:
{task_setup_api_string}
```

```
EXAMPLES:
#########

task: add 3 red blocks and 3 blue blocks
response:
for _ in range(3):
    self.add_block(env, "red")

for _ in range(3):
    self.add_block(env, "blue")

#########

task: add one big block and 4 blocks that are a quarter of the big blocks side length
response:
self.add_block(env, size=(0.08, 0.08, 0.08))
for _ in range(4):
    self.add_block(env, size=(0.02, 0.02, 0.02))

#########
"""
```

Listing A.5: Skill Parser prompts

```
generate_function_header_system_prompt = f"""
We are working in the context of controlling a robotic arm with python code.
The user proposes a certain skill they would like the robot to learn.
To enable this, you are supposed to translate this skill into a python function,
i.e. choose a clear, descriptive name for the function, choose appropriate arguments, and write a clear, descriptive docstring.

For example:
USER: "place one block on top of the other"
RESPONSE:
def place_block_on_other_block(block: TaskObject, otherBlock: TaskObject):
    \"\"\" Places one block on top of the other block \"\"\"
    pass

Do not try to implement the function yet, that happens later.
You should adhere to the following types:
{get_core_types_text()}

The functions don't need Workspace as an argument, since there is only one.
"""

def generate_skill_prompt(prompt, similar_skills: list[Skill]):
    return f"""
    you may use the following function headers as examples of what you are trying to generate:
    {"\n".join([skill.description for skill in similar_skills])}
    ---------------------------------------------------
    write a function header for the prompt: {prompt}.
    """

def refine_function_header_prompt(function_code, refinement):
    return f"""
    Your role is to refine an existing python function, for example by adding a function argument or changing the name.
    If the function is implemented (i.e. not just "pass"), you should also alter the implementation accordingly, making as little
          changes and assumptions as possible.
    Revise the following python function according to the user instructions:
    {function_code}
    Refinement prompt:
    {refinement}
    Do not make any assumptions.
    """

class ParsedList(BaseModel):
    parsed_list: list[str]

def parse_hint_to_list_prompt(hint):
    return f"""
    The user provided a list of tasks that are similar to the one you are currently trying to solve, in a single string.
    Retrieve each of the task descriptions from this string, and return them as a list.
    This is the string: {hint}
    """
```

B. Opening a Drawer

We want to provide a brief hypothetical case study of our method in a less abstract environment, along with a comparison with the most similar prior work Distilling and Retrieving Online Corrections for Generalisable Knowledge (DROC) [109].

In DROC, the authors retrieve *knowledge* relevant to a new task. Since knowledge can be anything that can be expressed as a string and inserted into a prompt, this is more flexible than our approach. Moreover, they similarly store the corrected codes.

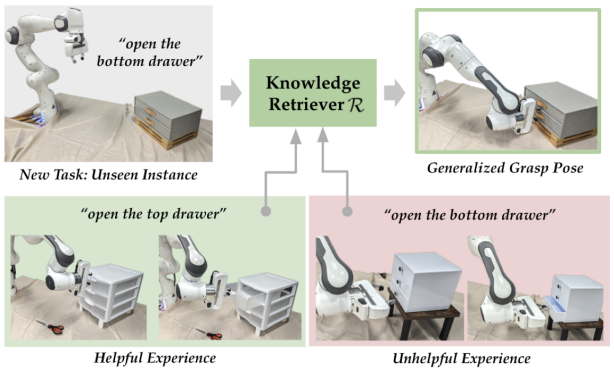


Figure B.1.: Example from DROC, demonstrating the usefulness of *visual* retrieval of prior knowledge to solve similar downstream tasks.

In this example, in DROC, the correct gripper orientation is retrieved from the "helpful experience", to enable gripping of an unseen drawer, and inserted into code for solving the current task. Importantly, in spite of the task description being the same ("open the drawer"), the "unhelpful experience" would motivate the agent to rotate the gripper horizontally, and fail the task. We made the same observation in section 6.1, where the agent successfully places a block next to an axis-aligned block, but the same code would fail if the block were slightly rotated.

Rather than attempting to retrieve such knowledge, in our method we would like to handle this behavioural variance in a skill, e.g. `open_drawer`. One solution would be to tell it to *always* use the vertical gripper orientation. Another would be to introduce another subskill that first determines whether the drawer handle is vertical or horizontal

(e.g. using a VLM, or dimensions of the segmentation mask, ...), and to adjust the gripper handle this way.

This is a reliable and interpretable way to alter the robots behaviour. In DROC, if the robot failed to produce the correct behaviour and we wanted to get an understanding of why, we would have to inspect the retrieval, the LLM modules that process the retrieval and pass it down, and then the Code-as-Policies generated skill-code.

The clear advantage of DROC is that it enables a seamless user interaction. In our method, the user needs to take a very active role, both proposing skills (work would need to be done to make this intuitive for end-users), and setting up tasks that meaningfully test these skills. Further work would have to be done to determine how limiting (or perhaps the opposite) this is.

List of Figures

List of Tables

Bibliography

[1] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. Sri, A. Barrett, D. Christianson, et al. "Pddl| the planning domain definition language." In: *Technical Report, Tech. Rep.* (1998).

[2] G. Ajaykumar, M. Steele, and C.-M. Huang. "A survey on end-user robot programming." In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–36.

[3] J. Aldaco, T. Armstrong, R. Baruch, J. Bingham, S. Chan, K. Draper, D. Dwibedi, C. Finn, P. Florence, S. Goodrich, et al. "Aloha 2: An enhanced low-cost hardware for bimanual teleoperation." In: *arXiv preprint arXiv:2405.02292* (2024).

[4] M. G. Arenas, T. Xiao, S. Singh, V. Jain, A. Ren, Q. Vuong, J. Varley, A. Herzog, I. Leal, S. Kirmani, et al. "How to prompt your robot: A promptbook for manipulation skills with code as policies." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 4340–4348.

[5] A. G. Barto and S. Mahadevan. "Recent advances in hierarchical reinforcement learning." In: *Discrete event dynamic systems* 13 (2003), pp. 341–379.

[6] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. "Curriculum learning." In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[7] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, et al. "Graph of thoughts: Solving elaborate problems with large language models." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 16. 2024, pp. 17682–17690.

[8] Z. Bing, A. Koch, X. Yao, K. Huang, and A. Knoll. "Meta-reinforcement learning via language instructions." In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 5985–5991.

[9] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. "On the opportunities and risks of foundation models." In: *arXiv preprint arXiv:2108.07258* (2021).

[10] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn, et al. "Rt-2: Vision-language-action models transfer web knowledge to robotic control." In: *arXiv preprint arXiv:2307.15818* (2023).

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. "Language models are few-shot learners." In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[12] Y. Cao, H. Zhao, Y. Cheng, T. Shu, Y. Chen, G. Liu, G. Liang, J. Zhao, J. Yan, and Y. Li. "Survey on large language model-enhanced reinforcement learning: Concept, taxonomy, and methods." In: *IEEE Transactions on Neural Networks and Learning Systems* (2024).

[13] X. Chen, M. Lin, N. Schärli, and D. Zhou. "Teaching large language models to self-debug." In: *arXiv preprint arXiv:2304.05128* (2023).

[14] C. Chi, Z. Xu, S. Feng, E. Cousineau, Y. Du, B. Burchfiel, R. Tedrake, and S. Song. "Diffusion policy: Visuomotor policy learning via action diffusion." In: *The International Journal of Robotics Research* (2023), p. 02783649241273668.

[15] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. Jordan, J. E. Gonzalez, et al. "Chatbot arena: An open platform for evaluating llms by human preference." In: *Forty-first International Conference on Machine Learning*. 2024.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding." In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.

[17] J. Duan, W. Yuan, W. Pumacay, Y. R. Wang, K. Ehsani, D. Fox, and R. Krishna. "Manipulate-anything: Automating real-world robots using vision-language models." In: *arXiv preprint arXiv:2406.18915* (2024).

[18] J. Eschmann. "Reward function design in reinforcement learning." In: *Reinforcement learning algorithms: Analysis and Applications* (2021), pp. 25–33.

[19] R. M. French. "Catastrophic forgetting in connectionist networks." In: *Trends in cognitive sciences* 3.4 (1999), pp. 128–135.

[20] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang. "Retrieval-augmented generation for large language models: A survey." In: *arXiv preprint arXiv:2312.10997* 2 (2023).

[21] L. Guan, Y. Zhou, D. Liu, Y. Zha, H. B. Amor, and S. Kambhampati. "Task success is not enough: Investigating the use of video-language models as behavior critics for catching undesirable agent behaviors." In: *arXiv preprint arXiv:2402.04210* (2024).

[22] H. Guo, F. Wu, Y. Qin, R. Li, K. Li, and K. Li. "Recent trends in task and motion planning for robotics: A survey." In: *ACM Computing Surveys* 55.13s (2023), pp. 1–36.

[23] H. Ha, P. Florence, and S. Song. "Scaling up and distilling down: Language-guided robot skill acquisition." In: *Conference on Robot Learning*. PMLR. 2023, pp. 3766–3777.

[24] S. Hao, Y. Gu, H. Ma, J. J. Hong, Z. Wang, D. Z. Wang, and Z. Hu. "Reasoning with language model is planning with world model." In: *arXiv preprint arXiv:2305.14992* (2023).

[25] T. Howell, N. Gileadi, S. Tunyasuvunakool, K. Zakka, T. Erez, and Y. Tassa. "Predictive sampling: Real-time behaviour synthesis with mujoco." In: *arXiv preprint arXiv:2212.00541* (2022).

[26] Y. Hu, Q. Xie, V. Jain, J. Francis, J. Patrikar, N. Keetha, S. Kim, Y. Xie, T. Zhang, H.-S. Fang, et al. "Toward general-purpose robots via foundation models: A survey and meta-analysis." In: *arXiv preprint arXiv:2312.08782* (2023).

[27] Y. Hu, F. Lin, T. Zhang, L. Yi, and Y. Gao. "Look before you leap: Unveiling the power of gpt-4v in robotic vision-language planning." In: *arXiv preprint arXiv:2311.17842* (2023).

[28] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei. "Voxposer: Composable 3d value maps for robotic manipulation with language models." In: *arXiv preprint arXiv:2307.05973* (2023).

[29] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, et al. "Inner monologue: Embodied reasoning through planning with language models." In: *arXiv preprint arXiv:2207.05608* (2022).

[30] X. Huang, W. Liu, X. Chen, X. Wang, H. Wang, D. Lian, Y. Wang, R. Tang, and E. Chen. "Understanding the planning of LLM agents: A survey." In: *arXiv preprint arXiv:2402.02716* (2024).

[31] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al. "Gpt-4o system card." In: *arXiv preprint arXiv:2410.21276* (2024).

[32] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. "How to train your robot with deep reinforcement learning: lessons we have learned." In: *The International Journal of Robotics Research* 40.4-5 (2021), pp. 698–721.

[33] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison. "Rlbench: The robot learning benchmark & learning environment." In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 3019–3026.

[34] D. Kahneman. *Thinking, fast and slow*. macmillan, 2011.

[35] S. Kambhampati, K. Valmeekam, L. Guan, M. Verma, K. Stechly, S. Bhambri, L. P. Saldyt, and A. B. Murthy. "Position: LLMs can't plan, but can help planning in LLM-modulo frameworks." In: *Forty-first International Conference on Machine Learning*. 2024.

[36] U. B. Karli, J.-T. Chen, V. N. Antony, and C.-M. Huang. "Alchemist: Llm-aided end-user development of robot applications." In: *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*. 2024, pp. 361–370.

[37] T. Kaufmann, P. Weng, V. Bengs, and E. Hüllermeier. "A survey of reinforcement learning from human feedback." In: *arXiv preprint arXiv:2312.14925* 10 (2023).

[38] O. Khatib. "A unified approach for motion and force control of robot manipulators: The operational space formulation." In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53.

[39] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair, R. Rafailov, E. Foster, G. Lam, P. Sanketi, et al. "Openvla: An open-source vision-language-action model." In: *arXiv preprint arXiv:2406.09246* (2024).

[40] Y. Kim, D. Kim, J. Choi, J. Park, N. Oh, and D. Park. "A survey on integration of large language models with intelligent robots." In: *Intelligent Service Robotics* 17.5 (2024), pp. 1091–1107.

[41] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, et al. "Segment anything." In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2023, pp. 4015–4026.

[42] B. Kouvaritakis and M. Cannon. "Model predictive control." In: *Switzerland: Springer International Publishing* 38.13-56 (2016), p. 7.

[43] H. Koziolek, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, and N. Eskandani. "LLM-based and retrieval-augmented control code generation." In: *Proceedings of the 1st International Workshop on Large Language Models for Code*. 2024, pp. 22–29.

[44] H. Le, H. Chen, A. Saha, A. Gokul, D. Sahoo, and S. Joty. "Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules." In: *arXiv preprint arXiv:2310.08992* (2023).

[45] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks." In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[46] Z. Li, K. Yu, S. Cheng, and D. Xu. "League++: Empowering continual robot learning through guided skill acquisition with large language models." In: *ICLR 2024 Workshop on Large Language Model (LLM) Agents*. 2024.

[47] 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: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 9493–9500.

[48] J. Liang, F. Xia, W. Yu, A. Zeng, M. G. Arenas, M. Attarian, M. Bauza, M. Bennice, A. Bewley, A. Dostmohamed, et al. "Learning to learn faster from human feedback with language model predictive control." In: *arXiv preprint arXiv:2402.11450* (2024).

[49] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone. "Llm+ p: Empowering large language models with optimal planning proficiency." In: *arXiv preprint arXiv:2304.11477* (2023).

[50] J. Luketina, N. Nardelli, G. Farquhar, J. Foerster, J. Andreas, E. Grefenstette, S. Whiteson, and T. Rocktäschel. "A survey of reinforcement learning informed by natural language." In: *arXiv preprint arXiv:1906.03926* (2019).

[51] C. Lynch and P. Sermanet. "Language conditioned imitation learning over unstructured data." In: *arXiv preprint arXiv:2005.07648* (2020).

[52] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar. "Eureka: Human-level reward design via coding large language models." In: *arXiv preprint arXiv:2310.12931* (2023).

[53] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. "Robot operating system 2: Design, architecture, and uses in the wild." In: *Science robotics* 7.66 (2022), eabm6074.

[54] A. Mandlekar, Y. Zhu, A. Garg, J. Booher, M. Spero, A. Tung, J. Gao, J. Emmons, A. Gupta, E. Orbay, et al. "Roboturk: A crowdsourcing platform for robotic skill learning through imitation." In: *Conference on Robot Learning*. PMLR. 2018, pp. 879–893.

[55] J. W. Mao. "A framework for llm-based lifelong learning in robot manipulation." PhD thesis. Massachusetts Institute of Technology, 2024.

[56] O. Mees, L. Hermann, and W. Burgard. "What matters in language conditioned robotic imitation learning over unstructured data." In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 11205–11212.

[57] M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, et al. "Orbit: A unified simulation framework for interactive robot learning environments." In: *IEEE Robotics and Automation Letters* 8.6 (2023), pp. 3740–3747.

[58] M. Murray, A. Gupta, and M. Cakmak. "Teaching Robots with Show and Tell: Using Foundation Models to Synthesize Robot Policies from Language and Visual Demonstration." In: *8th Annual Conference on Robot Learning*. 2024.

[59] O. Nachum, H. Tang, X. Lu, S. Gu, H. Lee, and S. Levine. "Why does hierarchy (sometimes) work so well in reinforcement learning?" In: *arXiv preprint arXiv:1909.10618* (2019).

[60] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone. "Curriculum learning for reinforcement learning domains: A framework and survey." In: *Journal of Machine Learning Research* 21.181 (2020), pp. 1–50.

[61] A. O'Neill, A. Rehman, A. Maddukuri, A. Gupta, A. Padalkar, A. Lee, A. Pooley, A. Gupta, A. Mandlekar, A. Jain, et al. "Open x-embodiment: Robotic learning datasets and rt-x models: Open x-embodiment collaboration 0." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 6892–6903.

[62] O. Ovadia, M. Brief, M. Mishaeli, and O. Elisha. "Fine-tuning or retrieval? comparing knowledge injection in llms." In: *arXiv preprint arXiv:2312.05934* (2023).

[63] M. Parakh, A. Fong, A. Simeonov, T. Chen, A. Gupta, and P. Agrawal. "Lifelong robot learning with human assisted language planners." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 523–529.

[64] Y. Park, G. B. Margolis, and P. Agrawal. "Automatic environment shaping is the next frontier in rl." In: *arXiv preprint arXiv:2407.16186* (2024).

[65] Y. Qin, S. Hu, Y. Lin, W. Chen, N. Ding, G. Cui, Z. Zeng, X. Zhou, Y. Huang, C. Xiao, et al. "Tool learning with foundation models." In: *ACM Computing Surveys* 57.4 (2024), pp. 1–40.

[66] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al. "Learning transferable visual models from natural language supervision." In: *International conference on machine learning*. PmLR. 2021, pp. 8748–8763.

[67] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. "Improving language understanding by generative pre-training." In: (2018).

[68] H. Ravichandar, A. S. Polydoros, S. Chernova, and A. Billard. "Recent advances in robot learning from demonstration." In: *Annual review of control, robotics, and autonomous systems* 3.1 (2020), pp. 297–330.

[69] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, et al. "A generalist agent." In: *arXiv preprint arXiv:2205.06175* (2022).

[70] N. Reimers and I. Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." In: *arXiv preprint arXiv:1908.10084* (2019).

[71] T. Schaul, D. Horgan, K. Gregor, and D. Silver. "Universal value function approximators." In: *International conference on machine learning*. PMLR. 2015, pp. 1312–1320.

[72] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou. "Large language models can be easily distracted by irrelevant context." In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31210–31227.

[73] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. "Reflexion: Language agents with verbal reinforcement learning." In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 8634–8652.

[74] A. Simeonov, Y. Du, A. Tagliasacchi, J. B. Tenenbaum, A. Rodriguez, P. Agrawal, and V. Sitzmann. "Neural descriptor fields: Se (3)-equivariant object representations for manipulation." In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 6394–6400.

[75] 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: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 11523–11530.

[76] J. Skalse, N. Howe, D. Krasheninnikov, and D. Krueger. "Defining and characterizing reward gaming." In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 9460–9471.

[77] S. Song, A. Zeng, J. Lee, and T. Funkhouser. "Grasping in the wild: Learning 6dof closed-loop grasping from low-cost demonstrations." In: *IEEE Robotics and Automation Letters* 5.3 (2020), pp. 4978–4985.

[78] M. T. Spaan. "Partially observable Markov decision processes." In: *Reinforcement learning: State-of-the-art*. Springer, 2012, pp. 387–414.

[79] S. Stepputtis, J. Campbell, M. Phielipp, S. Lee, C. Baral, and H. Ben Amor. "Language-conditioned imitation learning for robot manipulation tasks." In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 13139–13150.

[80] T. Sumers, S. Yao, K. Narasimhan, and T. Griffiths. "Cognitive architectures for language agents." In: *Transactions on Machine Learning Research* (2023).

[81] R. S. Sutton, A. G. Barto, et al. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[82] R. S. Sutton, D. Precup, and S. Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning." In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[83] C. Tang, B. Abbatematteo, J. Hu, R. Chandra, R. Martín-Martín, and P. Stone. "Deep reinforcement learning for robotics: A survey of real-world successes." In: *Annual Review of Control, Robotics, and Autonomous Systems* 8 (2024).

[84] O. M. Team, D. Ghosh, H. Walke, K. Pertsch, K. Black, O. Mees, S. Dasari, J. Hejna, T. Kreiman, C. Xu, et al. "Octo: An open-source generalist robot policy." In: *arXiv preprint arXiv:2405.12213* (2024).

[85] M. Toussaint. "Logic-Geometric Programming: An Optimization-Based Approach to Combined Task and Motion Planning." In: *IJCAI*. 2015, pp. 1930–1936.

[86] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017).

[87] S. H. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor. "Chatgpt for robotics: Design principles and model abilities." In: *Ieee Access* (2024).

[88] N. Wake, A. Kanehira, K. Sasabuchi, J. Takamatsu, and K. Ikeuchi. "Gpt-4v (ision) for robotics: Multimodal task planning from human demonstration." In: *IEEE Robotics and Automation Letters* (2024).

[89] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. "Voyager: An open-ended embodied agent with large language models." In: *arXiv preprint arXiv:2305.16291* (2023).

[90] J. Wang, E. Shi, H. Hu, C. Ma, Y. Liu, X. Wang, Y. Yao, X. Liu, B. Ge, and S. Zhang. "Large language models for robotics: Opportunities, challenges, and perspectives." In: *Journal of Automation and Intelligence* (2024).

[91] L. Wang, Y. Ling, Z. Yuan, M. Shridhar, C. Bao, Y. Qin, B. Wang, H. Xu, and X. Wang. "Gensim: Generating robotic simulation tasks via large language models." In: *arXiv preprint arXiv:2310.01361* (2023).

[92] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. "Executable code actions elicit better llm agents." In: *Forty-first International Conference on Machine Learning*. 2024.

[93] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. "Self-consistency improves chain of thought reasoning in language models." In: *arXiv preprint arXiv:2203.11171* (2022).

[94] Y. Wang, Z. Sun, J. Zhang, Z. Xian, E. Biyik, D. Held, and Z. Erickson. "Rl-vlm-f: Reinforcement learning from vision language foundation model feedback." In: *arXiv preprint arXiv:2402.03681* (2024).

[95] Z. Wang, S. Cai, G. Chen, A. Liu, X. Ma, and Y. Liang. "Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents." In: *arXiv preprint arXiv:2302.01560* (2023).

[96] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. "Chain-of-thought prompting elicits reasoning in large language models." In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[97] 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." In: *Autonomous Robots* 47.8 (2023), pp. 1087–1102.

[98] W. Xie, M. Valentini, J. Lavering, and N. Correll. "DeliGrasp: Inferring Object Properties with LLMs for Adaptive Grasp Policies." In: *arXiv preprint arXiv:2403.07832* (2024).

[99] K. Yang, J. Liu, J. Wu, C. Yang, Y. R. Fung, S. Li, Z. Huang, X. Cao, X. Wang, Y. Wang, et al. "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: *arXiv preprint arXiv:2401.00812* (2024).

[100] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan. "Tree of thoughts: Deliberate problem solving with large language models." In: *Advances in neural information processing systems* 36 (2023), pp. 11809–11822.

[101] W. Ye, Y. Zhang, H. Weng, X. Gu, S. Wang, T. Zhang, M. Wang, P. Abbeel, and Y. Gao. "Reinforcement Learning with Foundation Priors: Let the Embodied Agent Efficiently Learn on Its Own." In: *arXiv preprint arXiv:2310.02635* (2023).

[102] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine. "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning." In: *Conference on robot learning*. PMLR. 2020, pp. 1094–1100.

[103] W. Yu, N. Gileadi, C. Fu, S. Kirmani, K.-H. Lee, M. G. Arenas, H.-T. L. Chiang, T. Erez, L. Hasenclever, J. Humplik, et al. "Language to rewards for robotic skill synthesis." In: *arXiv preprint arXiv:2306.08647* (2023).

[104] W. Yuan, A. Murali, A. Mousavian, and D. Fox. "M2t2: Multi-task masked transformer for object-centric pick and place." In: *arXiv preprint arXiv:2311.00926* (2023).

[105] K. Zakka. *Mink: Python inverse kinematics based on MuJoCo*. Version 0.0.4. July 2024.

[106] M. Zare, P. M. Kebria, A. Khosravi, and S. Nahavandi. "A survey of imitation learning: Algorithms, recent developments, and challenges." In: *IEEE Transactions on Cybernetics* (2024).

[107] A. Zeng, M. Attarian, B. Ichter, K. Choromanski, A. Wong, S. Welker, F. Tombari, A. Purohit, M. Ryoo, V. Sindhwani, et al. "Socratic models: Composing zero-shot multimodal reasoning with language." In: *arXiv preprint arXiv:2204.00598* (2022).

[108] A. Zeng, P. Florence, J. Tompson, S. Welker, J. Chien, M. Attarian, T. Armstrong, I. Krasin, D. Duong, V. Sindhwani, et al. "Transporter networks: Rearranging the visual world for robotic manipulation." In: *Conference on Robot Learning*. PMLR. 2021, pp. 726–747.

[109] L. Zha, Y. Cui, L.-H. Lin, M. Kwon, M. G. Arenas, A. Zeng, F. Xia, and D. Sadigh. "Distilling and retrieving generalizable knowledge for robot manipulation via language corrections." In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2024, pp. 15172–15179.

[110] S. Zhang, P. Wicke, L. K. Şenel, L. Figueredo, A. Naceri, S. Haddadin, B. Plank, and H. Schütze. "Lohoravens: A long-horizon language-conditioned benchmark for robotic tabletop manipulation." In: *arXiv preprint arXiv:2310.12020* (2023).

[111] A. Zhao, D. Huang, Q. Xu, M. Lin, Y.-J. Liu, and G. Huang. "Expel: Llm agents are experiential learners." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 17. 2024, pp. 19632–19642.

[112] H. Zhou, X. Yao, Y. Meng, S. Sun, Z. Bing, K. Huang, and A. Knoll. "Language-conditioned learning for robotic manipulation: A survey." In: *arXiv preprint arXiv:2312.10807* (2023).

[113] Y. Zhu, J. Wong, A. Mandlekar, R. Martín-Martín, A. Joshi, S. Nasiriany, and Y. Zhu. "robosuite: A modular simulation framework and benchmark for robot learning." In: *arXiv preprint arXiv:2009.12293* (2020).