Universität Bremen

Robot Planning and Reasoning with Large Language Models: An Investigation Using PyCRAM as an Example

submitted by

Julius Harnack

Matriculation number: 6035044

in the course of study Computer Science

to obtain the academic degree

Bachelor of Science (B.Sc.)

Examiners:

Prof. Michael Beetz PhD

Dr. Rene Weller

January 31, 2025

Contents

List of Figures

Abstract

The integration of Large Language Models (LLMs) into robotic planning and reasoning presents a promising yet challenging frontier in artificial intelligence. This work addresses the difficulty of enabling LLMs to generate executable code for specialized and less-known frameworks like PyCRAM, which are not extensively represented in the models' pretraining data. The significance of this problem lies in its potential to revolutionize the development of intelligent and autonomous robots, facilitating more intuitive human-robot interactions and simplifying the programming of complex tasks.

To tackle this challenge, we developed a pipeline that augments LLMs with precisely structured prompts, advanced algorithms, and external knowledge sources. Techniques such as Retrieval Augmented Generation (RAG), sophisticated prompting strategies, and the ReWOO architectural framework were employed to enhance the models' understanding and code generation capabilities for PyCRAM. The approach involved optimizing prompt quality and integrating vector databases containing documentation, code examples, and environmental models.

The findings reveal that, while LLMs can be augmented to generate code for unfamiliar frameworks to a certain extent, they face limitations when handling complex tasks requiring intricate reasoning and planning. The models demonstrated proficiency in reproducing known patterns and executing simple tasks but struggled with novel instructions and error correction. Fine-tuning the models on domain-specific data showed mixed results, highlighting the delicate balance between specialization and generalization.

This work contributes valuable insights into the use of LLMs for robotic planning and reasoning, presenting an architecture that bridges natural language instructions and executable robot plans. The investigation underscores the potential of LLMs to simplify coding for specialized libraries and advances the understanding of their capabilities and limitations in robotics. The results pave the way for future developments that could further enhance the synergy between natural language processing and autonomous robotic systems.

The code for the architecture can be found in this GitHub repository: github.com/Julius6043/PyCRAM-LLM

Acknowledgments

I would like to express my sincere gratitude to the IAI and the research team for the supportive atmosphere, the exciting topic, the workplace with the Ubuntu computer, the helpfulness, and the access to the GPU server. I thank Dominic Kastens for the support in accessing the GPUs for testing the local LLM and Jonas Dech for the help regarding PyCRAM.

Special thanks go to my supervisors Alina Hawkin and Mihai Pomarlan, who have always supported, advised, and networked me very well, and have always responded promptly to my questions.

Declaration

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those indicated.

Place, Date Signature

1 Introduction

1.1 Motivation and Goals

Artificial intelligence is the top topic of our time. More and more start-ups in this field are emerging, and the big tech companies are investing billions in development. The goal is AGI, Artificial General Intelligence. In short: artificial intelligence that can perform all economically valuable tasks that a human is capable of [OpenAI, 2018].

The ongoing development has led to significant advances in natural language processing. In particular, Large Language Models (LLMs) like GPT-4 from OpenAI have proven to be extremely powerful in a variety of tasks, including text generation, translation, and even basic forms of reasoning. Despite these successes, there are still significant challenges when it comes to deploying LLMs in dynamic and real-world scenarios, such as in robot planning and control.

In robotics, the ability to plan and reason is crucial to autonomously handle complex tasks. Traditional approaches to robot planning, such as symbolic planners, offer efficient solutions but often rely on complete and precise descriptions of the domain [Singh et al., 2023a, P.1]. This significantly limits their applicability in real, unstructured, big environments. In contrast, LLMs can handle distorted and incomplete information, making them potentially powerful tools for robot planning.

In various demo videos from large companies, we can already see the impressive execution of highly complex tasks using LLMs. Examples include Figure 01 from Figure, GPT-4o from OpenAI and Optimus from Tesla. In the first example, we see a robot equipped with OpenAI's speech-to-speech feature responding to a given task and handing an apple [Figure, 2024].

The second example shows a demo video of the yet-to-be-released live call feature of GPT-4o, where the model is streamed a screen and is supposed to help the student understand a math problem. The model shows very good reasoning in understanding the task and the student's notes and questions. The live reasoning and video understanding shows potential for possible application in robots [OpenAI, 2024e].

In the last example, the Optimus robot from Tesla is presented on the "We, robot" event. There the robot is shown interacting with humans in real time and serving them drinks [Tesla, 2024].

The work was created as part of a research project at the University of Bremen and the IAI. The aim of this work is to investigate the role and capabilities of LLMs in the context of robot planning and reasoning. This is to be done using PyCRAM, a framework for cognitive robotics, as

an example. The possibilities and limitations of integrating LLMs into robot planning systems are to be demonstrated and illustrated through practical experiments with this framework.
To this end, a pipeline is being developed that enables the creation of complex action instructions for robots from user inputs in natural language to solve planing problems. Various techniques are applied to increase the performance of LLMs in this task area and to expand the capabilities of a single LLM.
The research questions to be answered include:
To what extent can LLMs be augmented to generate code from new research frameworks that are little or not at all captured by the pretraining data? How efficient and reliable can these models be used in robotics? What challenges and limitations currently exist in the implementation and use of this technology, and what developments can arise in the future?

1.2 Structure of the Work

The work is structured as follows:
Chapter 2 provides an overview of related work in the field of integrating Large Language Models (LLMs) into robotics. It discusses both closed and open research efforts, highlighting key developments and approaches in utilizing LLMs for robot planning and reasoning.
Chapter 3 delves into the technical background necessary for understanding the integration of LLMs into robot planning. It introduces the core concepts, technologies, and frameworks used in the thesis, including the PyCRAM framework, LangChain, LangGraph, and various LLMs. Additionally, it covers prompting methods and the use of vector databases.
Chapter 4 outlines the methodology adopted for developing the LLM pipeline. It details the design considerations, including preliminary investigations, the development of a complex architecture, and the incorporation of techniques such as ReWOO, RAG, and AutoCoT. The approach to fine-tuning models for PyCRAM and the presentation of the final architecture are also discussed.
Chapter 5 describes the implementation of the LLM pipeline. It discusses the practical steps taken to build the architecture, including the setup of the environment, integration of various components, and the challenges encountered during development.
Chapter 6 presents the evaluation of the developed pipeline. It details the test procedures, including component tests and the definition of test cases. The chapter analyzes the test results, providing insights into the performance, strengths, and weaknesses of the pipeline.

Chapter 7 summarizes the key findings of the research. It answers the research questions posed, discussing the extent to which LLMs can be augmented to generate code for new frameworks, their efficiency and reliability in robotics, and the challenges and limitations observed.
Chapter 8 offers an outlook on future work. It suggests potential improvements to the pipeline, such as incorporating new models, further fine-tuning, multimodality, and enhancing prompts and architecture. The chapter also discusses potential developments in the field and their implications for the future of robotics and AI.

2 Related Work

The following chapter the related research areas of the work are presented. The Topics are LLMs in Robotik, Planing with LLms and Reasoning with LLMs.

2.1 LLMs in Robotics

Like LLMs as well, Robotics is strongly divided into closed research and open research. The closed research is done by companies, who have the intention to develop a product for sale, and therefore they mostly keep their research for competitive advantage. Open research, are publicly available research paper. Closed research is typically ahead of the open research, because many of the companies have large amounts of capital and information. This is especially important to train neural networks on large amounts of data.

2.1.1 Closed research

Figure

Figure AI is a company specializing in the development of humanoid robots designed to work in various industrial environments. They have partnered with Nvidia, OpenAI, and Microsoft for the use, development, and training of AI models. One of their demos was already mentioned in the introduction. There the robot is asked questions about the environment and given undefined action prompts like "Can I have something to eat?". The robot then responds with "Of course!" and gives the person an apple that is on the kitchen counter. Even though it is a demo under laboratory conditions and the functionality cannot be seen, it definitely shows the potential and interest in multimodal LLMs in conjunction with robots.[1] The following Figure 1

[1] www.youtube.com/watch?v=Sq1QZB5baNw

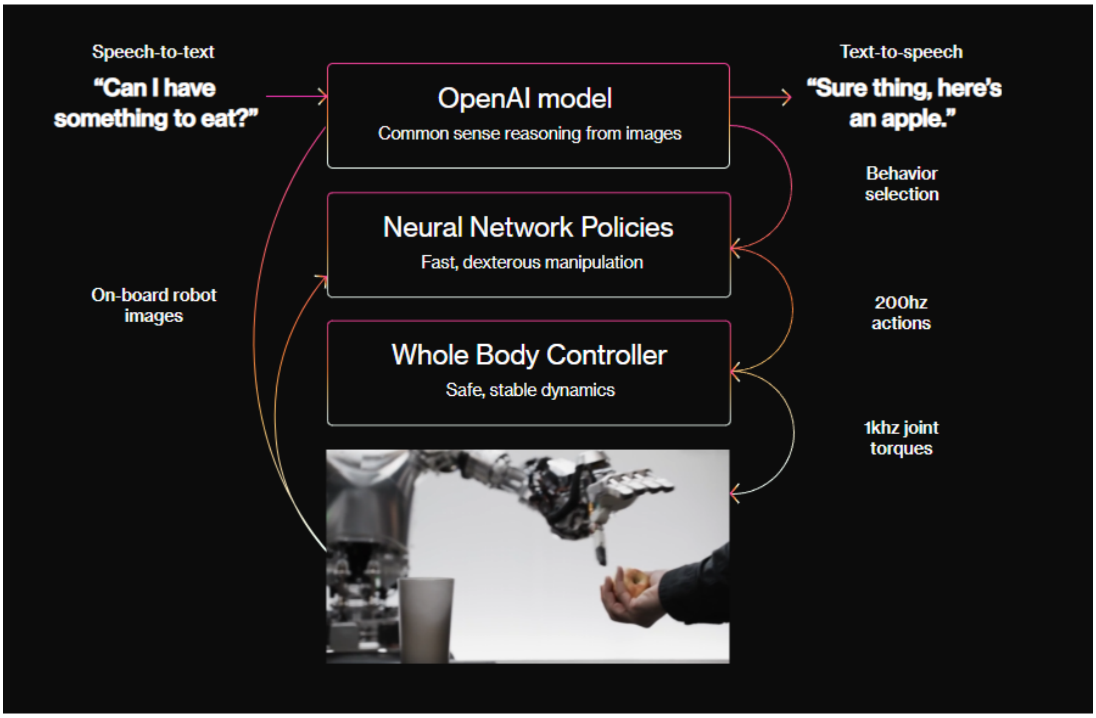

Figure 1: Figures Speech-to-Speech Robot Reasoning [Figure AI, 2024]

shows a rough visualization of the technical implementation of that presented demo. It can be seen that an OpenAI model is used for "Common Sense Reasoning" as well as for "Behavior Selection" for another neural network.

1x

1X Technologies develops humanoid robots that assist in households and other areas through AI and "embodied learning". Their latest model, NEO, builds on the experiences of their first robot, EVE, and is trained via teleoperation and natural language control. On their website, they state, "The AI 'mind' strings together actions" [1X Technologies, 2024]. Additionally, a YouTube video by the channel 'S3' features an interview with 1X, demonstrating live demos by living with NEO in his house[2]. In this video, it is explained that the robot is initially teleoperated, and the data collected is then used to train an AI model based on neural networks, which can predict the next move from the input data. The skills learned by the AI model in this way are subsequently linked together by another AI model [S3, 2024]. It is likely that this involves a type of fine-tuned multimodal LLM.

Optimus

[2]youtu.be/Sb6LMPXRdVc?si=DqqY2Tne2jG4EoHT

Optimus is a humanoid robot developed by Tesla. Aside from demos, little is known about the robot. As mentioned in the introduction, there is a recent demo where the robots interact live in large crowds at an event, distributing drinks to attendees.[3] Visitors also published videos showing full conversations with Optimus, which was able to quickly respond to requests such as "Do a dance move" within seconds [Chaudhary Parvez, 2024]. While some robots acted as though they were fully autonomous, with no transparent indication of control at the event, one robot in a video clarified that they were primarily teleoperated, and the conversations with people were conducted by humans [Troy Teslike, 2024a, Troy Teslike, 2024b]. Therefore, this was simply a concept demo, which, though still impressive, did not showcase the level of autonomy that an initial naive observation might suggest.
To execute the demonstrated capabilities, a system similar to GPT-4's live call feature would be required, which, as described in Figure 1, directs and integrates other neural models to perform tasks.

2.1.2 Open research

In the following, the open research paper are presented. There it is certain that they use LLMs, how they accomplished their work and that the demos are not fake.

Vision Models

Wang et al. [Wang et al., 2024] provide a comprehensive overview of the integration of LLMs into various robotic tasks and discuss the associated opportunities, challenges, and perspectives. They propose a framework that utilizes multimodal LLMs such as GPT-4V to improve the planning of embodied tasks by combining natural language instructions with visual perception. In their experiments across various datasets, they demonstrate that GPT-4V can effectively enhance robot performance in embodied tasks [Wang et al., 2024, S. 11]. Despite these advancements, the authors emphasize existing challenges, particularly regarding model transparency, robustness, and safety in real-world applications.
They note that generated plans struggle to handle complex scenarios, that LLMs require lengthy and detailed prompts to work robust, and that robots are limited to predefined actions [Wang et al., 2024, S. 10]. Additionally, the black-box nature of LLMs is highlighted as an issue, making it difficult to understand the decision-making processes of these models [Wang et al.,

[3] youtube.com/watch?v=6v6dbxPlsXs&t=1200s

2024, S. 11].

Robocup Team TRAIL

The TRAIL team from the University of Tokyo [Shirasaka et al., 2024] has developed a general-purpose service robot system that integrates multiple foundation models. Foundation models are large pre-trained neural networks trained on extensive datasets and can be used in various tasks. The system is fully promptable, meaning it can be adapted to different tasks and environments by adjusting the input prompts without additional training.

They identified three main types of errors in realistic applications of General Purpose Service Robots (GPSR): insufficient information, faulty plan creation, and execution errors. To address these challenges, they introduced self-recovery prompting, where the system determines necessary information and modifies the prompts to fix errors.

In the context of large language models (LLMs) for robotics, this work demonstrates how LLMs can be used in combination with other foundation models to increase the flexibility and adaptability of robots in real environments. Experimentally, the TRAIL team demonstrated that their system, with the self-recovery mechanism, is capable of solving various error scenarios and successfully executing tasks.

LLM for CRAM

Töberg and Cimiano examined in their study [Töberg and Cimiano, 2023] the ability of LLMs to generate robot manipulation plans within the CRAM framework. They conducted experiments with ChatGPT and GPT-4, in which the models were tasked with generating a plan (designator) for a target action based on a given reference action as an example. The quality of the generated designators was evaluated using machine translation and code generation metrics, as well as their compilability.

The results showed that GPT-4 slightly outperformed GPT-3.5, with both models delivering solid overall results. However, only about 36% of the generated designators were successfully compiled. Additionally, they found that a higher similarity between the reference and target actions unexpectedly correlated negatively with the success of compilation. In a further attempt to use the models interactively for the incremental refinement of designators, ChatGPT's behavior proved to be insufficiently reliable to support this process effectively. [Töberg and Cimiano, 2023, P. 195-196]

ProgPrompt

Singh et al. present ProgPrompt, a programmatic prompt structure for LLMs that enables the generation of executable robot task plans [Singh et al., 2023b]. By providing program-like specifications of available actions and objects in the environment, along with executable example programs, LLMs can generate situated plans that work across various environments, robotic capabilities, and tasks. The recommended prompt structure includes import statements for actions, a list of environmental objects, and function definitions that represent tasks through action sequences. Ablation experiments demonstrate that incorporating programming structural elements such as comments and assertions into prompts significantly enhances plan generation. Their approach achieves state-of-the-art success rates in VirtualHome household tasks and is successfully applied to a physical robotic arm for tabletop tasks.

2.2 Reasoning with LLMs

The reasoning capabilities of LLMs are a complex topic due to the black-box nature of these models, making it difficult to determine if true reasoning is occurring or if they are merely using memorized patterns and heuristics to solve problems [Huang and Chang, 2023, P.9]. There are also various types of reasoning, each with differing levels of difficulty and requirements. A distinction can be made between formal and informal reasoning, as well as between categories such as deductive, inductive, and abductive reasoning. Therefore, reasoning abilities must be evaluated with different benchmarks and comparisons. Current model benchmarks are presented in section 3.2.6.

Prompting

There are various input techniques that influence the performance and reasoning capabilities of LLMs. These range from simple tricks to complex architectures. The techniques used in this work are presented in subsection 3.3. One well-known technique is Chain of Thought (CoT), where the LLM is prompted to respond in a series of thought steps. This has been shown to improve performance on specific reasoning tasks [Huang and Chang, 2023, P.8].

LRM

OpenAI has introduced a new type of model with their recent release on September 12, referred to as o1. In the paper presented in section 2.3, this type of model is called a Large Reasoning Model (LRM). The model is trained through reinforcement learning to draw complex logical inferences. The o1 model reflects intensively before responding, using an internal chain of thought. It significantly outperforms GPT-4o on various challenging benchmarks, achieving the 89th percentile in competitive programming, solving over 74% of AIME math problems, and surpassing human experts with PhDs in scientific tests [OpenAI, 2024d]. An early version, called o1-preview, is already available in ChatGPT [OpenAI, 2024b].

The authors who coined the term Large Reasoning Model speculate that the complete system improves its ability to generate appropriate CoT steps useful for reasoning in a pretraining reinforcement learning (RL) phase with synthetic data, and executes prompt-specific rollouts at inference time. They suggest it may be an RL-trained system similar to AlphaGo, the RL-based chess AI, but in this case, the 'moves' generated and evaluated are chains of thought [Valmeekam et al., 2024]. These models represent a significant advancement, demonstrating the potential of well-coordinated AI components within a complex architecture involving LLMs.

2.3 Planning with LLMs

In the studies presented in subsubsection 2.1.2 regarding LLMs in robotics, they are utilized to solve planning problems.

In principle, classical planning problems consist of finding a sequence of actions within a domain, composed of state variables, known as fluents, as well as possible actions that can change the world. The goal is to transform an initial state into a target state [Valmeekam et al., 2023].

Compared to traditional planners, which require users to specify exact goal conditions and extensive domain knowledge, an LLM-based planner offers significant advantages. Classical planners encounter limitations in complex environments as the search space grows exponentially without built-in knowledge, and the domain specification, as well as the search space, increase non-linearly with environmental complexity. An LLM planner, however, can derive achievable goal states from simple natural language instructions, without requiring users to specify detailed goal states or domain knowledge. By leveraging world knowledge and common sense, it can effectively constrain the search space, enabling more efficient planning processes and greatly simplifying the handling of complex environments [Singh et al., 2023a,ProgPrompt, 2023, Dagan et al., 2023].

PlanBench

Valmeekam et al. evaluate the planning capabilities of LLMs, specifically OpenAI's o1 model, on PlanBench [Valmeekam et al., 2024]. PlanBench is a benchmark with 600 Blocksworld planning problems. They find that, despite advancements, current LLMs still have limitations in planning tasks. The o1 model, referred to as a Large Reasoning Model (LRM), shows significant improvements over previous LLMs, though it does not yet fully solve all tasks. The authors recommend targeted prompt structuring to enhance planning performance and emphasize the need for evaluation metrics that include accuracy, efficiency, and reliability.
While the results are still not flawless and especially less efficient than the traditional Fast Downward planner, they represent considerable progress over previous work from approximately one year ago [Valmeekam et al., 2023, Valmeekam et al., 2024]. Notably, o1 shows impressive capabilities for an initial preview version. If development continues and costs and execution times decrease, a fully capable LRM could be on the market within a year.
Until then, LLMs can primarily be used in conjunction with traditional planning methods as planning aids, for which there are already approaches [Kambhampati et al., 2024, Dagan et al., 2023].

3 Technical Background

In this chapter, concepts, technologies, and frameworks that have been used and are important for the built architecture are discussed. By combining these technical foundations, a solid basis for the implementation of the LLM pipeline is created, which is described in detail in the following chapters. The pipeline is intended to translate a natural language instruction from a user into executable PyCRAM plan code using LLMs, enabling it to be executed in a simulation.

3.1 Technology Stack

3.1.1 Python

Python is used as the programming language because it is one of the most popular languages and therefore supports all the important modern frameworks that are used in the work.

3.1.2 ROS

The Robot Operating System (ROS) is a flexible framework used for developing software for robots. It provides a collection of tools, libraries, and conventions that facilitate the development of complex and robust robot applications. Originally developed by Willow Garage and now maintained by Open Robotics, ROS has become the standard for robot applications in research and industry [Open Robotics, 2024]. ROS is required in this work for the use of PyCRAM.

3.1.3 PyCRAM

PyCRAM [Dech, 2024a] is a framework for cognitive robotics developed in Python 3 that facilitates the implementation and execution of complex robot tasks on various robot platforms. It provides a collection of tools and libraries that support both software development for robots and geometric reasoning and fast simulation mechanisms. This allows for the development of cognitive control programs that enable a high degree of robot autonomy [PyCRAM Developers, 2024].

The core elements of PyCRAM follow.

Abstraction of Robot Hardware:

Allows the execution of the same high-level plan on different robots, such as PR2 and IAI's Boxy, by abstracting hardware-specific details.

Designators:

Provides a flexible system of designators that allow objects, places, and actions to be symbolically described and referenced.

Integration with ROS:

PyCRAM is integrated into the Robot Operating System (ROS) to enable communication between different software components and the robot.

Geometric Reasoning and Simulation:

Enables the execution of geometric calculations and simulations to optimize the planning and execution of robot actions [Dech, 2024b].

Bullet World:

PyCRAM has a simulation environment where robots and various environments can be loaded using URDF files. URDF files have a hierarchical structure that defines the relationships and relative positions of the components.

In this work, this simulation environment is used to execute and test the generated action sequences of the robots.

3.1.4 LangChain

LangChain [LangChain, 2024b] is a framework for developing applications based on LLMs. It simplifies every phase of the lifecycle of an LLM application, from development to production to deployment. LangChain provides open-source building blocks, components, and third-party integrations to simplify the development of LLM applications.
langchain: Chains, agents, and retrieval strategies that form the cognitive architecture of an application.
LangGraph: It enables the creation of robust and stateful multi-actor applications with LLMs.
LangServe: For the deployment of LangChain chains as REST APIs.
LangSmith: It is developer platform for debugging, testing, evaluating, and monitoring LLM applications.

3.1.5 LangGraph

LangGraph [LangChain Inc, 2024] is a library developed by LangChain Inc. to create stateful multi-actor applications with LLMs. It is used to develop agent and multi-agent workflows and offers some core advantages over other LLM frameworks: cycles, controllability, and persistence.
LangGraph allows the definition of flows with cycles, which are essential for most agent architectures. This means that workflows do not have to be linear but can contain loops and conditional branches to handle more complex tasks. As a low-level framework, LangGraph offers fine-grained control over the flow and state of the application. This is crucial for developing reliable agents, as developers can precisely control how information is processed and decisions are made. LangGraph has a built-in persistence feature that allows the state of the application to be automatically saved after each step. This enables advanced features such as human-in-the-loop workflows, error recovery, and "time travel" through the state history.
LangGraph integrates seamlessly with LangChain and LangSmith but does not require them. Outputs can be streamed directly as they are produced by the nodes. The flow can be interrupted to approve or edit actions of the agent.

Central Concepts:
State: Each graph execution is assigned a state that is passed between the nodes. Each node updates this state with its return value.
Nodes: Nodes represent individual processing steps in the workflow.
Edges: Edges define the connections between the nodes and control the flow of the workflow.
Checkpoints: Checkpoints save the state of the application and enable recovery and analysis of the flow.

LangGraph is suitable for developing complex, stateful applications with LLMs, especially for agent and multi-agent systems. The library offers the flexibility and control required for developing robust and reliable AI applications. That is why it is used in this work.

3.1.6 Supabase

Supabase is used for the vector databases. It is an online database hosting provider for Postgres databases, accessible via API calls and directly integrated into LangChain. A free version is available, which is sufficient for this project.

3.2 Large Language Models (LLMs)

LLMs represent a significant advancement in natural language processing (NLP). They are capable of learning generic knowledge from extensive training datasets and generating high-quality texts. Particularly noteworthy are the models from OpenAI, such as GPT-3 and GPT-4, which, due to their size and the diversity of training data, possess impressive capabilities in text generation and understanding complex instructions. These models form the basis for the LLM pipeline developed in this work.

3.2.1 Functionality

Neural networks form the foundation for LLMs. In particular, deep neural networks have enabled the capture of complex patterns and dependencies in language data. The introduction of Recurrent Neural Networks (RNNs) and later Long Short-Term Memory (LSTM) networks [Hochreiter, 1997] enabled the effective modeling of sequential data, which is essential for language processing tasks.
With the development of the Transformer architecture by [Vaswani, 2017], a paradigm shift was

initiated. Transformer models use self-attention mechanisms that allow dependencies between words to be captured regardless of their distance in the sequence. This led to significant performance improvements in various NLP tasks and forms the basis for modern LLMs such as BERT [Kenton and Toutanova, 2019] and GPT models [Radford, 2018, Radford et al., 2019]. These deep neural networks are capable of learning the statistical properties of language by training on large text corpora. This allows them to perform tasks such as text generation, machine translation, and question answering with high accuracy [Brown, 2020].

3.2.2 Pretraining and Finetuning

In pretraining, models are trained on large unannotated text datasets to learn general language representations. The model is trained to predict the next words in a sentence or fill in gaps in texts. A pioneering model in this area is the Transformer by [Vaswani, 2017], which significantly improved sequence processing through the introduction of the self-attention mechanism. GPT (Generative Pretrained Transformer) models [Radford, 2018, Radford et al., 2019] rely on predicting the next word in a sequence, enabling the generation of coherent texts.

After pretraining, the model is fine-tuned for specific tasks. It is trained on smaller, annotated datasets relevant to the respective application, such as sentiment analysis or machine translation. [Howard and Ruder, 2018] showed that fine-tuning pretrained language models significantly enhances performance in various NLP tasks.

Through this two-stage training strategy, LLMs can combine their general language understanding with specific expertise, leading to improved results in diverse application areas.

3.2.3 Tokens

Tokens are the smallest units into which text is divided in NLP. They typically represent words, word parts, or characters, depending on the tokenization method used [OpenAI, 2024c]. In LLMs, these tokens are used to convert text into a form that computers can process. The number of tokens an LLM can accept as input is called the context window. This indicates the length of the prompt used to calculate the subsequent tokens. These tokens are converted into numerical IDs and then transformed into complex numerical vectors, called embedding vectors, that capture the semantic meaning and context of the text [OpenAI, 2024a]. LLMs use these embeddings to understand linguistic patterns and make predictions for the next token, allowing

the models to generate texts or perform other linguistic tasks.
A token is approximately 4 characters, which corresponds to about 0.75 words on average [OpenAI, 2024j].
Here is an overview with the frequently mentioned token numbers. 124,000 Token are comparable with one average novel. Whereby, 2,000,000 Token are approximately 15 average novels.

Tokens	**Words**	**Unicode Characters**
100	~75	~400
4,000	~3,000	~16,000
16,385	~12,289	~65,540
124,000	~93,000	~496,000
200,000	~150,000	~800,000
2,000,000	~1,500,000	~8,000,000

Table 1: Comparison of Tokens, Words, and Unicode Letters [OpenAI, 2024j]

3.2.4 Vector Databases

Vector databases are specialized database systems optimized for storing, managing, and retrieving embedding vectors. This enables semantic similarity search through large amounts of data [LangChain, 2024a].
To generate an embedding for a text embedding-models are used [OpenAI, 2024a].

3.2.5 Companies and Models

In the following, companies and their LLM models are presented, that were relevant for the work and are used in this context for development. There are many other LLM models and developers, but listing and testing them all would exceed the scope. Therefore, the focus is on the leading and most advantageous LLMs for this use case. A good overview of the various models is provided by the Chatbot Arena, which is discussed in section 3.2.6. Many models are mentioned here because new major models were released throughout the development of the architecture. This also shows the incredibly rapid development in this field. Important properties of the models are shown in tabular form, which contributed to the selection of the ultimately used models. These are the context window and the maximum output, both calculated in tokens, the knowledge level, which indicates up to which date the training data extends, as well as input format and the input/output costs.
The costs are calculated in $ per million tokens.

In subsubsection 3.2.6, evaluation tests of the models are presented to classify the strength of the models.

OpenAI

Name	Latest Version	Context Window	Maximum Output	Knowledge Level	Input Format	Input/Output Costs
GPT-3	gpt-3.5-turbo-0125	16,385 Tokens	4,096 Tokens	Up to Sep 2021	Text	$0.50/$1.50
GPT-4	gpt-4-turbo-2024-04-09	128,000 Tokens	4,096 Tokens	Up to Dec 2023	Text/Image	$10.00/$30.00
GPT-4o	gpt-4o-2024-08-06	128,000 Tokens	16,384 Tokens	Up to Oct 2023	Text/Image	$2.50/$10.00
GPT-4o-mini	gpt-4o-mini-2024-07-18	128,000 Tokens	16,384 Tokens	Up to Oct 2023	Text/Image	$0.15/$0.60
o1-preview	o1-preview-2024-09-12	128,000 Tokens	32,768 Tokens	Up to Oct 2023	Text	$15.00/$60.00
o1-mini	o1-mini-2024-09-12	128,000 Tokens	65,536 Tokens	Up to Oct 2023	Text	$3.00/$12.00

Table 2: Comparison of GPT Models [OpenAI, 2024h, OpenAI, 2024g]

OpenAI is a leading research company in the field of artificial intelligence, founded in 2015 [OpenAI, 2024f]. Their goal is to develop advanced AI technologies that benefit humanity. OpenAI places a strong emphasis on safety and ethical standards, especially in the development of Artificial General Intelligence (AGI). Their most well-known developments include the GPT models, which are used for natural language processing and other AI applications [OpenAI, 2024f].

The company has released the models shown in Table 2. There are also different versions of the models, but for simplicity, the most recently released version is presented which was used in the work.
GPT-3 was the first model from OpenAI, publicly available under ChatGPT on 30.11.2022. It is an LLM capable of generating natural language and code and optimized for chat messages.
OpenAI released a special reasoning model called o1. This model uses chain-of-thought intermediate steps to prepare for solving complex tasks. It performed very well in difficult tests and benchmarks according to evaluations [OpenAI, 2024d].

Anthropic

Anthropic, a significant company in the field of artificial intelligence, plays an important role in the research and development of LLMs [Anthropic, 2024b]. The company is funded by

Name	Latest Version	Context Window	Maximum Output	Knowledge Level	Input Format	Input/Output Costs
Haiku	claude-3-haiku-20240307	200,000 Tokens	4,096 Tokens	Up to Aug 2023	Text/Image	$0.25/$1.25
Sonnet	claude-3-sonnet-20240229	200,000 Tokens	4,096 Tokens	Up to Aug 2023	Text/Image	$3.00/$15.00
Opus	claude-3-opus-20240229	200,000 Tokens	4,096 Tokens	Up to Aug 2023	Text/Image	$15.00/$75.00
Sonnet-3.5	claude-3-5-sonnet-20240620	200,000 Tokens	8,192 Tokens	Up to Apr 2024	Text/Image	$3.00/$15.00

Table 3: Comparison of Anthropic Models [Anthropic, 2024a]

Amazon [Jannis Brühl, 2023] and is known for its advanced techniques in AI, including the development and use of competing models to OpenAI's GPT series. The series is called Claude, and during the work, the models Haiku, Sonnet, and Opus were released under Claude 3.

Google

Name	Latest Version	Context Window	Maximum Output	Knowledge Level	Input Format	Input/Output Costs
Gemini 1.5 Pro	gemini-1.5-pro	2,097,152 Tokens	8,192 Tokens	Up to Nov 2023	Audio/Image/Video/Text	$3.50/$10.50
Gemini 1.5 Flash	gemini-1.5-flash	1,048,576 Tokens	8,192 Tokens	Up to Nov 2023	Audio/Image/Video/Text	$0.075/$0.30

Table 4: Comparison of Gemini Models [Google, 2024]

The large tech company Google is, of course, also involved in the AI arms race. With Gemini, it has released its own LLM series. The models are actually available for free via the API up to a certain rate limit. For more details, I refer to the sources [Google, 2024].

Meta

Name	Latest Version	Context Window	Maximum Output	Knowledge Level	Input Format	Input/Output Costs
LLama-3	llama-3-2024-07-18	8,000 Tokens	4,000 Tokens	Up to Dec 2023	Text	Local
LLama-3.1	llama-3.1-2024-07-18	128,000 Tokens	4,000 Tokens	Up to Dec 2023	Text	Local

Table 5: Comparison of LLama Models [Meta AI, 2024a, Meta AI, 2024b]

Meta, formerly Facebook, also has its own LLM models. They rely on open source and make their models largely freely available. Their model series is called LLama. The LLama models each have three different sizes: 8 billion parameters, 70 billion parameters, and 405 billion parameters [Meta AI, 2024a, Meta AI, 2024b].

3.2.6 Comparison and Evaluation

Benchmarks

There are various benchmark tests to examine the capabilities of chatbots in different areas. Here is an overview over import once. For more information, refer to the sources [Anthropic, 2024c, Meta AI, 2024c, OpenAI, 2024i].

MMLU: Measuring Massive Multitask Language

MATH: Measuring Mathematical Problem Solving With the MATH Dataset

GPQA: A Graduate-Level Google-Proof Q&A Benchmark

DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs

MGSM: Multilingual Grade School Math Benchmark (MGSM)

HumanEval: Evaluating Large Language Models Trained on Code

These test give an impression on the capabilities of the models. In the sources, big benchmarks from 3 big companies for the top models can be found.[4]

The benchmarks are used in this work to determine which model to use for which case.

Chatbot Arena

Rank* (UB)	Model	Arena Score	95% CI	Votes	Organization	License	Knowledge Cutoff
1	ChatGPT-4o-latest (2024-09-03)	1338	+3/-5	24135	OpenAI	Proprietary	2023/10
1	o1-preview	1335	+5/-5	13725	OpenAI	Proprietary	2023/10
3	o1-mini	1314	+5/-5	13988	OpenAI	Proprietary	2023/10
3	Gemini-1.5-Pro-002	1304	+7/-6	8872	Google	Proprietary	Unknown
4	Gemini-1.5-Pro-Exp-0827	1299	+4/-4	32414	Google	Proprietary	2023/11
5	Grok-2-08-13	1293	+4/-4	31887	xAI	Proprietary	2024/3
7	GPT-4o-2024-05-13	1285	+3/-3	96501	OpenAI	Proprietary	2023/10
8	GPT-4o-mini-2024-07-18	1273	+4/-4	36122	OpenAI	Proprietary	2023/10
8	Gemini-1.5-Flash-Exp-0827	1269	+3/-4	25520	Google	Proprietary	2023/11
8	Claude 3.5 Sonnet	1268	+3/-3	71813	Anthropic	Proprietary	2024/4
8	Grok-2-Mini-08-13	1268	+4/-3	27913	xAI	Proprietary	2024/3
8	Meta-Llama-3.1-405b-Instruct-bf16	1267	+6/-5	11826	Meta	Llama 3.1 Community	2023/12
8	Gemini-1.5-Flash-002	1265	+9/-7	6339	Google	Proprietary	Unknown
8	Meta-Llama-3.1-405b-Instruct-fp8	1266	+3/-4	36694	Meta	Llama 3.1 Community	2023/12
9	Gemini Advanced App (2024-05-14)	1266	+3/-3	52211	Google	Proprietary	Online
9	GPT-4o-2024-08-06	1264	+4/-3	28238	OpenAI	Proprietary	2023/10
13	Gemini-1.5-Pro-001	1259	+3/-3	86453	Google	Proprietary	2023/11
15	Qwen2.5-72b-Instruct	1257	+5/-5	11306	Alibaba	Qwen	2024/9

Figure 2: Leaderboard of the Chatbot Arena [LMSYS, 2024]

The website "Chatbot Arena" on lmarena.ai offers a platform where various Large Language

[4] anthropic.com/news/claude-3-5-sonnet; github.com/openai/simple-evals; ai.meta.com/blog/meta-llama-3-1/

Models (LLMs) can be evaluated in a direct comparison. Users can choose between two models in anonymized duels, deciding which one they believe provides the better answer. The ratings are based on the Elo rating system, known from chess, and used to calculate the relative strength of the models. In Figure 2 there is the Leaderboard with the Elo scores displayed
A main goal of the platform is to provide a scalable and user-friendly system to evaluate the quality of LLMs in open and unpredictable scenarios, which is often challenging through traditional benchmarking methods [LMSYS, 2024].

3.2.7 API vs. Local

When utilizing Large Language Models (LLMs), users face a choice between API-based and locally hosted models. API LLMs, provided by companies such as Google or OpenAI, are easy to integrate and remain up-to-date as updates are automatically managed by the provider. They offer scalability, as the computing power is supplied by the provider, and can be cost-efficient due to usage-based fees replacing hardware and maintenance costs. Moreover, they often outperform smaller open-source models [Abid Ali Awan, 2024, Community, 2024].
However, API LLMs create dependency on the provider; outages or price changes can affect operations. There are also privacy and security concerns, as data must be transmitted to external servers, posing potential risks [Abid Ali Awan, 2024, Hafsa Jabeen, 2024]. Additionally, data transfer over the internet can lead to higher latency.
Local LLMs, on the other hand, offer full control and customization options, including fine-tuning for specific use cases. Since all data is processed locally, privacy risks are reduced, which is particularly important for sensitive information. Furthermore, there is no reliance on external providers [Timothy, 2023].
The drawbacks of local LLMs include high initial costs for hardware and maintenance, as well as technical challenges in implementation and optimization. Scalability may be limited, as necessary resources must be locally available. Although smaller models can run on standard computers through quantization techniques, they often do not achieve the desired performance [Hafsa Jabeen, 2024, Community, 2024].
In summary, API LLMs are well-suited for development and exploration, as they are ahead of open-source models, easily accessible, and require no upfront investment. However, for robotic applications, smaller local LLMs may be preferred, as robots should not rely on internet connections or experience latency, nor should their data be transmitted to external providers.

With models like LLama, significant improvements and compression of small local LLMs are observed. Smaller LLMs can even run on conventional gaming GPUs through quantization.

3.3 Prompting Methods

A key component of the work is the application of various prompting methods to improve the performance of LLMs. A prompt is the input for an LLM. The prompt is the basis on which the next token is determined. The prompt is therefore a crucial factor in directing the output of an already trained LLM. In the following, I present various techniques that have been investigated to achieve optimal outputs.

3.3.1 Few-Shot

A simple task or question in natural language is also called a zero-shot prompt. A one-shot prompt, on the other hand, is given an example of a similar task before the task itself. Few-shot prompts have more than one example.
Using few-shot helps the LLM better match the desired output in style and content. [Brown, 2020]

3.3.2 Role-Play

In a role-play prompt, the Large Language Model is instructed to assume a specific role and scenario, which should be directly related to the task at hand. A paper demonstrates that this method has significant positive effects on reasoning tasks relevant to the assigned roles [Kong et al., 2023].

3.3.3 Chain-of-Thought

Chain-of-Thought Prompting [Wei et al., 2022] is a method that enables language models to perform complex reasoning by imitating the human thought process in solving multi-step problems. Instead of generating an answer directly, language models are encouraged through chain-of-thought prompting to produce a series of intermediate steps—a "thought chain"—that leads to the final solution. This method is based on the observation that humans break down complex tasks, such as math word problems, into smaller, more manageable steps. An example of this would be solving the problem how many flowers Jane still have "After Jane gave her mother 2 flowers, she still has 10 ... after giving her father 3, she still has 7 ... so the answer is 7." Chain-of-Thought Prompting attempts to transfer this step-by-step approach to language models [Wei et al., 2022, S. 2–3]. By providing examples that demonstrate the solution path in the form of a thought chain, LLMs can learn to generate similar chains for new, unknown problems. A chain-of-thought can also be zero-shot by appending "Think step by step." to the prompt. The Role-Play Prompts can also trigger CoT [Kong et al., 2023].

Chain-of-Thought Prompting offers several advantages:

LLMs can break down multi-step problems into smaller, more manageable steps, allowing more computing power to be distributed to more complex tasks.

The generated thought chain provides insights into the model's solution path and allows for the identification of errors in the thought process.

Chain-of-Thought Prompting can be used for various tasks, such as mathematical word problems, reasoning based on everyday knowledge, and symbolic manipulation.

3.3.4 Automatic Chain of Thought

Auto-CoT [Zhang et al., 2022] offers an effective method to enhance the performance of LLMs without manual intervention. It combines chain-of-thought prompting with k-shot prompting. By intelligently selecting examples and automatically generating thought chains for each of these examples, the process of chain-of-thought prompting is optimized. This method allows for leveraging the reasoning capabilities of LLMs in a scalable and efficient manner, which is particularly advantageous in applications with large datasets.

3.3.5 Retrieval Augmented Generation (RAG)

Particularly important is the method of Retrieval Augmented Generation (RAG) [Zhao et al., 2024], where information is retrieved from external data sources and integrated into the generation process. This allows for increasing the accuracy and relevance of the generated instructions. It also circumvents limitations due to a low context window and ensures that predominantly relevant information is used, saving costs. In Figure 3 there is a visual expression on how the architecture is played out. Vector databases and embeddings play a crucial role in this, as they enable efficient storage and quick retrieval of relevant information. [Zhao et al., 2024]

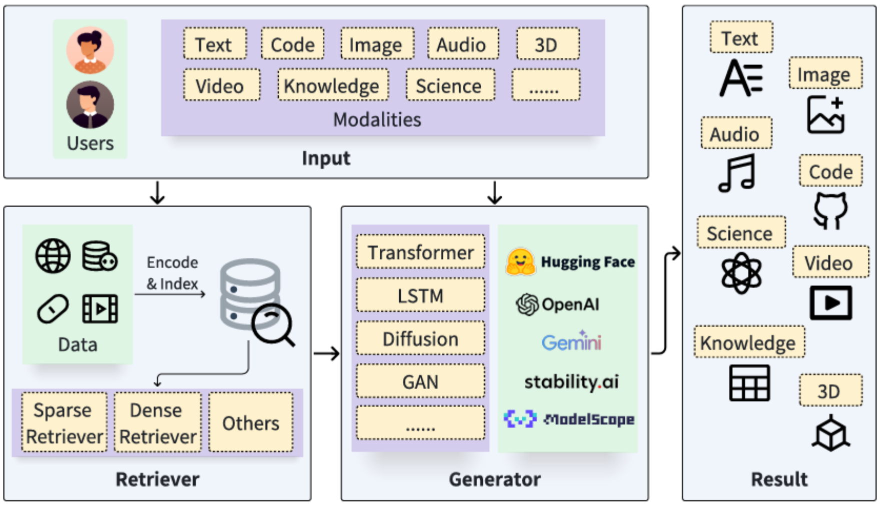

Figure 3: The RAG Architekture [Zhao et al., 2024, P.2]

3.3.6 Tool Augmented Agent

Just like with RAG, results from external functions and tools can be integrated into the LLM's prompt. An LLM can also be given a description of the function and then independently call it when needed, using the result for the generation process.

3.3.7 ReAct

ReAct [Yao et al., 2022] is a paradigm that combines thinking and acting in language models to solve a variety of tasks in the field of language understanding and decision-making. The model alternately generates verbal thought traces (thoughts) and actions related to a task.

Thinking helps the model reflect on the current context, create plans, and integrate knowledge from previous observations.

Acting means that the model performs concrete actions to influence the environment, such as retrieving additional information. This interplay allows the model to develop dynamic and flexible problem-solving strategies that incorporate both internal considerations and external environmental information. ReAct has proven to be particularly effective as it increases the interpretability and robustness of the model in various task areas.

Langgraph also has a tutorial for a ReAct type Agent called Plan-and-Execute, which can also be explored in the future [LangChain, 2024d].

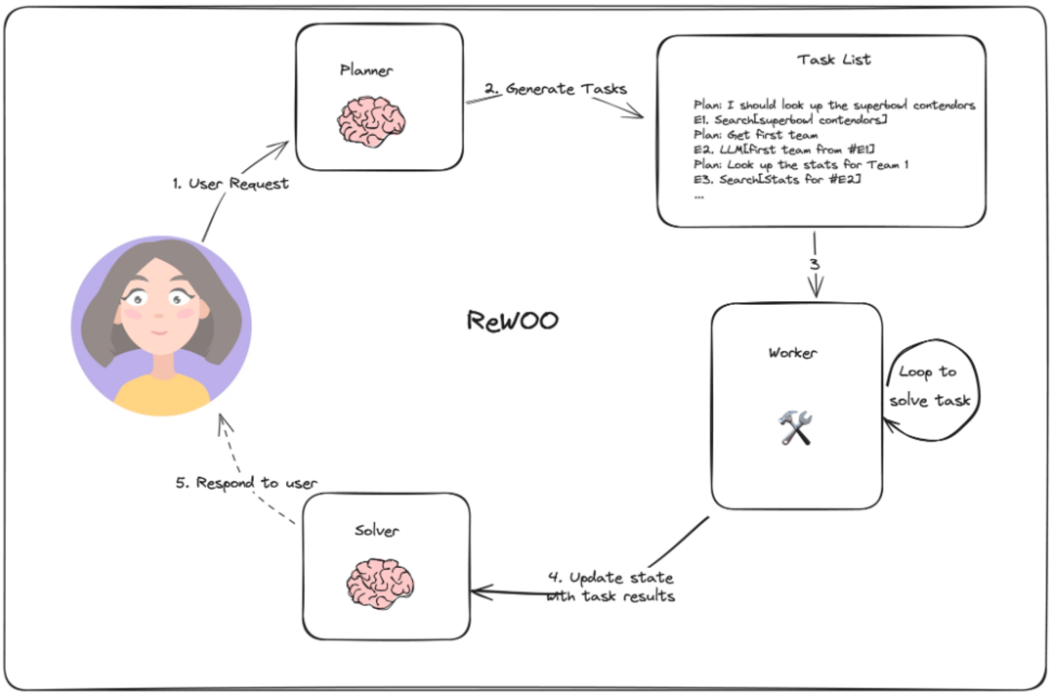

Figure 4: Representation of the ReWOO Architecture [LangChain, 2024]

3.3.8 ReWOO (Reason Without Observation)

Another important aspect of the work is the use of the ReWOO architecture. This model was described in a scientific paper and offers innovative approaches to solving complex problems through LLMs [Xu et al., 2023]. In Figure 4, a graphical overview of the architecture can be seen. ReWOO is divided into three separate modules, as also seen in the figure: Planner, Worker, and Solver.

The Planner breaks down a task into individual steps and creates a plan, which is then passed on to the Worker.

The Worker retrieves external knowledge or tools to provide evidence.

The Solver finally integrates all plans and evidence to generate the final answer to the original task.

ReWOO separates the thought process of the language model from the external tools, reducing the redundancy of interlocking prompts and increasing the efficiency of prompting [Xu et al., 2023, S. 2]. There is already a tutorial on the rough structure of this model in LangGraph [LangChain, 2024].

4 Methodology

4.1 Development of the LLM Pipeline

The methodology of this work focuses on the development and implementation of an LLM pipeline for generating, testing, and automatically adapting robot plans. The process is divided into several phases:

4.1.1 Preliminary Investigation

Here, initial tests with GPT-4 in zero-shot mode are conducted to check whether existing models can already generate PyCRAM plans without specific training.
Under subsubsection 10.2.1 and subsubsection 10.2.2, tests of GPT-4o-2024-08-06 can be found, once with zero-shot and the second with zero-shot-CoT. In a prompt, the very simple instruction "Pick-up the bowl from the table." was given with corresponding world knowledge.
The result makes it clear that information about PyCRAM was present in the training data. However, the generated code was incorrect in both cases and consistently contained hallucinations.
Subsequently, one-shot prompting and creating a custom GPT at OpenAI were tried, as it was clear that knowledge about the functions and code of PyCRAM was lacking. By providing code examples and the built-in retriever function in the custom GPT, it was investigated how providing context affects the output.
The one-shot test can be found under subsubsection 10.2.3 and the custom GPT test under subsubsection 10.2.5.
These attempts showed significant progress. Plan code for similar tasks as in the examples could be created well. The LLM needs more context-based information about the environment and PyCRAM and its function. For this, a well-thought-out multi-agent architecture was developed, combining proven prompting methods.

4.1.2 Development of a Complex Architecture

For the development of the architecture, LangChain and the related LangGraph are used to easily access the various API interfaces of the LLMs and chain the calls. This also supported the integration of a vector database and setting up a retriever. The vector database retriever is needed to insert the appropriate information about the PyCRAM framework into the LLM's

prompt.
User instructions in natural language can contain very abstract and complex action requirements, so an LLM call should first break them down.
For such a component that translates natural language inputs from the user into a plan structure, two different architectures were considered: the ReAct architecture 3.3.7 and the ReWOO architecture 3.3.8. Both approaches aim to break down the instructions given by the user into structured, processable plans that can then be executed. However, there are significant differences in how the respective architectures realize this task. In ReAct, the tasks are executed sequentially, and the result is appended to the prompt for solving the next task. This is how the plan is processed, and a result is produced at the end. ReWOO, on the other hand, executes tools for the tasks sequentially but separately and then inserts the result as evidence into the plan structure. The filled plan is then used by a final call for problem-solving. This saves tokens.

At the beginning of the work, the costs of the API calls were still very high. Therefore, due to the higher efficiency in resource usage and the associated reduction in costs, the ReWOO architecture was chosen for the project.
The tools are then the various vector database retrievers. There is a tool for the documentation, for the source code of PyCRAM, and for the URDF files of the objects and environments in the BulletWorld. In the tools, the results of the retriever are given in a call to a fast, inexpensive LLM to compress the content and limit it to what is important for solving the task.
In addition to ReWOO, the architecture was extended by integrating a variant of few-shot Auto-CoT. A vector database in which examples are stored. These examples already contain pre-made plans and the code that should be generated at the end of the execution. In addition, the corresponding user instructions and the description of the environment in which the plan is executed are stored in the database.
The vector database serves as a reference for the planner agents of the ReWOO architecture. During planning, the planner accesses the examples in the database to learn the structure and logic of the planning. Examples that are semantically related to the new task are selected. This makes the planning process more efficient and precise, as the agent can access a variety of successful, already executed examples.
To fill the vector database with relevant examples, suitable cases were extracted from the documentation in advance and plans were created in collaboration with ChatGPT. These plans were

included in the database and now serve as the basis for the planner agent. In addition, the database is designed to be continuously expanded.
This dynamic expansion of the vector database supports the continuous improvement of the planning process and ensures that the system can create increasingly precise and adaptable plans over time.

4.1.3 Extension through Autocorrection

Despite the successful implementation of the ReWOO architecture and its ability to generate promising plans, the generated code could not be executed without errors. To address this issue, an additional autocorrection architecture was developed and integrated into the system. This architecture is based on a loop structure that executes the generated PyCRAM plan, detects any errors that occur, and makes appropriate corrections. It is based on the code assistant on the Langgraph tutorial site [LangChain, 2024c].
The autocorrection architecture works as follows: If a code contains errors, it is returned with the error code to a slightly modified version of the ReWOO architecture. This version is tasked with reviewing the faulty code and making appropriate corrections. This process is iteratively performed until either the specified number of iterations is reached or the code can be executed without errors.
Once the code has been successfully executed without errors, it is automatically merged and re-integrated into the Auto-CoT system. The error-free code is then stored as a new example in the plan example database. In this way, the database continuously grows with additional, functional examples that can be used as a basis for future planning.
This loop structure ensures that the generated plans not only look good but can also be executed without errors. Through iterative correction, the system continuously improves and enables the resolution of more complex errors while simultaneously learning to generate error-free plans and code examples.

Prethinking

The appearance of the o1 reasoning model led to another idea. A preprocessing step for the user instruction was added before the ReWOO model call. This step is intended to break down the instruction with an LLM and formulate it as a planning problem in natural language. The world knowledge and URDF files are used to narrow down the initial and target states. Although the

ReWOO planner also breaks down the task, its main goal is to obtain information for implementing individual steps. o1 has shown that many complex problems can be better solved by thinking through the task beforehand.

4.1.4 Finetuning for PyCRAM

In the preliminary tests, it was observed that there were strong hallucinations in zero-shot and significant improvements in one-shot. The assumption was that PyCRAM is only represented in the training data through research papers that briefly introduce it, and the code and documentation are not present there. Therefore, finetuning the models with the documentation and code could give the LLM a much better understanding.
Towards the end of the work, finetuning of GPT-4o and GPT-4o-mini was released, so this approach could also be pursued. For this, code, documentation, and examples were translated into data points in JSONL format. A data point is composed of a list of messages. The List should have at least three messages one system message, which defines the role, one user message with the instruction and one assistant message where the preferred answer of the LLM is filled. These were then used to extend the models on the OpenAI API page. The JSONL file can be found on GitHub under 'data/finetuning'.
Zero-shot tests were also conducted with the finetuning models. The tests can be found under subsubsection 10.2.6 for GPT-4o and under subsubsection 10.2.7 for GPT-4o-mini. The same prompt as for the test with the standard models was used to compare performance. The results were promising. In the test with GPT-4o, correct imports were used, and the code looked more like PyCRAM code compared to the previous zero-shot test. However, it still did not work because, for example, 'with simulated_robot:' was not used, which is important for execution.
The test with the GPT-4o-mini variant was surprising. Unfortunately, it also did not work and had obvious errors, even in the imports. However, 'with simulated_robot:' was used, and the code in the block already looked similar to a functioning PyCRAM plan. Although motion designators were used, which are undesirable for this task.

The tests left a predominantly positive impression. It was seen that the models had a much better understanding of the structure and implementation of a PyCRAM plan than the standard versions. The finetuning dataset was by no means perfect. It was a first test, and there is still a lot of potential here.

There was also hope that the finetuning models, with their better understanding of PyCRAM, could positively influence the architecture. Therefore, a test was conducted after the main test of the architecture. You can find this under subsection 6.2.

4.2 Final Architecture

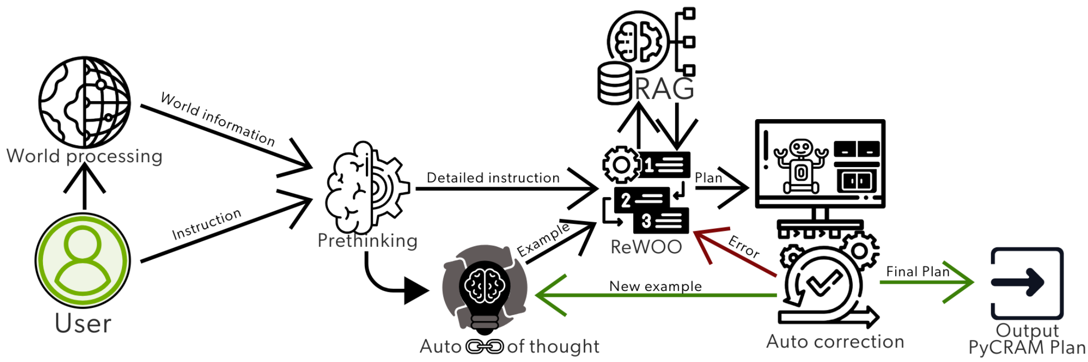

Figure 5: Simple overview of the Architecture

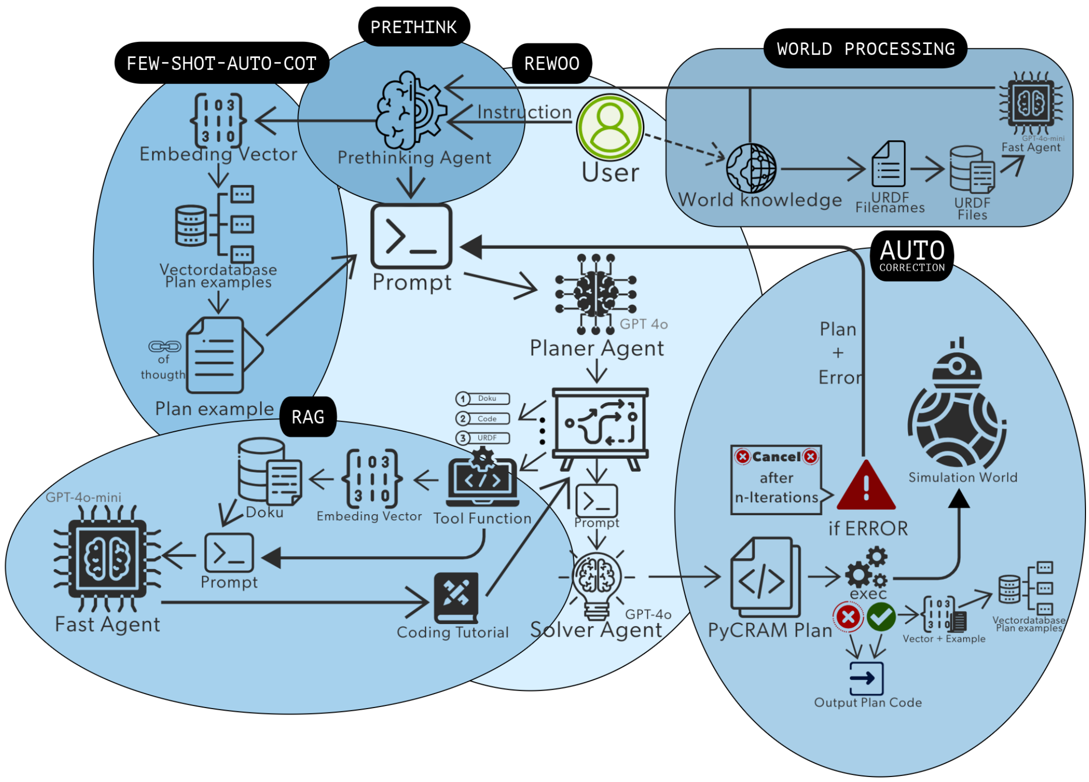

Figure 6: Detailed overview of the Architecture

Figures Figure 5 and Figure 6 provide a visual overview of the composite pipeline. Figure 5 offers a simple, concise representation of the architecture, while Figure 6 further breaks down the components shown. The following section summarizes the structure and interplay of these elements.

The architecture begins with a user prompt. Additionally, knowledge about the objects, the robot and the environment is required, which is also provided by the user for simplicity in this context. With this domain knowledge, the relevant URDF file for the environment is retrieved from a database and compressed into natural language using an affordable LLM. The domain knowledge, compressed URDF, and user instruction are then combined by the pre-thinking agent to form a detailed instruction. This prompt is converted into an embedding vector, which is used to retrieve a similar plan example from a vector database. This plan example, along with the user prompt, is converted into a prompt template, which serves as input for the planning agent.

The planning agent is an LLM that, based on the prompt template, generates a step-by-step plan to create the PyCRAM plan. It breaks down the instruction into smaller action steps and plans their implementation in PyCRAM code. Each step of the plan consists of a description, a tool, and an input for the tool.

The corresponding tools are executed for each plan. The tool instruction is converted into an embedding vector to find the relevant texts for the tool instruction from the corresponding vector database for the tool. These texts are then passed to an inexpensive language model, such as GPT-4o-mini, to fulfill the instructions. In the documentation tool, this would be a coding tutorial added at the appropriate point in the plan. In the coding tool, it would be an instruction for using a PyCRAM element, and in the URDF tool, a description of the important data of the specific URDF file for the task. This way, the plan steps are provided with helpful sources.

The solver agent generates the corresponding PyCRAM code using the filled plan. This PyCRAM code is then executed to check if the plan is functional. Successful PyCRAM plans are inserted into the plan example vector database. If the PyCRAM plan does not work, an error code is output. The error code and the generated plan code are then passed to another iteration with adjusted prompt templates to review and correct it. This process of reviewing and correcting the PyCRAM plan is repeated until the code runs error-free or a certain number of iterations is reached.

4.3 Test Procedure

At the end of the development process, a comprehensive test procedure was conducted to evaluate the efficiency and reliability of the developed pipeline. The individual components were tested at the beginning. Furthermore, various test scenarios were defined, and evaluation criteria were established. It should be noted that this is not the only test, but testing was conducted repeatedly throughout the development phase. Documenting all tests would have significantly exceeded the capacities. Especially with the prompts, you notice aspects that could still be adjusted and improved in every test.

4.3.1 Component Test

The architecture consists of several components and LLM calls. Therefore, it makes sense to test them individually before the main test. This also makes the interaction between them more apparent.

Retriever

First, the retrievers of the tools were tested individually. Short questions about PyCRAM were asked which could be generated by the planner agent, and the next two entries from the respective database were retrieved for the question.

For the documentation tool, the retriever was given the question "What are Action Designators and how are they used?". The output can be found in the appendix under subsubsection 10.2.8. The first page on `Action Designators` and the second page on `Designators` were returned. The pages match the question. This shows that the retriever can search for information about the question from the documentation.

For the code tool, the retriever was given the question “How is CostmapLocation defined?”. The output is documented under subsubsection 10.2.10. The first document contains the content from the file 'src/pycram/designators/specialized_designators/location/database_location.py', and the second document contains the content from the file 'src/pycram/designators/location_designator.py'. The first file does not exactly contain what is being searched for but defines `AbstractCostmapLocation` and `DatabaseCostmapLocation`. Since `CostmapLocation` appears frequently in the file, the content was semantically more similar than the content of the second file. The second file contains the definition of CostmapLocation. Thus, the definition of CostmapLocation was found, even if only on the second attempt.

Documentation Tool

The documentation tool was tested with the same question as the associated retriever. For this tool, the first four documents from the retriever are used. The test was conducted under the context of Test 4 defined in subsubsection 4.3.2. The test output can be found under subsubsection 10.2.9.

A detailed and structured tutorial on action designators is output. It lists the important action designators and provides a code snippet with examples of their use in the context of the task. Additional tips and information are also provided, which are not necessarily needed. Overall, this coding tutorial makes a helpful and correct impression. The information from the retriever was well used. It is clear that context in the prompt significantly improves the LLM's capabilities. However, errors, minor hallucinations, and irrelevant information can also occur here. Since this tool is applied multiple times in the architecture, it can be assumed that correct information predominates and is preferred due to overlaps.

Code Tool

The code tool was also tested with the same question as the associated retriever. For this tool, the first four documents from the retriever are also used. This test was also conducted under the context of Test 4. The test output can be found under subsubsection 10.2.11.

The output contains a description of the `CostmapLocation` class, along with associated parameters, the full code, a demonstration of usage, and further explanations of the class. Again, there is information that is not useful. Overall, the report on the class provides a thorough overview and offers all the necessary hints to use this class.

URDF Tool

The URDF tool was given 'kitchen.urdf' as input. This test was also conducted under the context of Test 4. The test output can be found under subsubsection 10.2.13. It first describes the content of the URDF file, then provides important data for the task, and translates the content into a compressed world model in natural language. In this case, the task was not satisfactorily executed, as the correct names of the links in the URDF file and their relative positions were not mentioned. These are important for accessing components of the kitchen. As a result, the world model is also superfluous, as only vague or already known information is summarized.

PreThinking

The test for the PreThinking component was also conducted under the context of Test 4. The test output can be found under subsubsection 10.2.12. In the output, an initial state, a goal state, and a step-by-step plan in natural language are defined for the task.

The positions were estimated, and the generated plan appears promising. However, since inaccuracies may certainly occur, the output is accompanied by a disclaimer stating that these are preliminary ideas and should not be considered entirely accurate.

Conclusion of Component Tests

Overall, the components perform their tasks satisfactorily. However, there is room for improvement in terms of robustness, layout, and the conciseness of essential information. In particular, the URDF tool should be adjusted, as it does not consistently produce accurate outputs and, as seen in the test, occasionally omits link names.

4.3.2 Definition of Test Cases

Creation of eight test cases covering different scenarios and requirements.

Since the tests are planning problems, they consist of a prompt that describes the target state in natural language and the world knowledge that defines the initial state and the list of objects.

Use of the defined test cases for comprehensive verification of pipeline functionality.

Test 1. Task Known from Documentation

Prompt: 'Place the cereal box on the kitchen island.'

World Knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL,'breakfast_cereal.stl', pose=Pose([1.4,
↪  1, 0.95]))]
```

Justification: This test case serves as a baseline and checks whether the pipeline can solve a simple task described in the documentation.

Test 2. Single, Non-Abstract Task

Prompt: 'Move to position (-2.5,1,0).'

World Knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')]
```

Justification: Tests the basic navigation ability of the robot.

Test 3. Small and Simple Task

Prompt: 'Pick up the bowl from the table.'

World Knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0,
↪ 1])]
```

Justification: Similar to the previous test case, but with a different object and action.

Test 4. Medium Task

Prompt: 'Place cereal and a bowl side by side on the kitchen island.'

World Knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 0.50, 0.89]), color=[1, 1,
↪ 0, 1]),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL, 'breakfast_cereal.stl', pose=Pose([1.4,
↪ 1, 0.95]))]
```

Justification: Increased complexity due to the manipulation of multiple objects.

Test 5. Complex Task

Prompt: 'Prepare breakfast. I want cereal with milk.'

World Knowledge:

```
[robot = Object("pr2", ObjectType.ROBOT, 'pr2.urdf', pose=Pose([1, 2, 0])),
apartment = Object('apartment', ObjectType.ENVIRONMENT, 'apartment.urdf'),
milk = Object('milk', ObjectType.MILK, 'milk.stl', pose=Pose([2.5, 2, 1.02]), color=[1, 0, 0,
↪ 1]),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL, 'breakfast_cereal.stl', pose=Pose([2.5,
↪ 2.3, 1.05]), color=[0, 1, 0, 1]),
spoon = Object('spoon', ObjectType.SPOON, 'spoon.stl', pose=Pose([2.4, 2.2, 0.85]), color=[0,
↪ 0, 1, 1]),
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([2.5, 2.2, 1.02]), color=[1, 1,
↪ 0, 1]),
apartment.attach(spoon, 'cabinet10_drawer_top')]
```

Justification: Tests the understanding of complex tasks that require multiple subtasks. A plan for this is in the examples, but with a different prompt.

Test 6. Task with Spatial Relationship

Prompt: 'Place the cereal box directly next to the refrigerator.'

World Knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL, 'breakfast_cereal.stl', pose=Pose([1.4,
↪ 1, 0.95]))]
```

Justification: Tests the understanding of spatial relationships.

Test 7. Task with Unknown Object

Prompt: 'Pick up the thing from the kitchen counter.'

World Knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
spoon = Object('spoon', ObjectType.SPOON, 'spoon.stl', pose=Pose([1.4, 1, 0.87]), color=[0, 0,
↪ 1, 1])]
```

Justification: Tests the pipeline's ability to handle unknown objects.

Test 8. Task with Negative Instruction

Prompt: 'Pick up a spoon, but not one that is already on a table.'

World Knowledge:

```
[robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf', pose=Pose([1, 2, 0])),
apartment = Object('apartment', ObjectType.ENVIRONMENT, 'apartment.urdf'),
milk = Object('milk', ObjectType.MILK, 'milk.stl', pose=Pose([2.5, 2, 1.02]), color=[1, 0, 0,
↪ 1]),
spoon1 = Object('spoon1', ObjectType.SPOON, 'spoon.stl', pose=Pose([2.4, 2.2, 0.85]),
↪ color=[0, 0, 1, 1]),
spoon2 = Object('spoon2', ObjectType.SPOON, 'spoon.stl', pose=Pose([2.5, 2.3, 1.00]),
↪ color=[0, 0, 1, 1]),
spoon3 = Object('spoon3', ObjectType.SPOON, 'spoon.stl', pose=Pose([0.4, 3.1, 0.96]),
↪ color=[0, 0, 1, 1]),
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([2.5, 2.2, 1.00]), color=[1, 1,
↪ 0, 1]),
apartment.attach(spoon1, 'cabinet10_drawer_top')]
```

Justification: Tests the understanding of negative instructions and the ability to distinguish objects.

4.3.3 Evaluation Criteria

To evaluate the tests, the following criteria are used:

Termination: Did the code run error-free? If yes, the number of loop steps needed is also given.
Success: Was the task successfully completed?
Efficiency: How long did the solution take?
Code Quality: How good was the generated code?
Costs: How much did the API calls cost?

These criteria for the tests allow the current state of the architecture to be evaluated and weaknesses in quality, capability, as well as time and cost identified.

5 Implementation

In the following, the implementation phase is presented. The name of the python file with a link to the repository is provided for specific code.
First, ROS was set up, followed by the installation of PyCRAM according to the documentation.
For the implementation of the LLM pipeline, a Python repository was created, with Langchain and Langgraph chosen as the main packages due to their ability to facilitate API calls, the creation of LLM agents, and their networking. Initially, an attempt was made to create a simple RAG agent. During this process, the ReWoo architecture was discovered and replicated based on a tutorial from the Langgraph documentation page. This was then adapted to the use case by rewriting the LLM chains and modifying the prompts (src/langgraph_ReWOO.py).
Prompt design also played a crucial role in the implementation. Known prompt engineering techniques were integrated into the prompt templates to ensure that the LLM understood its task optimally.
Next, a vector database was set up using Supabase, and the interface to it was added in Python (src/vector_store_SB.py). This allowed for the setup of a RAG chain. A script was developed to scrape the PyCRAM documentation page and load it into the vector database (code_scraper.py). After several tests, the prompts were further adjusted, and another vector database was created for code examples, which an agent automatically generated.

Since the code was still prone to errors, a more complex graph structure was implemented, which utilizes the ReWoo architecture for code generation and then simulates and tests the generated code in the Bullet World. If an error occurs, it is caught, and a new iteration is initiated. To achieve this, a tailored version of the ReWoo architecture was developed to optimize code correction. The code, the error message, and the original user task are passed along. These two versions of the ReWoo architecture were integrated into another Langgraph architecture, creating a loop in which code is generated, executed, and corrected if necessary. This loop continues until either the generated code runs without errors or a pre-defined number of iterations is reached (src/langgraph_code_assistant.py). Additionally, this code correction architecture was equipped with a vector database containing the entire PyCRAM codebase.
During the implementation phase, new LLM models from various providers were continuously being released. Initially, Anthropic introduced three new models with Claude 3. Of particular

interest was the Haiku model, which surpassed GPT-3.5-Turbo in terms of costs, input window size, and performance benchmarks. Previously, GPT-3.5 had been used in the tools to reduce costs. However, due to the low rate limit of Anthropic, a try-catch block was implemented to catch rate-limit errors and fall back to GPT-3.5.

To evaluate how well local models performed in this use case, LLama 3 (70 billion parameters) was set up on a GPU server at IAI using Ollama. The LLM was integrated into the architecture via an SSH tunnel and Python script. Unfortunately, the model was unable to process the complex prompts satisfactorily, had much longer response times, and its context window was too small. Additionally, it exhibited increased hallucinations, making the cost benefit less relevant (src/run_llm_local.py).

In discussions with the tutors and through the review of the generated codes, it became evident that the model lacked an understanding of world objects, leading to the creation of another vector database with the URDF files of the environments. Shortly afterward, OpenAI released the GPT-4o model, and Anthropic responded with the Sonnet 3.5 model. The introduction of GPT-4o-mini led to the replacement of Anthropic Haiku and GPT-3.5. Previously, Google's models had been overlooked, but with the release of the compelling Gemini-1.5-pro-exp model, the decision was made to integrate the Google API as well.

GPT-4o-mini was added as an inexpensive model for the tools. Finetuning for GPT-4o and GPT-4o-mini was made available, so the databases were updated to the latest PyCRAM update, and a finetuning dataset was compiled.

Since executing the tools sequentially took a long time and the sequential execution of the plan steps was not used, the ReWOO architecture was switched to parallel execution (src/ReWOO_parallel.py) (src/ReWOO_codeCheck_parallel.py) (src/langgraph_code_assistant_parallel.py). Additionally, a prethinking step was added, which initially considers the user instruction, which can also be abstract, complex, and unclear, with regard to the world. The URDF tool always returned the entire URDF, making the ReWOO plan very long. To save expensive tokens, the URDF files were also preprocessed by an inexpensive model in the context of world knowledge and instruction. Finally, logging for the tests was expanded, and the final models were determined for this. The model for the solvers was 'gpt-4o-2024-08-06', due to good ratings in the Chatbot Arena and promising coding benchmarks.

The model for the planners was 'gemini-1.5-pro-exp-0827', as it was available for free up to a certain rate limit and also had a good rating in the Chatbot Arena. For the tools, as already

mentioned, ’gpt-4o-mini’ was used. This is strong in terms of price-performance ratio.
All the current prompts and tool chains are stored in ’src/prompts.py’ and the LLM integrations are in ’src/helper_func.py’

6 Evaluation

6.1 Test Evaluation

Test	Result	Termination	Success	Runtime (m:s)	Costs	Code Quality	Further Observations
Test 1	GitHub Test 1	Yes on 1	Yes	1:46	0.05 $	Clean, functional code	Code is identical to the example
Test 2	GitHub Test 2	Yes on 2	Yes	2:31	0.09 $	Clean, functional code	First test iteration remained the same
Test 3	GitHub Test 3	No	Yes	3:44	0.16 $	Error in code	Error self-corrected, then 'IKError'
Test 4	GitHub Test 4	Yes on 1	No	2:02	0.14 $	Clean, running code	Object placed inside the cereal
Test 5	GitHub Test 5	No	No	8:10	0.21 $	Good code, but incorrect position recognition	Similar to the demo in the documentation
Test 6	GitHub Test 6	No	No	3:41	0.13 $	Error in position recognition	Access to list instead of attribute
Test 7	GitHub Test 7	Yes on 1	Yes	1:43	0.17 $	Clean, functional code	Correct conclusion about 'thing' = spoon
Test 8	GitHub Test 8	No	No	04:12	0.19 $	Error due to undefined function	Incorrect conclusion about the spoon

Table 6: Table of Test Evaluations

The table includes the evaluation criteria established under subsubsection 4.3.3. The outputs and results of the intermediate steps of the tests can be found in the GitHub repository. The pages are linked in the 'Result' column. Under Code Quality and Observations, a brief summary is provided in the table. A more detailed examination follows:

Test 1

Code Quality:
The code is clean and runs successfully. The task is also executed correctly. The code is uncommented.

Observation:
The exact task is present in the example database. The code in the database and the output code are mostly the same.

Test 2

Code Quality:
Clean code quality and the task is successfully executed. The code is uncommented.

Observation:
The code from the first iteration remained the same, even though the test did not seem to run there.

Test 3

Code Quality:
The code does not run. The error `TypeError: TransportAction.init() got an`

`unexpected keyword argument 'object_designator'` was output. Here, 'object_designator_description' was written instead of 'object_designator'. The code is uncommented.

Observation:

Despite the errors in the code, the task was executed. The error that occurred was actually self-corrected, but then an `IKError` occurred. The correction step apparently could not handle this and instead reversed its positive correction.

Test 4

Code Quality:

The quality of the code is clean and it runs through. The code is commented.

Observation:

Despite the running code, the task was not fulfilled. The bowl is placed inside the cereal instead of next to it.

Test 5

Code Quality:

In general, the code looks good, but 'SemanticCostmapLocation' is used to determine positions, which is not suitable in this case. It is also assumed that `cereal_box` and milk_carton' are part of the `apartment`, even though they were previously set as objects and object designators under the correct names. The following error code is output:

```
---------------------------------------------------------------------------
AttributeError Traceback (most recent call last)
Cell In[1], line 32
29 MoveTorsoAction([0.25]).resolve().perform()
31 # Retrieve the spoon
---> 32 cabinet_location = SemanticCostmapLocation(urdf_link_name='cabinet10_drawer_top', part_of=apartment).resolve()
33 NavigateAction(target_locations=[cabinet_location.pose]).resolve().perform()
34 OpenAction(object_designator_description=ObjectPart(names=['handle_cab10_t'], part_of=apartment),
    arms=[Arms.RIGHT]).resolve().perform()
File ~/ros/ros_ws/src/pycram/src/pycram/designators/location_designator.py:321, in SemanticCostmapLocation.ground(self)
315 def ground(self) -> Location:
316 """ 317 Default specialized_designators which returns the first element of the iterator of this instance.
318 :return: A resolved location
319 """
--> 321 return next(iter(self))
File ~/ros/ros_ws/src/pycram/src/pycram/designators/location_designator.py:331, in SemanticCostmapLocation.__iter__(self)
323 def __iter__(self):
324 """ 325 Creates a costmap on top of a link of an Object and creates positions from it. If there is a specific Object for
326 which the position should be found, a height offset will be calculated which ensures that the bottom of the Object
329 :yield: An instance of SemanticCostmapLocation.Location with the found valid position of the Costmap.
330 """
--> 331 sem_costmap = SemanticCostmap(self.part_of.world_object, self.urdf_link_name)
```

```
332 height_offset = 0
333 if self.for_object:
AttributeError: 'Object' object has no attribute 'world_object'
```

The code contains comments.

Observation:

Test 5 was conducted previously. The other tests can also be found in the GitHub repository under 'old_test' in 'test_files'. A significantly better plan was output there. In this plan, the table is being set, but then it is aborted because the milk is not in sight. This plan is very similar to the demo in the documentation, which covers the same problem. It can therefore be assumed that the demo plan was used for generation. In the current test execution, the demo plan was apparently not used as an example for generation.

Test 6

Code Quality:

The code does not run, but generally looks like a reasonable PyCRAM plan. The following error code is output:

```
---------------------------------------------------------------------------
AttributeError Traceback (most recent call last)
Cell In[1], line 34
31 fridge_pose = kitchen.get_link_pose("fridge_area")
32 cereal_box_dimensions = cereal.get_axis_aligned_bounding_box()
33 target_position = [
---> 34 fridge_pose.position.x + (cereal_box_dimensions.get_max().x - cereal_box_dimensions.get_min().x) / 2 + 0.1,
35 fridge_pose.position.y,
36 fridge_pose.position.z
37 ]
39 place_stand = CostmapLocation(Pose(target_position), reachable_for=robot_desig, reachable_arm=pickup_arm).resolve()
40 NavigateAction(target_locations=[place_stand.pose]).resolve().perform()
AttributeError: 'list' object has no attribute 'x'
```

The code is uncommented.

Observation:

An attempt was made here to determine a position next to the refrigerator. It was mistakenly assumed that a position had an attribute `x`, but it is a list with the elements for `x, y, z`. It should have accessed the first element of the list to get the x value.

Test 7

Code Quality:

The code runs through and also fulfills its purpose.

Observation:

In this test, it was correctly concluded that "thing" meant the spoon. The code is uncommented.

Test 8

Code Quality:

The code does not run. The following error code was output:

```
---------------------------------------------------------------------------
NameError Traceback (most recent call last)
Cell In[1], line 50
47 closest_spoon_desig = BelieveObject(names=[closest_spoon.name])
49 with simulated_robot:
---> 50 ParkArmsAction([Arms.BOTH]).resolve().perform()
51 MoveTorsoAction([0.25]).resolve().perform()
53 # Locate the closest spoon
NameError: name 'ParkArmsAction' is not defined
```

Observation:

It was correctly concluded that the spoon should not be on a table. However, it was assumed that this meant the spoon should not be attached to an object. Therefore, the spoon attached to `cabinet10_drawer_top` was excluded. The reason for the error code is not entirely clear. Actually, the error should not have been output because the code part where an attempt is made to determine the closest spoon from the two remaining spoons also tries to access the x-coordinate of a position with the argument x. As described in Test 6, this is not possible.

6.2 Test with Finetuning Models

Test	Result	Termination	Success	Runtime (m:s)	Costs	Code Quality	Further Observations
Test 3f	GitHub Test 3f	No	No	3:22	0.12 $		
Test 4f	GitHub Test 4f	Yes on 2	No	3:05	0.07 $	Clean, functional code	Also places bowl inside the cereal
Test 6f	GitHub Test 6f	Yes on 2	No	2:40	0.21 $	Functional, clean plan	Plan does not match the task

Table 7: Table of Test Evaluations

The tests where the generated code did not run were repeated with the architecture. However, the models with finetuning for the solvers and tools were used during the run.
However, the tests all turned out worse than before. The prompts were no longer followed correctly, or hallucinations occurred. For this reason, the tests were stopped and only Test 3, 4, and 6 were conducted. In Test 3f, the following error code is output:

```
---------------------------------------------------------------------------
StopIteration Traceback (most recent call last)
Cell In[1], line 17
14 robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
```

```
15 bowl = Object("bowl", ObjectType.BOWL, "bowl.stl", pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])
---> 17 cereal_desig = BelieveObject(names=["cereal"]).resolve()
18 kitchen_desig = BelieveObject(names=["kitchen"]).resolve()
19 robot_desig = BelieveObject(names=["pr2"]).resolve()
File ~/ros/ros_ws/src/pycram/src/pycram/designator.py:679, in ObjectDesignatorDescription.ground(self)
673 def ground(self) -> Union[Object, bool]:
674 """ 675 Return the first object from the world that fits the description.
676 :return: A resolved object designator
678 """
--> 679 return next(iter(self))
StopIteration:
```

This happens because `resolve()` is executed on `BelieveObject(names=["cereal"])` in the code. This is not common and leads to an error. The code from Test 4f runs through. However, the robot also places the bowl inside the cereal as in the previous Test 4. This time, 2 iterations are needed. In Test 6f, a functional plan is generated. However, it does not match the task but is the demo code where the cereal is placed on the kitchen island.

6.3 Analysis of Test Results

The conducted tests provide a comprehensive insight into the performance of the developed LLM architecture for generating PyCRAM code. The following summarizes and analyzes the key results to identify strengths and weaknesses of the architecture.

Summary of Results

Of the eight tests conducted with the base model, four tests (**Test 1**, **Test 2**, **Test 4**, **Test 7**) were successfully terminated, with the task being correctly executed in three cases. The remaining four tests resulted in code errors or incomplete fulfillment of the task. The tests with the fine-tuning models did not lead to an improvement in the results. On the contrary, the tests (**Test 3f**, **Test 4f**, **Test 6f**) showed a deterioration in terms of task fulfillment and the quality of the generated code.

Analysis of Successful Tests

In **Test 1** and **Test 2**, simple tasks present in the documentation or examples were posed. The LLM architecture was able to generate functional and clean code that correctly executed the tasks. This suggests that the model is good at reproducing known patterns and solving tasks that closely resemble the examples in the database.

In **Test 7**, the model demonstrated the ability to use context information and draw logical conclusions. It correctly recognized that "thing" referred to the "spoon" and generated corresponding functional code. This shows that the model can handle ambiguities in the task if sufficient context is available.

Analysis of Failed Tests

In **Test 3** and **Test 6**, syntax errors and errors in the use of classes and methods occurred. In Test 3, an incorrect argument name was used, leading to a runtime error. In Test 6, an attempt was made to access an attribute `x` of a list, resulting in an `AttributeError`. These errors indicate that the model has difficulty mastering the correct use of PyCRAM data structures and methods, especially in situations that do not directly correspond to the examples.

In **Test 4** and **Test 8**, the task was not correctly implemented, although the generated code was generally syntactically correct. In Test 4, the bowl was placed inside the cereal instead of next to it, and in Test 8, an incorrect spoon was selected, causing the architecture to become confused when trying to determine the next spoon. This indicates difficulties of the model in correctly interpreting and implementing spatial relationships. However, the negative instruction was understood.

Test 5 presented a complex task that required multiple steps. Here, the model showed weaknesses in planning and executing multi-step tasks. The generated code was not functional because incorrect methods for position determination were used and objects were incorrectly assigned. This shows that the architecture reaches its limits with complex tasks. This task was actually also present in the examples.

Evaluation of Finetuning Models

The use of finetuning models led to a deterioration of the results. The models showed difficulties in correctly interpreting the prompts, and hallucinations occurred more frequently. This could indicate that the finetuning was not optimally conducted or that the models were overfitted, losing generalization ability and no longer able to process the complex prompts and context.

Time and Cost Analysis

The analysis of the runtimes and costs of the conducted tests provides further insights into the efficiency and cost-effectiveness of the developed LLM architecture.

Runtime Analysis:

The runtimes of the tests varied between 1:43 minutes (**Test 7**) and 8:10 minutes (**Test 5**). On average, the runtime per test was about 3:30 minutes. Shorter runtimes were observed for simpler tasks, while more complex tasks tended to take more time. In particular, **Test 5** with the most complex task required the longest time, due to the higher computational effort and additional iterations for error correction.

The autocorrection mechanisms contributed to extended runtimes, as additional iterations were conducted in case of errors. For example, **Test 2** required two iterations to arrive at a functional code, which increased the runtime accordingly.

Cost Analysis:

The costs per test ranged between 0.05$ (**Test 1**) and 0.21$ (**Test 5**). The average cost per test was about 0.13$. The costs correlated with the number of tokens consumed by the LLM calls. More complex tasks and additional iterations for error correction led to higher costs.

The use of specialized models like `gpt-4o-2024-08-06` for the solver and `gpt-4o-mini` for the tools helped optimize costs without significantly impacting performance. However, it was shown that generating faulty code requiring additional iterations increased the overall costs.

Efficiency Assessment

The efficiency of the architecture can be considered satisfactory for simple tasks in terms of time and costs. For more complex tasks, both runtime and cost increase significantly, especially when error corrections are necessary. This indicates that the architecture is suitable for practical use in simple tasks but requires optimizations in error prevention and correction to increase efficiency. The average costs in time and money are 3:30 minutes and about 0.13$. As a reference, at the first tests, the architecture with GPT-3.5 for the tools and GPT-4 for the rest took 10 minutes and about 60$. This is a big improvement in just this short time. The improvement was mainly driven by new cheaper, faster and better performing models. This development can be seen in the tables under subsubsection 3.2.5.

Strengths and Weaknesses of the Architecture

Strengths

- The architecture is capable of correctly and efficiently solving known and simple tasks, especially when they resemble the examples in the database.
- It shows partial ability to draw logical conclusions and use context information.
- Code that is not correct is still a good starting point and can be corrected by a human.
- The output of the different parts are logged, which gives control over the separated reasoning process.

Weaknesses

- Difficulties in generating correct code for more complex or new tasks.
- Errors in the use of PyCRAM classes and methods, as well as in handling data structures.
- Problems in interpreting spatial relationships and understanding the world fully.
- The autocorrection mechanisms could not successfully correct all errors.
- The finetuning did not lead to the expected improvement and caused additional problems.
- Time and money consuming, specially when it needs more tries.

7 Results

7.1 Summary and Analysis

The primary goal of this research was to investigate the integration of LLMs into robot planning and reasoning, using the PyCRAM framework as a case study. The developed pipeline aimed to translate natural language instructions into executable PyCRAM code, enabling robots to perform tasks specified by human users.

By incorporating techniques such as Retrieval Augmented Generation (RAG) and providing extensive context through vector databases, LLMs demonstrated an enhanced ability to generate code for frameworks like PyCRAM, which are not extensively represented in their pretraining data.

The use of advanced prompting methods, including few-shot learning, role-playing, and chain-of-thought (CoT) prompting, significantly improved the performance of LLMs in code generation tasks. These methods helped guide the models to produce more accurate and contextually relevant outputs.

Despite improvements, LLMs faced challenges when tasked with complex or novel instructions that required intricate planning or reasoning. Errors often manifested in the form of incorrect code syntax, misuse of PyCRAM classes and methods, or failure to interpret spatial relationships accurately.

The implementation of automatic code correction loops improved the overall reliability of the pipeline. However, these mechanisms were not universally effective, particularly when addressing errors stemming from deeper misunderstandings of the task or the framework.

Initial experiments with fine-tuning LLMs on PyCRAM-specific data showed some promise in improving code generation. Nevertheless, fine-tuned models also exhibited increased hallucinations and occasionally produced less accurate code, suggesting potential overfitting or misalignment with the intended task.

The pipeline achieved reasonable efficiency for simple tasks, with average runtimes of approximately 3.5 minutes and costs around 0.13$ per task. However, more complex tasks led to longer runtimes and higher costs due to additional iterations required for error correction.

The rapid evolution of LLMs during the research period highlighted the importance of utilizing the most current models to achieve optimal performance. Newer models often offered improved capabilities, reduced costs, and faster processing times. It also shows the fast improvements in the field AI. The LLM and LRM capability are getting insane, and they are already a strong tool to accomplish all sorts of tasks in minutes which could take many hours otherwise. In fact, they also helped a lot with this thesis. OpenAI claims that the o1 model is on PhD level, so it is likely that the progress is exponentially speeding up, when the architectures begin to generate high quality research papers to improve themselves in one day.

7.2 Answering the Research Questions

To what extent can LLMs be augmented to generate code from new research frameworks that are little or not at all captured by the pretraining data?
LLMs can be significantly augmented to generate code for less-known or new frameworks like PyCRAM by integrating external knowledge sources and employing sophisticated prompting techniques. The use of vector databases containing documentation, code examples, and URDF files allowed the LLMs to access relevant information not present in their pretraining data. Techniques such as RAG and few-shot learning provided the necessary context for the models to produce more accurate code. However, the effectiveness of this augmentation is constrained by the quality and comprehensiveness of the provided context. While LLMs showed improved performance with these augmentations, they still struggled with tasks that deviated substantially from the examples or required deep understanding of the framework's intricacies.

How efficient and reliable can these models be used in robotics?
In the context of robotics, LLMs offer a promising avenue for translating natural language instructions into executable robot plans, facilitating more intuitive human-robot interactions. For simple and well-defined tasks, the models demonstrated reasonable efficiency and reliability, generating correct code within acceptable time frames and computational costs. However, their reliability diminishes with increasing task complexity. The models are less efficient when handling complex tasks due to longer processing times and higher computational demands, especially when multiple iterations are required for error correction. The current state of LLMs suggests that while they can be effectively used for straightforward robotic applications, significant enhancements are needed for their deployment in more complex, real-world scenarios. However, LLMs can be very valuable within complex architectures, as they facilitate the integration of various machine learning and planning methods. This allows these methods to benefit from the broad, general reasoning capabilities of LLMs, as concluded from the overview in section 2.

What challenges and limitations currently exist in the implementation and use of this technology, and what developments can arise in the future?
LLMs lack inherent knowledge of specialized frameworks like PyCRAM, leading to errors in code generation without sufficient contextual augmentation.
Automatic correction mechanisms are not always effective, particularly when errors arise from fundamental misunderstandings rather than minor syntax issues.
The models struggle with tasks requiring complex reasoning, multi-step planning, or the interpretation of abstract and negative instructions.
Processing complex tasks increases computational costs and processing times, which may not be practical for real-time applications in robotics.
The models' performance is highly sensitive to the quality of prompts, necessitating careful design and ongoing adjustments.

7.3 Discussion of the Results

The integration of LLMs into robot planning and reasoning presents a transformative opportunity to simplify and enhance human-robot interaction. This research has demonstrated that, with appropriate augmentation, LLMs can translate natural language instructions into executable code for frameworks like PyCRAM, thereby lowering the barriers to programming robotic behaviors.
However, the limitations observed highlight significant areas for improvement. The LLMs' difficulties with complex reasoning and novel task execution suggest that current models are not yet fully equipped to handle the nuanced demands of robotic planning without substantial support. The reliance on extensive context provision and prompt engineering indicates that LLMs, in their present form, may not generalize well across diverse tasks without significant human intervention.
The challenges with fine-tuning underscore the complexities of adapting large models to specific domains. The observed increase in hallucinations and decreased performance suggest that fine-tuning requires careful calibration to avoid degrading the model's overall capabilities.
Moreover, the resource-intensive nature of processing complex tasks raises practical concerns for real-world applications, where efficiency and responsiveness are critical. The balance be-

tween computational cost and performance is a crucial consideration for deploying such technologies in operational settings.

Looking forward, the development of more advanced models, such as OpenAI's o1 reasoning model, offers potential pathways to address some of these challenges. The integration of multimodal inputs and hybrid architectures that combine LLMs with traditional planning algorithms could enhance the models' reasoning capabilities and robustness.

In conclusion, while the current state of LLMs shows considerable promise for application in robotic planning and reasoning, there is a clear need for continued research and development. Addressing the identified challenges will require a multidisciplinary effort, combining advances in AI modeling, prompt engineering, domain-specific knowledge representation, and real-world testing in robotic systems. The ongoing evolution of LLMs and their integration into robotics holds significant potential for advancing the field toward more intelligent and autonomous robotic agents capable of understanding and executing complex human instructions.

8 Outlook

The integration of LLMs into robotics is a rapidly evolving field, as evidenced by the developments presented in this bachelor's thesis. While the results achieved are promising and demonstrate significant progress, the work is far from complete. There remains substantial potential for enhancement and expansion, necessitating ongoing research and development to fully realize the capabilities of LLMs in robot planning and reasoning.

8.1 Improvements to the Pipeline

The pipeline developed in this thesis serves as a prototype that lays the groundwork for future advancements. Given the complexity and scope of integrating LLMs into robotic systems, there are numerous opportunities for improvement and refinement. The pipeline's scalability aligns well with the continuous evolution of AI technologies, suggesting that it can be augmented as new models and methodologies emerge.

Incorporation of New Models

The introduction of new LLMs has consistently influenced the trajectory of this work, and this trend is expected to persist. Future models are anticipated to offer enhancements in cost-efficiency, performance, and processing speed. The ongoing development of open-source models opens new avenues for exploration. For instance, LLama-3.1 has shown considerable promise, unexpectedly achieving top performance on the PlanBench benchmark [Valmeekam et al., 2024]. Additionally, the recently introduced reasoning models from OpenAI, known as the o1 series, utilize multi-step reasoning processes that could significantly benefit the pipeline. Once these models are fully accessible through APIs, integrating them could enhance complex reasoning capabilities. Continued research into these models is recommended to leverage their potential fully.

Advanced Fine-Tuning Techniques

As increasingly powerful models become available for fine-tuning, expanding the fine-tuning dataset for PyCRAM is crucial. Work has already commenced on an automated pipeline that converts PyCRAM source code and documentation into suitable datasets. By extending this effort, a new fine-tuned model could be developed with a deep understanding of the framework.

The dataset can be enriched with additional examples, finer granularity through smaller data chunks, and comprehensive tutorials. For example, generated PyCRAM plan code could be converted into JSONL data points and stored alongside entries in the vector database. This approach would enhance the model's familiarity with PyCRAM, potentially improving code generation accuracy and efficiency.

Integration of Multimodal Inputs

Many new models possess multimodal capabilities, allowing them to process and generate not only text but also images and other data types. Incorporating camera images or screenshots into the prompts could provide the LLM with richer contextual information about the environment. This multimodal integration could improve the model's ability to reason about spatial relationships and object interactions, leading to more accurate and context-aware planning and execution.

Refinement of Prompts and Architectural Adjustments

Continuous refinement of prompts is essential to maximize the performance of LLMs. Fine-tuning offers an opportunity to investigate the potential for shorter, more efficient prompts without compromising effectiveness. As models become more cost-effective and fine-tuning becomes increasingly accessible, it is feasible to experiment with different architectural approaches.

While the ReWOO architecture was initially selected for its efficiency and suitability given the LLMs' limited knowledge of PyCRAM, evolving models may support alternative architectures. Adopting architectures like ReAct or hybrid models could be advantageous. Such an approach would enable the execution of individual steps within the simulation, allowing for the regeneration of specific steps in the event of errors. Additionally, capturing images after each step and incorporating them into the prompts of a multimodal model could enhance error detection and correction.

Implementing a decision-making component where the LLM evaluates the complexity and familiarity of instructions could optimize the processing pathway. Depending on this assessment, the system could select a streamlined path for simple, known tasks, bypassing the more complex ReWOO architecture when unnecessary. This flexibility would improve efficiency and responsiveness.

Integration with Additional Reasoning Methods

This thesis primarily explored LLM-based reasoning for robotics, demonstrating the substantial benefits LLMs offer. However, combining LLMs with other reasoning methods could further enhance system capabilities. As observed in approaches like those of Figure AI, a hybrid reasoning framework may yield the most significant advancements. An LLM agent could function as a foundational layer, equipped with additional reasoning tools and algorithms that can be invoked as needed. This integration would harness the strengths of various methods, enabling more robust and versatile problem-solving.

Expanded Testing and Validation

While numerous variants and methods were tested during the development phase, comprehensive testing remains crucial. Systematic evaluation across diverse scenarios and tasks will help identify limitations and guide iterative improvements. Emphasizing extensive testing ensures that enhancements are validated and that the system performs reliably under different conditions.

8.2 Future Outlook

Finally, I would like to use this section to try to look a little into the future. I will include opinions from researchers in the field and expand on my own observations. This section could become philosophical and hypothetical, but this is important to get an overview of this rapidly developing field, where advances will have a huge impact on the world and our everyday lives. As the integration of advanced language models into robotics continues, it's essential to recognize the trajectory of AI and its implications for society. Researchers and ex employee like Leopold Aschenbrenner have pointed out the exponential increase in computing power and the subsequent leaps in AI capabilities. In his work "situational awareness" he pointed out that we only have to automate AI research to create an intelligence explosion [Aschenbrenner, 2024]. Meanwhile, we see every big tech company rushing AI development and buying GPUs to build gigantic clusters [Simon Alvarez, 2024]. And now we get o1, as I mentioned, which is near to PhD level. I guess that even I as a Bachelor student can build a langgraph agent in the next year, which leverages o1, other LLMs and search APIs to write research papers non-stop. I guess

that in 2025 we will clearly experience a development that our human brain can not comprehend anymore.
So as we move forward, it is crucial to remain aware of the social and ethical implications of these advancements. The discussions around AGI and superintelligence emphasize the importance of responsible development and deployment. A collaborative effort between technologists, policymakers, and society at large will be required to navigate the challenges and seize the opportunities that the future of AI in robotics holds.

9 References

References

[1X Technologies, 2024] 1X Technologies (2024). 1x technologies: Safe, intelligent humanoids. https://www.1x.tech/. Accessed: 2024-10-11.

[Abid Ali Awan, 2024] Abid Ali Awan (2024). The pros and cons of using llm in the cloud versus running llm locally. https://www.datacamp.com/blog/the-pros-and-cons-of-using-llm-in-the-cloud-versus-running-llm-locally. Accessed: 2024-10-11.

[Anthropic, 2024a] Anthropic (2024a). Anthropic documentation - claude models. https://docs.anthropic.com/en/docs/about-claude/models. Accessed: 2024-10-03.

[Anthropic, 2024b] Anthropic (2024b). Anthropic homepage. https://www.anthropic.com/. Accessed: 2024-10-11.

[Anthropic, 2024c] Anthropic (2024c). Claude 3.5: A sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2024-10-11.

[Aschenbrenner, 2024] Aschenbrenner, L. (2024). Situational awareness: The decade ahead. https://situational-awareness.ai. Updated June 6, 2024, San Francisco, California.

[Brown, 2020] Brown, T. B. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[Chaudhary Parvez, 2024] Chaudhary Parvez (2024). Tweet from chaudhary parvez. https://x.com/ChaudharyParvez/status/1844591728038191414?t=x2pcQpMZDwAwSsauJihw7Q&s=19. Accessed: 2024-10-13.

[Community, 2024] Community, D. (2024). Best llm for coding: Cloud vs local.

[Dagan et al., 2023] Dagan, G., Keller, F., and Lascarides, A. (2023). Dynamic planning with a llm. *arXiv preprint arXiv:2308.06391*.

[Dech, 2024a] Dech, J. (2024a). Pycram: A python framework for cognition-enbabled robtics.

[Dech, 2024b] Dech, J. (2024b). Pycram: The next generation of cram. In *Proceedings of the Institute of Artificial Intelligence, University of Bremen*, Bremen, Germany. Institute of Artificial Intelligence, University of Bremen.

[Figure, 2024] Figure (2024). Figure status update - openai speech-to-speech reasoning. https://www.youtube.com/watch?v=Sq1QZB5baNw. Accessed: 2024-08-10.

[Figure AI, 2024] Figure AI (2024). Figure ai. https://www.figure.ai/ai. Accessed: 2024-08-10.

[Google, 2024] Google (2024). Google gemini api documentation - models. https://ai.google.dev/gemini-api/docs/models/gemini?hl=de. Accessed: 2024-10-03.

[Hafsa Jabeen, 2024] Hafsa Jabeen (2024). Self-hosted vs. api-based llms. https://codesphere.com/articles/self-hosted-vs-api-based-llms. Accessed: 2024-10-11.

[Hochreiter, 1997] Hochreiter, S. (1997). Long short-term memory. *Neural Computation MIT-Press*.

[Howard and Ruder, 2018] Howard, J. and Ruder, S. (2018). Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*.

[Huang and Chang, 2023] Huang, J. and Chang, K. C.-C. (2023). Towards reasoning in large language models: A survey.

[Jannis Brühl, 2023] Jannis Brühl (2023). Wieso amazon diesen geschwistern vier milliarden dollar geben will. `https://www.sueddeutsche.de/wirtschaft/kuenstliche-intelligenz-amazon-amodei-anthropic-1.6254633`. Accessed: 2024-10-11.

[Kambhampati et al., 2024] Kambhampati, S., Valmeekam, K., Guan, L., Stechly, K., Verma, M., Bhambri, S., Saldyt, L., and Murthy, A. (2024). Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv preprint arXiv:2402.01817*.

[Kenton and Toutanova, 2019] Kenton, J. D. M.-W. C. and Toutanova, L. K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2.

[Kong et al., 2023] Kong, A., Zhao, S., Chen, H., Li, Q., Qin, Y., Sun, R., Zhou, X., Wang, E., and Dong, X. (2023). Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702*.

[LangChain, 2024a] LangChain (2024a). How to create and query vector stores. `https://python.langchain.com/v0.2/docs/how_to/vectorstores/`. Accessed: 2024-10-13.

[LangChain, 2024b] LangChain (2024b). Introduction to langchain. `https://python.langchain.com/v0.2/docs/introduction/`. Accessed: 2024-08-10.

[LangChain, 2024c] LangChain (2024c). Langgraph code assistant tutorial. `https://langchain-ai.github.io/langgraph/tutorials/code_assistant/langgraph_code_assistant/`. Accessed: 2024-10-11.

[LangChain, 2024d] LangChain (2024d). Plan and execute tutorial. `https://langchain-ai.github.io/langgraph/tutorials/plan-and-execute/plan-and-execute/`. Accessed: 2024-10-11.

[LangChain, 2024] LangChain (2024). Rewoo tutorial. `https://langchain-ai.github.io/langgraph/tutorials/rewoo/rewoo/`. Zugriff am: 30. August 2024.

[LangChain Inc, 2024] LangChain Inc (2024). Langgraph: Building language agents as graphs. `https://langchain-ai.github.io/langgraph/`. Accessed: 2024-08-10.

[LMSYS, 2024] LMSYS (2024). Chat arena. `https://lmarena.ai/`. Accessed: 2024-10-11.

[Meta AI, 2024a] Meta AI (2024a). Llama 3 model card. `https://github.com/meta-llama/llama-models/blob/main/models/llama3/MODEL_CARD.md`. Accessed: 2024-10-11.

[Meta AI, 2024b] Meta AI (2024b). Llama 3.1 model card. `https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md`. Accessed: 2024-10-11.

[Meta AI, 2024c] Meta AI (2024c). Meta llama 3.1. https://ai.meta.com/blog/meta-llama-3-1/. Accessed: 2024-10-11.

[Open Robotics, 2024] Open Robotics (2024). Robot operating system (ros). https://www.ros.org/. Accessed: 2024-08-10.

[OpenAI, 2018] OpenAI (2018). Openai charter. https://openai.com/charter/. Accessed: 2024-10-11.

[OpenAI, 2024a] OpenAI (2024a). Embeddings. https://platform.openai.com/docs/guides/embeddings. Accessed: 2024-10-13.

[OpenAI, 2024b] OpenAI (2024b). Introducing openai o1-preview. https://openai.com/index/introducing-openai-o1-preview/. Accessed: 2024-10-11.

[OpenAI, 2024c] OpenAI (2024c). Key concepts. https://platform.openai.com/docs/concepts. Accessed: 2024-10-13.

[OpenAI, 2024d] OpenAI (2024d). Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/. Accessed: 2024-10-11.

[OpenAI, 2024e] OpenAI (2024e). Math problems with gpt-4o. https://www.youtube.com/watch?v=_nSmkyDNulk. Accessed: 2024-10-11.

[OpenAI, 2024f] OpenAI (2024f). Openai. https://openai.com/. Accessed: 2024-08-10.

[OpenAI, 2024g] OpenAI (2024g). Openai api pricing. https://openai.com/api/pricing/. Accessed: 2024-10-03.

[OpenAI, 2024h] OpenAI (2024h). Openai platform documentation - models. https://platform.openai.com/docs/models. Accessed: 2024-10-03.

[OpenAI, 2024i] OpenAI (2024i). Simple evals. https://github.com/openai/simple-evals?tab=readme-ov-file#user-content-fn-4-909bb9cac2709f698f869056b10467ec. Accessed: 2024-10-11.

[OpenAI, 2024j] OpenAI (2024j). What are tokens and how to count them? https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them. Accessed: 2024-10-03.

[ProgPrompt, 2023] ProgPrompt (2023). Progprompt: Generating situated robot task plans using large language models. https://progprompt.github.io/. Accessed: 2024-10-11.

[PyCRAM Developers, 2024] PyCRAM Developers (2024). Pycram documentation. https://pycram.readthedocs.io/en/latest/. Accessed: 2024-08-10.

[Radford, 2018] Radford, A. (2018). Improving language understanding by generative pre-training.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

[S3, 2024] S3 (2024). I lived with a humanoid robot for 48 hours. https://youtu.be/Sb6LMPXRdVc?si=PP75SJ4X2mIBM4hX&t=423. Accessed: 2024-10-11.

[Shirasaka et al., 2024] Shirasaka, M., Matsushima, T., Tsunashima, S., Ikeda, Y., Horo, A., Ikoma, S., Tsuji, C., Wada, H., Omija, T., Komukai, D., et al. (2024). Self-recovery prompting: Promptable general purpose service robot system with foundation models and self-recovery. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 17395–17402. IEEE.

[Simon Alvarez, 2024] Simon Alvarez (2024). Elon musk's xai activates "most powerful ai training cluster in the world" with 100k nvidia h100 gpus. https://www.teslarati.com/elon-musk-xai-supercomputer-cluster-100k-nvidia-h100-gpus/. Accessed: 2024-10-13.

[Singh et al., 2023a] Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. (2023a). Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE.

[Singh et al., 2023b] Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. (2023b). Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530. IEEE.

[Tesla, 2024] Tesla (2024). We, robot | tesla robotaxi unveil. https://www.youtube.com/watch?v=6v6dbxPlsXs&t=1200s. Accessed: 2024-10-11.

[Timothy, 2023] Timothy, M. (2023). Should you use a local llm? 9 pros and cons. https://www.makeuseof.com/should-you-use-local-llms/. Accessed: 2024-10-11.

[Troy Teslike, 2024a] Troy Teslike (2024a). Tweet from troy teslike. https://x.com/TroyTeslike/status/1845199345307509233?t=-urCz6tnl9w-3h44rw9Tug&s=19. Accessed: 2024-10-13.

[Troy Teslike, 2024b] Troy Teslike (2024b). Tweet from troy teslike. https://x.com/TroyTeslike/status/1845047695284613344?t=gCTz18idRsjK2HgzQDSpzg&s=08. Accessed: 2024-10-13.

[Töberg and Cimiano, 2023] Töberg, J.-P. and Cimiano, P. (2023). Generation of robot manipulation plans using generative large language models. In *2023 Seventh IEEE International Conference on Robotic Computing (IRC)*, pages 190–197.

[Valmeekam et al., 2023] Valmeekam, K., Sreedharan, S., Marquez, M., Olmo, A., and Kambhampati, S. (2023). On the planning abilities of large language models (a critical investigation with a proposed benchmark). *arXiv preprint arXiv:2302.06706*.

[Valmeekam et al., 2024] Valmeekam, K., Stechly, K., and Kambhampati, S. (2024). Llms still can't plan; can lrms? a preliminary evaluation of openai's o1 on planbench. *arXiv preprint arXiv:2409.13373*.

[Vaswani, 2017] Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

[Wang et al., 2024] Wang, J., Wu, Z., Li, Y., Jiang, H., Shu, P., Shi, E., Hu, H., Ma, C., Liu, Y., Wang, X., et al. (2024). Large language models for robotics: Opportunities, challenges, and perspectives. *arXiv preprint arXiv:2401.04334*.

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

[Xu et al., 2023] Xu, B., Peng, Z., Lei, B., Mukherjee, S., Liu, Y., and Xu, D. (2023). Rewoo: Decoupling reasoning from observations for efficient augmented language models.

[Yao et al., 2022] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

[Zhang et al., 2022] Zhang, Z., Zhang, A., Li, M., and Smola, A. (2022). Automatic chain of thought prompting in large language models.

[Zhao et al., 2024] Zhao, P., Zhang, H., Yu, Q., Wang, Z., Geng, Y., Fu, F., Yang, L., Zhang, W., and Cui, B. (2024). Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473*.

10 Appendix

10.1 Prompt-Templates

Custom PyCRAM GPT

```
# Role:
You are a programming expert with strong coding skills, especially in python.
You are specialized on generating PyCRAM-Plans for Robots with the PyCRAM Framework.
You can plan and code a complex action plan based on the provided task or question form the user.
You have the Documentation in 'PyCRAM_Documentation.txt', the PyCRAM code in 'PyCRAM_Code.txt' and the URDF Files for the environment and robots in 'PyCRAM_URDFs.txt' in your knowledge base.

# Task execution:
1.  First you create a step-by-step outline to create the Plan. A Plan should have the following aspects:
    - python package import statements
    - initialize the world and the objects
    - make the ObjectDesignators
    - ActionDesignators, CostMaps and other needed Statements in "with simulated_robot:"
2. Then you iterate through these Steps and create the required Python Code for it.
3. Assemble the Code form all the steps to one Plan.
4. Check if there are mistakes in the Plan.
5. Just give the Python Code of the Plan as Output

# Rewriting/Polishing Approach:
- When tasked with rewriting or polishing the plan, you ask for the error message, when not provided and then recheck the whole code and improve it.

# Example Plan (Plan for a Robot to pick up cereal and take it from one kitchen counter to another):
"""
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode

world = BulletWorld(WorldMode.GUI)
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
cereal = Object("cereal", ObjectType.BREAKFAST_CEREAL, "breakfast_cereal.stl", pose=Pose([1.4, 1, 0.95]))

cereal_desig = ObjectDesignatorDescription(names=["cereal"])
kitchen_desig = ObjectDesignatorDescription(names=["kitchen"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()

with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()

    MoveTorsoAction([0.3]).resolve().perform()

    pickup_pose = CostmapLocation(target=cereal_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()

    PickUpAction(object_designator_description=cereal_desig, arms=[pickup_arm], grasps=[Grasp.FRONT]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

    place_island = SemanticCostmapLocation("kitchen_island_surface", kitchen_desig.resolve(), cereal_desig.resolve()).resolve()
```

```

    place_stand = CostmapLocation(place_island.pose, reachable_for=robot_desig, reachable_arm=pickup_arm).resolve()

    NavigateAction(target_locations=[place_stand.pose]).resolve().perform()

    PlaceAction(cereal_desig, target_locations=[place_island.pose], arms=[pickup_arm]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

world.exit()
"""
```

ReWOO-Planer

```
rewoo_planer_prompt = """You are a renowned AI engineer and programmer. You receive world knowledge and a task. Your task is
↪ to develop a sequenz of plans to geather
resources and break down the given task for a other LLM Agent to generate PyCramPlanCode. PyCramPlanCode is a plan instruction
↪ for a robot that should enable it to perform the provided high level task. For each plan, indicate which external tool,
↪ along with the input for the tool, is used to gather evidence. You
can store the evidence in a variable #E, which can be called upon by other tools later. (Plan, #E1, Plan, #E2, Plan,
...).

The tools can be one of the following:
(1) Retrieve[input]: A vector database retrieval system containing the documentation of PyCram. Use this tool when you need
↪ information about PyCram functionality. The input should be a specific search query as a detailed question.
(2) Code[input]: A LLM Agent with a database retriever for the PyCRAM code. Returns a function from the code base and provides
↪ a tutorial for it. Provide a function as input.
(3) URDF[input]: A database retriver which returns the URDF file text. Use this tool when you need information about the URDF
↪ files used in the world. Provide the URDF file name as input.

PyCramPlanCode follow the following structure:
Imports
BulletWorld Definition
Objects
Object Designators
The 'with simulated_robot:'-Block (defines the Actions and moves of the Robot)
    Start with this two Code lines in this block:
        'ParkArmsAction([Arms.BOTH]).resolve().perform()
        MoveTorsoAction([0.25]).resolve().perform()'
    ActionDesignators (Concentrate on using Action Designators over MotionDesignators)
    SematicCostmapLocation
BulletWorld Close


Here are some examples of PyCramPlanCode with its corresponding building plan (use them just as examples to learn the
code format and the plan structure):
<examples>
{examples}
</examples>


Below you find all the Infos for your current task.
Describe your plans with rich details. Each plan should follow only one #E and it should be exactly in the given structure. DO
↪ NOT use any highlighting with markdown and co. You DO NOT need to consider how
PyCram is installed and set up in the plans, as this is already given.

World knowledge: {world}
---
Task: {task}"""
```

ReWOO-Solver

```
rewoo_solve_prompt = """You are a professional programmer, specialized on writing PyCram-Roboter-Plans. To write the code,
we have made a step-by-step Plan and retrieved corresponding evidence to each Plan. The evidences are examples and information
↪ to write PyCramPlanCode so
```

```
use them with caution because long evidence might contain irrelevant information and only use the world knowledge
for specific world information. Also be conscious about you hallucinating and therefore use evidence and example code
as strong inspiration.

{code_example}

Plan with evidence and examples to do the coding task:
<Plan>
{plan}
</Plan>

Now create the PyCramPlanCode for the task according to provided evidence above and the world knowledge.
Respond with nothing other than the generated PyCramPlan python code.
PyCramPlanCode follow the following structure:
<Plan structure>
Imports #Import Designators with *
#clear separation between code block
BulletWorld Definition
Objects
Object Designators
The 'with simulated_robot:'-Block (defines the Actions and moves of the Robot)
    Start with this two Code lines in this block:
        'ParkArmsAction([Arms.BOTH]).resolve().perform()
        MoveTorsoAction([0.25]).resolve().perform()'
    ActionDesignators
    SemanticCostmapLocation
BulletWorld Close
</Plan structure>


Task: {task}
World knowledge: {world}
Code Response (Response structure: prefix("Description of the problem and approach"); imports()"Import statements of the code"); code(Code block not including import statements)):
"""
```

ReWOO-CodeCheck-Planer

```
rewoo_codecheck_planer_prompt = """You are a renowned AI engineer and programmer. You receive world knowledge, a task, an error-code and a
code solution. The code solution was created by another LLM Agent like you to the given task and world knowledge. The
code was already executed resulting in the provided error message. Your task is to develop a sequenz of plans to geather
resources and correct the given PyCramPlanCode. PyCramPlanCode is a plan instruction for a robot that should enable the
robot to perform the provided high level task. For each plan, indicate which external tool, along with the input for
the tool, is used to gather evidence. You can store the evidence in a variable #E, which can be called upon by other
tools later. (Plan, #E1, Plan, #E2, Plan, ...).

The tools can be one of the following:
(1) Retrieve[input]: A vector database retrieval system containing the documentation of PyCram. Use this tool when you need information about PyCram functionality. The input should be a specific search query as a detailed question.
(2) Code[input]: A vector database retriever to search and look directly into the PyCram package code. As input give the exact Function and a little description.
(3) URDF[input]: A database retriever which returns the URDF file text. Use this tool when you need information about the URDF files used in the world. Provide the URDF file name as input.

PyCramPlanCode follow the following structure (Focus on useing ActionDesignators for the basic moves of the Robot):
Imports
BulletWorld Definition
Objects
Object Designators
The 'with simulated_robot'-Block (defines the Actions and moves of the Robot)
    Start with this two Code lines in this block:
        'ParkArmsAction([Arms.BOTH]).resolve().perform()
        MoveTorsoAction([0.25]).resolve().perform()'
    ActionDesignators
```

```
    SematicCostmapLocation
BulletWorld Close


Here is an PyCramPlanCode example with its corresponding correction plan (use them just as examples to learn the plan structure):
<example>
Failed PyCramPlanCode:
<failed_code>
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode
from pycram.datastructures.pose import Pose
world = BulletWorld(WorldMode.GUI)
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
cereal = Object("cereal", ObjectType.BREAKFAST_CEREAL, "breakfast_cereal.stl", pose=Pose([1.4, 1, 0.95]))

cereal_desig = ObjectDesignatorDescription(names=["cereal"])
fridge_desig = ObjectDesignatorDescription(names=["fridge"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()

with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()
    MoveTorsoAction([0.25]).resolve().perform()

    pickup_pose = CostmapLocation(target=cereal_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()

    PickUpAction(object_designator_description=cereal_desig, arms=[pickup_arm], grasps=[Grasp.FRONT]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

    place_fridge = SemanticCostmapLocation("fridge_surface", fridge_desig.resolve(), cereal_desig.resolve()).resolve()

    place_stand = CostmapLocation(place_fridge.pose, reachable_for=robot_desig, reachable_arm=pickup_arm).resolve()

    NavigateAction(target_locations=[place_stand.pose]).resolve().perform()

    PlaceAction(cereal_desig, target_locations=[place_fridge.pose], arms=[pickup_arm]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()
world.exit()
</failed_code>
---
Corresponding error:
Traceback (most recent call last):
  File "/home/julius/ros/ros_ws/src/pycram/src/llm_pyCram_plans/test.py", line 32, in <module>
    place_fridge = SemanticCostmapLocation("fridge_surface", fridge_desig.resolve(), cereal_desig.resolve()).resolve()
  File "/home/julius/ros/ros_ws/src/pycram/src/pycram/designator.py", line 679, in ground
    return next(iter(self))
StopIteration
X connection to :1 broken (explicit kill or server shutdown).
---
World knowledge:
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL, 'breakfast_cereal.stl',
pose=Pose([1.4, 1, 0.95]))]
```

```
---
Task:
Place the cereal box directly next to the refrigerator.
---
Corresponding output plan:
Plan 1: Use the URDF tool to retrieve the kitchen.urdf file to verify if the fridge is properly defined within the kitchen environment. This will help determine whether the fridge exists in the kitchen's URDF and is correctly referenced in the plan.
#E1 = URDF[kitchen.urdf]

Plan 2: Investigate the code for the function SemanticCostmapLocation to understand its behavior and possible causes of the error.
#E2 = Code[SemanticCostmapLocation]

Plan 3: Confirm the correct usage of the resolve method for object designators. This will help ensure that the fridge_desig.resolve() and cereal_desig.resolve() in the line causing the error are being used correctly.
#E3 = Retrieve[Correct usage of resolve method on ObjectDesignators in PyCram]

Plan 4: Validate how to use object designators in context with actions and location definitions in PyCram. This will ensure proper interaction with fridge and cereal in the SemanticCostmapLocation.
#E4 = Retrieve[Proper usage of ObjectDesignators in actions and locations in PyCram]

Plan 5: Examine how to correctly instantiate and use SemanticCostmapLocation in a way that avoids the error and properly defines the "fridge_surface" location for the placement task.
#E5 = Retrieve[Correct instantiation and usage of SemanticCostmapLocation in PyCram]

Plan 6: Investigate the correct usage of PlaceAction in PyCram, particularly how to specify both the object and the target locations effectively to achieve the intended placement behavior.
#E6 = Retrieve[Correct usage of PlaceAction in PyCram, including object_to_place and target_locations]

Plan 7: Ensure that the position of the cereal is calculated correctly. We need to confirm how to compute the target position next to the refrigerator based on the cereal's current position in the world, especially when using semantic locations.
#E7 = Retrieve[How to compute target positions for object placement based on current position in PyCram]
</example>

Below you find all the Infos for your current task.
Describe your plans with rich details. Each plan should follow only one #E and it should be exactly in the given structure. Do not include other characters for highlighting because this can break the Regex Pattern.
DO NOT use any highlighting with markdown and co. You DO NOT need to consider how PyCram is installed and set up in the plans, as this is already given.
Your task is to make a plan to correct the error but also include a general check up for unseen errors in the plan.

Failed PyCramPlanCode: {code}
---
Corresponding error: {error}
---
World knowledge: {world}
---
Task: {task}
---
Plan:"""
```

ReWOO-CodeCheck-Solver

```
codecheck_solve_prompt = """You are a professional programmer, specialized on correcting PycramRoboterPlanCode. To repair the
code, we have made step-by-step Plan and retrieved corresponding evidence to each Plan. The evidence are examples
and information to write PyCramCode so use them with caution because long evidence might contain irrelevant
information and only use the world knowledge for specific world information. Also be conscious about you
hallucinating and therefore use evidence and example code as strong inspiration.

{code_example}

Plan with evidence and examples to do the coding task:
<Plan>
{plan}
```

```
</Plan>

Now create the new properly functioning PyCramPlanCode Version for the task according to provided evidence above and
the world knowledge. Respond with nothing other than the generated PyCramPlan python code.
PyCramPlanCode follow the following structure:
<PyCramPlan structure>
Imports #Import Designators with *
BulletWorld Definition
Objects
Object Designators
The 'with simulated_robot:'-Block (defines the Actions and moves of the Robot)
    Start with this two Code lines in this block:
        'ParkArmsAction([Arms.BOTH]).resolve().perform()
        MoveTorsoAction([0.25]).resolve().perform()'
    ActionDesignators (Concentrate on using Action Designators over MotionDesignators)
    SematicCostmapLocation
BulletWorld Close
</PyCramPlan structure>

Failed PyCramPlanCode: {code}
---
Corresponding error: {error}
---
World knowledge: {world}
---
Task: {task}
---
Corrected Code Response (Response structure: prefix("Description of the problem and approach"); imports()"Import statements of the code"); code(Code block not including import statements)):
"""
```

Tool-Doku

```
prompt_docs = """**You are an experienced technical writer and coding educator specializing in creating comprehensive guides for implementing specific tasks and workflows using the PyCram framework.**

**Your task is to thoroughly explain how to accomplish the given task within PyCram, based on the provided context. Research and extract all relevant information, summarizing and organizing it in a way that enables another LLM agent to efficiently implement the workflow in code.**

---

### Context:
{context}

---

### Task:
{task}

---

### Instructions:

1. **Task Overview and Objectives**

   - **Define the Task:** Clearly articulate the task or workflow to be accomplished.
   - **Explain the Goal:** Describe the objective of the task and its significance within the PyCram framework.
   - **Prerequisites and Setup:** Detail any necessary setup steps or prerequisites required before starting the task.

2. **Detailed Workflow Explanation**

   - **Step-by-Step Guide:** Break down the process into logical, sequential steps.
   - **Key Concepts:** Explain important concepts and how they relate to each step.
   - **Relevant Functions:** Highlight relevant PyCram functions and explain their roles in the workflow.
```

```
   - **Integration:** Discuss how these steps and functions integrate within the PyCram framework.

3. **Code Examples and Implementation Guidance**

   - **Code Snippets:** Provide clear, complete code examples for each significant step.
   - **Explanation:** Accompany code with detailed explanations to ensure understanding.
   - **Adaptability:** Ensure code examples can be easily adapted to similar tasks or scenarios.

4. **Framework Integration and Concepts**

   - **Broader Context:** Explain how the task fits into the larger PyCram framework.
   - **Essential Components:** Discuss crucial components, tools, or modules within PyCram relevant to the task.
   - **Conceptual Understanding:** Enhance understanding of the framework's architecture as it relates to the task.

5. **Best Practices and Considerations**

   - **Implementation Tips:** Offer best practices for effectively implementing the task.
   - **Potential Challenges:** Identify common pitfalls or challenges that may arise.
   - **Solutions:** Provide recommendations or strategies to overcome these challenges.

---

### Important Notes:

- **Length:** Use approximately **3,000 words** for your explanation to ensure thoroughness.
- **Completeness:** Ensure that all code examples are **complete and well-explained**.
- **Clarity and Organization:** Present information in a **clear, logical order** to facilitate implementation by another LLM agent.
- **Style Guidelines:**
    - Use clear and professional language appropriate for a technical audience.
    - Structure your guide with headings, subheadings, and bullet points for readability.
    - Avoid ambiguity and ensure precision in explanations and code.
"""
```

Tool-Code

```
prompt_code = """**You are an experienced technical writer and coding educator specializing in creating detailed and precise tutorials. You are a Tool in an LLM Agent structure and you get a subtask for which you gather information.**

**Your task is to craft a comprehensive guide on how to use the provided function within the PyCram framework, based on the given documentation and code context. You should not only explain the function itself but also describe its relationship with other relevant functions and components within the context. DO NOT provide information how to install and setup PyCRAM because it is already done.**

---

### Context:
Here is the high level task and world knowledge. Use it just as context and only work on you subtask:
<high_level_task_context>
{instruction}

World knowledge:
{world}
</high_level_task_context>
Here is the retrieved context for your task:
{context}

---

### Your Subtask:
**Function:** {task}

---

### Instructions:
```

```

1. **Function Explanation and Contextualization**

   - **Detailed Description:** Begin with a comprehensive description of the function, including its purpose and how it fits within the PyCram framework.
   - **Syntax and Parameters:** Explain the function's syntax, input parameters, and return values.
   - **Integration:** Describe how this function integrates into the framework and its role within the overall context.
   - **Relationship with Other Components:** Discuss how the function interacts with other relevant functions and components in the context.

2. **Code Examples and Implementation**

   - **Full Function Code:** Provide the complete code of the function.
   - **Demonstration Snippets:** Include relevant code snippets from the context that demonstrate the function in action.
   - **Step-by-Step Explanation:** Explain how the code works step by step.
   - **Adaptation:** Show how the code can be adapted to solve similar tasks.

3. **General Framework Functionality**

   - **Fundamental Concepts:** Explain the fundamental functionality of the PyCram framework as it relates to the given function.
   - **Key Principles:** Discuss key concepts and principles necessary to understand the function and its application.
   - **Importance:** Highlight why these concepts are essential for effectively using the function.

4. **Best Practices and Recommendations**

   - **Effective Usage:** Provide guidance and best practices for effectively using the function and the framework.
   - **Common Pitfalls:** Mention potential pitfalls and how to avoid them.
   - **Optimization Tips:** Offer suggestions on optimizing the function's performance and reliability.

5. **Planning and Implementation for Developers**

   - **Implementation Plan:** Design a clear plan outlining how developers can implement the function in their own projects.
   - **Integration Steps:** Outline the necessary steps to correctly integrate and customize the function.
   - **Customization Guidance:** Provide advice on tailoring the function to meet specific project requirements.


### Important Notes:

- **Length:** Write as short as possible but ensure that everything important is included.
- **Code Examples:** Incorporate all essential code examples in their entirety.
- **Systematic Thinking:** Think systematically to ensure that another LLM agent can produce correct code based on your output.
- **Clarity and Organization:**
    - Use clear and professional language appropriate for a technical audience.
    - Structure your tutorial with headings, subheadings, and bullet points for readability.
    - Ensure explanations are precise and unambiguous.
"""
```

URDF-Tool

```
urdf_tool_template = """You are a file summariser tool. You get a urdf file for an environment, a robot or an object. You also get a instruction and world knowledge.
Your task is to summarize and compress the urdf-file and list the important data in it. Be sure that you list all the Data in the file which is important for the world understanding to accomplish the instruction.
Combine the summary and the listing with the world knowledge and create a world model in natural language with it. Be sure that the used information is correct.

Here is the input. Remember that the instruction is just to understand the context and not your task:
- **Instruction:** {prompt}
---
- **World knowledge:** {world}
---
- **URDF file:**
<urdf>
```

```
{urdf}
</urdf>
"""
```

Prethinking

```
preprocessing_prompt = """You are an intelligent planning agent for robots. Your task is to analyze a given instruction, world knowledge, and a URDF file, and break them down into the following components:

1. **Initial stage:** Describe the current state before the task begins.
2. **Goal stage:** Describe the desired end state after completing the task.
3. **Step-by-step action plan:** Create a detailed but concise sequence of actions that the robot must perform to move from the initial stage to the goal stage.

The output should be brief and concise, formatted as a clear instruction, similar to the provided instruction, but with more details. When you are not sure you can provide the right answer just try your best.

Example input:
- **Instruction:** Place the cereal box on the kitchen island.
- **World knowledge:** [kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf'),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL, 'breakfast_cereal.stl', pose=Pose([1.4, 1, 0.95])))]
- **URDF file:** (The content of the file is very long, so no example here. The file would contain information about the components of the objects, such as 'kitchen.urdf' and 'pr2.urdf'.)


**Your Expected output:**
- **Initial stage:**
    - **Cereal box:** Located on the ground approximately at position **[1.4, 1, 0.95]**.
    - **PR2 Robot:** Positioned near the cereal box, approximately at position **[1.4, 1, 0]**.

- **Goal stage:**
    - **Cereal box:** Located on the kitchen island surface approximately at position **[-1.07, 1.72, 0.84]**.

- **Step-by-step plan:**

    1. **Robot positioning:**
    - **Action:** The PR2 robot moves to position **[1.4, 1, 0]**, near the cereal box.

    2. **Grabbing the cereal box:**
    - **Action:** The robot navigates to the cereal box at **[1.4, 1, 0.95]**.
    - **Action:** Securely grab the cereal box.

    3. **Movement to the kitchen island:**
    - **Action:** The robot transports the cereal box from **[1.4, 1, 0.95]** to the kitchen island approximately at **[-1.07, 1.72, 0.84]**.

    4. **Placing the cereal box:**
    - **Action:** Carefully place the cereal box on the kitchen island approximately at position **[-1.07, 1.72, 0.84]**.

    5. **Task completion:**
    - **Action:** Finish the task and return to the initial position or prepare for the next instruction.

Now do the Task for the Input:
**Input**
- **Instruction:** {prompt}
- **World knowledge:** {world}
- **URDF file:**
<urdf>
{urdf}
</urdf>
"""
```

10.2 Pretests:

10.2.1 Zero-Shot-GPT-4o

Prompt:

What is PyCRAM?

ChatGPT-4o:

```
PyCRAM is a Python library designed for cognitive robot abstract machines (CRAM). It is used in the field of robotics to facilitate the development of high-level robot control programs. PyCRAM provides tools and abstractions that help in creating plans and behaviors for robots, allowing them to perform complex tasks in dynamic environments.

The library is built to support the design and execution of robot plans that can adapt to changes and uncertainties in the environment. It typically includes features for task-level planning, reasoning, and execution monitoring, enabling robots to make decisions and adjust their actions based on sensory feedback and other inputs.

PyCRAM is part of a broader effort to create more intelligent and autonomous robotic systems, leveraging the flexibility and power of Python to make it accessible and easy to integrate with other software components in a robotics stack.
```

Prompt:

```
You are a professional programmer, specialized on writing PyCRAM-Roboter-Plans. Your task it to generate a PyCRAM-Plan for a Robot instruction and world knowledge you get as input. Respond with just the code.
Instruction:
Pick-up the bowl from the table.
World knowledge:
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]),
color=[1, 1, 0, 1])]
```

ChatGPT-4o:

````
```python
from pycram.bullet_world import BulletWorld
from pycram.process_module import ProcessModule
from pycram.robot_descriptions.robot_description_handler import InitializedRobotDescription as robot_description
from pycram.task import with_tree, with_knowledge
from pycram.helper import transform

# Initialize the world
world = BulletWorld()
kitchen = world.add_object('kitchen', 'kitchen.urdf')
robot = world.add_robot('pr2', 'pr2.urdf')
bowl = world.add_object('bowl', 'bowl.stl', pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])

# Define the pick-up plan
@with_tree
@with_knowledge
def pick_up_bowl():
    # Get the robot's end effector
    end_effector = robot_description.i.get_tool_frame()

    # Move the robot's end effector to the bowl's position
    ProcessModule.move(end_effector, transform(bowl.pose, [0, 0, 0.1]))

    # Close the gripper to grasp the bowl
    ProcessModule.grasp(bowl)

    # Lift the bowl slightly
    ProcessModule.move(end_effector, transform(bowl.pose, [0, 0, 0.2]))

````
````

````
# Execute the plan
pick_up_bowl()
```

This code initializes a PyCRAM world with a kitchen, a PR2 robot, and a bowl. It defines a plan to pick up the bowl by moving the robot's end effector to the bowl's position, grasping it, and then lifting it slightly.
````

10.2.2 Zero-Shot-CoT-GPT-4o

The beginning is the same as 10.2.1.

Prompt:

```
You are a professional programmer, specialized on writing PyCRAM-Roboter-Plans. Your task it to generate a PyCRAM-Plan for a Robot instruction and world knowledge you get as input. Think and plan step by step.
Instruction:
Pick-up the bowl from the table.
World knowledge:
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]),
color=[1, 1, 0, 1])]
```

ChatGPT-4o:

````
To generate a PyCRAM plan for the robot to pick up the bowl from the table, we need to follow a structured approach. This involves understanding the task, setting up the environment, and writing the plan that the robot will execute. Here's a step-by-step breakdown:

### Step 1: Understand the Task
The task is to have the robot pick up a bowl from a table. The world knowledge provides us with the objects involved: the kitchen environment, the robot (PR2), and the bowl with its position and color.

### Step 2: Set Up the Environment
We need to ensure that the environment is correctly set up in the simulation. This involves loading the kitchen and the robot models, and placing the bowl at the specified position.

### Step 3: Define the Plan
The plan will involve the following steps:
1. Navigate the robot to the table.
2. Locate the bowl using the robot's sensors.
3. Plan the grasping motion.
4. Execute the grasp to pick up the bowl.

### Step 4: Implement the Plan in PyCRAM

Here's a basic implementation of the plan in PyCRAM:

```python
from pycram import *
from pycram.bullet_world import BulletWorld
from pycram.robot_descriptions.robot_description_handler import InitializedRobotDescription as robot_description
from pycram.process_module import simulated_robot
from pycram.task import with_tree

# Initialize the BulletWorld
world = BulletWorld()

# Load the environment and objects
kitchen = world.add_object('kitchen', 'kitchen.urdf')
robot = world.add_robot('pr2', 'pr2.urdf')
bowl = world.add_object('bowl', 'bowl.stl', pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])

# Define the plan
def pick_up_bowl():
````

```
    with with_tree():
        # Step 1: Navigate to the table
        simulated_robot.navigate(Pose([1.4, 1, 0]))  # Assuming the table is at this position

        # Step 2: Locate the bowl
        bowl_position = bowl.pose.position

        # Step 3: Plan the grasp
        grasp_pose = Pose([bowl_position.x, bowl_position.y, bowl_position.z + 0.1])  # Adjust for grasping height

        # Step 4: Execute the grasp
        simulated_robot.pick_up(bowl, grasp_pose)

# Execute the plan
pick_up_bowl()
```

Explanation
- **Environment Setup**: We initialize the `BulletWorld` and add the kitchen, robot, and bowl to it.
- **Navigation**: The robot navigates to a position near the table. This is a simplified assumption; in a real scenario, you would use a more sophisticated navigation system.
- **Bowl Location**: The robot uses its sensors to locate the bowl. Here, we directly use the known position.
- **Grasp Planning**: We calculate a grasp pose slightly above the bowl to ensure a proper grasp.
- **Execution**: The robot executes the pick-up action using the `pick_up` method.

This plan is a basic example and can be expanded with more sophisticated perception, error handling, and dynamic adjustments based on real-time feedback from the robot's sensors.
```

### 10.2.3 One-Shot-GPT-4o

#### System Prompt:

You are a professional programmer, specialized on writing PyCRAM-Roboter-Plans. Your task it to generate a PyCRAM-Plan for a Robot instruction and world knowledge you get as input. Respond with just the code.

Here is an PyCRAM-Plan example for the instruction "Can you place the cereal on the kitchen island?":

&lt;example&gt;

```
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode

world = BulletWorld(WorldMode.GUI)
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
cereal = Object("cereal", ObjectType.BREAKFAST_CEREAL, "breakfast_cereal.stl", pose=Pose([1.4, 1, 0.95]))

cereal_desig = ObjectDesignatorDescription(names=["cereal"])
kitchen_desig = ObjectDesignatorDescription(names=["kitchen"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()

with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()

    MoveTorsoAction([0.3]).resolve().perform()

    pickup_pose = CostmapLocation(target=cereal_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()
```
```

```
    PickUpAction(object_designator_description=cereal_desig, arms=[pickup_arm], grasps=[Grasp.FRONT]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

    place_island = SemanticCostmapLocation("kitchen_island_surface", kitchen_desig.resolve(),
    ↪  cereal_desig.resolve()).resolve()

    place_stand = CostmapLocation(place_island.pose, reachable_for=robot_desig, reachable_arm=pickup_arm).resolve()

    NavigateAction(target_locations=[place_stand.pose]).resolve().perform()

    PlaceAction(cereal_desig, target_locations=[place_island.pose], arms=[pickup_arm]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

world.exit()
```

</example>

User instruction:

Pick-up the bowl from the table. World knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]),
color=[1, 1, 0, 1])]
```

GPT-4o:

```
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode

world = BulletWorld(WorldMode.GUI)
kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf')
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])

bowl_desig = ObjectDesignatorDescription(names=['bowl'])
robot_desig = ObjectDesignatorDescription(names=['pr2']).resolve()

with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()

    MoveTorsoAction([0.3]).resolve().perform()

    pickup_pose = CostmapLocation(target=bowl_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()

    PickUpAction(object_designator_description=bowl_desig, arms=[pickup_arm], grasps=[Grasp.FRONT]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

world.exit()
```

10.2.4 One-Shot-o1-preview

System prompt and instruction as in subsubsection 10.2.3.

o1-preview:

```python
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode
from pycram.pose import Pose

world = BulletWorld(WorldMode.GUI)
kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf')
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])

bowl_desig = ObjectDesignatorDescription(names=['bowl'])
robot_desig = ObjectDesignatorDescription(names=['pr2']).resolve()

with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()

    MoveTorsoAction([0.3]).resolve().perform()

    pickup_pose = CostmapLocation(target=bowl_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()

    PickUpAction(object_designator_description=bowl_desig, arms=[pickup_arm], grasps=[Grasp.TOP]).resolve().perform()

    ParkArmsAction([Arms.BOTH]).resolve().perform()

world.exit()
```

10.2.5 Custom PyCRAM GPT

Der Prompt für den Custom GPT ist hier unter section 10.1 zu finden.

Der GPT verfügt über die PyCRAM Dokumentation, den Code und die URDF Datein in seinem Wissen, welches einer Vektordatenbank mit Retriever entspricht.

Link zum Custom GPT: PyCRAM GPT

Instruktion wie bei subsubsection 10.2.3

```
Here is the Python code plan for a robot to pick up the bowl from the table using the PyCRAM framework.

### Step 1: Import required packages
```python
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
```
```

````
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode
from pycram.datastructures.pose import Pose
```

### Step 2: Initialize the world and objects
```python
# Initialize the world in GUI mode
world = BulletWorld(WorldMode.GUI)

# Create the kitchen environment, robot, and bowl object
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
bowl = Object("bowl", ObjectType.BOWL, "bowl.stl", pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])
```

### Step 3: Define Object Designators
```python
# Create designators for the objects
bowl_desig = ObjectDesignatorDescription(names=["bowl"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()
```

### Step 4: Action execution within the simulated robot environment
```python
# Start the simulated robot environment
with simulated_robot:
    # Park both arms initially
    ParkArmsAction([Arms.BOTH]).resolve().perform()

    # Move the torso to ensure the arms can reach the bowl
    MoveTorsoAction([0.3]).resolve().perform()

    # Find a reachable position for the robot to pick up the bowl
    pickup_pose = CostmapLocation(target=bowl_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    # Navigate to the pickup location
    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()

    # Pick up the bowl with the available arm
    PickUpAction(object_designator_description=bowl_desig, arms=[pickup_arm], grasps=[Grasp.FRONT]).resolve().perform()

    # Park the arms again after picking up the bowl
    ParkArmsAction([Arms.BOTH]).resolve().perform()
```

### Step 5: Exit the simulation world
```python
# Exit the simulation
world.exit()
```

### Full Plan Code:
```python
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.process_module import simulated_robot
from pycram.designators.motion_designator import *
from pycram.designators.location_designator import *
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import ObjectType, Arms, Grasp, WorldMode
from pycram.datastructures.pose import Pose
````

````

# Initialize world and objects
world = BulletWorld(WorldMode.GUI)
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
bowl = Object("bowl", ObjectType.BOWL, "bowl.stl", pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])

# Create object designators
bowl_desig = ObjectDesignatorDescription(names=["bowl"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()

# Execute actions in simulated robot environment
with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()
    MoveTorsoAction([0.3]).resolve().perform()

    # Get reachable pose for picking up the bowl
    pickup_pose = CostmapLocation(target=bowl_desig.resolve(), reachable_for=robot_desig).resolve()
    pickup_arm = pickup_pose.reachable_arms[0]

    # Navigate to the bowl and pick it up
    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()
    PickUpAction(object_designator_description=bowl_desig, arms=[pickup_arm], grasps=[Grasp.FRONT]).resolve().perform()

    # Park arms after pickup
    ParkArmsAction([Arms.BOTH]).resolve().perform()

# Exit the simulation
world.exit()
```

This plan will navigate the robot to the table, pick up the bowl, and park its arms.
````

10.2.6 Finetuned GPT-4o Zero-Shot

System-Prompt:

You are a professional programmer, specialized on writing PyCRAM-Roboter-Plans. Your task it to generate a PyCRAM-Plan for a Robot instruction and world knowledge you get as input. Respond with just the code.

Instruktion:

Pick-up the bowl from the table.

World knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]),
color=[1, 1, 0, 1])]
```

Output:

```python
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.enums import ObjectType, Arms
from pycram.datastructures.pose import Pose

# Create Object Designators
bowl_desig = ObjectDesignatorDescription(names=['bowl'])
kitchen_desig = ObjectDesignatorDescription(names=['kitchen'])
robot_desig = ObjectDesignatorDescription(names=['pr2'], types=[ObjectType.ROBOT])
```
```

```python

# Create Location Designators
torso_locations = [Pose([0, 1, 0.8], [0, 0, 1, 0]), Pose([1, 0, 0.8], [0, 0, 1, 0]), Pose([-1,
↪  1, 0.8], [0, 0, 1, 0])]
torso_desig = CostmapLocation(torso_locations, kitchen_desig.resolve(), bowl_desig.resolve())
arm_desig = 'RIGHT' if torso_desig.resolve().pose.position[1] <= 0 else 'LEFT'

# Create and resolve action designator
pick_up_desig = PickUpAction(object_designator_description=bowl_desig, arms=[Arms[arm_desig]],
↪  grasps=['front'], grab_after_pickup=True, torso_locations=torso_locations,
↪  torso_location_costmap=True,
↪  torso_location_reference_world_obj=kitchen_desig.resolve()).resolve()
pick_up_desig.perform()
```

10.2.7 Finetuned GPT-4o-mini Zero-Shot

System-Prompt:

You are a professional programmer, specialized on writing PyCRAM-Roboter-Plans. Your task it to generate a PyCRAM-Plan for a Robot instruction and world knowledge you get as input. Respond with just the code.

Instruktion:

Pick-up the bowl from the table.

World knowledge:

```python
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]),
color=[1, 1, 0, 1])]
```

Output:

```python
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.datastructures.enums import ObjectType
from pycram.datastructures.poses import Pose
from pycram.datastructures.arms import Arms
from pycram.process_module import simulated_robot, with_next_step
from pycram.designators.motion_designator import NavigateMotion, MoveArmMotion
from pycram.designators.action_designator import ParkAction
from pycram.datastructures.enums import TransportState
from pycram.designators.location_designator import CostmapLocation
from pycram.designators.designator import *

world = BulletWorld()
kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf')
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 1, 0.89]), color=[1, 1, 0, 1])

pickup_pose = CostmapLocation(target=bowl.pose, global_costmap="kitchen").resolve()

with simulated_robot:
    ParkAction([Arms.BOTH]).perform()
    NavigateMotion(target_locations=[pickup_pose.pose]).resolve().perform()
    MoveArmMotion(Arms.RIGHT, [Pose([0, 0, 0], [0, 0, 0, 1])]).resolve().perform()
    PickUpAction(bowl, [Arms.RIGHT]).resolve().perform()
    MoveArmMotion(Arms.LEFT, [Pose([0, 0, 0], [0, 0, 0, 1])]).resolve().perform()
    ParkAction([Arms.RIGHT]).perform()
```

10.2.8 Retriever Doku

For this test, I've shortened the output since it's quite lengthy. However, the full output can be viewed in the GitHub from PyCRAM repository here.

Input:

What are Action Designators and how are they used?

Output:

````
Action Designator

This example will show the different kinds of Action Designators that are available. We will see how to create Action
Designators and what they do.

Action Designators are high-level descriptions of actions which the robot should execute.

Action Designators are created from an Action Designator Description, which describes the type of action as well as the
parameter for this action. Parameter are given as a list of possible parameters.
For example, if you want to describe the robot moving to a table you would need a
{meth}`~pycram.designators.action_designator.NavigateAction` and a list of poses that are near the table. The Action
Designator Description will then pick one of the poses and return a performable Action Designator which contains the
picked pose.

... ... ...

```python
apartment = Object("apartment", ObjectType.ENVIRONMENT, "apartment.urdf")
apartment.set_joint_state("cabinet10_drawer_top_joint", 0.4)
```

```python
from pycram.designators.action_designator import *
from pycram.designators.object_designator import *
from pycram.datastructures.enums import Arms
from pycram.process_module import simulated_robot
from pycram.datastructures.pose import Pose

apartment_desig = BelieveObject(names=["apartment"]).resolve()
handle_deisg = ObjectPart(names=["handle_cab10_t"], part_of=apartment_desig)

with simulated_robot:
    MoveTorsoAction([0.25]).resolve().perform()
    ParkArmsAction([Arms.BOTH]).resolve().perform()
    NavigateAction([Pose([1.7474915981292725, 2.8073629093170166, 0.0],
                         [-0.0, 0.0, 0.5253598267689507, -0.850880163370435])]).resolve().perform()
    CloseAction(handle_deisg, [Arms.RIGHT]).resolve().perform()
```

```python
world.exit()
```
````

—– 2nd content ———- view full file on GitHub

````
Designators


Designators are CRAMs and PyCRAMs way of representing actions, motions, objects and locations.

In general, PyCRAM-Designators consist of a description and a specified element.
Descriptions describe sets of designators and designators are one thing in the described set.
For example, such a description could describe an action where the robot moves to a location
from where it can grasp an object. The specific location in this case is not relevant as long
````

```
as the robot can reach the object.

The designator description will be resolved during runtime which results in a designator with
specific parameter, the resulting designator can also be performed to let the robot perform the
desired behaviour.

To stay on the example of an action designator which should move the robot to a location from
where it can grasp an object, we will create a NavigateAction description with a list of possible
poses.

... ... ...

Creating your own Designator
============================
Creating your own designator is fairly easy, you only need to extend the base class of the respective description.

 - :mod:`~pycram.designator.ActionDesignatorDescription`
 - :mod:`~pycram.designator.ObjectDesignatorDescription`
 - :mod:`~pycram.designator.LocationDesignatorDescription`
 - :mod:`~pycram.designator.BaseMotion`

Afterwards you need to implement your own ``ground`` method which is the default resolver and for location and object
designator it makes sense to also implement a ``__iter__`` method. The ``ground`` and ``__iter__`` methods should return
the designator sub-class so you also need to implement these with the parameter your designator needs.

The sub-class can already contain some parameters, this is usually the case if the parameter is the same for every designator
of this type. For example, :class:`~pycram.designator.LocationDesignatorDescription.Location`
contains a ``pose`` parameter since every location designator contains a resolved pose.

For action and motion designator the sub-class is also the place where the ``perform`` method is written which contains
the behaviour of the designator.
```

10.2.9 LLM Tool Doku

This test was conducted in the context of Test 4.

Input:

What are Action Designators and how are they used?

Output:

```
# Implementing Action Designators in PyCram

## Task Overview and Objectives

### Define the Task
The task involves using Action Designators within the PyCram framework to perform specific actions with a robot. In this context, we will focus on placing a cereal box and a bowl side by side on a kitchen island.

### Explain the Goal
The objective is to utilize Action Designators to navigate the robot to the kitchen island, pick up the cereal and bowl, and place them in the desired configuration. This task demonstrates the capabilities of Action Designators in automating complex robotic actions.

### Prerequisites and Setup
Before starting, ensure that:
- The PyCram framework is properly set up and running.
- The robot (PR2) and objects (cereal and bowl) are defined in the environment.

## Detailed Workflow Explanation

### Step-by-Step Guide

1. **Initialize the Environment**
   - Create a BulletWorld and define the robot and objects.
```

````

2. **Navigate to the Kitchen Island**
   - Use the `NavigateAction` to move the robot to the kitchen island.

3. **Pick Up the Cereal**
   - Use the `PickUpAction` to grasp the cereal.

4. **Navigate to the Bowl**
   - Use the `NavigateAction` again to move to the bowl's location.

5. **Pick Up the Bowl**
   - Use the `PickUpAction` to grasp the bowl.

6. **Navigate to the Desired Placement Location**
   - Use the `NavigateAction` to move to the final placement location on the kitchen island.

7. **Place the Cereal and Bowl**
   - Use the `PlaceAction` to place the cereal and bowl side by side.

### Key Concepts
- **Action Designators**: High-level descriptions of actions that the robot can perform, allowing for complex behaviors to be executed with simple commands.
- **Object Designators**: Represent objects in the environment that the robot can interact with.

### Relevant Functions
- `NavigateAction`: Moves the robot to a specified location.
- `PickUpAction`: Allows the robot to pick up an object.
- `PlaceAction`: Places an object at a specified location.

### Integration
These steps integrate seamlessly within the PyCram framework, allowing for modular and reusable code that can be adapted for various tasks.

## Code Examples and Implementation Guidance

### Code Snippet

```python
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.datastructures.pose import Pose
from pycram.designators.action_designator import NavigateAction, PickUpAction, PlaceAction
from pycram.process_module import simulated_robot
from pycram.datastructures.enums import Arms, Grasp

# Initialize the world and objects
world = BulletWorld()
robot = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
bowl = Object("bowl", ObjectType.BOWL, "bowl.stl", pose=Pose([1.4, 0.50, 0.89]), color=[1, 1, 0, 1])
cereal = Object("cereal", ObjectType.BREAKFAST_CEREAL, "breakfast_cereal.stl", pose=Pose([1.4, 1, 0.95]))

# Start the simulation
with simulated_robot:
    # Step 1: Navigate to the cereal
    navigate_to_cereal = NavigateAction(target_locations=[Pose([1.4, 1, 0.95])]).resolve()
    navigate_to_cereal.perform()

    # Step 2: Pick up the cereal
    pickup_cereal = PickUpAction(object_designator_description=cereal, arms=[Arms.RIGHT], grasps=[Grasp.RIGHT]).resolve()
    pickup_cereal.perform()

    # Step 3: Navigate to the bowl
    navigate_to_bowl = NavigateAction(target_locations=[Pose([1.4, 0.50, 0.89])]).resolve()
    navigate_to_bowl.perform()

    # Step 4: Pick up the bowl
````
````

````
    pickup_bowl = PickUpAction(object_designator_description=bowl, arms=[Arms.RIGHT], grasps=[Grasp.RIGHT]).resolve()
    pickup_bowl.perform()

    # Step 5: Navigate to the placement location
    navigate_to_placement = NavigateAction(target_locations=[Pose([1.4, 0.75, 0.95])]).resolve()
    navigate_to_placement.perform()

    # Step 6: Place the cereal
    place_cereal = PlaceAction(object_designator_description=cereal, target_locations=[Pose([1.4, 0.75, 0.95])], arms=[Arms.RIGHT]).resolve()
    place_cereal.perform()

    # Step 7: Place the bowl
    place_bowl = PlaceAction(object_designator_description=bowl, target_locations=[Pose([1.4, 0.70, 0.95])], arms=[Arms.RIGHT]).resolve()
    place_bowl.perform()
```

### Explanation
- **Initialization**: The world and objects are created, defining their properties and positions.
- **Navigation and Actions**: Each action is resolved and performed sequentially, demonstrating how Action Designators facilitate complex workflows.

### Adaptability
This code can be adapted for similar tasks by changing the object types, positions, and actions as needed.

## Framework Integration and Concepts

### Broader Context
This task exemplifies how Action Designators can be used to automate robotic tasks in the PyCram framework, showcasing the modularity and flexibility of the design.

### Essential Components
- **Action Designators**: Central to executing high-level actions.
- **Object Designators**: Essential for identifying and interacting with objects in the environment.

### Conceptual Understanding
Understanding Action Designators is crucial for leveraging the full capabilities of the PyCram framework, enabling the creation of sophisticated robotic behaviors.

## Best Practices and Considerations

### Implementation Tips
- Ensure that object poses are accurately defined to avoid navigation errors.
- Test each action in isolation before integrating into a larger workflow.

### Potential Challenges
- Objects may not be reachable due to environmental constraints.
- The robot may misinterpret object locations if poses are not defined correctly.

### Solutions
- Use debugging tools within PyCram to visualize object positions and robot movements.
- Adjust poses and navigation paths based on real-time feedback from the robot's sensors.

By following this guide, you can effectively implement Action Designators in PyCram to automate complex robotic tasks, enhancing the robot's capabilities in a simulated environment.
````

10.2.10 Retriever Code

For this test, I have shortened the output since it's quite lengthy. However, the full output can be viewed in the PyCRAM GitHub repository.

Input:
````

How is CostmapLocation defined?

Output:

```
# Content from: #<src/pycram\designators\specialized_designators\location\database_location.py>#
from dataclasses import dataclass

import sqlalchemy.orm
import sqlalchemy.sql
from sqlalchemy import select, Select
from typing_extensions import List, Type
from ....costmaps import Rectangle, OccupancyCostmap
from ....designator import LocationDesignatorDescription
from ....designators.location_designator import CostmapLocation
from ....orm.views import PickUpWithContextView
from ....datastructures.pose import Pose

... ... ...

    def sample_to_location(self, sample: sqlalchemy.engine.row.Row) -> Location:
        """
        Convert a database row to a costmap location.

        :param sample: The database row.
        :return: The costmap location
        """

        target_x, target_y, target_z = self.target.pose.position_as_list()
        position = [target_x + sample[3], target_y + sample[4], 0]
        orientation = [sample[5], sample[6], sample[7], sample[8]]

        result = Location(Pose(position, orientation), sample.arm, sample.torso_height, sample.grasp)
        return result

    def __iter__(self) -> Location:
        statement = self.create_query_from_occupancy_costmap().limit(200)
        samples = self.session.execute(statement).all()
        if samples:
            for sample in samples:
                yield self.sample_to_location(sample)
        else:
            raise ValueError("No samples found")
```

—— 2nd content ——

```

# Content from: #<src/pycram\designators\location_designator.py>#
import dataclasses
import time

from typing_extensions import List, Union, Iterable, Optional, Callable

from .object_designator import ObjectDesignatorDescription, ObjectPart
from ..datastructures.world import World, UseProspectionWorld
from ..local_transformer import LocalTransformer
from ..world_reasoning import link_pose_for_joint_config
from ..designator import DesignatorError, LocationDesignatorDescription
from ..costmaps import OccupancyCostmap, VisibilityCostmap, SemanticCostmap, GaussianCostmap
from ..datastructures.enums import JointType, Arms
from ..pose_generator_and_validator import PoseGenerator, visibility_validator, reachability_validator
```

```python
from ..robot_description import RobotDescription
from ..datastructures.pose import Pose


class Location(LocationDesignatorDescription):
    """
    Default location designator which only wraps a pose.
    """

... ... ...

    def __iter__(self):
        """
        Creates a costmap on top of a link of an Object and creates positions from it. If there is a specific Object for
        which the position should be found, a height offset will be calculated which ensures that the bottom of the Object
        is at the position in the Costmap and not the origin of the Object which is usually in the centre of the Object.

        :yield: An instance of SemanticCostmapLocation.Location with the found valid position of the Costmap.
        """
        sem_costmap = SemanticCostmap(self.part_of.world_object, self.urdf_link_name)
        height_offset = 0
        if self.for_object:
            min_p, max_p = self.for_object.world_object.get_axis_aligned_bounding_box().get_min_max_points()
            height_offset = (max_p.z - min_p.z) / 2
        for maybe_pose in PoseGenerator(sem_costmap):
            maybe_pose.position.z += height_offset
            yield self.Location(maybe_pose)

```

10.2.11 LLM Tool Code

The prompt can be found in section 10.1. The test was conducted in the context of **??**.

Input:

How is CostmapLocation defined?

Output:

````
# Comprehensive Guide to Using `CostmapLocation` in PyCram

## 1. Function Explanation and Contextualization

### Detailed Description
`CostmapLocation` is a class within the PyCram framework that utilizes costmaps to create locations based on complex constraints such as reachability and visibility. It serves as a designator that helps in determining valid poses for a robot to interact with objects in its environment, taking into account various factors like obstacles and the robot's capabilities.

### Syntax and Parameters
The constructor for `CostmapLocation` is defined as follows:

```python
def __init__(self, target: Union[Pose, ObjectDesignatorDescription.Object],
             reachable_for: Optional[ObjectDesignatorDescription.Object] = None,
             visible_for: Optional[ObjectDesignatorDescription.Object] = None,
             reachable_arm: Optional[Arms] = None, resolver: Optional[Callable] = None):
```

- **Parameters:**
  - `target`: The location for which visibility or reachability should be calculated. This can be a `Pose` or an `ObjectDesignatorDescription.Object`.
  - `reachable_for`: An optional object (usually a robot) for which reachability should be calculated.
  - `visible_for`: An optional object (usually a robot) for which visibility should be calculated.
````

````
  - `reachable_arm`: An optional arm that should be used to reach the target.
  - `resolver`: An alternative specialized designator that returns a resolved location for the given input.

### Integration
`CostmapLocation` integrates into the PyCram framework as part of the designator system, which is responsible for determining valid locations for robot actions. It leverages costmaps to assess the environment and generate potential poses that meet the specified constraints.

### Relationship with Other Components
`CostmapLocation` interacts with several other components:
- **Costmaps**: It uses various types of costmaps (e.g., `OccupancyCostmap`, `VisibilityCostmap`, `GaussianCostmap`) to evaluate the environment.
- **Pose Generators**: It employs pose generators to create candidate poses based on the merged costmaps.
- **Validators**: It utilizes validators to ensure that generated poses are valid concerning reachability and visibility constraints.

## 2. Code Examples and Implementation

### Full Function Code
Here is the complete code for the `CostmapLocation` class:

```python
class CostmapLocation(LocationDesignatorDescription):
    """
    Uses Costmaps to create locations for complex constraints.
    """

    @dataclasses.dataclass
    class Location(LocationDesignatorDescription.Location):
        reachable_arms: List[Arms]

    def __init__(self, target: Union[Pose, ObjectDesignatorDescription.Object],
                 reachable_for: Optional[ObjectDesignatorDescription.Object] = None,
                 visible_for: Optional[ObjectDesignatorDescription.Object] = None,
                 reachable_arm: Optional[Arms] = None, resolver: Optional[Callable] = None):
        super().__init__(resolver)
        self.target: Union[Pose, ObjectDesignatorDescription.Object] = target
        self.reachable_for: ObjectDesignatorDescription.Object = reachable_for
        self.visible_for: ObjectDesignatorDescription.Object = visible_for
        self.reachable_arm: Optional[Arms] = reachable_arm

    def ground(self) -> Location:
        return next(iter(self))

    def __iter__(self):
        # Implementation of pose generation and validation
        ...
```

### Demonstration Snippets
To demonstrate how `CostmapLocation` can be used, consider the following example:

```python
# Assuming 'bowl' and 'robot' are defined Object instances
costmap_location = CostmapLocation(target=bowl, reachable_for=robot)
for location in costmap_location:
    print(location)
```

### Step-by-Step Explanation
1. **Initialization**: Create an instance of `CostmapLocation` by specifying the target object (e.g., a bowl) and the robot that will interact with it.
2. **Iteration**: Use the `__iter__` method to generate valid poses based on the constraints defined during initialization.
3. **Output**: Each iteration yields a valid location that the robot can reach or see.

### Adaptation
````

```
To adapt `CostmapLocation` for different tasks, modify the `target`, `reachable_for`, and `visible_for` parameters based on the specific objects and constraints relevant to your scenario.

## 3. General Framework Functionality

### Fundamental Concepts
The PyCram framework is designed for robotic manipulation tasks, focusing on the interaction between robots and their environments. Key concepts include:
- **Designators**: Abstractions that represent objects or locations in the environment.
- **Costmaps**: Data structures that represent the environment, indicating where a robot can or cannot go.
- **Pose Generation**: The process of creating potential robot poses based on environmental constraints.

### Key Principles
Understanding the principles of reachability and visibility is crucial for effectively using `CostmapLocation`. These principles dictate how the robot can interact with objects based on its physical capabilities and the layout of the environment.

### Importance
These concepts are essential for ensuring that robots can perform tasks safely and efficiently, avoiding collisions and ensuring that they can reach their targets.

## 4. Best Practices and Recommendations

### Effective Usage
- Always define the `target` and `reachable_for` parameters to ensure that the generated poses are relevant to the task.
- Utilize the `reachable_arm` parameter to specify which arm should be used for reaching, enhancing the specificity of the generated poses.

### Common Pitfalls
- Failing to specify the `reachable_for` or `visible_for` parameters can lead to invalid poses being generated.
- Not considering the robot's physical constraints may result in poses that are unreachable.

### Optimization Tips
- Use smaller costmap sizes for faster computations if the environment allows for it.
- Pre-compute costmaps for static environments to reduce runtime overhead.

## 5. Planning and Implementation for Developers

### Implementation Plan
1. **Define Objects**: Identify the objects and the robot that will interact with them.
2. **Create CostmapLocation**: Instantiate `CostmapLocation` with the appropriate parameters.
3. **Iterate for Poses**: Use the iterator to generate valid poses and implement logic to utilize these poses in your robotic tasks.

### Integration Steps
- Ensure that the necessary costmaps are initialized and available in the environment.
- Integrate `CostmapLocation` into your robotic control loop to dynamically generate poses based on the current state of the environment.

### Customization Guidance
- Modify the `__iter__` method to include additional constraints or logic specific to your application.
- Extend the `CostmapLocation` class to include additional functionality as needed for your specific use case.

By following this guide, developers can effectively utilize the `CostmapLocation` class within the PyCram framework to enhance robotic manipulation tasks.
```

10.2.12 Prethinking

The exact prompt can be found in section 10.1.

Input:

Instruction: Place the cereal and a bowl side by side on the kitchen island. World knowledge:

```
[kitchen = Object('kitchen', ObjectType.ENVIRONMENT, 'kitchen.urdf'),
robot = Object('pr2', ObjectType.ROBOT, 'pr2.urdf')
bowl = Object('bowl', ObjectType.BOWL, 'bowl.stl', pose=Pose([1.4, 0.50, 0.89]),
```

```
color=[1, 1, 0, 1]),
cereal = Object('cereal', ObjectType.BREAKFAST_CEREAL, 'breakfast_cereal.stl',
pose=Pose([1.4, 1, 0.95]))]
```

Output:

```
## Placing Cereal and Bowl on Kitchen Island

**Initial stage:**
- **Cereal box:** Located on the ground approximately at position **[1.4, 1, 0.95]**.
- **Bowl:** Located on the ground approximately at position **[1.4, 0.50, 0.89]**.
- **PR2 Robot:** Positioned near the objects, approximately at position **[1.4, 0.75, 0]**.
- **Kitchen Island:** Located in the kitchen, with its surface approximately at height **0.84** (assuming a standard kitchen island height based on the provided URDF). The target area for placement is approximately **[-1.07, 1.72, 0.84]** and **[-1.27, 1.72, 0.84]** for the cereal and bowl respectively (allowing space for both objects side by side).

**Goal stage:**
- **Cereal box:** Located on the kitchen island surface approximately at position **[-1.07, 1.72, 0.84]**.
- **Bowl:** Located on the kitchen island surface approximately at position **[-1.27, 1.72, 0.84]**, next to the cereal box.

**Step-by-step plan:**

1. **Robot positioning (Cereal):**
   - **Action:** The PR2 robot moves to position **[1.4, 1, 0]**, near the cereal box.

2. **Grabbing the cereal box:**
   - **Action:** The robot navigates to the cereal box at **[1.4, 1, 0.95]**.
   - **Action:** Securely grab the cereal box.

3. **Movement to the kitchen island (Cereal):**
   - **Action:** The robot transports the cereal box from **[1.4, 1, 0.95]** to the kitchen island approximately at **[-1.07, 1.72, 0.84]**.

4. **Placing the cereal box:**
   - **Action:** Carefully place the cereal box on the kitchen island approximately at position **[-1.07, 1.72, 0.84]**.

5. **Robot positioning (Bowl):**
   - **Action:** The PR2 robot moves to position **[1.4, 0.50, 0]**, near the bowl.

6. **Grabbing the bowl:**
   - **Action:** The robot navigates to the bowl at **[1.4, 0.50, 0.89]**.
   - **Action:** Securely grab the bowl.

7. **Movement to the kitchen island (Bowl):**
   - **Action:** The robot transports the bowl from **[1.4, 0.50, 0.89]** to the kitchen island approximately at **[-1.27, 1.72, 0.84]**.

8. **Placing the bowl:**
   - **Action:** Carefully place the bowl on the kitchen island approximately at position **[-1.27, 1.72, 0.84]**, next to the cereal box.

9. **Task completion:**
   - **Action:** Finish the task and return to the initial position or prepare for the next instruction.


**Note:** The exact positions for placing the cereal and bowl on the kitchen island might need slight adjustments based on the specific dimensions and layout of the kitchen island obtained from the 'kitchen.urdf' file. The provided positions are estimations based on typical kitchen layouts and the need to place the objects side by side.
```

10.2.13 URDF Tool

This test was conducted in the context of Test 4. **Input:**

kitchen.urdf

Output:

```
### Summary of the URDF File

The provided URDF file describes a kitchen environment, including various components such as walls, a kitchen island, a sink area, an oven area, and a fridge area. Each component is defined with its physical properties (mass, inertia), visual representation (mesh files),
and collision geometry. The kitchen is structured with fixed joints connecting different parts, ensuring they remain in place relative to one another.

### Important Data from the URDF File

1. **Kitchen Structure:**
   - **Walls:** Six walls with defined dimensions and positions.
   - **Kitchen Island:**
     - **Footprint:** Fixed joint with mass and inertia.
     - **Surface:** Box geometry for collision.
     - **Stove:** Mesh representation.
   - **Sink Area:**
     - **Components:** Sink, surface, and drawers with defined positions and meshes.
   - **Oven Area:**
     - **Components:** Oven, door, and knobs with defined positions and meshes.
   - **Fridge Area:**
     - **Components:** Fridge, door, and drawers with defined positions and meshes.

2. **Robot and Objects:**
   - **Robot:** PR2 robot is referenced but not detailed in the URDF.
   - **Bowl:** Positioned at `[1.4, 0.50, 0.89]` with a yellow color.
   - **Cereal:** Positioned at `[1.4, 1, 0.95]`.

3. **Joints:**
   - Fixed joints for walls, kitchen island, sink, oven, and fridge components.
   - Prismatic joints for drawers allowing sliding motion.

4. **Visual and Collision Geometry:**
   - Each component has associated mesh files for visual representation and collision detection.

### World Model

In the kitchen environment, there is a structured layout consisting of walls that enclose the space. The kitchen island is centrally located, featuring a stove on its surface. Adjacent to the island is a sink area equipped with drawers for storage. The oven area is positioned nearby, complete with an oven and its door. A fridge area is also present, containing drawers for food storage.

The robot (PR2) is capable of interacting with the environment, specifically with the bowl and cereal, which are positioned on the kitchen island. The bowl is yellow and located at a height of 0.89 meters, while the cereal is placed slightly higher at 0.95 meters. The instruction to place the cereal and bowl side by side on the kitchen island can be accomplished by the robot, utilizing its capabilities to navigate and manipulate objects within this structured kitchen environment.

Overall, the URDF file provides a comprehensive representation of a kitchen, detailing the physical properties and spatial relationships of its components, which are essential for understanding how to interact with the environment effectively.
```