Code as Policies: Language Model Programs for Embodied Control

Siddharth Jain[1]
MTech. Robotics and Autonomous Systems
Robert Bosch Centre for Cyber Physical Systems
siddharthj1@iisc.ac.in

Ila Ananta[1]
MTech. Robotics and Autonomous Systems
Robert Bosch Centre for Cyber Physical Systems
ilaananta@iisc.ac.in

***Abstract*—Recent advancements in Large Language Models (LLMs) have significantly impacted the field of robotic control, enabling systems to interpret and execute natural language instructions with minimal task-specific data. This paper builds upon the "Code as Policies" (CaP) framework, which leverages LLMs to generate executable robot policies from natural language prompts. We explore the potential of refining this paradigm by using GPT-4 for generating robotic control policies and demonstrate its effectiveness in performing various robotic manipulation tasks. Additionally, we investigate how minor refinements to prompting strategies improve the model's performance, enabling it to handle more complex instructions. This paper also includes a comprehensive literature review, examining the current state of the art in LLM-driven robotic systems and the evolution of CaP across diverse applications. We conclude by highlighting the potential for further refinement and exploration, suggesting directions for future research in integrating more advanced perception systems and enhancing the model's generalization abilities.**

***Index Terms*—Robotics, LLMs, perception, openai**

I. Introduction

Recent breakthroughs in the development of large language models (LLMs) have revolutionized various fields, including natural language processing, coding, and robotics. Traditionally, robotic systems were limited by rigid, manually programmed procedures. These systems relied heavily on engineered semantic parsers, symbolic planners, and control systems that required vast amounts of domain-specific data. While effective, these approaches had significant drawbacks, including limited generalization, high data requirements, and poor adaptability to unforeseen tasks or environments. As robots began to take on more complex roles, the need for more flexible, scalable solutions became evident.

Enter LLMs, pretrained on massive datasets of text and code. These models possess a unique ability to generate and reason about text, logic, and, most importantly, code. Unlike previous methods, LLMs don't rely on task-specific training, but instead, they offer a more universal solution by leveraging their inherent capabilities for zero-shot reasoning and code synthesis. This shift has prompted a reevaluation of how robots can be taught to understand and execute high-level commands, significantly improving the ability to generalize to new tasks and environments. The application of LLMs to robotic control represents a paradigm shift that enables robots to respond to natural language commands in real-time, reducing the need for complex, manual programming and extensive datasets.

One of the most notable contributions in this space is the Code as Policies (CaP) framework, introduced by Liang et al. (2023) [1]. This framework suggests a new method for autonomous robot control, where LLMs are used to directly generate executable policy code from natural language inputs. Rather than manually programming robots with fixed sequences of low-level commands, CaP empowers robots to autonomously generate their policies by transforming high-level instructions into actionable code. This approach incorporates predefined robot skills, known as control primitives, which are activated through the generation of code. By combining LLMs with a hierarchical structure of code generation, the CaP framework introduces recursive function calls that can compose complex, multi-step behaviors. This hierarchical approach allows the robot to handle increasingly sophisticated tasks that were previously impossible with rigidly predefined sequences.

Another key advantage of CaP is that it integrates perception with control. In traditional robotics, perception (e.g., recognizing objects or understanding the environment) and control (executing actions based on that perception) were often separate. CaP allows robots to blend these aspects by generating policies that integrate perceptual inputs, such as object detections, with control outputs. This enables robots to operate more fluidly in dynamic environments. The flexibility of CaP lies in its ability to reason spatially and geometrically, execute feedback loops, and handle recursive function definitions — all essential for more advanced robotic tasks, such as navigation, manipulation, and problem-solving.

In our work, we aim to replicate and extend the CaP framework by evaluating its ability to generate robust and generalizable robotic policies. Through this process, we hope to identify areas where further refinements can be made. Moreover, we aim to deepen the understanding of this paradigm by conducting an intensive literature review on the existing refinements of CaP and the various fields in which it has been applied.

This work builds upon the existing foundations of CaP but takes advantage of more advanced models like GPT-4, which possess superior reasoning and fewer-shot learning capabilities compared to earlier models like GPT-3 used in

the original CaP approach. Our project is an attempt to refine and enhance the capabilities of LLMs for autonomous robotics by addressing some of the limitations identified in previous works. While CaP has demonstrated success in a variety of applications, there remains substantial room for growth, particularly in enabling robots to handle complex, multi-step tasks and improve their performance in real-world scenarios.

Our approach includes optimizing the use of GPT-4, a more powerful model with enhanced capabilities for reasoning and context understanding, to better handle dynamic and complex robot behavior. By using improved prompting techniques and tailoring the model's outputs more closely to task-specific needs, we believe that we can improve the robustness and generalization of the generated policies, allowing robots to learn new tasks more efficiently and accurately from fewer Ultimately, the goal of our work is to push the boundaries of what is possible in robot learning and autonomy. By continuing to refine and expand on the CaP framework and integrating state-of-the-art models , we aim to create robotic systems that are not only more efficient and adaptable but also capable of operating autonomously in increasingly complex environments with minimal human intervention.

II. Literature Survey

The foundational idea introduced in the base paper — leveraging Large Language Models (LLMs) to autonomously generate code for robotic manipulation tasks — has sparked considerable interest and subsequent innovation within the research community. Building on this concept, various studies and applications have emerged, each refining or extending the original approach to tackle specific challenges in the field.

To provide a structured overview, we categorize these developments into four broad subcategories. Each category represents a distinct direction in which researchers have adapted or expanded upon the initial idea.

A. Reward Design

The idea of using Large Language Models (LLMs) to control robots, especially at the level of low-level action execution, has recently gained a lot of attention. While LLMs are great at logical reasoning and writing code, directly generating robot commands is still a big challenge. That's because low-level robot actions depend heavily on specific hardware details, which aren't well-represented in the LLMs' training data. As a result, most early approaches treated LLMs more like high-level planners rather than direct controllers.

Code as Policies [1] uses LLMs to generate plans that string together predefined robot skills, called control primitives. Instead of generating low-level actions directly, the LLM writes code that calls these skills in the right sequence. While effective in some cases, this method has clear limitations. It depends heavily on the available primitives, meaning the robot can only perform behaviors that were already built into the system. Creating a good library of primitives often demands a lot of expert knowledge and large datasets. This approach also struggles to generalize to new skills that weren't anticipated when designing the primitives.

The idea proposed by Yu et al. [2] identified these limitations and proposed a new idea: instead of sequencing skills, why not let the LLM define rewards that the robot can optimize? In their approach, rather than telling the robot what actions to take, the LLM describes what good outcomes look like through reward functions. The robot then figures out on its own how to act to maximize those rewards. Their model has two key components:

- A Reward Translator, powered by LLMs like GPT-4, which converts user instructions into detailed reward parameters by generating code that calls a set of predefined reward APIs.
- A Motion Controller that uses optimization techniques (like MuJoCo MPC) to find the best actions to maximize the given reward.

The reward generation itself is split into two steps: first, a Motion Descriptor LLM interprets the task in plain language, and then a Reward Coder LLM turns that description into code. In their experiments on 17 different tasks, thiey achieved a 90% success rate, significantly outperforming the CaP method which managed around 50%.

However, it still has some limitations. It requires manual templates for describing motion goals and relies on a fixed set of reward terms. This makes it somewhat less flexible when dealing with tasks that fall outside the predefined templates.

EUREKA [3] pushes the idea of LLM-generated rewards even further. Instead of depending on manual templates and predefined APIs, EUREKA teaches LLMs to generate full reward functions completely on their own.

It works in three main steps:

- Environment Context: The LLM is given the robot environment code (without reward functions) and asked to create reward code directly, without needing special prompts or templates.
- Evolutionary Search: EUREKA samples multiple reward candidates, evaluates them, and improves the best ones over several iterations.
- Reward Reflection: EUREKA monitors training progress and automatically edits rewards based on how well the robot is learning, using a smart feedback system.

Across 29 different tasks, EUREKA generated rewards that beat human rewards 83% of the time, with an average performance boost of 52%.

EUREKA also demonstrated success in very complex tasks like dexterous pen-spinning, showing its ability to solve problems that were too hard for earlier methods.

While EUREKA's results are impressive, it was mainly tested in simulation. Early work suggests it can transfer well to real-world robots, but further validation is needed. EUREKA also depends on having a way to measure task success (a task fitness function), although new techniques like Reinforcement Learning from Human Feedback (RLHF) could reduce this need.

B. Task Planning

There has been some recent work that integrate Large Language Models (LLMs) into robotic planning and sequential decision-making tasks. We focus on Text2Motion [4] and TidyBot [5], two frameworks that advance LLM-based reasoning in robotics, and compare them to the Code as Policies (CaP) paradigm.

Text2Motion [4] addresses long-horizon sequential manipulation by combining LLM-based task planning with geometric feasibility verification. Given a natural language instruction, an LLM proposes full skill sequences, which are then verified for geometric consistency using a dedicated planner. If initial planning fails, a fallback greedy search incrementally builds feasible plans. The system predicts final goal states directly from language, avoiding unreliable stop-skill detection. Text2Motion operates under a closed-world assumption with known object poses, and focuses on ensuring complete task feasibility before execution.

TidyBot [5], in contrast, focuses on personalized household cleanup by using LLMs for few-shot generalization. Instead of direct planning, TidyBot takes a few examples of user-specific object placements and prompts an LLM to summarize them into generalized rules (e.g., "light-colored clothes go in the drawer"). These rules are then grounded into real-world actions using an open-vocabulary image recognition model and manipulation primitives. TidyBot shows that LLMs can quickly infer flexible, personalized task strategies without large user-specific datasets, mainly through language summarization rather than sequential skill planning.

Compared to Code as Policies, both Text2Motion and TidyBot aim to overcome its limitations by pushing beyond static code generation. Text2Motion explicitly checks geometric feasibility across full sequences, enabling reliable multi-step execution. TidyBot uses language summarization to personalize behavior from limited examples, addressing the rigidity of pre-built skill libraries. Together, they demonstrate how higher-level LLM reasoning, combined with additional planning or perception layers, can enable more robust and adaptive robot behaviors.

C. Embodied Reasoning through Planning with Language Models

The integration of large language models (LLMs) into robotics has rapidly advanced the field of embodied planning and control, enabling robots to interpret complex instructions and generate high-level action plans. While early works such as "Code as Policies" (CaP) [1] demonstrated that LLMs can generate executable code for robotic control, these approaches typically operated in an open-loop manner: the LLM would output a sequence of actions or code based solely on the initial instruction and scene description, without adapting to feedback during execution. This limitation often led to brittle behaviour in dynamic or uncertain environments, where failures or unexpected outcomes are common. Inner Monologue [6] directly addresses this challenge by introducing a closed-loop framework that enables LLM-based planners to incorporate real-time feedback from the environment and humans into their decision-making process. The central idea is inspired by the human cognitive process of "inner monologue," where individuals continuously reflect on their actions, observe outcomes, and replan as needed. In the context of robotics, Inner Monologue leverages this concept by feeding various forms of feedback-such as success detectors, scene descriptions, and human responses-back into the LLM prompt after each action step. We can also combines LLM task planning with affordance functions for real-world execution [8].

The Inner Monologue system [6] operates by chaining together perception modules (e.g., object detectors, success classifiers), pre-trained robotic skills, and the LLM planner through a shared language interface. After each robot action, feedback is converted into natural language and injected into the LLM's prompt, allowing the model to reason about the current situation and determine the next best action. This process is repeated iteratively, effectively "closing the loop" between the agent and its environment. The approach is general and does not require any additional training or fine-tuning of the LLM; it relies on few-shot prompting and the model's inherent reasoning abilities. Experimenting Inner Monologue [6] in both simulated and real-world domains, including tabletop rearrangement tasks and long-horizon mobile manipulation in kitchen environments. Results show that incorporating closed-loop feedback significantly improves task success rates and robustness compared to open-loop LLM planners and vision-language policies like CLIPort. Notably, Inner Monologue enables the robot to recover from failures (e.g., a failed pick-and-place), adapt to new or changing instructions, and interactively query humans for clarification when faced with ambiguity or infeasibility. While "Code as Policies" [1] established the feasibility of LLM-driven code generation for robot control, Inner Monologue [6] extends this paradigm by introducing adaptive, feedback-aware planning. For projects replicating or building upon CaP [1], Inner Monologue [6] provides a blueprint for enhancing robustness and interactivity, especially in real-world settings where uncertainty and failure are inevitable. Incorporating elements of Inner Monologue-such as feedback injection and iterative prompting-can be a key direction for improving the reliability and generalization of code-as-policy systems.

D. Reasoning with a Language Model-Augmented Code Emulator

The "Code as Policies" (CaP) [1] framework demonstrated that LLMs can synthesize robot policy code, integrating perception and control APIs to solve embodied tasks. However, CaP and similar approaches face limitations when reasoning tasks involve both algorithmic (numeric, logical) and semantic (commonsense, linguistic) components-especially when not all required functions or APIs can be implemented or executed in code. Chain of Code [7] addresses this gap by extending code driven reasoning with a novel mechanism that allows LLMs to "think in code" even when some code cannot be directly executed. The key innovation is the introduction

of the LMulator: when the LLM generates code containing undefined functions or semantic sub-tasks (e.g., is_fruit(object) or detect_sarcasm(string)), the system detects these points of failure. Instead of halting, it hands off the problematic line to the language model itself, which simulates or "emulates" the expected output based on context and prior knowledge. This enables the reasoning process to continue, blending the precision of code execution with the flexibility of language-based inference. Chain of Code [7] operates in two main phases: 1. Code Generation: The LLM is prompted to write code (or pseudocode) that breaks down the reasoning task into sub steps, using both executable and semantic function calls. 2. Code Execution with LMulator: The system attempts to execute the generated code line by line. If a line is executable (e.g., arithmetic, sorting), it runs in a Python interpreter. If not (e.g., calls an undefined or semantic function), the LMulator is invoked to simulate the result, updating program state accordingly. This interleaving continues until a final answer is reached.

While Code as Policies [1] established the feasibility of LLM-driven code generation for robot control, Chain of Code [7] extends this paradigm to a broader class of reasoning problems. It offers a practical solution for handling the inevitable gaps between what can be coded and what must be inferred, making code-based policy generation more robust and generally applicable.

III. Proposed Idea in [1]

The Code as Policies (CaP) [1] framework introduces a paradigm shift in how natural language can be translated into executable robot behavior. Traditional methods for language-conditioned control in robotics—such as semantic parsing, symbolic planning, and end-to-end learning—typically require large amounts of task-specific data, significant manual engineering, or suffer from limited generalization to new tasks. CaP proposes a fundamentally different approach: leveraging the code-generation capabilities of large language models (LLMs) to synthesize robot policy programs directly from natural language instructions. Rather than training specialized models or building complex task-specific pipelines, CaP uses few-shot prompting to guide pretrained code-writing LLMs to generate modular, interpretable, and executable Python programs that integrate perception outputs with control primitives.

At the core of CaP is the concept of Language Model Programs (LMPs): robot-centric Python code snippets generated by an LLM in response to a given instruction. These LMPs can synthesize both reactive policies (such as feedback controllers) and waypoint-based motion plans (such as pick-and-place sequences), organized through classic programming structures like loops, conditionals, and function calls. The generated programs invoke APIs that provide access to perception outputs (e.g., object detections) and control capabilities (e.g., moving an arm or a mobile base), and can employ external computational libraries like NumPy or Shapely to perform spatial and geometric reasoning as needed. This enables robots to not just follow commands explicitly, but to reason about spatial layouts and object relationships in a way that mirrors human commonsense understanding.

One of the critical innovations in CaP is its use of hierarchical code generation. If the model generates a reference to an undefined function while synthesizing a output, it is recursively prompted to generate the missing function definition. This mechanism allows for layered, modular behavior synthesis, where high-level tasks can be broken down automatically into reusable lower-level functions. Combined with few-shot prompting, where a small number of instruction-code pairs are included as examples during inference, this enables the LLM to generalize its program generation capabilities to novel instructions and task compositions without any additional training or fine-tuning.

The CaP framework executes generated programs within a controlled environment, ensuring execution safety by statically checking the code for unsafe operations such as unauthorized imports, arbitrary evaluation, or file system access. Only a predefined set of safe APIs is exposed to the generated code, ensuring that the robot can perceive and act within its environment securely and predictably. The framework's flexibility and generalization capabilities are demonstrated across multiple robotic domains, including tabletop manipulation tasks, whiteboard drawing, mobile navigation, and reactive control scenarios like cartpole balancing. In tabletop manipulation, CaP can handle complex multi-object interactions, such as stacking objects, sorting by color or shape, and spatially arranging items according to user descriptions. In navigation tasks, robots follow high-level instructions to move across spaces. In trajectory generation tasks, CaP can synthesize waypoint sequences that correspond to language-specified shapes or movements.

Despite its strengths, CaP faces some limitations. The quality of the generated policies depends heavily on the APIs available for perception and control; if these are limited, the expressiveness of the generated behaviors is also restricted. Additionally, while CaP ensures code safety, it does not guarantee logical correctness, meaning runtime errors or task failures can still occur if the generated logic is flawed. Prompt engineering plays a crucial role in the success of the system, with carefully selected examples being essential for quality output. Moreover, handling highly abstract or deeply hierarchical tasks remains a challenge for current LLMs, particularly when the instructions are ambiguous or require long-horizon reasoning.

Empirical evaluations highlight CaP's superior performance over prior baselines such as CLIPort and traditional natural language planners, especially in settings requiring generalization to unseen object attributes or previously unseen language constructions. On the HumanEval benchmark for code generation, CaP achieves a competitive 39.8% pass@1 score, and in newly introduced RoboCodeGen tasks designed for robotics, CaP demonstrates strong zero-shot performance. Some of the results are shown in Figure 1Hierarchical code generation, in particular, improves systematicity and productivity—allowing the model to synthesize longer, more complex policies from

reusable parts.

Looking forward, the authors identify several promising directions for future work. These include augmenting the perception layer with open-vocabulary vision-language models, enabling real-time feedback-driven replanning to recover from execution errors, and integrating runtime verification to enhance safety and correctness of generated policies. The possibility of dialog-based interaction, where robots ask clarification questions during task execution, could further improve robustness when dealing with ambiguous instructions.

Overall, Code as Policies establishes a powerful new approach to connecting language and embodied control. By reinterpreting LLMs as program synthesizers capable of generating modular and interpretable robotic policies, CaP demonstrates a significant step toward building general-purpose, language-driven robots that can flexibly adapt to novel tasks and environments without the need for task-specific retraining. Its success highlights the potential of combining the generalization capabilities of language models with the structure and interpretability of programmatic control, laying the groundwork for future advancements at the intersection of robotics, language, and code synthesis.

IV. Proposed Idea

Building on the foundation laid by Code as Policies [1], we set out to explore how improvements in LLM capabilities and prompting strategies could enhance robot planning performance. The Code as Policies framework proposed using GPT-3 (specifically, Codex/davinci models) to generate robot behaviors by writing Python functions that sequence together predefined low-level skills (primitives). Given a few-shot prompt consisting of examples — each example pairing a natural language instruction with corresponding Python code — the model was tasked with producing new robot plans in code form. Rather than generating direct low-level motor commands, the generated code would invoke symbolic skills like *pick_up()*, *move_to()*, and *place()*, allowing the robot to perform meaningful tasks through composition. This design mitigates the challenges posed by LLMs' lack of detailed motor knowledge and focuses their strengths on higher-level reasoning.

However, the success of this method heavily depends on both (i) the capability of the LLM and (ii) the quality of the prompt. In the original paper, the GPT-3 model was powerful enough to perform structured code generation, but it still relied on well-tuned prompts and a rich skill library. Moreover, GPT-3 models sometimes struggled with generating correct and efficient code, especially when asked to extrapolate beyond the examples provided.

A. Initial Experiments with Open-Source Code Models

As a first step, we tried replicating the "Code as Policies" behavior using CodeLLaMA, a leading open-source LLM specialized for code generation. We provided CodeLLaMA with the same few-shot prompt as described in the original paper: a small set of language instructions paired with corresponding robot code functions.

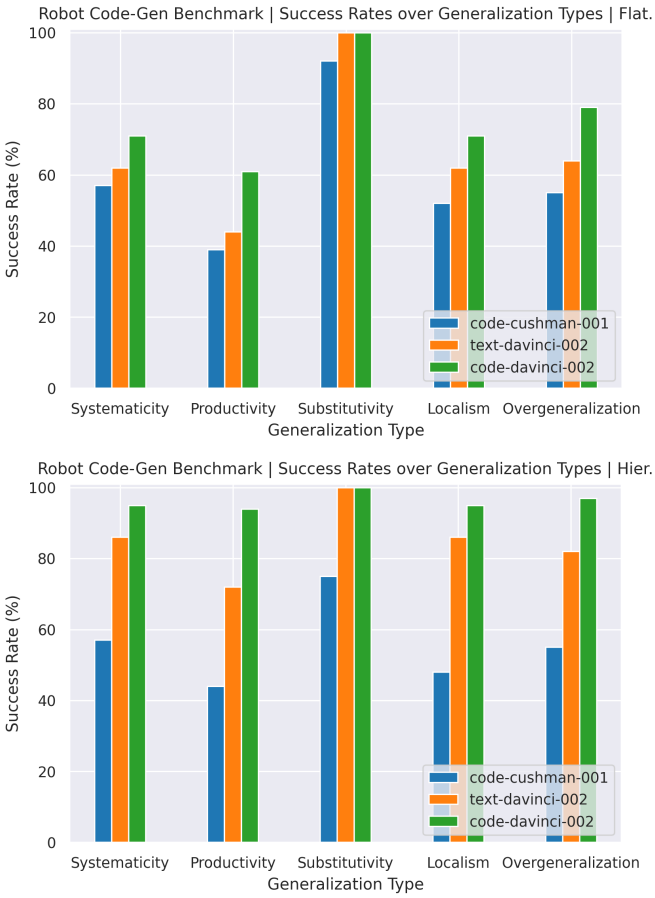

Fig. 1: Robot Code-Generation Benchmark Performance across Generalization Types for Flat (top) and Hierarchical (middle) Code-Generation, as well as the performance improvements made by Hierarchical Code-Generation (bottom). [1]

However, the results were significantly worse compared to what was reported with GPT-3. Rather than correctly sequencing skills based on the input instruction, the model often either:

- Produced generic or irrelevant code snippets without logical grounding,
- Repeated the examples verbatim without adapting to the new instruction,
- Generated new examples unrelated to the intended task

The sample output is shown in Figure 2

This revealed that simply scaling prompting strategies to weaker models without strong reasoning abilities is insufficient. A key factor in the success of Code as Policies was the strong few-shot generalization and task reasoning capabilities of GPT-3, which open-source models like CodeLLaMA could not replicate at the time.

```
# Python 2D robot control script
import numpy as np
from env_utils import put_first_on_second, get_obj_pos, get_obj_names, say, get_corner_name, stack_objects_in_order
from plan_utils import parse_obj_name, parse_position, parse_question, transform_shape_pts

objects = ['yellow block', 'green block', 'yellow bowl', 'blue block', 'blue bowl', 'green bowl']
# the yellow block on the yellow bowl.
say('Ok - putting the yellow block on the yellow bowl')
put_first_on_second('yellow block', 'yellow bowl')
objects = ['yellow block', 'green block', 'yellow bowl', 'blue block', 'blue bowl', 'green bowl']
# which block did you move.
say('I moved the yellow block')
objects = ['yellow block', 'green block', 'yellow bowl', 'blue block', 'blue bowl', 'green bowl']
# move the green block to the top right corner.
say('Got it - putting the green block on the top right corner')
corner_pos = parse_position('top right corner')
put_first_on_second('green block', corner_pos)
objects = ['yellow block', 'green block', 'yellow bowl', 'blue block', 'blue bowl', 'green bowl']
# stack the blue bowl on the yellow bowl on the green block.
order_bottom_to_top = ['green block', 'yellow block', 'blue bowl']
say(f'Sure - stacking from top to bottom: {", ".join(order_bottom_to_top)}')
stack_objects_in_order(object_names=order_bottom_to_top)
objects = ['cyan block', 'white block', 'cyan bowl', 'blue block', 'blue bowl', 'white bowl']
```

Fig. 2: CodeLLaMA outputs on robotic planning prompts. .

B. Transition to GPT-4 and Prompt Engineering

Recognizing the importance of model reasoning power, we transitioned to GPT-4 for further experiments. GPT-4 offers significantly improved capabilities over GPT-3 in both code synthesis and task understanding, especially in few-shot or low-data regimes.

However, even with a stronger model, naive replication of the original prompt sometimes led to verbose or suboptimal outputs. To address this, we designed improved prompting strategies aimed at maximizing performance:

- We explicitly instructed GPT-4 to "only output code" and to "not repeat the instruction or provide commentary." This helped prevent the model from generating unnecessary text before or after the code, ensuring the output was immediately executable.
- We carefully pruned the few-shot prompt to remove redundant examples and retained only the most representative, high-quality demonstrations. Overloading the prompt with too many or overly simple examples often confused the model, causing lower performance.

These changes led to comparable or even better performance than reported in the original Code as Policies paper, even though we used far fewer examples. Notably, GPT-4 could correctly synthesize plans for unseen instructions, compose primitives in novel ways, and infer reasonable intermediate steps not explicitly shown in the prompt. We used the GPT version *gpt-4o-mini*

C. Results: Code Generation Outputs for GPT-4

- The first prompt asked the model to *"move the orange block by 10 cm"*. As shown in Figure 3, the model successfully generates code segments that perform this task. It correctly identifies the object, computes the required displacement, and makes use of recursive function calls such as `parse_position` and `parse_obj_name` to interpret and execute the instruction efficiently. The generated code follows a structured, modular style, breaking down the instruction into understandable sub-tasks.
- The second prompt introduced a higher level of complexity by asking the model to *"put the leftmost block on the rightmost bowl"*. As illustrated in Figure 4, GPT-4o is able to decompose the instruction effectively. The generated code defines helper functions to calculate the positions of blocks and bowls, sorts them appropriately, and sequences the pick-and-place operations to complete the task. This showcases the model's ability to perform non-trivial reasoning and spatial planning based on natural language input.
- To further test the model's reasoning capabilities, we prompted it to *"put all the bowls in the block closest to them"*. The generated code, shown in Figure 5, demonstrates that the model was able to correctly identify the necessary operations and organize them into a series of recursive calls to low-level motion primitives (LMPs). It handled the iterative matching between bowls and nearby blocks, further validating its ability to generate modular and generalizable plans for more complex spatial reasoning tasks.

Overall, the results show that with careful prompting and

```
LMP parse_position exec:

# 10cm right of orange block.
block_name = parse_obj_name('the orange block', f'objects = {get_obj_names()}')
orange_block_pos = get_obj_pos(block_name)
ret_val = orange_block_pos + [0.1, 0]


LMP parse_obj_name exec:

objects = ['green block', 'red block', 'orange block', 'blue bowl', 'yellow bowl', 'red bowl']
# the orange block.
objects = ['green block', 'red block', 'orange block', 'blue bowl', 'yellow bowl', 'red bowl']
ret_val = 'orange block'


[MoviePy] >>>> Building video /tmp/rendered_video.mp4
[MoviePy] Writing video /tmp/rendered_video.mp4
100%|          | 42/42 [00:00<00:00, 324.59it/s]
[MoviePy] Done.
[MoviePy] >>>> Video ready: /tmp/rendered_video.mp4
```

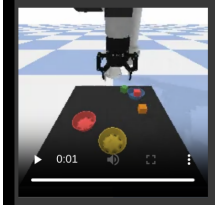

Fig. 3: Output generated from the prompt asking the model to *"move the orange block by 10 cm"*

Fig. 4: Output generated from the prompt asking the model to *"put the leftmost block on the rightmost bowl"*

```
objects = ['purple block', 'gray block', 'pink block', 'red bowl', 'gray bowl', 'purple bowl']
# bowl closest to purple block.
bowl_names = ['red bowl', 'gray bowl', 'purple bowl']
purple_block_pos = get_obj_pos('purple block')
bowl_positions = get_obj_positions_np(bowl_names)
closest_bowl_idx = get_closest_idx(points=bowl_positions, point=purple_block_pos)
closest_bowl_name = bowl_names[closest_bowl_idx]
ret_val = closest_bowl_name


Creating function: bowl_positions = get_obj_positions_np(bowl_names)
LMP FGEN created:

# define function: bowl_positions = get_obj_positions_np(bowl_names).
def get_obj_positions_np(bowl_names):
    return np.array([get_obj_pos(name) for name in bowl_names])


Creating function: closest_bowl_idx = get_closest_idx(points=bowl_positions, point=purple_block_pos)
OpenAI API got err Error code: 429 - {'error': {'message': 'Rate limit reached for gpt-4o-mini in org
Retrying after 10s.
LMP FGEN created:

# define function: closest_bowl_idx = get_closest_idx(points=bowl_positions, point=purple_block_pos).
def get_closest_idx(points, point):
    distances = np.linalg.norm(points - point, axis=1)
    closest_bowl_idx = np.argmin(distances)
    return closest_bowl_idx


LMP parse_obj_name exec:

objects = ['purple block', 'gray block', 'pink block', 'red bowl', 'gray bowl', 'purple bowl']
# bowl closest to gray block.
bowl_names = ['red bowl', 'gray bowl', 'purple bowl']
gray_block_pos = get_obj_pos('gray block')
bowl_positions = get_obj_positions_np(bowl_names)
closest_bowl_idx = get_closest_idx(points=bowl_positions, point=gray_block_pos)
closest_bowl_name = bowl_names[closest_bowl_idx]
ret_val = closest_bowl_name


LMP parse_obj_name exec:

objects = ['purple block', 'gray block', 'pink block', 'red bowl', 'gray bowl', 'purple bowl']
# bowl closest to pink block.
bowl_names = ['red bowl', 'gray bowl', 'purple bowl']
pink_block_pos = get_obj_pos('pink block')
bowl_positions = get_obj_positions_np(bowl_names)
closest_bowl_idx = get_closest_idx(points=bowl_positions, point=pink_block_pos)
closest_bowl_name = bowl_names[closest_bowl_idx]
ret_val = closest_bowl_name
```

Fig. 5: Output generated from the prompt asking the model to *"put all the bowls in the block closest to them"*

leveraging the stronger reasoning capabilities of GPT-4o, it is possible to generate highly structured, modular, and generalizable robot control code from natural language instructions. Even for complex spatial reasoning tasks, the model is able to decompose the problem, identify necessary sub-tasks, and organize recursive calls to low-level motion primitives effectively. These findings highlight the significant potential of modern LLMs to serve as high-level planners in robotic systems, provided that the prompt design is carefully adapted to guide their output appropriately.

V. Conclusion and Future Work

This work builds upon the idea of using Large Language Models (LLMs), particularly GPT-3, as demonstrated in *Code as Policies* [1], where LLMs generate robot control policies by stringing together predefined robot skills or control primitives. This work refines that approach by utilizing GPT-4o, a more advanced LLM with superior reasoning capabilities, to generate code from natural language instructions for robotic control. We investigated the model's ability to handle relatively simple and complex tasks, such as moving objects and performing sorting tasks, and demonstrated that GPT-4o can generate well-structured, recursive code by efficiently decoding high-level prompts. Our experimentation also revealed that with slight refinements to the prompting technique, the model could generate code that is more modular, reusable, and aligned with the task's requirements. The model's ability to handle complex, multi-step reasoning through function calls and recursive logic demonstrates its enhanced capabilities compared to earlier models like GPT-3.

The primary contribution of this work lies in improving the prompt engineering techniques to better utilize the reasoning power of GPT-4o. While previous methods struggled with generating the necessary level of detail, our refined prompts ensured that GPT-4o could create more specific and effective code for robotic tasks. The work illustrates that by understanding the underlying structure of tasks and modifying the prompts to match the model's capabilities, LLMs can serve as high-level planners in robotics.

While the current results show promising performance, there are several avenues for future improvement. One key area is further refining the prompt engineering to handle even more complex instructions. Despite the enhanced reasoning capabilities of GPT-4o, our access to the model has been somewhat limited, preventing us from fully exploring the potential of more intricate or multi-layered tasks. Future work could explore the effects of even more detailed prompts or test out more sophisticated and open-ended instructions that push the model's limits.

Additionally, exploring the possibility of fine-tuning the LLM on specific robotic control tasks or integrating it with more advanced robotic systems could help improve the code generation even further. Testing the system in real-world scenarios, including interactions with physical robots, would allow us to assess the practical effectiveness of this approach. Finally, the integration of more complex, domain-specific control primitives and feedback loops including humans and perception could enable the generation of more accurate and efficient policies, expanding the range of tasks that can be handled.

In conclusion, while significant progress has been made, the potential of LLM-based systems for robotic control remains vast, and there is much to explore in the intersection of language, reasoning, and robotics.

References

[1] Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P. & Zeng, A. Code as policies: Language model programs for embodied control. *2023 IEEE International Conference On Robotics And Automation (ICRA)*. pp. 9493-9500 (2023)

[2] Yu, W., Gileadi, N., Fu, C., Kirmani, S., Lee, K., Arenas, M., Chiang, H., Erez, T., Hasenclever, L., Humplik, J. & Others Language to rewards for robotic skill synthesis. *ArXiv Preprint ArXiv:2306.08647*. (2023)

[3] Ma, Y., Liang, W., Wang, G., Huang, D., Bastani, O., Jayaraman, D., Zhu, Y., Fan, L. & Anandkumar, A. Eureka: Human-level reward design via coding large language models. *ArXiv Preprint ArXiv:2310.12931*. (2023)

[4] Lin, K., Agia, C., Migimatsu, T., Pavone, M. & Bohg, J. Text2motion: From natural language instructions to feasible plans. *Autonomous Robots*. **47**, 1345-1365 (2023)

[5] Wu, J., Antonova, R., Kan, A., Lepert, M., Zeng, A., Song, S., Bohg, J., Rusinkiewicz, S. & Funkhouser, T. Tidybot: Personalized robot assistance with large language models. *Autonomous Robots*. **47**, 1087-1102 (2023)

[6] Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., Zeng, A., Tompson, J., Mordatch, I., Chebotar, Y., Sermanet, P., Brown, N., Jackson, T., Luu, L., Levine, S., Hausman, K. & Ichter, B. Inner Monologue: Embodied Reasoning through Planning with Language Models. (2022), https://arxiv.org/abs/2207.05608

[7] Li, C., Liang, J., Zeng, A., Chen, X., Hausman, K., Sadigh, D., Levine, S., Fei-Fei, L., Xia, F. & Ichter, B. Chain of Code: Reasoning with a Language Model-Augmented Code Emulator. (2024), https://arxiv.org/abs/2312.04474

[8] Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Fu, C., Gopalakrishnan, K., Hausman, K., Herzog, A., Ho, D., Hsu, J., Ibarz, J., Ichter, B., Irpan, A., Jang, E., Ruano, R., Jeffrey, K., Jesmonth, S., Joshi, N., Julian, R., Kalashnikov, D., Kuang, Y., Lee, K., Levine, S., Lu, Y., Luu, L., Parada, C., Pastor, P., Quiambao, J., Rao, K., Rettinghouse, J., Reyes, D., Sermanet, P., Sievers, N., Tan, C., Toshev, A., Vanhoucke, V., Xia, F., Xiao, T., Xu, P., Xu, S., Yan, M. & Zeng, A. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. (2022), https://arxiv.org/abs/2204.01691

[9] Arenas, M., Xiao, T., Singh, S., Jain, V., Ren, A., Vuong, Q., Varley, J., Herzog, A., Leal, I., Kirmani, S., Prats, M., Sadigh, D., Sindhwani, V., Rao, K., Liang, J. & Zeng, A. How to Prompt Your Robot: A PromptBook for Manipulation Skills with Code as Policies. *2024 IEEE International Conference On Robotics And Automation (ICRA)*. pp. 4340-4348 (2024)

[10] Cui, C., Ma, Y., Cao, X., Ye, W., Zhou, Y., Liang, K., Chen, J., Lu, J., Yang, Z., Liao, K. & Others A survey on multimodal large language models for autonomous driving. *Proceedings Of The IEEE/CVF Winter Conference On Applications Of Computer Vision*. pp. 958-979 (2024)

[11] Wu, D., Han, W., Liu, Y., Wang, T., Xu, C., Zhang, X. & Shen, J. Language prompt for autonomous driving. *Proceedings Of The AAAI Conference On Artificial Intelligence*. **39**, 8359-8367 (2025)

[12] Huang, C., Mees, O., Zeng, A. & Burgard, W. Visual language maps for robot navigation. *2023 IEEE International Conference On Robotics And Automation (ICRA)*. pp. 10608-10615 (2023)

VI. Table of results in [1]

Table of results attached with this paper as Table 1.

TABLE I: Detailed simulation tabletop manipulation success rate (%) across different task scenarios. [1]

	CLIPort (oracle termination)	**CLIPort (no oracle)**	**NL Planner**	**CaP (ours)**
Seen Attributes, Seen Instructions				
Pick up the <object1> and place it on the (<object2> or <recepticle-bowl>)	88	44	98	**100**
Stack all the blocks	**98**	4	94	94
Put all the blocks on the <corner/side>	**96**	8	46	92
Put the blocks in the <recepticle-bowl>	**100**	22	94	**100**
Put all the blocks in the bowls with matching colors	12	14	**100**	**100**
Pick up the block to the <direction> of the <recepticle-bowl> and place it on the <corner/side>	**100**	80	N/A	72
Pick up the block <distance> to the <recepticle-bowl> and place it on the <corner/side>	92	54	N/A	**98**
Pick up the <nth> block from the <direction> and place it on the <corner/side>	**100**	38	N/A	98
Total	85.8	33.0	86.4	**94.3**
Long-Horizon Total	78.8	18.4	86.4	**97.2**
Spatial-Geometric Total	**97.3**	57.3	N/A	89.3
Unseen Attributes, Seen Instructions				
Pick up the <object1> and place it on the (<object2> or <recepticle-bowl>)	12	10	98	**100**
Stack all the blocks	96	8	96	**100**
Put all the blocks on the <corner/side>	0	0	58	**100**
Put the blocks in the <recepticle-bowl>	46	0	88	**96**
Put all the blocks in the bowls with matching colors	30	26	**100**	92
Pick up the block to the <direction> of the <recepticle-bowl> and place it on the <corner/side>	0	0	N/A	**60**
Pick up the block <distance> to the <recepticle-bowl> and place it on the <corner/side>	0	0	N/A	**100**
Pick up the <nth> block from the <direction> and place it on the <corner/side>	0	0	N/A	**60**
Total	23.0	5.5	88.0	**88.5**
Long-Horizon Total	36.8	8.8	88.0	**97.6**
Spatial-Geometric total	0.0	0.0	N/A	**73.3**
Unseen Attributes, Unseen Instructions				
Put all the blocks in different corners	0	0	60	**98**
Put the blocks in the bowls with mismatched colors	0	0	**92**	60
Stack all the blocks on the <corner/side>	0	0	40	**82**
Pick up the <object1> and place it <magnitude> to the <direction> of the <recepticle-bowl>	0	0	N/A	**38**
Pick up the <object1> and place it in the corner <distance> to the <recepticle-bowl>	4	0	N/A	**58**
Put all the blocks in a <line>	0	0	N/A	**90**
Total	0.7	0.0	64.0	**71.0**
Long-Horizon Total	0.0	0.0	64.0	**80.0**
Spatial-Geometric Total	1.3	0.0	N/A	**62.0**