Improving LLM code generated policies using Multi-Agent Reasoning Frameworks

Ori Spector
Department of Computer Science
Stanford University
orispec@stanford.edu

***Abstract*—I aim to improve on the results from the simulated tabletop manipulation task experiments conducted by Liang et al's in their foundational paper *Code as Policies* (CaP). Specifically, their tabletop manipulation experiments garnered high success rates for seen attributes and instructions in their few-shot prompting approach, but fell short (failing 25+%) in multiple scenarios where policies were generated using unseen attributes and/or instructions in the user's prompt. To enhance the generalization of their method, my approach is motivated by using multiple LLM based agents to complete high-order reasoning using chain of thought (CoT). Prior research has shown that multi-agent systems can improve code generation and other high-order reasoning problems (e.g. mathematics, debating) through the use of state-of-the-art CoT frameworks. Additionally, it has been showed that Embodied Chain-of-Thought (ECoT) reasoning improves policy generalization. In this paper, I create a up to date (simplified) baseline adapted from CaP's original code using gpt-3.5-turbo and build a multi-agent reasoning model using OpenAI's swarm library. Overall, this paper serves as a proof of concept for generating code policies using higher-order reasoning systems, guiding future research that can more sufficiently show final results.**

I. Introduction

Robotic manipulation involves interpreting and executing commands within dynamic environments. Traditional methods often struggle with generalizing to new tasks without extensive retraining or data collection. The "Code as Policies" (CaP) framework from Google Robotics by Liang et al [11] demonstrates the potential of large language models (LLMs) to synthesize robot policy code from natural language commands, allowing robots to leverage spatial-geometric reasoning and behavioral commonsense. However, these systems rely on predefined logic structures, in the form of few-shot prompting, to simulate basic reasoning for policy generation. This few-shot approach demonstrates high success rates with tabletop manipulation tasks in situations where attributes (e.g. object color, location, direction, distance, count) or instructions (e.g. Pick up the <block> and place it on <bowl>) are 'seen' or explicitly written in prompts fed into the code generation pipeline. For instance, in the results from the simulated table top experiments done in CaP the success rate was as high as 100% for tasks with **Seen Attributes** and **Seen Instructions**[1]. On the other hand, for user prompts containing **Unseen Attributes** and **Unseen Instructions** there were drastically worse results including 38% success on the following task.

[1]To view the detailed results from CaP see Appendix 6.

> Pick up the <object> and place it <magnitude> to the <direction> of the <bowl>

The findings suggest that while LLMs excel at implementing policies when given clear contextual information, they struggle to create appropriate policies when context is missing from their input prompts. Consider how a human would approach a complex task - such as positioning blocks at specific locations on a table or creating stacks in various orientations. They would likely pause to analyze the situation first, distinguishing between different objects like blocks and bowls, rather than relying on automatic responses. Similarly, we aim to develop robotic systems that can balance two capabilities: executing well-trained control sequences and carefully analyzing new situations before converting them into actionable code policies. This analytical process should encompass several steps: recognizing and pinpointing relevant objects, developing a strategic approach to the task, and effectively translating both subtasks and environmental observations into more precise code execution.

Thus, my goal is to present a method where multi-agent CoT helps breakdown tasks in order to reason more deeply for unseen scenarios. While, LLMs perform well when given context, as seen by the strong results for manipulations tasks in "seen" cases feeding more context is not always a solution. Research [9] has shown that long context does not always yield better results, meaning that few-shot approaches are not scalable for multiple reasons. Therefore, deeper reasoning strategies with known context might be a better solution for dealing with cases where there is deviance from the prompt (causing hallucination and failing tasks). For example, here is a seen instruction defining positions functions for the top left corner and bottom side from CaP.

```
# the top right corner.
top_left_pos_norm = [0, 1]
top_left_pos = denormalize_xy(top_left_pos_norm)
ret_val = top_left_pos
# the bottom side.
bottom_pos_norm = [0.5, 0]
bottom_pos = denormalize_xy(bottom_pos_norm)
```

When tasked with moving a block to the bottom side, the generated policy will be successful. However, there can be some generalization mistakes that occur when trying to move a block to the top right corner. This is a rather trivial case, but when one starts stacking unknowns factors, such as colors,

directions, and magnitudes all in one prompt, the few-shot system is not as robust. Thus, with smart multi-step reasoning system, these large unknown scenarios can be handled better by breaking down similar scenarios with known attributes/instructions in order to code yet-to-be-generated functions that can compute moving a block "top right" or "further" without needed this explicit knowledge in the prompt.

II. Previous Investigations

The intersection of language models, robotics, and reasoning frameworks has seen significant developments in recent years. My work tries to build upon these advances while addressing the crucial challenge of improving generalization in code-based policies. I organize my discussion around three main themes: code as policies, chain-of-thought reasoning, and multi-agent frameworks.

A. Code Generation and Robot Control

Code as Policies (CaP) [11] introduced a novel paradigm for robot control by directly generating executable code through large language models. This approach differs fundamentally from traditional methods that rely on semantic parsing [8] [10][14] or predefined skills [13] [12]. While CaP demonstrated impressive capabilities in generating precise control policies, its generalization to unseen attributes and instructions remains a challenge, particularly in complex manipulation tasks. Prior works in language-based robot control have primarily focused on high-level interpretation [6][1][5] or planning approaches. These methods typically decompose tasks into discrete steps but often assume the existence of pre-trained skills for execution. In contrast, CaP generates complete policy code that includes both high-level logic and low-level control primitives, eliminating the need for extensive pre-training of specific skills.

B. Chain-of-Thought Reasoning

Recent advances in chain-of-thought (CoT) reasoning have shown remarkable improvements in various complex tasks. The Embodied Chain-of-Thought (ECoT) framework from Zawalski et al [17] has demonstrated that incorporating multiple steps of reasoning about plans, sub-tasks, and visually grounded features can significantly enhance policy performance. This approach achieved a 28% improvement in success rates across challenging generalization tasks without additional robot training data. Traditional CoT approaches [16] focus primarily on simple language-only reasoning. However, robotics applications require grounding this reasoning in physical states and observations. The integration of embodied reasoning with code generation presents a promising direction for improving policy generalization, as demonstrated by recent works[2][15].

C. Multi-Agent Frameworks

Multi-agent systems have shown promise for enhancing reasoning capabilities. AgentCoder [4] demonstrated that collaborative code generation through multiple specialized agents can significantly improve code quality and testing effectiveness. Their approach achieved 96.3% pass rates on standard benchmarks, substantially outperforming single-agent approaches. The Chain of Agents (CoA) framework [18] has shown particular promise in handling long-context tasks through sequential agent collaboration. This approach is especially relevant for complex robotics tasks that require processing and reasoning about extensive contextual information. Recent work by Hegazy [3] showed that diversity in multi-agent systems can lead to stronger reasoning capabilities. Their findings show that combining different model architectures in a debate framework can outperform even the most advanced single models, achieving state-of-the-art performance on complex reasoning tasks.

D. Research Gaps and Future Directions

Despite these advances, several crucial challenges remain:

- The integration of multi-agent reasoning frameworks with code generation for robotics has not been fully explored especially using vision language models.
- The balance between high-level reasoning and low-level control generation in multi-agent systems requires further investigation.

III. Methods & Experiments

This section outlines my designed multi-agent reasoning system and the experiments designed to evaluate the proposed system's performance. My work hopes to combine some of the existing research frameworks highlighted in the previous section including using multi-agent systems with CoT reasoning to enhance the generalization capabilities of code-based policies. This approach hypothesizes improvements of both the robustness and interoperability of generated robot control policies while maintaining the flexibility and expressiveness of the original CaP framework.

A. Methods

My system is designed to compartmentalize complex tasks into distinct reasoning stages in order to execute new policies specifically to help generalize language model programs for table top manipulation tasks.

1) Mulit-Agent Overview: The system consists of five specialized agents working in sequence to process natural language commands into executable robot policies: a UI Agent, Object Parser Agent, Position Parser Agent, Function Generator Agent, and Executor Agent.

The UI Agent, serves as the initial interface for processing user commands performing preliminary parsing of task requirements and determines which specialized agent should handle the next stage of processing. The Object Parser Agent identifies and categorizes objects mentioned in commands, handles disambiguation of object references, maps natural language object descriptions to specific simulation objects, and maintains awareness of available objects in the environment. [2]. Moreover, the Position Parser Agent processes spatial

[2]A simplified implementation of this agent is given in the Appendix A

relationships and positioning requirements, handles relative positioning (e.g., ”to the left of”, ”behind”), translates natural language spatial descriptions into coordinate frames, and validates spatial feasibility of requested positions. Additionally, the Function Generator Agent, creates helper functions needed for complex task execution, breaks down multi-step tasks into executable sub-components, generates code for geometric calculations and path planning and handles edge cases and error conditions. Finally, the Executor Agent is responsible for final code execution and robot control, monitors execution progress and handles failures, provides feedback on task completion status, and manages robot state and safety constraints.

2) Swarm Implementation Architecture: The multi-agent system leverages OpenAI's Swarm framework [7]8, which provides a lightweight and flexible approach to agent orchestration. At its core, Swarm operates through two fundamental primitives: Agents and handoffs. Each Agent is instantiated with a distinct name, system-level instructions that define its role and behavior, a list of available functions or tools it can utilize, and a specified language model (in this case, GPT-3.5-turbo). This structure allows for clear separation of concerns while maintaining a unified interface for agent communication and control.

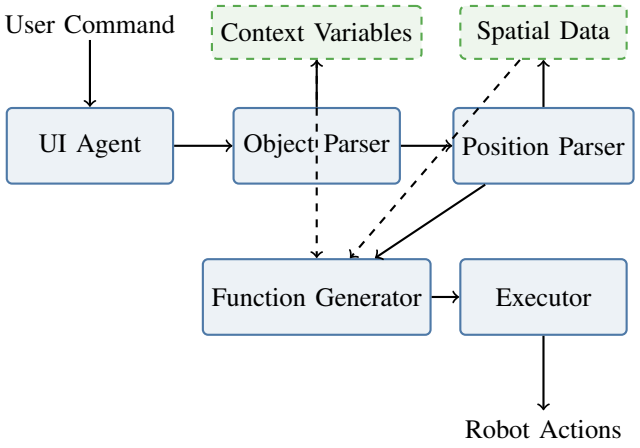

Fig. 1: Multi-Agent System Architecture using Swarm Framework. Solid arrows represent direct agent handoffs, while dashed arrows show context variable flow. Each agent specializes in a specific aspect of task processing, from initial command interpretation to final robot execution.

The system's communication flow is orchestrated through a custom Result dataclass that encapsulates three key components: a return value from function execution, a reference to the next agent that should handle the task, and a dictionary of context variables that maintains shared state across the agent network. This design enables smooth transitions between agents while preserving critical context and state information throughout the execution pipeline. State management is achieved through a comprehensive context variables system that maintains and updates task-relevant information throughout the execution cycle. These variables include the original command, available objects in the environment, parsed object references, computed positions, and generated helper functions. As each agent processes the task, it can both access and modify these variables, ensuring that subsequent agents have access to the accumulated context and reasoning from previous steps.

Agent transitions are implemented through explicit handoff functions that package the current state and designate the next appropriate agent in the pipeline. For example, when the UI Agent completes its initial parsing, it triggers a handoff to the Object Parser Agent by returning a Result object containing the updated context and the next agent reference. This explicit handoff system ensures clear control flow and maintains traceability throughout the execution process.

This architecture enables robust task decomposition and specialized processing while maintaining coherent state management and clear execution flow. The combination of explicit agent roles, structured handoffs, and shared context creates a system that can effectively reason about and execute complex robotic manipulation tasks.

3) Reasoning Framework Benefits: This multi-agent architecture offers several advantages over single-agent approaches including, task specialized reasoning, improved generalization, and better contextual understanding.

Each agent focuses on a specific aspect of the task, allowing for deeper domain expertise in each component, which reduces the complexity of individual agent responsibilities. Furthermore, there is potential for better handling of unseen attributes through decomposition and explicit reasoning about spatial relationships. Also, by maintaining state between processing stages there is accumulated context from previous steps leading to a more robust knowledge pipeline.

Overall, the goal was to mirror human problem-solving approaches - breaking down complex tasks into manageable subproblems while maintaining overall context and goals. While, I've only done preliminary research into this methodology it seems like a promising direction to try and handle novel situations by applying specialized reasoning at each stage rather than relying solely on end-to-end pattern matching.

B. Experiments

Simulated Tabletop Manipulation

To reproduce results from the original CaP paper I adapted the code provided in their Tabletop Manipulation Interactive Demo. This code uses a PyBullet-based environment that simulates a UR5e robot with a 2-finger gripper to perform complex manipulation tasks. These include:

- Spatial Reasoning: Tasks like placing objects relative to each other (e.g., ”to the left of the red block”).
- Sequential Actions: Multi-step commands such as ”stack blocks on the bottom-right corner”.
- Contextual Commands: Handling instructions that depend on previous states, like ”undo that”.

However, rapid model innovation has lead to OpenAI deprecating the exact model used in the original paper. Thus, I updated their code to support OpenAI's newer models and API schemas specifically around Codex and Edit which have

become deprecated in favor of using general intelligence models. Specifically I made two major changes [3]:

- The chat completion model uses *gpt-3.5-turbo* instead of *code-davinci-002*
- The code completion and editing to use *gpt-3.5-turbo* instead of the Edit API and Codex models

I performed a simple new baseline evaluation on a few of the largest fail cases in order to ensure fair comparison to the multi-agent system. The following failcases were used to establish the baseline to evaluate my new approach:

1. **Seen Attributes, Seen Instructions**
   - Pick up the block to the `<direction>` of the `<bowl>` and place it on the `<corner/side>`
2. **Unseen Attributes, Seen Instructions**
   - Pick up the block to the `<direction>` of the `<bowl>` and place it on the `<corner/side>`
   - Pick up the `<nth>` block from the `<direction>` and place it on the `<corner/side>`
3. **Unseen Attributes, Unseen Instructions** [4]
   - Pick up the `<object>` and place it `<magnitude>` to the `<direction>` of the `<bowl>`
   - Pick up the `<object>` and place it in the corner `<distance>` to the `<bowl>`
   - Put the blocks in the bowls with mismatched colors

Specifically, failcases – cases where the original baseline in CaP failed at least 25% of the time – across three categories were my focus. To see a full list of the attributes and instructions (seen and unseen) view section A in the appendix.

The multi-agent system ran with the same model (gpt-3.5-turbo) on the same set of tasks to ensure fairness. Also, I tested the baseline using OpenAI's state-of-the-art gpt-4o to test if a larger model by itself could increase success rates without additional reasoning scaffolding.

IV. Preliminary Results

A. Model Comparison

Notably the results for the new model baselines and multi-agent methods are preliminary and not extensive compared to any of the results detailed in CaP. Due to limited time and monetary resources only 15 samples were run on each fail case, thus, this serves more as a directional proof of concept that further research can explore and evaluate thoroughly in order to show more meaningful results.

The first notable result, is that a larger LLM model with better code generation capabilities out of the box improves the original results of CaP in every fail case category. Therefore, gpt-4o yielded higher success rates tham gpt-3.5-turbo which beat code-davinci-002. The multi-agent scaffolding seems to be on par or slightly better than pure gpt-3.5-turbo [5]. This

[3]The API now uses chat completions schema instead of completions create pipeline. Additionally, I used updated version numbers and prompt messaging frameworks supported by the new models using defined roles. There was minimal system prompt tweaks for the baseline.

[4]The largest amount of fail cases and worst success rates lie in this category.

[5]No results collected for multi-agent with gpt-4o due to time constraints and errors running the code. Future work should explore this avenue.

Fail Cases	Original CaP	Baseline (gpt-3.5-turbo)	Baseline gpt-4o	Multi-Agent
Seen Attributes, Seen Instructions				
Pick up the block to the `<direction>` of the `<bowl>` and place it on the `<corner/side>`	72%	80%	**93.3%**	80%
Unseen Attributes, Seen Instructions				
Pick up the block to the `<direction>` of the `<bowl>` and place it on the `<corner/side>`	60%	60%	86.6%	73.3%
Pick up the `<nth>` block from the `<direction>` and place it on the `<corner/side>`	60%	66.6%	**93.3%**	80%
Unseen Attributes, Unseen Instructions				
Pick up the `<object>` and place it `<magnitude>` to the `<direction>` of the `<bowl>`	38%	46.6%	80%	60%
Pick up the `<object>` and place it in the corner `<distance>` to the `<bowl>`	58%	60%	86.6%	66.6%
Put the blocks in the bowls with mismatched colors	60%	66.6%	86.6%	73.3%

Fig. 2: Success rate (%) across different failcase scenarios

sheds light on the directionality that multi-agent CoT frameworks can be helpful in generalizing robotic manipulation tasks, but that larger LLMs play bigger factors in overall success.

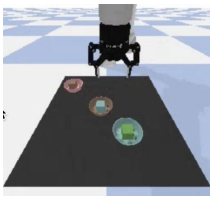

Fig. 3: Successful Attempt

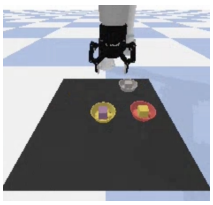

Fig. 4: Unsuccessful Attempt

Fig. 5: Placing blocks in mismatched colored bowls

B. Limitations

While this paper shows preliminary directional results, it is by no means extensive enough to yield significant results. Each metric needs to be tested on at least $\approx 30k$ simulated samples as done in CaP. This requires more compute, funding, and time. Additionally, experimenting using a variety of state-of-the-art code models and different agent architectures is a important future direction to explore. Finally, the testing was done using the simulated environment given in the CaP paper, which is constrained to set of tasks with certain instructions and attributes. Thus, it is a relatively contained environment and might not generalize to other situations which are more noisy or dynamic. Furthermore, there is still a reliance on few-shot prompting, necessitating knowledge of a given environment. Ultimately, there are limitations in this current approach that needs to be worked on to yield considerable results.

Acknowledgments

Special thanks to Liang et al [11] at Google Robotics for their foundational work *Code as Policies* and for all the resources available in there github repository.

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022. URL https://arxiv.org/abs/2204.01691.

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[3] Mahmood Hegazy. Diversity of thought elicits stronger reasoning capabilities in multi-agent debate frameworks, 2024. URL https://arxiv.org/abs/2410.12853.

[4] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

[5] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *CoRR*, abs/2201.07207, 2022. URL https://arxiv.org/abs/2201.07207.

[6] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models, 2022. URL https://arxiv.org/abs/2207.05608.

[7] OpenAI Ilan Bigio. Swarm - orchestrating agents: Routines and handoffs. https://github.com/openai/swarm?tab=readme-ov-file, 2024.

[8] Thomas Kollar, Stefanie Tellex, Deb Roy, and Nicholas Roy. Toward understanding natural language directions. In *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 259–266, 2010. doi: 10.1109/HRI.2010.5453186.

[9] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023. URL https://arxiv.org/abs/2307.03172.

[10] Cynthia Matuszek, Evan V. Herbst, Luke Zettlemoyer, and Dieter Fox. Learning to parse natural language commands to a robot control system. In *International Symposium on Experimental Robotics*, 2012. URL https://api.semanticscholar.org/CorpusID:1658890.

[11] Google Robotics. Code as policies: Language model programs for embodied control. https://github.com/google-research/google-research/tree/master/code_as_policies, 2022.

[12] Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Cliport: What and where pathways for robotic manipulation. *CoRR*, abs/2109.12098, 2021. URL https://arxiv.org/abs/2109.12098.

[13] Simon Stepputtis, Joseph Campbell, Mariano J. Phielipp, Stefan Lee, Chitta Baral, and Heni Ben Amor. Language-conditioned imitation learning for robot manipulation tasks. *CoRR*, abs/2010.12083, 2020. URL https://arxiv.org/abs/2010.12083.

[14] Jesse Thomason, Shiqi Zhang, Raymond Mooney, and Peter Stone. Learning to interpret natural language commands through human-robot dialog. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 19231929. AAAI Press, 2015. ISBN 9781577357384.

[15] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. *CoRR*, abs/2108.13643, 2021. URL https://arxiv.org/abs/2108.13643.

[16] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL https://arxiv.org/abs/2201.11903.

[17] Micha Zawalski, William Chen, Karl Pertsch, Oier Mees, Chelsea Finn, and Sergey Levine. Robotic control via embodied chain-of-thought reasoning. *arXiv preprint arXiv:2407.08693*, 2024.

[18] Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan . Arik. Chain of agents: Large language models collaborating on long-context tasks, 2024. URL https://arxiv.org/abs/2406.02818.

APPENDIX

TABLE VI: Detailed simulation tabletop manipulation success rate (%) across different task scenarios.

	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**

Fig. 6: Original Detailed Results from Code as Policies Simulated Tabletop Experiments

Detailed Breakdown of Attributes and Instructions

Seen Instructions.

1) Pick up the <block1> and place it on the <block2>
2) Stack all the blocks
3) Put all the blocks on the <corner/side>
4) Put the blocks in the <bowl>
5) Put all the blocks in the bowls with matching colors
6) Pick up the block to the <direction> of the <bowl> and place it on the <corner/side>
7) Pick up the block <distance> to the <bowl> and place it on the <corner/side>
8) Pick up the <nth> block from the <direction> and place it on the <corner/side>

Unseen Instructions.

1) Put all the blocks in different corners
2) Put the blocks in the bowls with mismatched colors
3) Stack all the blocks on the <corner/side>
4) Pick up the <block1> and place it <magnitude> to the <direction> of the <bowl>
5) Pick up the <block1> and place it in the corner <distance> to the <bowl>
6) Put all the blocks in a <line> line

Seen Attributes.

1) <block>: blue block, red block, green block, orange block, yellow block
2) <bowl>: blue bowl, red bowl, green bowl, orange bowl, yellow bowl
3) <corner/side>: left side, top left corner, top side, top right corner
4) <direction>: top, left
5) <distance>: closest
6) <magnitude>: a little
7) <nth>: first, second
8) <line>: vertical, horizontal

Unseen Attributes.

1) <block>: pink block, cyan block, brown block, gray block, purple block
2) <bowl>: pink bowl, cyan bowl, brown bowl, gray bowl, purple bowl
3) <corner/side>: bottom right corner, bottom side, bottom left corner
4) <direction>: bottom, right
5) <distance>: farthest
6) <magnitude>: a lot
7) <nth>: third, fourth
8) <line>: diagonal

.
.
.

```
# Example of Object Parser Agent
self.object_parser_agent = Agent(
name = "ObjectParser",
instructions = prompts['prompt_parse_obj_name']
['prompt_text'][system],
functions = [self.transfer_to_position_parser,
self.parse_objects],
model="gpt-3.5-turbo"
)
```

Parsing Agent System Prompt

You are an agent specialized in Object Parssing for code generation used for a robotic simulation that handles:
- Object identification by properties (color, size, type, position)
- Spatial relationships between objects
- Geometric patterns and arrangements
- Relative positions and distances

Only output the exact code needed - no explanations or markdown. Use the examples provided as parsing templates but generalize to handle variations in:
- Object descriptions and references
- Position specifications
- Spatial relationships
- Pattern formations

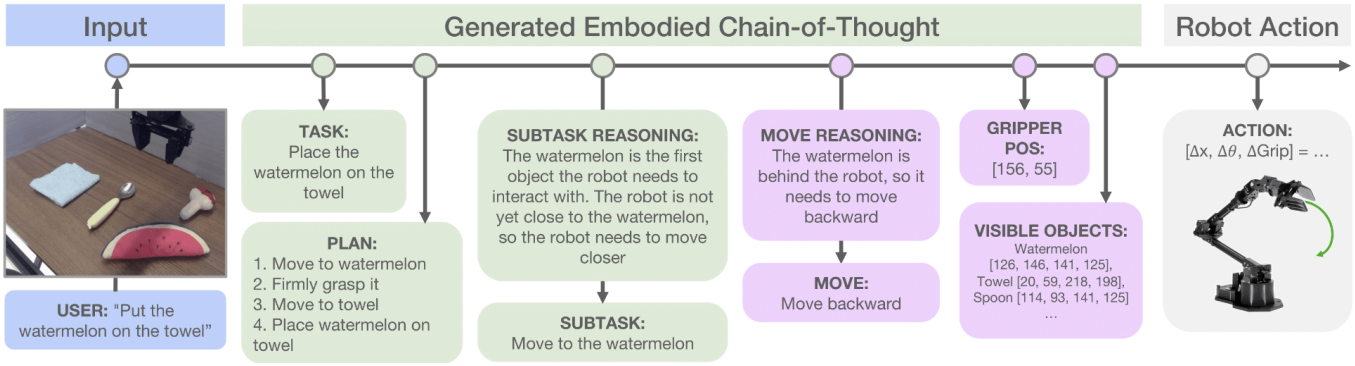

Fig. 7: Embodied Chain-of-Thought Reasoning (ECoT)

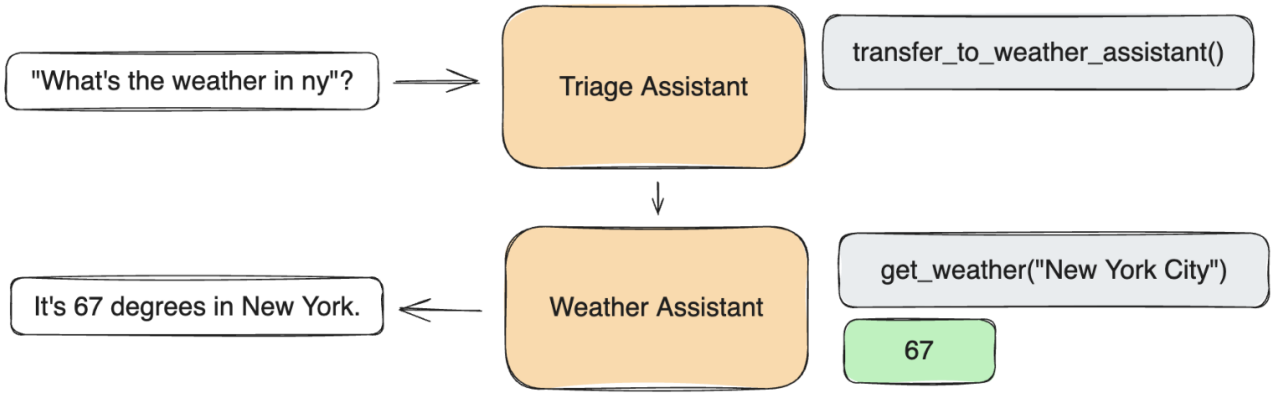

Fig. 8: Swarm Simple Agent Example

Fail Cases	Original CaP	Baseline (gpt-3.5-turbo)	Baseline gpt-4o	Multi-Agent
Seen Attributes, Seen Instructions				
Pick up the block to the <direction> of the <bowl> and place it on the <corner/side>	72%	80%	**93.3**%	80%
Unseen Attributes, Seen Instructions				
Pick up the block to the <direction> of the <bowl> and place it on the <corner/side>	60%	60%	86.6%	73.3%
Pick up the <nth> block from the <direction> and place it on the <corner/side>	60%	66.6%	**93.3**%	80%
Unseen Attributes, Unseen Instructions				
Pick up the <object> and place it <magnitude> to the <direction> of the <bowl>	38%	46.6%	80%	60%
Pick up the <object> and place it in the corner <distance> to the <bowl>	58%	60%	86.6%	66.6%
Put the blocks in the bowls with mismatched colors	60%	66.6%	86.6%	73.3%

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