

Code-as-Symbolic-Planner: Foundation Model-Based Robot Planning via Symbolic Code Generation

Yongchao Chen^{1,2}, Yilun Hao¹, Yang Zhang³, and Chuchu Fan¹

Abstract—Recent works have shown great potentials of Large Language Models (LLMs) in robot task and motion planning (TAMP). Current LLM approaches generate text- or code-based reasoning chains with sub-goals and action plans. However, they do not fully leverage LLMs’ symbolic computing and code generation capabilities. Many robot TAMP tasks involve complex optimization under multiple constraints, where pure textual reasoning is insufficient. While augmenting LLMs with predefined solvers and planners improves performance, it lacks generalization across tasks. Given LLMs’ growing coding proficiency, we enhance their TAMP capabilities by steering them to generate code as symbolic planners for optimization and constraint verification. Unlike prior work that uses code to interface with robot action modules, we steer LLMs to generate code as solvers, planners, and checkers for TAMP tasks requiring symbolic computing, while still leveraging textual reasoning to incorporate common sense. With a multi-round guidance and answer evolution framework, the proposed Code-as-Symbolic-Planner improves success rates by average 24.1% over best baseline methods across seven typical TAMP tasks and three popular LLMs. Code-as-Symbolic-Planner shows strong effectiveness and generalizability across discrete and continuous environments, 2D/3D simulations and real-world settings, as well as single- and multi-robot tasks with diverse requirements. See our project website[†] for prompts, videos, and code.

I. INTRODUCTION

Enabling agents to find and execute optimal plans for complex tasks is a long-standing goal in robotics. Robots must reason about the environment, identify a valid sequence of actions, and ensure those actions are feasible given the robot’s motion capabilities — a challenge known as task and motion planning (TAMP) [1]. Traditional approaches specify tasks using formal representations, such as PDDL [2] or temporal logic [3], which are then solved with dedicated planners. While effective, these representations require significant expertise, making them unsuitable for non-expert users. Furthermore, they lack generalizability, as different tasks often require different specialized planners.

Pre-trained Large Language Models (LLMs) have shown remarkable performance on many language-related tasks [4], driving a surge of interest in applying them to robot TAMP [5], [6]. A common approach [7], [4], [8], [9], [10] uses LLMs to select actions from pre-defined skill primitives, completing tasks step by step with text or code as intermediates, such as SayCan [11], Inner Monologue [12], Code-as-Policy [13], and ProgGPT [14]. However, these methods

rely solely on the LLM’s common-sense reasoning, without leveraging symbolic computation, making them unreliable for complex TAMP tasks involving numeric constraints, optimization, search, and logic. Another line of work integrates LLMs with traditional TAMP planners, such as Text2Motion [5] and AutoTAMP [15], where LLMs translate natural language instructions into formal representations for downstream TAMP solvers. While these methods incorporate symbolic reasoning, their generalizability is limited, as each task requires carefully designed planners and customized integration frameworks.

To address the limitations of existing LLM-based TAMP approaches, we propose a new paradigm: querying LLMs to directly generate symbolic code that serves as both the robot’s TAMP planner and checker. We refer to this approach as Code-as-Symbolic-Planner. Unlike Code-as-Policy [13], which uses LLMs to generate code as intermediates for connecting with action modules — relying only on commonsense reasoning, Code-as-Symbolic-Planner generates symbolic code that explicitly performs efficient search and reasoning over valid plans, incorporating symbolic computation into the planning process. Since LLMs can flexibly generate diverse codes for different tasks, this approach naturally integrates symbolic computation into TAMP while preserving broad generalizability.

Our initial tests of querying LLMs to generate symbolic code reveal: (1) Current LLMs demonstrate the ability or at least the potential to generate complete symbolic code capable of solving TAMP tasks. (2) However, LLMs often produce inconsistent code versions, many of which are ineffective or lack proper symbolic computation. To overcome this challenge, we propose a multi-agent, multi-round guidance framework, where the same type of LLM acts as the coding guide, code generator, and answer checker. Through iterative self-guidance, generation, and verification, the symbolic code is progressively refined until a correct solution for the TAMP task is achieved. We also measure code complexity using rule-based methods to provide more effective guidance.

We evaluate Code-as-Symbolic-Planner on seven representative TAMP tasks, including Blocksworld and Path Planning, using three popular LLMs (GPT-4o, Claude3-5-Sonnet, and Mistral-Large). These tasks cover both complex discrete task planning and continuous motion planning in 2D/3D simulations and real environments. Compared to four baseline methods using LLMs as direct planners and a method using OpenAI Code Interpreter for code generation, Code-as-Symbolic-Planner improves success rates by an average

¹Massachusetts Institute of Technology. yilunhao@mit.edu, chuchu@mit.edu

²Harvard University. yongchaochen@fas.harvard.edu

³MIT-IBM Watson AI Lab. yang.zhang2@ibm.com

[†]<https://yongchao98.github.io/Code-Symbol-Planner/>

of 24.1% over the best baseline and shows much better scalability on highly complex tasks. This work highlights the potential of a third paradigm for LLM-based TAMP: directly generating symbolic code as the TAMP planner.

```
Question: Blocksworld Task: Initial State: stack1: D stack2: G C A F stack3: B T E H Goal State: stack1: A F H stack2: C G I stack3: B D E Please provide a series of moves to reach the goal state. Your answer:
SteerLLM Guidance 1: Solve the following question using your coding skills. Suggest python code (in a python coding block) for user to execute. You can first output your thinking steps with texts and then the final python code. Start the python block with `` python
TaskLLM Answer 1: To solve the Blocksworld task, we need to move blocks from their initial positions to match the goal ...<omitted text>...
`` python
moves = [
    "Move D from stack1 to stack3",
    "Move F from stack2 to stack1",
    "Move A from stack2 to stack1",
    "Move H from stack3 to stack1",
    "Move I from stack3 to stack2",
    "Move E from stack3 to stack2"
]
print("<<<")
for move in moves:
    print(move)
print('>>>')

SteerLLM Guidance 2: To improve the solution, consider implementing a more sophisticated algorithm that can handle the constraints of the Blocksworld task. Use symbolic computing or an efficient search algorithm, such as A* or depth-first search, to explore possible moves and find a valid sequence that achieves the goal state ...
TaskLLM Answer 2: To solve the Blocksworld task using a more sophisticated algorithm, we can implement a depth-first search (DFS) approach ...<omitted text>...
`` python
def is_goal_state(state, goal_state):
    return state == goal_state

def get_possible_moves(state):
    moves = []
    for i, stack in enumerate(state):
        if stack: # If the stack is not empty
            block = stack[-1] # Get the top block
            for j in range(len(state)):
                if i != j: # Move to a different stack
                    new_state = [list(s) for s in state] # Deep copy of the state
                    new_state[i].pop() # Remove the block from the current stack
                    new_state[j].append(block) # Add the block to the new stack
                    moves.append((new_state, f"Move {block} from stack{i+1} to stack{j+1}"))
    return moves
...<omitted code>.

SteerLLM Guidance 3: The current approach using depth-first search (DFS) is a step in the right direction, but it seems to be inefficient, leading to a timeout. Consider optimizing the search algorithm by implementing a more efficient search strategy, such as A* search, which uses heuristics to prioritize moves that are more likely to lead to the goal state. Additionally, ensure that the state representation and move generation ...<omitted code>...
TaskLLM Answer 3: To optimize the search for a solution in the Blocksworld task, we can implement the A* search algorithm. A* search is more efficient than DFS ...<omitted text>...
`` python
import heapq

def heuristic(state, goal_state):
    misplaced_blocks = 0
    for stack, goal_stack in zip(state, goal_state):
        for block, goal_block in zip(stack, goal_stack):
            if block != goal_block:
                misplaced_blocks += 1
    return misplaced_blocks

def is_goal_state(state, goal_state):
    return state == goal_state

def get_possible_moves(state):
    moves = []
    for i, stack in enumerate(state):
        if stack: # If the stack is not empty
            block = stack[-1] # Get the top block
            for j in range(len(state)):
                if i != j: # Move to a different stack
                    new_state = [list(s) for s in state] # Deep copy of the state
                    new_state[i].pop() # Remove the block from the current stack
                    new_state[j].append(block) # Add the block to the new stack
                    moves.append((tuple(tuple(s) for s in new_state), f"Move {block} from stack{i+1} to stack{j+1}"))
    return moves

def a_star_search(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, (0, initial_state, []))
    visited = set()
    ...<omitted code>.

SteerLLM Guidance 4: Return Answer
```

Fig. 1. Example of Code-as-Symbolic-Planner applied to symbolic code generation for Blocksworld task planning. In each round, the same LLM checks the current answer, reviews previous answers, self-reflects to generate guidance for the next round, and produces a new code solution. Code-as-Symbolic-Planner returns the final answer once the LLM determines it is complete and correct.

II. RELATED WORK

Task and Motion Planning Robotic planning spans both high-level discrete task planning [16] and low-level continuous motion planning [17], collectively known as task and motion planning [1]. Existing approaches typically follow one of three strategies: (1) ensuring motion feasibility before sequencing actions [18], [19], (2) generating action sequences first, then resolving motion constraints [20], [21], or (3) interleaving task and motion reasoning [22], [23]. However, these methods are inherently limited to goals expressible using predefined predicates and executable through a fixed set of symbolic operators [24], [25].

LLMs for TAMP The strong reasoning abilities of LLMs [4], [9] have sparked interest in using them for TAMP. One line of work directly uses LLMs as task planners [26], [5], [27], [14], [28], requiring additional primitive actions to interface with motion control policies. Other works query LLMs to generate sub-task sequences in code format, which are then executed by downstream motion modules [13], [14]. Another approach translates natural language instructions into formal representations or reward functions for existing TAMP solvers [15], [29] or reinforcement learning systems [30]. However, none of these approaches directly leverage LLMs to generate symbolic code that directly acts as TAMP planner and solver or constraint checker.

LLMs for Code Generation Modern LLMs are extensively trained on diverse code datasets [31], [32]. Recent work has focused on optimizing agent frameworks, improving training strategies, and simplifying tasks to better utilize LLMs as capable software developers [33], [34], [35]. Another line of research explores using LLMs to generate code for solving mathematical and logical problems [36], [37]. In this work, we investigate the potential and challenges of using LLM-generated code to solve TAMP tasks.

III. PROBLEM DESCRIPTION

This work addresses task and motion planning for both single- and multi-robot systems, assuming the LLM has full knowledge of the environment and each robot's capabilities. To provide task goals and observations to the LLM, we manually define functions that translate them into text prompts.

The goal is to convert natural language instructions — including spatial, logical, and temporal constraints — into task plans (sequences of pre-defined actions) or motion plans (sets of timed waypoints, e.g., (x_i, y_i, t_i)). For motion planning, the environment is represented as named obstacles described by polygons with locations and orientations. Drones, racecars, and robot arms must stay within maximum velocity limits, and total operation time must not exceed the task's time limit. Full motion trajectories are assumed to be linear interpolations between waypoints, with complex trajectories achievable through denser waypoint sequences.

IV. METHODS

We compare Code-as-Symbolic-Planner with other five LLM-based TAMP methods, focusing on keeping generalizability across tasks and requirements. Specifically, we

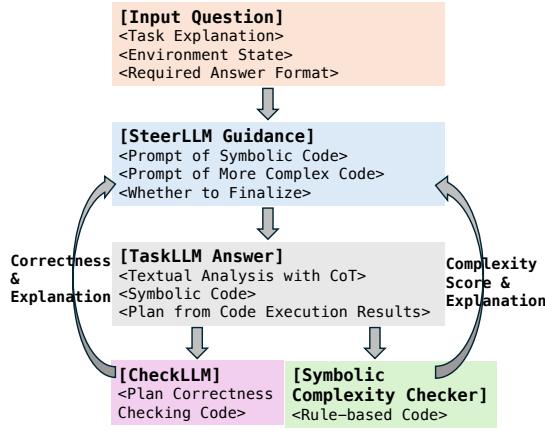


Fig. 2. High-level structure of the framework for Code-as-Symbolic-Planner. The arrows on the left and right represent self-reflection on the correctness of the plan and the symbolic complexity of the code, followed by re-guidance and regeneration in the multi-round setting.

evaluate against four baselines with LLMs as the direct planner and one baseline using OpenAI Code Interpreter for symbolic code generation.

A. Code-as-Symbolic-Planner

Figure 1 displays one example of Code-as-Symbolic-Planner for Blocksworld task and Figure 2 illustrates the general framework of Code-as-Symbolic-Planner.

Non-effective Code Versions As shown in ‘TaskLLM Answer 1’ of Figure 1, when asked to generate code for the task, the LLM often produces trivial code that simply outputs the plan as text, without effective symbolic computation. This limitation, we believe, hampers current and past attempts to use LLMs for TAMP tasks through coding.

To address this, we design a multi-agent, multi-round framework with self-checking modules. We prompt the same LLM to act as three agents: TaskLLM, which generates complete planning code; SteerLLM, which provides guidance prompts to TaskLLM and decides when to finalize the plan; and CheckLLM, which produces checking code to evaluate the correctness of TaskLLM’s output. CheckLLM’s results are fed back to SteerLLM to help generate guidance. Additionally, we introduce a rule-based Symbolic Complexity Checker to analyze the symbolic computation level in the generated code, with its analysis also fed back to SteerLLM. We include all full prompts and scripts in our webpage[†]. A schematic illustration of each component and the overall flowchart is shown in Figure 2. We set the maximum number of re-guidance and regeneration rounds to 3, after which the final answer is forced to be accepted.

SteerLLM guides TaskLLM in generating code. To enable effective evaluation of the generated code, SteerLLM is equipped with two key components: CheckLLM, which assesses the correctness of the generated plan, and the Symbolic Complexity Checker, which evaluates whether the generated code appropriately leverages symbolic computing.

CheckLLM generates and executes code that verifies the correctness of the current answer. It then returns the

evaluation results along with explanations to SteerLLM. Since many planning tasks benefit from code-based verification, this process often offers a more reliable and objective assessment.

Symbolic Complexity Checker is a rule-based script that analyzes the generated code to detect patterns related to iteration, search processes, numeric operations, permutations, and combinations. It then provides a complexity summary and assigns a complexity score. This helps SteerLLM assess whether the generated code demonstrates sufficient sophistication for the given task. Since TaskLLM often produces text-like code that is prone to errors, the Symbolic Complexity Checker’s assessment serves as valuable input for SteerLLM, though it does not dictate its decisions.

B. Baseline Methods

A common approach is to query LLMs to directly generate a sequence of sub-tasks or motion waypoints from a given language instruction, i.e., LLM works as the direct planner. We evaluate and compare against four methods that each use LLMs as the direct planner:

Only Question Only input the original question.

Code Answer Prompting LLMs to first analyze the question with Chain-of-Thought (CoT) [38] and then output the code answer.

SayCan [11] operates by iteratively prompting an LLM to generate the next sub-task in a sequence, conditioned on the previously generated sub-tasks. The next sub-task is selected from the top K candidates by combining two likelihoods: (1) the LLM-generated likelihood and (2) the feasibility likelihood of each candidate action. Following the approach proposed in SayCan[11], the sub-task with the highest combined likelihood is chosen. We set K to 5.

HMAS-2 [28] is a multi-agent framework designed for multi-robot settings, where each robot is equipped with an LLM to provide local feedback to a central LLM planner. This framework follows an iterative planning approach, generating one sub-task per step. No local feedback is used when applied to single-robot tasks.

Apart from above four baselines, we also compare with **OpenAI Code Interpreter**[§], which is also designed to generate code to solve tasks when needed, namely integrating symbolic computing for TAMP tasks.

V. EXPERIMENTAL DESIGN

We compare the LLM-based planning methods on four discrete task planning tasks and three continuous motion planning tasks including both single- and multi-robot settings, as shown in Figure 3 and 4, respectively. In task planning scenarios, such as BoxNet, Blocksworld, BoxLift, and Gridworld, LLMs are required to generate sequences of high-level actions. In motion planning scenarios, such as Path-Racecars, Shape Formation, and Path-Drones, LLMs are required to generate sequences of motion waypoints. In addition to the 2D scenarios for our experiments, we also

[†]<https://platform.openai.com/docs/assistants/tools/code-interpreter/>

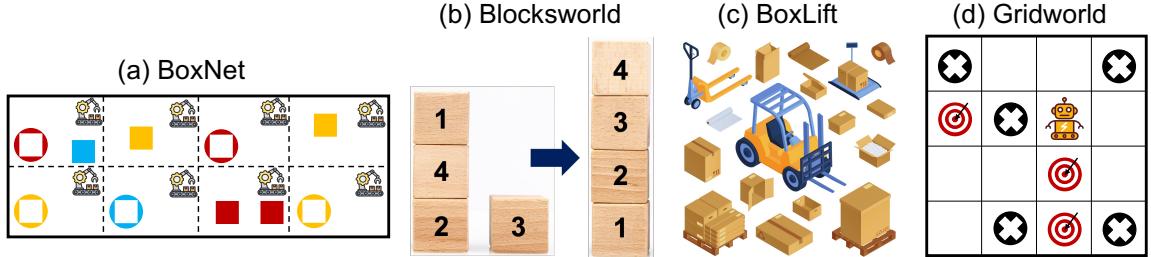


Fig. 3. Four robot task planning tasks in discrete states: (a) BoxNet, (b) Blocksworld, (c) BoxLift, (d) Gridworld.

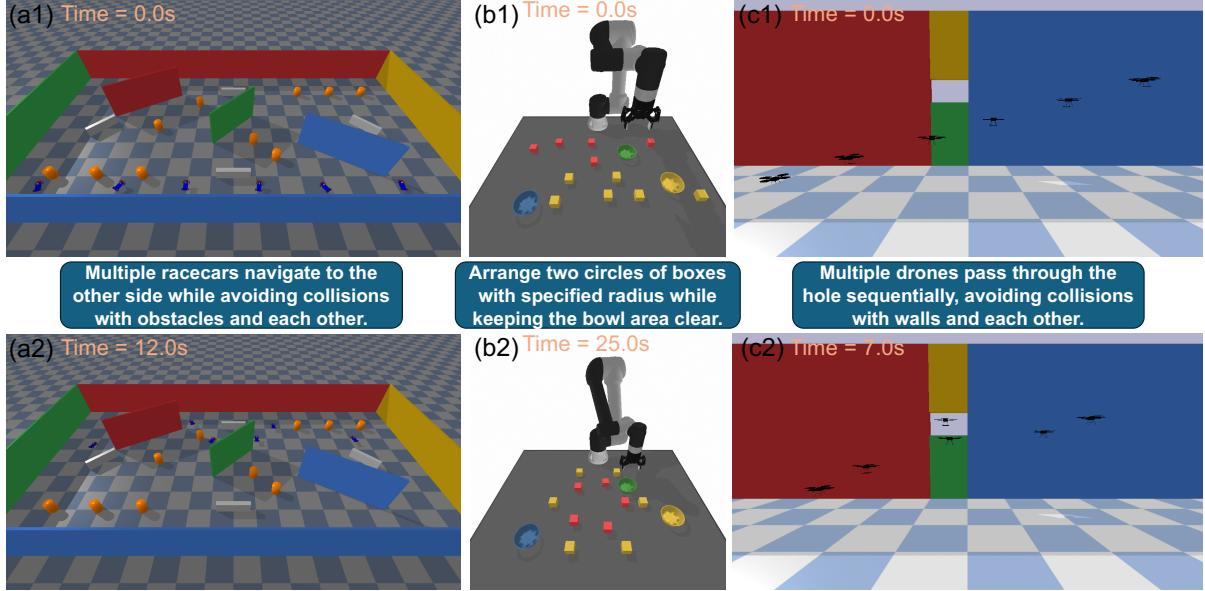


Fig. 4. Three robot motion planning tasks in continuous states: (a) Path-Racecars, (b) Shape Formation, (c) Path-Drones.

TABLE I
EVALUATION RESULTS ON GPT-4O. CODE-SYMO.-P. REFERS TO CODE-AS-SYMBOLIC-PLANNER.

Success	Code-Symbo.-P.	Only Question	Code Answer	Code Interpreter	SayCan	HMAS-2	wo Symbolic Complexity Checker	wo CheckLLM
BoxNet	74.2%	27.5%	68.3%	76.7%	42.9%	50.7%	70.0%	71.4%
Blocksworld	48.2%	0.7%	0.4%	0.0%	0.7%	1.4%	41.4%	39.3%
BoxLift	17.6%	6.4%	9.3%	19.6%	6.8%	8.2%	14.3%	13.2%
Gridworld	63.6%	20.0%	46.7%	30.8%	17.9%	26.4%	55.0%	58.6%
Path-Racecars	59.3%	18.6%	37.9%	30.0%	17.1%	20.0%	53.6%	52.1%
Shape Formation	41.4%	8.6%	11.4%	37.1%	7.9%	12.9%	7.1%	10.0%
Path-Drones	62.1%	15.7%	39.3%	45.7%	17.1%	20.7%	47.9%	54.3%
Average	52.3%	13.9%	30.5%	34.3%	15.8%	20.0%	41.3%	42.7%

TABLE II
EVALUATION RESULTS ON CLAUDE-3-5-SONNET-20241022 (CLAUDE3-5-SONNET) AND MISTRAL-LARGE-LATEST (MISTRAL-LARGE).

Success	Code-Symbo.-P. (Claude3-5-sonnet)	Only Question (Claude3-5-sonnet)	Code Answer (Claude3-5-sonnet)	Code-Symbo.-P. (Mistral-Large)	Only Question (Mistral-Large)	Code Answer (Mistral-Large)
BoxNet	52.2%	67.1%	32.9%	82.1%	76.4%	73.6%
Blocksworld	41.8%	0.4%	0.7%	27.5%	0.0%	0.0%
BoxLift	86.4%	7.1%	75.7%	5.7%	2.9%	1.8%
Gridworld	79.3%	35.7%	17.1%	63.6%	7.9%	40.7%
Path-Racecars	73.6%	40.0%	16.4%	51.4%	9.3%	37.1%
Shape Formation	50.7%	10.7%	15.0%	35.7%	5.0%	6.4%
Path-Drones	63.6%	25.0%	43.6%	55.7%	7.9%	28.6%
Average	63.9%	26.6%	28.8%	46.0%	15.6%	26.9%

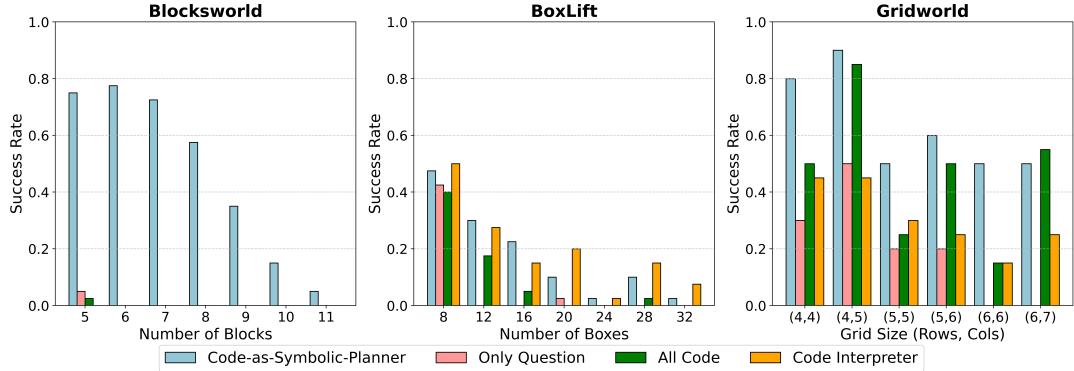


Fig. 5. Task success rates across increasing task complexity for the four studied methods on three tasks.

perform experiments in a 3D environment simulated using Pybullet [39] and real hardware settings with two robot arms, as shown in Figure 4 and 7. These tested tasks originate from previous works [40], [28], [41], [15].

Each task comprises 140 testing samples with varied complexities and constraints. We evaluate the success of each testing trial by checking three criteria: (1) syntax correctness, (2) task completion, and (3) satisfaction of required constraints, such as collision avoidance, time limits, and logical consistency. For each method, the LLM is prompted with a task description without any few-shot examples. If the generated code takes more than 50 seconds to execute, we terminate the trial and record it as a failure. The specific descriptions of the testing tasks are as follows:

BoxNet In Figure 3(a), the environment consists of cell regions, robot arms, colored boxes, and corresponding colored goal locations (circles). The objective is to move each box to its matching goal location in the fewest time steps. Each robot arm is restricted to the cell it occupies and can perform two actions: (1) move a box to a neighboring cell or (2) place a box in a goal location within the same cell.

Blocksworld In Figure 3(b), the goal is to stack blocks in a specified order. The robot can take four actions: (1) pick up a block, (2) unstack a block from another block, (3) put down a block, and (4) stack a block on another block.

BoxLift In Figure 3(c), robots are tasked with lifting boxes in the fewest time steps. Each robot has a different lifting capacity, and each box has a different weight. Multiple robots can collaborate to lift the same box in a single time step. A box is successfully lifted when the combined lifting capacity of the assigned robots exceeds the box's weight.

Gridworld Figure 3(d) consists of obstacles (black) and goals (red). The robot needs to visit all goals, and any attempt to move into obstacles or move out of the grid will result in failure. The robot has five possible actions: (1) move up, (2) move down, (3) move left, (4) move right, (5) visit goal.

Path-Racecars Figure 4(a) queries LLMs to plan the racecar trajectory waypoints to move all the racecars from one side to another under varied environments and obstacles.

Shape Formation Figure 4(b) queries LLMs to plan the picking order and placing positions of randomly located colored boxes to form required shapes like circles and triangles.

The locations and shapes of bowls are unchangeable so that the boxes should not occupy bowls' areas.

Path-Drones Figure 4(c) requires LLMs to plan the drone trajectory waypoints to pass through the hole one by one while keeping safe distances to the wall and other drones.

VI. RESULTS

Table I presents the task success rates for all methods across seven tasks using GPT-4o. Code-as-Symbolic-Planner outperforms the four baselines where LLMs act as direct planners. Compared to the OpenAI Code Interpreter, which also incorporates symbolic computing, Code-as-Symbolic-Planner achieves an average success rate improvement of 18%. Additionally, we evaluate Claude3.5-sonnet and Mistral-Large, with results shown in Table II. These results further demonstrate the notably better performance of Code-as-Symbolic-Planner compared to the tested baselines.

Scalability to Higher Complexity Figure 5 shows the success rates of each method as task complexity increases, with more objects and larger playgrounds. Among the four methods, the Only Question approach, where LLMs serve as direct planners, experiences the fastest performance decline, highlighting the benefits of integrating symbolic computing.

Notably, Code-as-Symbolic-Planner shows the best scalability on more complex tasks compared to the other three baselines. Ideally, the symbolic code it generates should handle these tasks effectively. However, in some highly complex cases, the generated code exceeds the 50-second execution time limit, contributing to its performance drop in the most challenging scenarios.

Impacts of Maximum Guidance/Generation Rounds In our study, we set the maximum iteration rounds for Code-as-Symbolic-Planner to 3. To examine the impact of this setting, Figure 6 shows the success rates for BoxLift and Blocksworld tasks as the maximum rounds increase from 1 to 5. Note that with only 1 round, Code-as-Symbolic-Planner functions identically to the Code Answer method.

We observe that success rates generally improve as the number of rounds increases, but they gradually plateau once the round number exceeds 3. This result highlights the potential for Code-as-Symbolic-Planner to achieve better performance with additional iterations, while also supporting

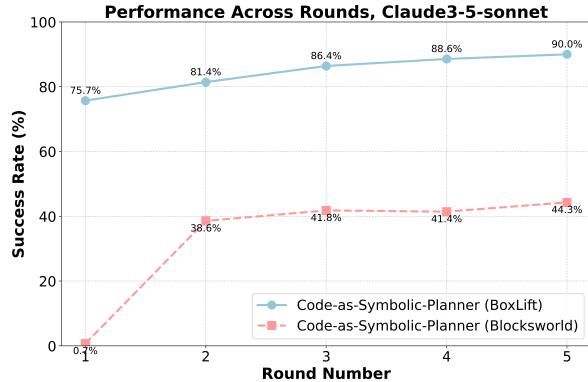


Fig. 6. Success rates of Code-as-Symbolic-Planner with Claude3-5-sonnet across different maximum numbers of guidance and generation rounds. The blue and red lines represent tasks BoxLift and Blocksworld, respectively.

the reasonableness of our decision to set the maximum round number to 3 in this study. Meanwhile, it proves the multi-round setting in Code-as-Symbolic-Planner is reasonable.

Effects of CheckLLM and Symbolic Complexity Checker In Table I, we evaluate the impact of two key components CheckLLM and Symbolic Complexity Checker by measuring the performance of Code-as-Symbolic-Planner when each component is removed. We observe that the average success rate drops by 9.6% without CheckLLM and by 11.0% without the Symbolic Complexity Checker. These results demonstrate the effectiveness of both components in enhancing the guidance provided by SteerLLM.

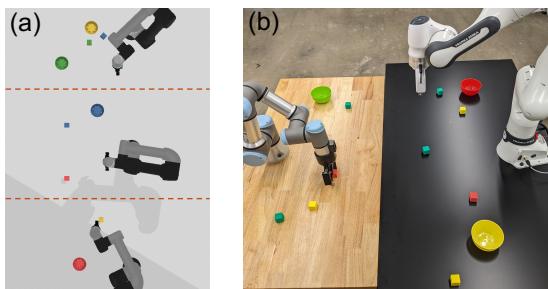


Fig. 7. 3D simulation environments and real hardware demonstrations: robot arms collaborate to move all the boxes into the same colored bowls. Each robot arm has a limited workspace and can only move within its assigned region (divided by the dotted lines or different colored tables).

TABLE III

EVALUATION RESULTS ON BOXNET FOR REAL ROBOT ARMS AND 3D SIMULATIONS, REPORTED AS AVERAGE SUCCESS RATES OVER 20 RUNS.

Success	Code-Symbo.-P.	Only Question	Code Answer
Two Robots (real)	100%	45%	70%
Three Robots (3D)	100%	20%	55%
Six Robots (3D)	95%	0%	35%

3D Simulation and Physical Demonstrations For the BoxNet task, we conduct experiments in both a 3D simulation environment using Pybullet [39] and on a real two-arm robotic system, as shown in Figure 7. The environment

includes colored boxes, colored bowls, and robot arms. The objective is to move each colored box into the bowl of the matching color using the fewest possible actions. Each robotic arm is stationary and limited to operating within its designated workspace (indicated by the dotted lines or the different colored tables). Arms can only pick up and place boxes within their workspace or on the border between workspaces. We also evaluate the system’s scalability by testing with two arms in the real-world experiments, and with either three or six arms in the simulation environment.

The 3D real-world and simulation environments introduce additional complexity by incorporating an image-to-text model, ViLD [42], which provides object bounding boxes and text descriptions. Additionally, the 3D simulation includes a richer environment model that allows for action execution errors caused by dynamic factors (e.g., boxes slipping from the gripper), requiring real-time re-planning. When such execution errors occur, we query the three tested methods to re-plan the next steps.

We conduct 20 trials for each scenario. The results, shown in Table III, demonstrate that Code-as-Symbolic-Planner consistently outperforms methods that do not incorporate symbolic computing, aligning with the findings from the other experiments.

VII. CONCLUSION

We present Code-as-Symbolic-Planner, a framework that leverages LLMs to generate symbolic code for general robot TAMP. Compared to the two existing types of approaches for applying LLMs to TAMP — using LLMs either as direct planners or as translators interfacing with external planners, Code-as-Symbolic-Planner directly integrates symbolic computing into the planning process while maintaining strong generalizability across diverse tasks and constraints.

Our experiments, conducted on seven representative TAMP tasks in both 2D/3D simulation environments and real hardware settings, demonstrate the strong potential of Code-as-Symbolic-Planner in solving complex robot TAMP problems. We believe this framework offers a promising third paradigm for applying LLMs to robot TAMP tasks.

Limitations and Future Directions First, while our results show that current LLMs can generate correct code to solve TAMP tasks under multi-round guidance, the generated code is often not optimal, especially for high-complexity tasks where it may exceed time limits. In robotics, many existing planners are highly efficient but lack generalizability. Combining Code-as-Symbolic-Planner with these efficient planners is a promising direction for future work.

Second, helping LLMs formalize tasks into code remains a key challenge. In our study, all task information and environment states are provided as detailed text descriptions. To improve usability, future work could explore incorporating additional input modalities, such as vision and force sensing.

Third, We also observe that the required output format for plans strongly influences the coding success rate, underscoring the need to enhance LLMs’ self-formalization capabilities for TAMP problems.

REFERENCES

- [1] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, “Integrated task and motion planning,” *Annual review of control, robotics, and autonomous systems*, vol. 4, pp. 265–293, 2021.
- [2] M. Fox and D. Long, “Pddl2. 1: An extension to pddl for expressing temporal planning domains,” *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.
- [3] E. A. Emerson, “Temporal and modal logic,” in *Formal Models and Semantics*. Elsevier, 1990, pp. 995–1072.
- [4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [5] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg, “Text2motion: From natural language instructions to feasible plans,” *arXiv preprint arXiv:2303.12153*, 2023.
- [6] Y. Ding, X. Zhang, C. Paxton, and S. Zhang, “Task and motion planning with large language models for object rearrangement,” *arXiv preprint arXiv:2303.06247*, 2023.
- [7] N. Wake, A. Kanehira, K. Sasabuchi, J. Takamatsu, and K. Ikeuchi, “Chatgpt empowered long-step robot control in various environments: A case application,” *arXiv preprint arXiv:2304.03893*, 2023.
- [8] M. Skreta, N. Yoshikawa, S. Arellano-Rubach, Z. Ji, L. B. Kristensen, K. Darvish, A. Aspuru-Guzik, F. Shkurti, and A. Garg, “Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting,” *arXiv preprint arXiv:2303.14100*, 2023.
- [9] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” in *ICML 2022 Workshop on Knowledge Retrieval and Language Models*, 2022. [Online]. Available: <https://openreview.net/forum?id=6p3AuaHAFiN>
- [10] L. Guan, K. Valmecikam, S. Sreedharan, and S. Kambhampati, “Leveraging pre-trained large language models to construct and utilize world models for model-based task planning,” *arXiv preprint arXiv:2305.14909*, 2023.
- [11] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, *et al.*, “Do as i can, not as i say: Grounding language in robotic affordances,” *arXiv preprint arXiv:2204.01691*, 2022.
- [12] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, *et al.*, “Inner monologue: Embodied reasoning through planning with language models,” *arXiv preprint arXiv:2207.05608*, 2022.
- [13] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, “Code as policies: Language model programs for embodied control,” *arXiv preprint arXiv:2209.07753*, 2022.
- [14] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, “ProgPrompt: Generating situated robot task plans using large language models,” in *International Conference on Robotics and Automation (ICRA)*, 2023. [Online]. Available: <https://arxiv.org/abs/2209.11302>
- [15] Y. Chen, J. Arkin, C. Dawson, Y. Zhang, N. Roy, and C. Fan, “Autotamp: Autoregressive task and motion planning with llms as translators and checkers,” in *2024 IEEE International conference on robotics and automation (ICRA)*. IEEE, 2024, pp. 6695–6702.
- [16] R. E. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” *Artificial Intelligence*, vol. 2, no. 3, pp. 189–208, 1971. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0004370271900105>
- [17] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.
- [18] J. Ferrer-Mestres, G. Frances, and H. Geffner, “Combined task and motion planning as classical ai planning,” *arXiv preprint arXiv:1706.06927*, 2017.
- [19] C. R. Garrett, T. Lozano-Perez, and L. P. Kaelbling, “Ffrob: Leveraging symbolic planning for efficient task and motion planning,” *The International Journal of Robotics Research*, vol. 37, no. 1, pp. 104–136, 2018.
- [20] F. Lagriffoul and B. Andres, “Combining task and motion planning: A culprit detection problem,” *The International Journal of Robotics Research*, vol. 35, no. 8, pp. 890–927, 2016.
- [21] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, “Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 440–448.
- [22] M. Colledanchise, D. Almeida, and P. Ögren, “Towards blended reactive planning and acting using behavior trees,” in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 8839–8845.
- [23] E. Fernandez-Gonzalez, B. Williams, and E. Karpas, “Scottyactivity: Mixed discrete-continuous planning with convex optimization,” *Journal of Artificial Intelligence Research*, vol. 62, pp. 579–664, 2018.
- [24] D. Sun, J. Chen, S. Mitra, and C. Fan, “Multi-agent motion planning from signal temporal logic specifications,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 3451–3458, 2022.
- [25] M. Katayama, S. Tokuda, M. Yamakita, and H. Oyama, “Fast ltl-based flexible planning for dual-arm manipulation,” in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 6605–6612.
- [26] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, “Language models as zero-shot planners: Extracting actionable knowledge for embodied agents,” in *International Conference on Machine Learning*. PMLR, 2022, pp. 9118–9147.
- [27] J. Wu, R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser, “Tidybot: Personalized robot assistance with large language models,” *arXiv preprint arXiv:2305.05658*, 2023.
- [28] Y. Chen, J. Arkin, Y. Zhang, N. Roy, and C. Fan, “Scalable multi-robot collaboration with large language models: Centralized or decentralized systems?” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 4311–4317.
- [29] Z. Yang, C. Garrett, D. Fox, T. Lozano-Pérez, and L. P. Kaelbling, “Guiding long-horizon task and motion planning with vision language models,” *arXiv preprint arXiv:2410.02193*, 2024.
- [30] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jaramanian, Y. Zhu, L. Fan, and A. Anandkumar, “Eureka: Human-level reward design via coding large language models,” *arXiv preprint arXiv:2310.12931*, 2023.
- [31] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [32] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, *et al.*, “The llama 3 herd of models,” *arXiv preprint arXiv:2407.12783*, 2024.
- [33] C. E. Jimenez, J. Yang, A. Wetzig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” *arXiv preprint arXiv:2310.06770*, 2023.
- [34] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, 2023.
- [35] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, *et al.*, “Opendedvin: An open platform for ai software developers as generalist agents,” *arXiv preprint arXiv:2407.16741*, 2024.
- [36] K. Wang, H. Ren, A. Zhou, Z. Lu, S. Luo, W. Shi, R. Zhang, L. Song, M. Zhan, and H. Li, “Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning,” *arXiv preprint arXiv:2310.03731*, 2023.
- [37] Y. Chen, H. Jhamtani, S. Sharma, C. Fan, and C. Wang, “Steering large language models between code execution and textual reasoning,” 2024. [Online]. Available: <https://arxiv.org/abs/2410.03524>
- [38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [39] E. Coumans and Y. Bai, “Pybullet, a python module for physics simulation for games, robotics and machine learning,” <http://pybullet.org>, 2016–2021.
- [40] K. Valmecikam, M. Marquez, A. Olmo, S. Sreedharan, and S. Kambhampati, “Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [41] A. Curtis, N. Kumar, J. Cao, T. Lozano-Pérez, and L. P. Kaelbling, “Trust the proc3s: Solving long-horizon robotics problems with llms and constraint satisfaction,” *arXiv preprint arXiv:2406.05572*, 2024.
- [42] X. Gu, T.-Y. Lin, W. Kuo, and Y. Cui, “Open-vocabulary object detection via vision and language knowledge distillation,” *arXiv preprint arXiv:2104.13921*, 2021.