Coding Agents for Generalized Task and Motion Planning Problems

***Abstract*— Task and motion planning (TAMP) problems remain difficult even with full observability and object-centric states because discrete decisions are tightly coupled to geometric, kinematic, and dynamic constraints. Generalized TAMP addresses this difficulty by exploiting regularities across problem instances to reduce planning effort on new instances. However, existing methods require substantial TAMP-specific engineering. We investigate whether coding agents can automate this process by synthesizing programs that generalize across instances. Given a task description and simulator access, each agent chooses how to investigate the environment while developing a program within a fixed synthesis budget. The program is then frozen and evaluated on unseen instances. We evaluate *Claude Code* (Opus 5) and *Codex* (GPT-5.6 Sol) on 28 simulated environments from KinDER and PDDLStream, spanning geometric, kinematic, and dynamic constraints, with object counts beyond those evaluated in the original benchmark. Across all program synthesis methods, we evaluate 700 generated programs on 100 held-out instances each, 70,000 evaluation episodes in total. Overall, we find that coding agents are surprisingly effective at generalized TAMP: both agents outperform hand-engineered planners, one-shot generation, and an LLM-based generalized planning baseline in mean success (74% and 51% across all environments, versus 28% for the strongest baseline and 47% for the planners on the 16 environments they cover). As object counts grow, *Claude Code*'s programs maintain higher success than the planner, using an order of magnitude less computation per instance on average. Synthesis logs show agents using interaction to calibrate physical models, test edge cases, and refine strategies. We release all code, including the full prompts given to the agents. These findings support coding agents as an important baseline for generalized TAMP.**

I. Introduction

We are interested in the extent to which state-of-the-art coding agents can solve the constrained manipulation problems that typify task and motion planning (TAMP). Even with full observability and object-centric states, these problems remain formally hard [1] and challenging in practice because horizons are long, feedback is sparse, and geometric, kinematic, and dynamic constraints are tightly coupled [2]–[7]. To mitigate these difficulties, generalized TAMP [8]–[14] exploits regularities across problem instances to produce reusable solutions, for example by learning samplers, feasibility predictors, search heuristics, or abstractions. We ask whether coding agents can similarly discover regularities that enable fast and effective planning, while relying on far less TAMP-specific scaffolding than previous methods.

Prior evidence points in opposite directions. Coding agents are improving rapidly on software-engineering benchmarks [15], [16], and large language models (LLMs) have shown increasing success in classical planning domains [17], [18]. Yet when LLMs are asked to make the geometric and physical decisions within a TAMP system, they perform poorly, even when the relevant geometry is included in the prompt [19], [20]. It therefore remains unclear whether advances in coding transfer to the physical reasoning that distinguishes TAMP. Either answer matters for the field. Success would suggest that coding agents can reduce much of the domain-specific engineering required by TAMP methods; failure would help identify concrete limitations of current agents.

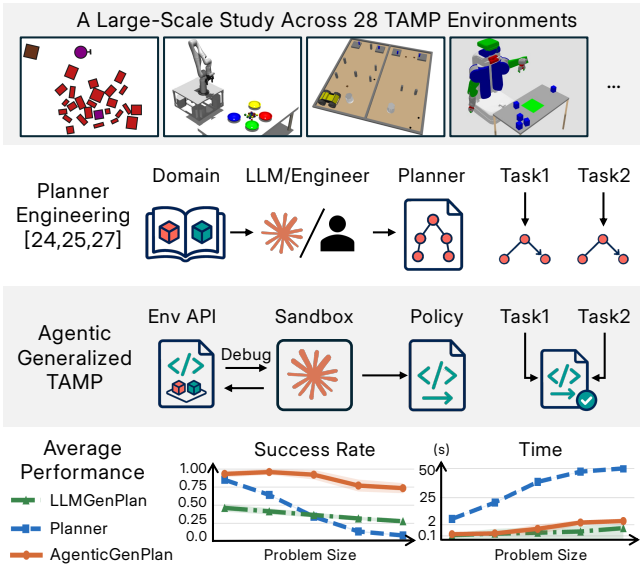

Fig. 1: **Coding agents for generalized TAMP.** A coding agent (Claude Opus 5) synthesizes a programmatic policy shared across problem instances, without hand-designed planning components. Compared with the planner and LLM-based generalized planner, the agentic method achieves a higher success rate while maintaining efficiency.

To answer this question, we present a large-scale, systematic study of off-the-shelf coding agents for generalized TAMP (AgenticGenPlan). The study covers 28 environments, five program synthesis methods, five runs per method per environment, and 100 held-out test instances per resulting program, 70,000 evaluation episodes in total.[1] For each environment, we give a coding agent a task description, simulator access, and a fixed synthesis budget (Figure 1). Within this budget, the agent chooses which experiments to run, writes code to probe the simulator, and uses the results to develop and test its program. The agent returns a single program, which is frozen and evaluated on unseen instances, with no LLM involved at test time. In our main setting, the agent receives no environment source code and no hand-designed planning abstractions. Beyond the task description and simulator access, nothing is engineered for the agent.

We evaluate this approach with two different agentic backends (*Claude Code* with Opus 5 and *Codex* with GPT-5.6 Sol) on environments spanning kinematic and dynamic tasks in two and three dimensions, including the KinDER

[1] Website: `r8508271-lang.github.io`
Code: `anonymous.4open.science/r/agentamp-3481`

benchmark [20] (comprising 25 environments across four families) and three PDDLStream domains [19]. We compare the synthesized programs with TAMP planners given the hand-designed predicates, operators, samplers, and skills supplied by the benchmarks, and we measure both success and runtime as the number of objects increases. We also compare interactive synthesis with the LLM-based generalized planning method LLMGenPlan [21] and the one-shot generation variant, both given environment source code. We additionally evaluate *Claude Code* with environment source code, which bounds the cost of the black-box restriction. We release all code, including the full agent prompts.

Overall, both coding agents outperform the hand-engineered planners on the 16 environments where one is available: *Claude Code* averages 82% success, 1.7 times the planner's 47%, and *Codex* averages 56%. As object counts grow, they maintain higher success and low computation times, while the planner takes longer and solves fewer instances (Figure 1). Both coding agents also outperform one-shot generation and LLMGenPlan in mean success over all 28 environments, and the source-access reference enables successful policies in environments where *Claude Code*'s main-setting runs fail.

Analysis of the synthesized programs and interaction logs shows agents using targeted probing to infer environment dynamics and developing strategies that exploit regularities across instances. Some programs narrow a search over actions by first identifying which obstacles block a route to the goal, then planning how to move them; others adapt a fixed manipulation sequence to each instance without searching. The agents also discovered unexpected strategies, including non-prehensile maneuvers and uses of the environment layout that, to our knowledge, have not appeared in prior work on these benchmarks (Figure 2). We consider this qualitative evidence among the strongest in the study, both for what it says about agentic physical reasoning and because strategies absent from any published solution are hard to attribute to memorized training data. Failures remain: the hardest dynamic three-dimensional environments, including both sweeping tasks, are unsolved in the main setting. Together, these results establish coding agents as an important baseline for future work on generalized TAMP.

II. Problem Setting

A. MDPs and Simulator Access

We study finite-horizon, goal-directed Markov decision processes (MDPs). An MDP is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \rho, H \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces, P is the transition model, R is a sparse reward function indicating goal achievement, ρ is the initial-state distribution, and H is the horizon. States are fully observed and object-centric: a state maps each typed object to a real-valued feature vector describing properties such as pose, geometry, joint configurations, and velocity [20].

Each environment is represented by an MDP and a task description d. The initial-state distribution ρ generates problem instances with varying object counts, configurations, and geometry. For example, in an object-retrieval task, instances may differ in the number and placement of obstacles surrounding the target object. The state and action spaces, transition model, and reward function are shared.

We assume simulator access through `reset` and `step`: a method can sample an initial state $s_0 \sim \rho$ and execute an action a_t from the current state s_t to obtain $s_{t+1} \sim P(\cdot \mid s_t, a_t)$, together with the reward and termination signal.

B. TAMP Environments

We study TAMP environments that are simplified relative to real-world manipulation, following common practice in TAMP research [2]–[7], [19], [20]. With fully observed, object-centric states and no need for perception or language understanding, the remaining challenge is physical reasoning [20]. Horizons are long, rewards are sparse, and the instance distributions are broad enough that no fixed action sequence works. Most importantly, discrete choices are tightly coupled to continuous geometric, kinematic, and dynamic constraints: which object to manipulate, which tool to use, or which subgoal to pursue determines which motions remain feasible, collision constraints grow with the number of objects, and feasible actions can occupy small regions of the action space.

C. Synthesis and Evaluation

At learning time, given the task description d and simulator access, a synthesis method produces a program for $\mathcal{M}$ within a fixed budget. At step t, the program receives $s_t \in \mathcal{S}$ and returns $a_t \in \mathcal{A}$. Since the agent can generate programs that maintain internal state between steps, we denote the induced policy by $a_t = \pi(h_t)$, where $h_t = (s_0, a_0, \ldots, s_t)$ is the interaction history.

At evaluation time, the program is frozen and the synthesis method is no longer invoked. In particular, any coding agent or LLM used during learning is unavailable. The program is evaluated on initial states $s_0 \sim \rho$ that were intentionally hidden during learning.

Our primary metric is success rate: the probability that the program reaches the goal within the horizon H and a wall-clock limit of τ seconds. We additionally measure per-instance computation time spent during planning and deciding on actions, excluding synthesis and time spent advancing the evaluation environment.

III. Generalized TAMP with Coding Agents

A. Coding Agents

We instantiate the synthesis method described in Section II using off-the-shelf coding agents, specifically *Claude Code* running Opus 5 and *Codex* running GPT-5.6 Sol, both with their default settings. These integrate frontier LLMs with harnesses that enable them to read files, write programs, and execute arbitrary commands. We therefore run the agents inside a sandboxed Docker container with a separate filesystem and no network access. We test this isolation through red-teaming, including attempts to recover environment source code, import forbidden libraries, access the host filesystem, or reach the network.

Fig. 2: **Unexpected successful strategies found by the coding agent.** Each row shows one execution from left to right; the annotation describes the unexpected behavior. The strategies come from both default settings and "source code available".

For our main setting, we keep the sandbox bare: the agents have only a Python interpreter with NumPy and SciPy, forcing them to generate end-to-end policies instead of relying on existing libraries. Each agent receives an initial prompt containing the task description d, which includes the environment name and descriptions of the task, observation and action spaces, and goal. We further explain that the environment can vary in object count, and that solutions must support any valid number of objects. We finally explain the evaluation setting, providing the agent with the time τ it will have to run its programs. We provide no hand-written TAMP predicates, operators, samplers, or skills.

To enable interactive learning, we provide the agents with simulator access to the environment: each receives a class

that implements `reset` (ρ), `step` (P), and other helpers, including ones to render states as images. The agents see only a client, while the server with the actual implementation of these functions runs outside the container and is inaccessible to the agents. Using this interface, each agent chooses what to run: it can write and execute custom tests, inspect states, and revise its program based on the results.

We also evaluate an additional *Claude Code + source* setting, where the environment source code is available inside the container. The agents can inspect the implementation and import helper functions, *e.g.*, inverse kinematics solvers. During synthesis, source access also lets the agent set arbitrary states and otherwise manipulate the simulator directly, providing generative access to the transition model [22]. This setting lets us study policy synthesis when the environment's mechanics are explicitly available, so the agents can concentrate on developing behavior with less need to infer how the environment works through interaction.

Each agent implements the programmatic policy π as a class with a `reset` method for episode initialization and a `get_action` method for computing $a_t = \pi(h_t)$. Beyond this interface, we do not constrain the program to any particular abstractions or solution strategy, such as symbolic representations, planning algorithms, or skills.

B. Baselines

We compare AgenticGenPlan against *TAMP planners* that combine symbolic search with sampling to satisfy continuous constraints [7]; the benchmarks provide planners for 16 of the 28 environments. We follow official implementations [7], [20] for the hand-written predicates, operators, samplers, and motion planners (skills) for generating feasible trajectories [10]. They compute a new plan per evaluation instance.

Furthermore, we re-implement *LLMGenPlan* [21], which we run using Opus 5, as a representative non-agentic LLM generalized planning method. In this setting, the LLM cannot use tools or access the filesystem. It receives a prompt adapted from the original work to our environment and program interfaces, together with the full source code of the environment, and must write a program implementing the same specifications as in our main setting. As the LLM cannot run code, a fixed pipeline evaluates each program generated by the LLM and returns specific pre-defined feedback (an exception with its traceback, an invalid action, or an unsolved instance with its seed). LLMGenPlan receives the same synthesis budget as the coding agents, but the LLM never chooses what to run, cannot write custom tests, and sees only the pre-defined feedback. We finally report the performance of the first program generated by LLMGenPlan as a *One-shot* baseline, to evaluate the LLM without any kind of refinement loop.

IV. Experiments

We design experiments to answer the following questions about the efficacy and efficiency of AgenticGenPlan:

Q1. Can agents write generalized programs for TAMP?
Q2. What strategies do agents discover?
Q3. What advantage does being "agentic" give?
Q4. How efficient are the synthesized programs?
Q5. Does access to environment source code help?

A. Environment Setup

We select the KinDER benchmark [20] as our main evaluation setting. This covers 25 environments, grouped into four families: *Kinematic 2D*, *Dynamic 2D*, *Kinematic 3D*, and *Dynamic 3D*. The kinematic environments are free of dynamics, while in the dynamic environments, outcomes depend on contact, velocity, and friction, so successful programs may need behaviors such as sweeping, pouring, and tossing. Twenty of these environments feature variants with different object counts, including counts beyond those evaluated in the original benchmark. We further include three PDDLStream domains, originally introduced by Garrett et al. [6], where LLMs have been shown ineffective [19]: Packing, Blocked, and Rovers.

For each method, we perform five independent runs per environment. Each coding-agent and LLMGenPlan synthesis run has a budget of $20 in model usage. We evaluate the resulting programs and planners on the same 100 instances per environment, sampled from the same initial-state distribution ρ used during synthesis. We generate the evaluation seeds randomly to make it unlikely that agents test them during synthesis, and verify afterward that none were used. We use a timeout τ of 60 seconds per evaluation instance for all methods, matching the original KinDER protocol.

B. Results and Analysis

Tables I and II report success rates across the 28 environments. *Claude Code* is the best performing agent, averaging 96% success on Kinematic2D, 92% on Dynamic2D, 90% on Kinematic3D, 45% on the harder Dynamic3D, and 78% on PDDLStream. Both coding agents exceed the planning baseline in most environments with one available (12 of 16 for *Claude Code*, 9 for *Codex*), despite lacking the hand-designed models, skills, and samplers supplied to these TAMP methods, and their programs exploit regularities within each environment to restrict the decisions considered at test time (**Q1**).

Discovering Unexpected Strategies: We find that the coding agents discover unexpected manipulation strategies, depicted in Figure 2 (**Q2**). To name a few, in Dynamic2D ScoopPour (row two), the agent decides to rotate the tool and regrasp it from the left side, which enables a huge left-side scoop. This strategy shows higher efficiency than right-side scoops. In SweepIntoDrawer (AgenticGenPlan + source), the agent ignores the tool (sweeper) and uses the gripper to sweep the crumbs one by one.

Testing Edge-Cases: One advantage that agentic methods provide over static LLM calls is the flexibility to write and execute custom tests (**Q3**). These let agents investigate failures and evaluate their policies on selected configurations. In Kinematic2D StickButton, for example, *Claude Code* writes a script that repeatedly calls `reset` with different seeds, inspects the sampled states, and saves edge cases

Method	Kinematic 2D						Dynamic 2D				Kinematic 3D				
	StickButton	Obstruction	ClutteredStorage	ClutteredRetrieval	Motion	PushPullHook	Obstruction	PushPullHook	PushT	ScoopPour	Obstruction	Packing	Transport	Table	BaseMotion
Planner	0.36 [0.36–0.36]	0.41 [0.41–0.41]	0.19 [0.19–0.19]	0.51 [0.51–0.51]	0.71 [0.68–0.73]	–	0.24 [0.24–0.24]	0.13 [0.13–0.13]	–	–	–	**0.67** [0.66–0.67]	0.63 [0.59–0.69]	–	**1.00** [1.00–**1.00**]
One-shot Opus 5	0.11 [0.03–0.16]	0.20 [0.00–0.54]	0.02 [0.00–0.06]	0.13 [0.02–0.23]	0.81 [0.17–1.00]	0.01 [0.00–0.04]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.17 [0.01–0.36]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.06 [0.00–0.30]	0.00 [0.00–0.00]
LLMGenPlan Opus 5	0.83 [0.70–0.97]	0.82 [0.53–**1.00**]	0.02 [0.00–0.06]	0.32 [0.21–0.41]	0.97 [0.95–**1.00**]	0.12 [0.02–0.34]	0.49 [0.36–0.77]	0.00 [0.00–0.00]	0.17 [0.00–0.41]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.14 [0.00–0.58]	0.05 [0.00–0.27]	0.85 [0.41–**1.00**]	**1.00** [1.00–**1.00**]
AgenticGenPlan GPT-5.6 Sol	0.94 [0.90–0.99]	0.96 [0.93–0.99]	0.37 [0.19–0.65]	0.21 [0.12–0.27]	**1.00** [1.00–**1.00**]	0.69 [0.46–0.90]	0.77 [0.60–0.90]	0.52 [0.37–0.64]	0.76 [0.21–**1.00**]	0.83 [0.20–**1.00**]	0.26 [0.00–0.69]	0.37 [0.00–0.65]	0.17 [0.00–0.53]	0.99 [0.97–**1.00**]	**1.00** [1.00–**1.00**]
AgenticGenPlan Opus 5	**1.00** [1.00–**1.00**]	**1.00** [1.00–**1.00**]	**0.99** [0.99–**1.00**]	**0.81** [0.36–**0.98**]	**1.00** [1.00–**1.00**]	**0.98** [0.91–**1.00**]	**0.88** [0.79–**0.94**]	**0.83** [0.63–**0.95**]	**1.00** [1.00–**1.00**]	**0.97** [0.89–**1.00**]	**0.85** [0.41–**1.00**]	0.66 [0.52–**0.79**]	**0.99** [0.94–**1.00**]	**1.00** [1.00–**1.00**]	**1.00** [1.00–**1.00**]
AgenticGenPlan + source Opus 5	1.00 [1.00–1.00]	1.00 [1.00–1.00]	1.00 [1.00–1.00]	0.96 [0.92–0.99]	1.00 [1.00–1.00]	1.00 [1.00–1.00]	0.92 [0.87–0.96]	0.92 [0.77–0.99]	1.00 [1.00–1.00]	0.88 [0.65–1.00]	1.00 [0.99–1.00]	0.98 [0.94–1.00]	1.00 [1.00–1.00]	1.00 [1.00–1.00]	1.00 [1.00–1.00]

TABLE I: **Success rate over environments**. We report the mean over five runs, with [min–max] across run-level success rates below. Bold marks the best mean and the best max per environment among the main-setting rows. AgenticGenPlan + source additionally gives the agent the environment source code. A dash marks environments for which KinDER provides no planner. Dynamic 3D and PDDLStream environments are listed in Table II.

Method	Dynamic 3D										PDDLStream		
	BalanceBeam	ConstrainedCupboard	Dynamo	Rearrange	ScoopPour	Shelf	SortClutteredBlocks	SweepIntoDrawer	SweepSimple	Tossing	Packing	Blocked	Rovers
Planner	–	–	–	–	–	0.27 [0.22–0.30]	–	0.00 [0.00–0.00]	–	0.77 [0.77–0.78]	0.54 [0.50–0.56]	0.74 [0.70–0.77]	0.30 [0.24–0.38]
One-shot Opus 5	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.13 [0.01–0.34]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.12 [0.00–0.40]	0.00 [0.00–0.01]	0.26 [0.00–0.87]
LLMGenPlan Opus 5	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.98 [0.95–**1.00**]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.01]	0.21 [0.00–0.39]	0.00 [0.00–0.01]	0.86 [0.81–0.97]
AgenticGenPlan GPT-5.6 Sol	0.64 [0.00–**1.00**]	0.00 [0.00–0.00]	**1.00** [1.00–**1.00**]	0.09 [0.00–0.32]	0.00 [0.00–0.00]	0.24 [0.00–0.63]	0.01 [0.00–0.03]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	0.81 [0.41–**1.00**]	**0.75** [0.34–**0.99**]	0.78 [0.00–**1.00**]
AgenticGenPlan Opus 5	**0.97** [0.87–**1.00**]	**0.08** [0.00–**0.34**]	0.92 [0.60–**1.00**]	**0.58** [0.01–**0.96**]	**0.09** [0.00–**0.31**]	**0.60** [0.00–**1.00**]	**0.30** [0.13–**0.72**]	0.00 [0.00–0.00]	0.00 [0.00–0.00]	**0.99** [0.96–**1.00**]	**0.99** [0.96–**1.00**]	0.38 [0.25–0.56]	**0.98** [0.90–**1.00**]
AgenticGenPlan + source Opus 5	1.00 [0.99–1.00]	0.47 [0.05–1.00]	1.00 [1.00–1.00]	0.77 [0.15–0.97]	0.38 [0.21–0.57]	0.43 [0.00–0.98]	0.49 [0.20–0.96]	0.57 [0.00–0.97]	0.07 [0.00–0.21]	0.85 [0.31–1.00]	1.00 [1.00–1.00]	0.87 [0.43–1.00]	0.99 [0.98–1.00]

TABLE II: **Dynamic 3D and PDDLStream environments**. Same protocol and notation as Table I. The planner for the PDDLStream environments is PDDLStream itself.

Method	Time/action (ms)
AgenticGenPlan Opus 5	4.002 [0.043–29.177]
AgenticGenPlan + source Opus 5	27.900 [0.065–96.640]

TABLE III: **Policy-computation time per action** on 44 matched seeds across 13 environments where both settings achieve 100% held-out success; equal-environment means with [min–max] across environment-level means. AgenticGenPlan + source is slower because its programs invoke the environment source code during planning.

involving wall-adjacent sticks and high buttons. It combines these edge cases with typical instances to build a diverse test suite, then tests its revised policy on both to challenge it across a broad range of configurations (Figure 3, left).

Building Internal Models: Compared to static LLM-based synthesis methods, we also find that AgenticGenPlan draws on prior robotics knowledge to build initial models, then tests and calibrates them through interaction (**Q3**). In Dynamic3D Shelf, for example, one run constructs an initial kinematic model of the Kinova Gen3 arm from the agent's own knowledge, without internet access. It then uses a grasped block as a marker because the state exposes object positions but not the hand's position. Moving the arm through different configurations lets it calibrate the model's predictions of block positions from joint angles (Figure 3, right). From a few observations, it fits six parameters describing the robot mount and grasp offsets, reducing the RMSE between predicted and observed block positions from 38.9 to 1.8 mm on the calibration observations. It later uses the fitted model to solve inverse kinematics (IK) as part of the solution.

Building on what they learn about the environment, agents continue interacting with it to refine control parameters and manipulation strategies. In BalanceBeam, for example, one *Codex* run moves its small-block placement targets closer to the beam's center, increasing success from 6% to 64%.

***Claude Code* versus *Codex*:** *Claude Code* is the stronger of the two coding agents, achieving higher mean success in 22 of 28 environments, with four ties. Blocked is an exception: *Codex* reaches 75%, compared with 38% for *Claude Code* and 74% for the planner. The robot can retrieve either a nearby obstructed block or, when available, an unobstructed

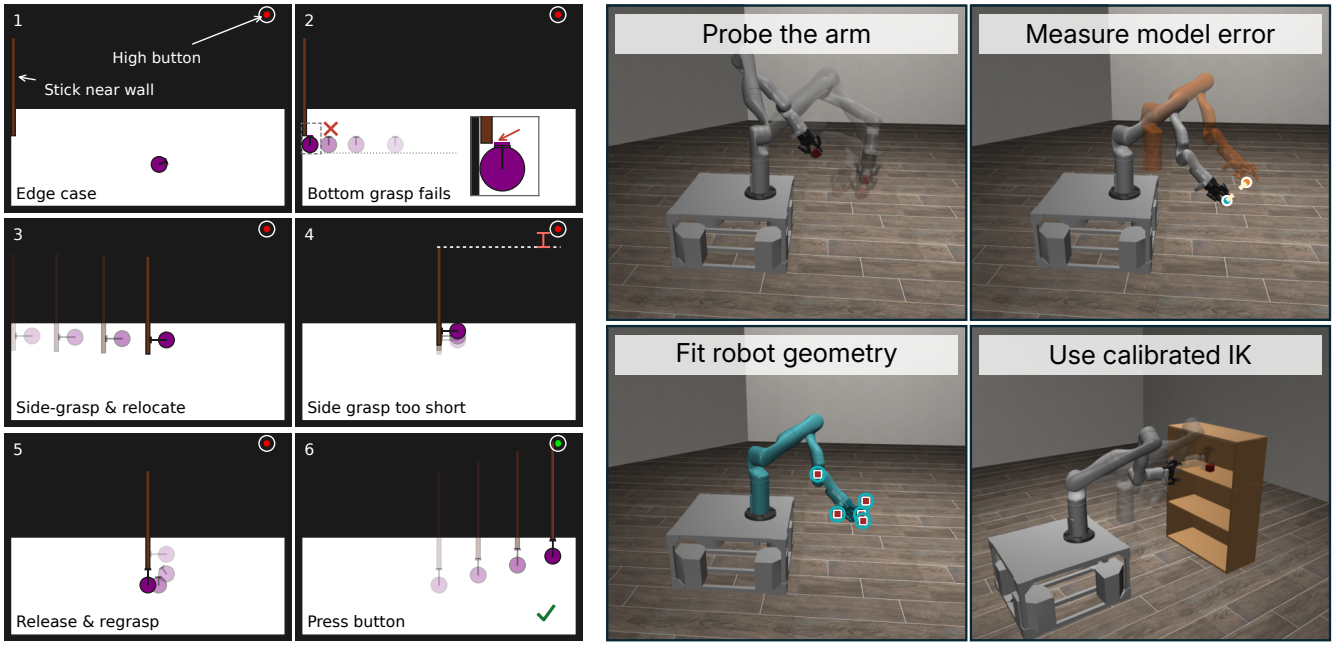

Fig. 3: **Left: custom tests reveal edge cases in StickButton.** *Claude Code* writes a script that repeatedly calls `reset` with different seeds to find wall-adjacent sticks and high buttons for testing its policy. The wall prevents a bottom grasp. A side grasp lets the robot move the stick away from the wall but leaves the high button out of reach; releasing and re-grasping from below provides the required height. **Right: learning robot geometry through interaction.** In Dynamic3D Shelf, *Claude Code* probes the arm while holding a block, fits its kinematic model to observed block positions, and uses the calibrated model for inverse kinematics (IK). The initial model discrepancy is exaggerated for visibility.

block on a distant table. *Claude Code* programs often persist with the nearby block, where obstacle orientations make grasping and retrieval difficult. *Codex* programs can instead switch to the distant block after unsuccessful attempts. In one *Codex* run, a revision adding this fallback raises success from 15% to 56%, with 41 newly solved instances and no lost successes (Figure 4). Subsequent revisions reach 83% final success, solving all 80 instances with an alternative block but only three of the 20 without one.

Program Computation Efficiency: Table III compares policy-computation time per action for *Claude Code* with and without source access, on 44 matched environment/seed pairs in which both settings reach 100% held-out success (**Q4**). Black-box programs choose actions quickly, averaging 4.0 ms per action. Source-access programs average about 7 times that, with large variation across environments, because many of them invoke the environment source code as part of planning, for example calling its motion planners at decision time. Black-box synthesis instead forces the agent to distill what it learns into self-contained code, which yields faster policies at evaluation time.

Environment Source Code Access: We further inspect whether providing the agents with environment source code helps them generate policies (**Q5**). Qualitatively, many black-box runs begin by trying to work out how the environment behaves, sometimes reasoning in natural language and sometimes writing exploratory code, before committing to a policy. As shown in Table I and Table II, source access improves *Claude Code*'s mean success in 19 of 28 environments, including the harder Dynamic3D tasks. In SweepIntoDrawer, where both coding agents score zero in the main setting, source access raises mean success to 57%, with one run reaching 97%. ConstrainedCupboard also improves from 8% to 47%. Environment source code provides both information for diagnosing failures and implementations that programs can reuse. In Kinematic3D Packing, one run uses internal collision information to identify that the gripper, rather than the held part, collides with the rack. It then changes the grasp to provide clearance. Other source-access programs reuse motion planners or test actions in private simulators.

V. Related Work

A. Generalized Planning and Learning for TAMP

TAMP couples discrete decisions with continuous feasibility constraints [7]. Systems such as PDDLStream provide interfaces between symbolic planning and procedures for sampling and checking continuous quantities [6]. Generalized planning seeks solutions that apply across related problem instances [8]. In TAMP, this objective has motivated learning reusable samplers, feasibility predictors, search heuristics, and state and action abstractions [9], [11], [13], [14]; see [10] for a recent survey. We ask whether general-purpose coding agents can likewise exploit regularities across instances without being confined by specific TAMP abstractions.

B. Foundation Models for Task and Motion Planning

One branch of foundation model-driven TAMP research leverages large language models (LLMs) to guide a TAMP planner. For example, Text2Motion combines language-guided task planning with skill affordances [23], and LLM^3 uses motion-planning failures to guide plan revision [24]. In

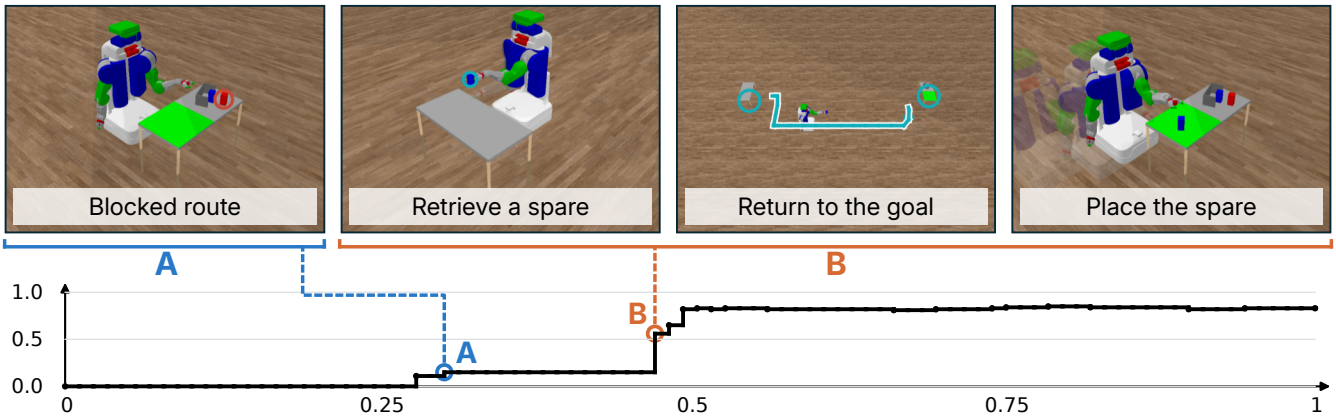

Fig. 4: **Refining manipulation strategies during synthesis.** In one *Codex* Blocked run, a revision adding a spare-block fallback changes success from 15% (A) to 56% (B); the pictured execution retrieves an alternative goal block from a distant table. The curve shows success fraction against normalized commit order.

addition to guidance, LLM-based program synthesis can also automate the engineering of TAMP components. PRoC3S generates programs whose continuous parameters are resolved through constraint satisfaction [25]. MOPS searches over programs specifying constraints for trajectory optimization [26], while OWL-TAMP uses vision-language models to generate constraints within a TAMP system [27]. Instead of committing to certain TAMP abstractions, our evaluation leaves the internal structure of the generated solution open: an agent may implement search, optimization, sampling, or task-specific procedures on its own, with the final objective of efficiently solving as many problem instances as possible.

C. Agentic Synthesis of Robotic Programs

Since Code-as-Policies [28], LLMs have demonstrated the ability to generate robot control code, integrating manipulation skills and perception interfaces. Later works such as GenCHiP [29] and InstructFlow [30] motivate code as a flexible policy representation, while highlighting the importance of the supplied action interface and feedback. In symbolic task planning, Silver et al. [21] use LLMs and execution feedback to synthesize reusable programs that solve new PDDL instances without further LLM calls.

Recent agentic systems provide even closer precedents for reusable policy synthesis. RHO introduces a reflective evolutionary optimizer over policy repositories, using coding agents and execution feedback [31]. ASPIRE develops mechanisms for discovering, repairing, and reusing robotic skills [32], and MEMENTO introduces memory-guided evolutionary search over policy programs [33]. The benchmarks used in these works, such as Robosuite [34] and LIBERO [35], focus on general-purpose manipulation rather than the physical reasoning challenges of TAMP problems [20]. Our focus is a systematic assessment of off-the-shelf coding agents generating code as generalized policies for constrained TAMP-like environments.

D. Systematic Evaluation of Embodied Agents

Mendez-Mendez's study of LLMs within PDDLStream is the closest precedent in evaluation focus [19]. It examines configurations in which LLMs replace task planning, continuous sampling, or both within established planning architectures, and compares them with engineered planners. Whereas it queries LLMs during problem solving, we use simulator feedback to develop a program, then freeze it for LLM-free evaluation on unseen instances. Several benchmarks and systematic studies also inform our evaluation. CaP-X benchmarks coding agents for general manipulation control [36]. Tsui et al. [37] evaluate an LLM agent SDK on general manipulation tasks with iterative execution. KinDER [20] provides procedurally generated environments designed to isolate physical reasoning and already compares planning and learning approaches; we build on these environments and their baseline implementations.

VI. Discussion

Limitations: Our setup assumes fully observed, object-centric states and simulator access during synthesis. It therefore does not establish performance under perception uncertainty. The models' training data are undisclosed, so prior exposure to benchmark code cannot be excluded. However, the synthesis logs show policies being developed through environment probing, testing, and substantial revision. Together with the unexpected strategies the agents synthesized, this provides evidence against simply recalling complete solutions seen during pretraining.

Conclusions: We show that coding agents are strong generalized TAMP planners, synthesizing reusable policies that outperform hand-engineered planners and existing LLM-based synthesis methods on our benchmarks. These results extend the promise of LLM-based generalized planning to environments with geometric, kinematic, and dynamic constraints. Through interaction, agents can investigate the physical behavior of an environment and develop effective strategies without being supplied with its symbolic models, skills, or samplers. This ability to discover both how an environment works and how to solve its tasks makes coding agents an important baseline for future TAMP research.

ACKNOWLEDGMENTS

We used *Claude Code* and *Codex* to help with our implementations and to refine the figures and text. The core scientific contributions, including the problem formulation, the design of the study, and the analysis of results, were developed by the authors, who take full responsibility for the accuracy and correctness of all AI-generated content.

REFERENCES

[1] A. Deshpande, "Exact geometry algorithms for robotic motion planning," Ph.D. dissertation, Massachusetts Institute of Technology, 2019.

[2] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[3] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[4] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[5] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "An incremental constraint-based framework for task and motion planning," *The International Journal of Robotics Research*, vol. 37, no. 10, pp. 1134–1151, 2018.

[6] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2020.

[7] 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.

[8] S. Jiménez, J. Segovia-Aguas, and A. Jonsson, "A review of generalized planning," *The Knowledge Engineering Review*, vol. 34, 2019.

[9] A. Curtis, T. Silver, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, "Discovering state and action abstractions for generalized task and motion planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022, arXiv:2109.11082.

[10] Y. Huang, B. Hedegaard, N. Shah, S. Srivastava, G. Konidaris, and T. Silver, "Learning by and for task and motion planning: A survey," 2026, manuscript. https://prpl-group.com/tamp-learning-survey.pdf.

[11] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez, "Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[12] Z. Wang, C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez, "Learning compositional models of robot skills for task and motion planning," *The International Journal of Robotics Research*, vol. 40, no. 6–7, pp. 866–894, 2021.

[13] A. M. Wells, N. T. Dantam, A. Shrivastava, and L. E. Kavraki, "Learning feasibility for task and motion planning in tabletop environments," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1255–1262, 2019.

[14] R. Chitnis, D. Hadfield-Menell, A. Gupta, S. Srivastava, E. Groshev, C. Lin, and P. Abbeel, "Guided search for task and motion plans using learned heuristics," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

[15] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in *International Conference on Learning Representations (ICLR)*, 2024.

[16] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[17] A. B. Corrêa, A. G. Pereira, and J. Seipp, "Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[18] A. B. Corrêa, A. G. Pereira, and J. Seipp, "Frontier large language models rival state-of-the-art planners," *arXiv preprint arXiv:2511.09378*, 2025.

[19] J. Mendez-Mendez, "A systematic study of large language models for task and motion planning with PDDLStream," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2026, arXiv:2510.00182.

[20] Y. Huang, B. Li, V. Saxena, Y. Liang, U. A. Mishra, L. Ji, L. Zha, J. Wu, N. Kumar, S. Scherer, D. Xu, and T. Silver, "KinDER: A physical reasoning benchmark for robot learning and planning," in *Robotics: Science and Systems (RSS)*, 2026, arXiv:2604.25788.

[21] T. Silver, S. Dan, K. Srinivas, J. B. Tenenbaum, L. P. Kaelbling, and M. Katz, "Generalized planning in PDDL domains with pretrained large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024, arXiv:2305.11014.

[22] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large Markov decision processes," *Machine Learning*, vol. 49, no. 2–3, pp. 193–208, 2002.

[23] K. Lin, C. Agia, T. Migimatsu, M. Pavone, and J. Bohg, "Text2Motion: From natural language instructions to feasible plans," *Autonomous Robots*, vol. 47, pp. 1345–1365, 2023.

[24] S. Wang, M. Han, Z. Jiao, Z. Zhang, Y. N. Wu, S.-C. Zhu, and H. Liu, "LLM^3: Large language model-based task and motion planning with motion failure reasoning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024.

[25] 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," in *Proceedings of the Conference on Robot Learning (CoRL)*, vol. 270, 2025, pp. 1362–1383.

[26] D. Shcherba, E. Cobo-Briesewitz, C. V. Braun, and M. Toussaint, "Meta-optimization and program search using language models for task and motion planning," in *Proceedings of the Conference on Robot Learning (CoRL)*, vol. 305, 2025, pp. 5339–5361.

[27] N. Kumar, W. Shen, F. Ramos, D. Fox, T. Lozano-Pérez, L. P. Kaelbling, and C. R. Garrett, "Open-world task and motion planning via vision-language model generated constraints," *IEEE Robotics and Automation Letters*, vol. 11, no. 3, pp. 3366–3373, 2026.

[28] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as policies: Language model programs for embodied control," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 9493–9500.

[29] K. Burns, A. Jain, K. Go, F. Xia, M. Stark, S. Schaal, and K. Hausman, "GenCHiP: Generating robot policy code for high-precision and contact-rich manipulation tasks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024, pp. 9596–9603.

[30] H. Chi, Z. Feng, Y. Lyu, C. Zheng, L. Luo, Y. S. Ong, I. Tsang, H. Chen, Y. Chang, and H. Yin, "InstructFlow: Adaptive symbolic constraint-guided code generation for long-horizon planning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 38, 2025.

[31] K. Elmaaroufi, J. Svegliato, S. Kalade, G. Schelle, S. A. Seshia, and M. Zaharia, "RHO: Your coding agent is secretly a roboticist," arXiv preprint arXiv:2606.16458, 2026.

[32] R. Lu, Y. Wu, E. Kou, L. Fu, W. Xiao, A. Mandlekar, Y. Xu, G. Shi, K. Goldberg, A. Chen, M. Chowdhury, Y. Zhu, L. Fan, and G. Wang, "ASPIRE: Agentic /skills discovery for robotics," arXiv preprint arXiv:2607.00272, 2026.

[33] A. Sygkounas, V. Aregbede, A. Loutfi, and A. Persson, "MEMENTO: Memory-guided memetic code-as-policy evolution," arXiv preprint arXiv:2607.22832, 2026.

[34] Y. Zhu, J. Wong, A. Mandlekar, R. Martín-Martín, A. Joshi, K. Lin, A. Maddukuri, S. Nasiriany, and Y. Zhu, "robosuite: A modular simulation framework and benchmark for robot learning," arXiv preprint arXiv:2009.12293, 2020.

[35] B. Liu, Y. Zhu, C. Gao, Y. Feng, Q. Liu, Y. Zhu, and P. Stone, "LIBERO: Benchmarking knowledge transfer for lifelong robot learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[36] M. Fu, J. Yu, K. El-Refai, E. Kou, H. Xue, H. Huang, W. Xiao, G. Wang, F.-F. Li, G. Shi, J. Wu, S. Sastry, Y. Zhu, K. Goldberg, and L. Fan, "CaP-X: A framework for benchmarking and improving coding agents for robot manipulation," arXiv preprint arXiv:2603.22435, 2026.

[37] B. Y. Tsui, A. Y. Fang, and T. J. Hwu, "Demonstration-free robotic control via LLM agents," arXiv preprint arXiv:2601.20334, 2026.