ANALYZING AGENTIC BENCHMARKS UNDER SUBGOAL INTERACTIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-calling is a core function of LLM agents, and a plethora of benchmarks attempt to evaluate it. These benchmarks divide into stateful benchmarks, where tool calls operate on shared state such as a database or environment, and stateless benchmarks, where a composition of tools computes or retrieves a value. Stateful benchmarks such as TauBench, AppWorld, and AgentBench turn out to be ergodic, or are ergodified by the harnesses agents run in, and so mask a failure mode studied extensively in the AI planning literature: negative subgoal interaction. In ergodic environments, negatively interacting subgoals increase the cost of achieving one another; in non-ergodic environments, they can make the goal unreachable entirely. While AI planning has studied this concept at length, no study has examined how it affects agentic planning. We introduce PlanBench-Gym, an interactive, tool-calling extension of PlanBench that evaluates LLM agents on PDDL Blocksworld instances constructed along a three-way taxonomy of no-interaction, positive-interaction, and negative-interaction goal structure. Even though these instances remain ergodic under our construction, we find that state-of-the-art reasoning models struggle with negative interactions, though the degree varies: closed source models such as GPT-5.2 fails outright on negative-interaction tasks under most agent scaffolds, while Claude Sonnet 5 completes them but at a measurable efficiency cost. Additional experiments on open-source models mirror the finding with Claude Sonnet 5, where Qwen3.6, GLM5.2, and Gemma4 are able to consistently solve tasks, but do so at a measurable efficiency cost. We further find that prompting mode, whether an agent observes its full interaction history or only the current state, has a substantial effect on completion, independent of interaction class.

1 INTRODUCTION

Widespread interest in agentic AI systems has led to the development of a multitude of benchmarks testing the tool-calling ability of LLMs. These tool-calling benchmarks can either be stateful or stateless. In stateful tool-calling benchmarks, an LLM's tool call affects an external state of a world; examples include TauBench (Yao et al., 2025), AppWorld (Trivedi et al., 2024), and AgentBench (Liu et al., 2023). Stateless tool-calling benchmarks instead require an agent to compose tool calls to compute or retrieve a result without altering any persistent world state; examples include ToolBench, ToolLLM, and versions 1 and 2 of the Berkeley Function Calling Leaderboard.

LLM agents operate in these environments in an observe-act loop. Since planning and execution are interwoven, the agent replans at every step, so there is no guarantee about what plan it is committing to; the only way to guarantee correctness is to generate the whole plan, verify it, and then execute it (Kambhampati et al., 2024; Gundawar et al., 2024). Current agentic harnesses do not do this; they execute directly and rely on execution feedback to correct the agent's behavior afterwards. LLMs are not guaranteed to produce correct plans, but in these benchmarks an incorrect plan rarely costs anything: a bad action can be retried, blocked by a safety check, or quickly reverted, so poor planning shows up as wasted steps rather than as failure. Agentic benchmarks mask poor planning in one of two ways. In some, the environment simply turns out to be ergodic by construction; in others, the agent is equipped with a harness that stabilizes the environment on its behalf. Such a harness curtails the agent's access to the non-ergodic parts of the environment by applying state-based checks on what the agent is allowed to do: in Sokoban, if an agent beside a corner attempts to push a box into

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

it, a check can prevent that operation from ever being executed. Under either arrangement, execution feedback is enough to support a directed-ish random walk toward the goal, since no action the agent takes can permanently close the goal off. In situations such as the HuggingFace incident, safety checks were implemented to prevent an agent from reaching unsafe or unintended states, effectively attempting to ergodify the environment. But safety checks cannot guarantee full coverage of the state space: if an edge case is missed, the agent can reach regions of the state space it was never intended to access, with consequences that a purely ergodic benchmark would never surface.

Both arrangements turn on ergodicity, which is worth stating precisely. An environment's states and actions form a directed graph: each action is an edge from the state it applies in to the state it produces. The environment is ergodic if every state in that graph is reachable from every other, and non-ergodic if some state cannot be reached back from another. What matters for an agent is the version restricted to where it actually starts: an environment is non-ergodic for an agent if two states are both reachable from its initial state, yet one cannot be reached from the other. A single irreversible action is enough to make an environment non-ergodic. Note that ergodicity is not solvability: an ergodic environment can still contain instances with no valid plan, which is a property of the instance, not a failure of the agent.

Since agents need to eventually work in the real world, where no harness stabilizes the environment on their behalf, the question is how fragile current agents become once this masking is relaxed. Negative subgoal interaction gives us a necessary but not sufficient condition for non-ergodicity, and with it a way to study that question. Subgoal interactions arise when the actions supporting different subgoals share state: subgoals are independent if they share no state, positively interacting if progress toward one also advances the other, and negatively interacting if achieving one impedes progress toward the other. A non-ergodic environment necessarily exhibits negative interaction among the subgoals its unreachable regions separate. The converse does not hold: negative interaction arises in fully ergodic environments too, where achieving one subgoal makes another more expensive without ever placing it out of reach. Standard agentic benchmarks contain essentially no negative subgoal interactions of either kind, leaving this behavior untested. PlanBench-Gym is designed to test it.

In this work, we design PlanBench-Gym, a stateful agentic AI benchmark comprising a simulator that preserves the interactive nature of existing agentic AI benchmarks while allowing efficient, fine-grained control over the state space through PDDL. PlanBench-Gym builds on PlanBench (Valmeekam et al., 2023), moving from a one-shot, non-interactive benchmark to one built on tool calls and an interactive structure. The harness itself returns environment feedback to the agent and rejects infeasible actions. To analyze the effect of subgoal interactions in agentic environments, we generate PDDL Blocksworld instances along a three-way axis of no-interaction, positive-interaction, and negative-interaction goal structure.

We evaluate both open-source and closed-source frontier models across three agent scaffolds and two prompting regimes. All models struggle with negative-interaction tasks, but the nature of the difficulty varies. GPT-5.2 fails outright under most scaffolds, while Claude Sonnet 5 completes them at a measurable efficiency cost. The open-source models, GLM5.2, Qwen3.6, and Gemma4 mirror Claude Sonnet 5's behavior: all three complete negative-interaction tasks but require more steps than the optimal plan.

In this study, we make the following contributions:

1. We design a new agentic AI benchmark called PlanBench-Gym, a stateful agentic AI benchmark extending PlanBench to include subgoal interactions and tool-calling.
2. We use the subgoal-interaction taxonomy from the planning literature to construct PDDL instances spanning no-interaction, positive-interaction, and negative-interaction goal structures.
3. We evaluate both closed-source and open-source frontier models: Claude Sonnet 5, GPT5.2, Qwen3.6, Gemma4, GLM5.2 on the resulting taxonomy, across multiple agent scaffolds and prompting regimes.

2 Related Works

2.1 Tool-calling benchmarks

To evaluate and assess the tool-calling ability of LLMs, the agentic AI field has built numerous benchmarks. These benchmarks can be split into two categories: stateful benchmarks and stateless benchmarks. In a stateful benchmark, tool calls carried out by an agent modify the state of an external system, and the agent is graded on whether the sequence of actions it took leaves that system in a specific target final state. TauBench (Yao et al., 2025), AgentBench (Liu et al., 2023), and AppWorld (Trivedi et al., 2024) are examples of stateful tool-calling benchmarks. In a stateless benchmark, an LLM instead makes tool calls to compute or retrieve a value, without any persistent system state being modified or graded; ToolLLM and ToolBench (Qin et al., 2023) are representative examples.

2.2 Implicit versus explicit causal structure

In stateful benchmarks, tool calls operate on a common, shared world state: each call can mutate the same underlying database or environment, which means one tool call's effects can silently alter the preconditions of another. This gives rise to causal dependencies between actions, but in most existing benchmarks that dependencies is *implicit*: it is embedded in tool source code, a natural-language domain policy, or a database schema rather than stated formally. τ-bench (Yao et al., 2025) situates an agent against a live backend database mutated by tool calls and constrained by a natural-language policy; conflicting actions exist, but identifying them requires reading the policy and tracing tool effects through the database schema. ToolSandbox (Lu et al., 2024) explicitly motivates its design around *implicit state dependencies between tools*, recognizing that one tool call can affect the preconditions of another without this relationship being formally declared anywhere. In both cases, whether two actions interact, and how severely, is not something the benchmark states: it must be recovered from the implementation. PlanBench-Gym takes the opposite approach: the causal structure is given explicitly by the PDDL precondition and effect schemas, making action interactions inspectable, verifiable, and controllable by construction.

3 Background

3.1 AI planning and subgoal interactions

AI planning formalisms such as STRIPS represent a task as a set of world-state predicates, a set of ground actions with preconditions and effects, and a goal condition; Blocksworld is a canonical STRIPS domain, and the International Planning Competition (IPC) has long used such domains to benchmark planner performance.

3.1.1 Subgoal Interactions

A planning problem requires an agent to achieve a goal state that consists of multiple fluents. A subgoal can best be viewed as a fluent to be achieved in the final goal state. In order to reach the goal, an agent must apply a series of actions. An agent will pick a certain subgoal to make progress towards, but in achieving that first subgoal it may impede progress towards another subgoal, or make progress towards it. The situation where achieving one subgoal affects the progress made towards another is known as a subgoal interaction. Following Nguyen et al. (2002), subgoals are negatively interacting if achieving one impedes progress towards the other, and positively interacting if achieving one also makes progress towards the other.

Heuristic-search planning commonly assumes *subgoal independence*: that the cost of achieving a conjunctive goal is the sum of the costs of achieving each subgoal separately (Bonet and Geffner, 2001). Subgoal interaction is precisely the failure of this assumption, and it fails in two directions (Nguyen et al., 2002). Assume that $C(G)$ is the cost of an optimal plan achieving G, where G_1 and G_2 are subgoals an agent needs to achieve to reach G. A *positive* interaction holds when

$$C(G_1 \wedge G_2) < C(G_1) + C(G_2),$$

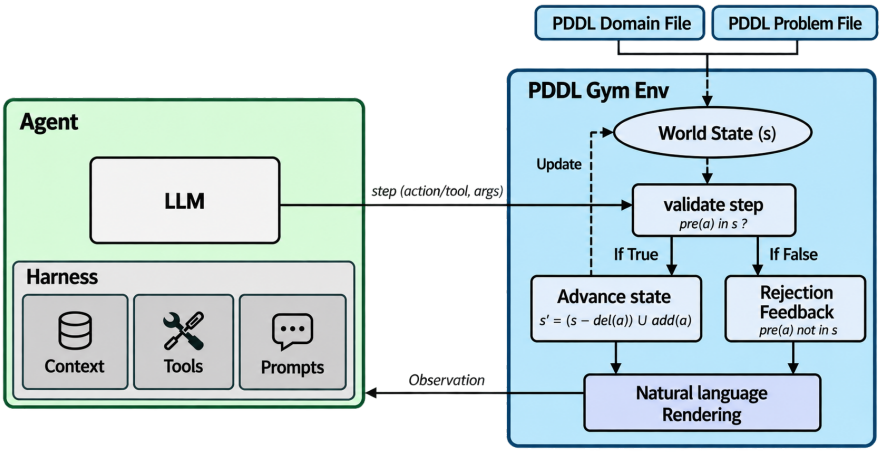


Figure 1: Internal structure of PlanBench-Gym.

since a single action may contribute to both subgoals and need not be paid for twice. A *negative* interaction holds when

$$C(G_1 \wedge G_2) > C(G_1) + C(G_2),$$

since achieving one subgoal undoes progress towards the other and additional actions are required to restore it. When the independence assumption holds exactly, $C(G_1 \wedge G_2) = C(G_1) + C(G_2)$, and the subgoals do not interact.

We adopt this cost-based characterization throughout. It is the definition our instance generation operationalizes directly: as described in Section 4.3, a candidate positive-interaction instance is accepted only if the optimal joint plan cost is strictly below the sum of the optimal costs of the two single-goal probes, with the analogous comparison defining the other classes. Because these comparisons are only meaningful over optimal plan costs, all costs are computed with Fast Downward under A* search with the admissible `lmcut` heuristic.

Finally, subgoal interactions behave differently depending on whether the underlying environment is ergodic. In an ergodic environment, a negative interaction only raises the cost of reaching the goal: achieving one subgoal makes the other more expensive, but never unreachable, so the interaction is fully captured by the cost inequality above. In a non-ergodic environment, where some states reachable from the initial state cannot reach one another, a negative interaction can be more severe: achieving one subgoal may foreclose the other entirely, placing the goal beyond reach at any cost. Under the cost-based characterization, this corresponds to the limiting case where $C(G_1 \wedge G_2) = \infty$, since no plan exists that satisfies both subgoals.

3.2 Bridging agentic and AI planning

PlanBench Valmeekam et al. (2023) established that LLMs can be evaluated on classical planning problems by rendering PDDL instances in natural language, but evaluates plans in a single, non-interactive shot rather than through the step-wise, tool-calling interaction pattern used by agentic benchmarks. This leaves a gap: no existing benchmark combines PlanBench's fine-grained, verifiable control over the underlying state space with the interactive, tool-calling structure of agentic evaluation. PlanBench-Gym, described in the next section, is designed to close this gap.

4 Methodology

4.1 PlanBench-Gym Design

PlanBench-Gym is built on top of PlanBench (Valmeekam et al., 2023), turning its static planning tasks into a closed-loop, tool-calling environment. Conceptually, PlanBench-Gym follows the same

agent-environment interaction loop as the OpenAI Gym framework (Brockman et al., 2016): an agent receives an observation, selects an action, and the environment advances its state and returns a new observation. Internally, PlanBench-Gym implements the same constructs as an OpenAI Gym environment, providing `reset()` and `step()` functions to manage simulator state across episodes.

The state space is the set of all possible assignments of ground atoms over the domain's object vocabulary, and the action space is the set of lifted PDDL action schemas defined in the domain; Blocksworld has four, Logistics six. Each tool exposed to the agent corresponds to exactly one lifted schema: the tool name is the PDDL action name verbatim, and its parameters, in declared order, each accept any object in the domain's object universe. The central design decision is that the agent never observes the state space or action space directly. Following PlanBench (Valmeekam et al., 2023), all states, goals, and available actions are rendered into natural language, so the agent reasons over a text world rather than over formal syntax. This is what allows PlanBench-Gym to evaluate agentic planning rather than PDDL parsing.

The environment maintains world state as a set of ground atoms and advances it using standard STRIPS progression semantics. A tool call `(action_name, args)` grounds the chosen schema against the supplied object bindings to produce one ground action instance; the environment then checks $\mathrm{pre}(a) \subseteq s$ and, if the precondition holds, computes the successor state as $s' = (s \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)$ and returns a new natural-language observation. If the precondition does not hold, including when an object of the wrong type is bound to a parameter since parameters are not type-filtered, the world state is left unchanged and the agent receives an explanation of which preconditions were unmet. The mapping from tool call to ground action is syntactically one-to-one; legality in the current state is what the environment decides at each step. This gives the agent a closed, consistent interface regardless of which PDDL domain is loaded.

A single control loop orchestrates every episode: it resets the environment, passes the resulting observation to the agent, receives an action, and feeds it to the environment's step function, repeating until the episode terminates or the agent signals it cannot solve the task. The environment and agent never communicate directly; all interaction is mediated through this loop. The agent-environment interaction loop is exposed as a pluggable interface, allowing any new agent implementation to be hooked in without modifying the environment.

4.2 Experimental Setup

Our experimental setup utilizes PlanBench-Gym to evaluate frontier LLMs in a closed-loop, tool-calling environment. There are three main components to the experimental setup: context management, the agent scaffold, and problem instance generation. Generating problem instances is an important component of this work, and we explain it in (Section 4.3).

The environment provides rejection feedback whenever an LLM makes a tool call whose preconditions are not met in the current state it returns an explanation of which preconditions failed and leaves the state unchanged.

Context management controls how much of the interaction history an LLM receives at each timestep. Given the structure of chat completions APIs, we can either provide the entire interaction history from the initial timestep (full history), or only the observation from the current timestep (current-state).

The prompting approach determines how the LLM is instructed to act and how its response is interpreted. We define an agent scaffold as the combination of a system prompt and an action-elicitation mechanism. All three scaffolds use the ReAct (Yao et al., 2023) prompt, requesting that the agent emit `<think> </think>` tokens before an `<Action>` token, and all describe the task as an interactive STRIPS planning problem in a Blocksworld environment. The three scaffolds differ only in how the action is elicited and parsed:

- **Prompted ReAct.** The agent emits its reasoning and action entirely as natural-language text, with the full set of available tool calls and an object legend embedded directly in the system prompt. The action is extracted from free-form text by a hand-written parser before being executed.

- **Hybrid ReAct.** Uses the same system prompt as Prompted ReAct, but issues the action through the chat completions API's native tool-calling mechanism rather than free-form text.
- **Native tool-calling.** Uses a ReAct-style prompt without embedding tool-call information in the system prompt body, relying entirely on the chat completions SDK's tool-calling API for action elicitation.

We leave all models at their default parameters as provided by each frontier lab. For each combination of model, agent scaffold, and context management mode, we evaluate 50 task instances per interaction class (no-interaction, positive-interaction, negative-interaction), yielding $3 \times 2 \times 3 = 18$ conditions per model. Each instance is assigned a default step budget $2\times$ the optimal plan length, computed via Fast Downward using A* search with the admissible heuristic `lmcut` heuristic; the episode terminates once the agent reaches the goal or exhausts this budget. In our experiments we set this value to 4
$\times$ the budget and retroactively threshold the budgets.

4.3 Problem instance generation

Each task instance is generated offline by a per-class constructor, then verified before being admitted to the evaluation corpus (Appendix A.1, Figure 3). A schema specifies the interaction class, sampling parameters (with the number of blocks pinned within a given cell to control for instance size as a confound), and symbolic init/goal templates; the corresponding constructor builds a candidate instance whose structure is designed so that the target class holds *by construction*. Every candidate is then checked rather than trusted: it must load and reset successfully in `PDDLGymEnv`, Fast Downward's solvability verdict must match the schema's expected class, and for positive-interaction instances the joint optimal plan cost must be strictly below the sum of the two single-goal probe costs. A candidate that fails any check is discarded and resampled (up to 25 times) rather than relabeled, so no instance in the reported corpus is included on the strength of the constructor alone.

4.3.1 Templates for each interaction class

A schema specifies an initial-state template, a goal template, and sampling ranges; a per-class constructor turns these into a concrete PDDL instance. Each class is designed so that its defining property holds *by construction*, not by post-hoc classification.

No interaction. All blocks start on the table; the goal pairs disjoint blocks into n independent `(on top bottom)` subgoals. Since no block appears in two goal atoms, each subgoal's supporting subplan touches disjoint objects, and joint cost equals the sum of independent costs exactly.

Positive interaction. Each unit consists of four disjoint blocks {base, burden, support, landing}, with burden pre-stacked on base; the goal requires base to support support, and burden to land on landing. Clearing base to satisfy the first subgoal forces an `unstack(burden, base)` action, and the only useful place to place burden is landing, its own goal position. One action thus serves both subgoals, so joint cost is strictly below the sum of independently solving each.

Negative interaction (solvable). We use two constructions. In the *tower* variant, blocks start on the table and the goal is a chain of `on` atoms forming a single tower; since goal atoms share blocks along the chain, only a bottom-up build order succeeds, and any subplan that stacks an upper atom first destroys a lower atom's precondition. In the *Sussman* variant, the goal chain is the same, but a subset of goal edges are pre-built in inverted order at initialization, so the agent must first dismantle an existing stack before it can build, a scaled instance of the classic Sussman anomaly.

Plan-length padding. Because all three subgoal interaction classes cost exactly two steps per subgoal, an optimal planner ignores irrelevant distractor blocks, so instance difficulty cannot be scaled simply by adding loose objects. We instead stack K extra blocks in a chain on a block already required by the goal, forcing $2K$ mandatory unstack/put-down steps in any valid plan. The blocking stack's target block is chosen per class so that padding cannot alter the class-defining causal structure; for positive-interaction instances, for example, padding is placed on the support block, which adds the same cost to both the joint solve and the single-subgoal probe, preserving the joint-vs-independent cost margin that defines the class.

5 RESULTS AND DISCUSSION

This section presents experiments evaluating agentic planning under subgoal interactions. We isolate components of the agent harness, varying prompting strategy, context management, and model, to study how each affects performance across subgoal interaction classes. We first measure task completion across interaction classes and various harness configurations. Based on the best performing harness configuration, we then analyze how subgoal interaction class affects plan efficiency (suboptimality) as we vary the environment's step budget. This becomes necessary as task completion is not a well-defined signal in an ergodic environment. An agent with a large enough environment step budget can random-walk its way to the goal.

5.1 TASK COMPLETION ACROSS SUBGOAL INTERACTIONS

	Claude Sonnet 5		GPT-5.2		Qwen3.6		GLM-5.2		Gemma 4	
Scaffold	CS	FH	CS	FH	CS	FH	CS	FH	CS	FH
No interaction										
Native	100	44	68	26	60	78	56	100	2	100
Hybrid	100	56	46	0	84	80	64	100	62	98
Prompted	100	2	98	100	88	100	60	100	98	100
Positive interaction										
Native	100	2	32	14	44	62	16	98	0	98
Hybrid	100	6	44	0	54	76	12	100	80	94
Prompted	100	0	80	100	88	100	30	100	98	100
Negative (solvable)										
Native	100	10	0	2	4	16	46	100	0	74
Hybrid	100	10	6	0	0	72	40	100	48	100
Prompted	100	0	0	58	76	100	54	100	88	98

Table 1: **Task completion rate (%) by interaction class, model, and scaffold.** CS = current-state context management; FH = full-history context management. $n = 50$ instances per cell. Environment interaction budget: $2\times$ optimal plan

Table 1 shows task completion across subgoal interaction class, model, scaffold, and context management mode. For these experiments the environment step budget is set to 2 times the optimal plan length for any instance. The table supports the following findings.

Our first finding is that Prompted ReAct is the strongest scaffold across all models, interaction classes, and context management modes. This is notable because Hybrid ReAct uses the same system prompt as Prompted ReAct, differing only in how the action is issued: Hybrid ReAct uses the provider's native tool-calling API, while Prompted ReAct emits actions as free-form text parsed by a hand-written parser. Native tool-calling, which relies entirely on the tool-calling API without embedding action information in the system prompt, performs worst. Prompted ReAct outperforms both Hybrid ReAct and Native Tool-calling which suggests that keeping the full action specification in the system prompt, where the model reasons in its native text modality, is more effective than delegating action elicitation to a structured API channel.

Our second finding is that GPT-5.2 suffers the most under negative subgoal interaction tasks, failing to solve them entirely under most scaffold and context management combinations. Claude Sonnet 5, Qwen3.6, GLM5.2, and Gemma4 all complete negative-interaction tasks with reasonable consistency, though as we show in the next section, they do so at a measurable efficiency cost.

Our third finding is that context management mode has a substantial effect on task completion, and its direction depends on the model. For Claude Sonnet 5, current-timestep context management consistently reaches 100% across all subgoal interaction classes. GPT-5.2 similarly performs better under current-timestep than full-history context management. For the open-source models,

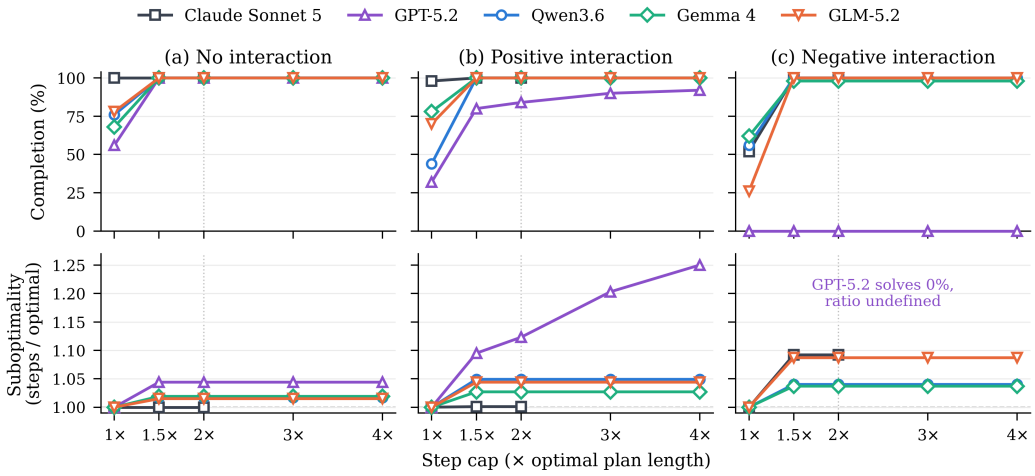


Figure 2: Plan Suboptimality and Task Completion across varying environment step budgets

Qwen3.6, GLM5.2, and Gemma4, the pattern reverses: full-history context management outperforms current-timestep by a noticeable margin across interaction classes.

5.2 Plan Suboptimality across Subgoal Interaction Classes

We next examine how subgoal interactions affect plan optimality, while varying the step budget provided to the agent. We set the step budget to 4 $\times$ the optimal budget for an task instance, and retroactively threshold the budget for analysis purposes. For each model we use the configuration that yielded the highest task completion in Table 1: current-state context management for Claude Sonnet and GPT-5.2, and Prompted ReAct with full-history context management for the open-source models. Claude Sonnet performs worse under full-history context management, so the two tables reflect each model's strongest setting rather than a matched comparison.

The suboptimality curves in Figure 2 show a split by interaction class. Under no-interaction and positive-interaction instances, suboptimality stays close to $1\times$ across every model and every step budget: once a model can solve an instance at all, it does so near-optimally, and giving it more budget does not change that. GPT-5.2 is the exception — under positive interaction, its suboptimality rises alongside its completion rate, meaning the additional instances it solves as budget increases are solved less efficiently, not just more often.

Under negative interaction, this consistency breaks down for the models that can solve these instances at all. Claude Sonnet 5 and GLM-5.2 climb to roughly 1.08–$1.09\times$ (8–9%) more steps than optimal), while Qwen3.6 and Gemma 4 stay closer to $1.05\times$ (5% more steps). GPT-5.2 has no suboptimality curve to plot in this regime: it solves 0% of negative-interaction instances at any step budget we test, so the ratio is undefined rather than simply low. Negative interaction is therefore the only condition under which plan efficiency, rather than just task completion, becomes a meaningful axis of difference between the models that succeed at all. This leaves GPT-5.2 as a qualitatively different case: where the other models trade efficiency for success, GPT-5.2 loses both at once.

Turning to completion rates directly, Claude Sonnet solves every instance regardless of interaction class, and only 52–54% of runs are exactly optimal under negative interaction tasks as shown in 2. The open-weight models follow the same pattern (Figure 2), requiring 3–6% more steps under negative interactions, with 68–80% of runs exactly optimal. GLM-5.2 also degrades mildly under positive interactions (88% optimal). These models still reach the goal, but they take actions that change simulator state without advancing the agent toward the goal.

Negative interactions therefore affect models in two distinct ways: for GPT-5.2, they impair task completion itself; for Claude Sonnet, GLM-5.2, Gemma 4, and Qwen3.6, they leave completion intact but reduce plan efficiency.

Table 2: **Closed Source Models - Plan optimality on completed runs (budget 2x), current-state prompting.** *% opt* = share of completed runs at exactly $1.00\times$ optimal plan length; n = completed runs.

		No interaction	Positive interaction	Negative (solvable)
Model	Scaffold	n / %opt	n / %opt	n / %opt
Claude Sonnet 5	Native	50 / 100	50 / 100	50 / 52
	Hybrid	50 / 100	50 / 100	50 / 54
	Prompted	50 / 100	50 / 100	50 / 52
GPT-5.2	Native	34 / 68	16 / 44	0 / –
	Hybrid	23 / 100	22 / 77	3 / 0
	Prompted	49 / 100	40 / 72	0 / –

6 Conclusion

In this study, we designed and proposed a new benchmark, PlanBench-Gym, which extends PlanBench into an interactive benchmark that allows LLMs to make tool calls against a PDDL-based simulator. We use the AI planning literature's taxonomy of subgoal interactions to generate task instances along three classes: no-interaction, positive-interaction, and negative-interaction. The generated Blocksworld instances are ergodic under this taxonomy, therefore negative subgoal interactions increase cost without ever foreclosing the goal. As shown in the study models struggle with negative subgoal interactions in different ways. Where GPT5.2 fails to complete tasks containing negative subgoal interactions, Claude Sonnet solves all tasks but struggles to solve tasks within the optimal plan budget. The open source models Qwen3.6, Gemma4, GLM5.2, exibhit the same failure mode as Claude Sonnet 5: all three complete negative-interaction tasks but require additional steps beyond the optimal plan, wasting tool-calls in the process.

7 Future Work

Several questions raised by this study remain open. Our evaluation studies subgoal interactions entirely within ergodic environments: Blocksworld instances where negatively interacting subgoals increase cost without ever foreclosing the goal. The natural next step is to assess agentic planning in non-ergodic environments, where subgoal interactions can foreclose the goal entirely. Our evaluation is also limited to the Blocksworld domain; since PlanBench-Gym accepts any PDDL domain and problem file, extending the evaluation to richer domains such as Logistics requires only generating the appropriate instance corpora, and would confirm that our understanding of how subgoal interactions degrade agentic planning holds across a broader range of environments.

REFERENCES

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Atharva Gundawar, Mudit Verma, Lin Guan, and Subbarao Kambhampati. Robust planning with compound llm architectures: An llm-modulo approach. *arXiv preprint arXiv:2408.09510*, 2024.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv preprint arXiv:2402.01817*, 2024.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities, 2024.

XuanLong Nguyen, Subbarao Kambhampati, and Romeo S. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and csp search. *Artificial Intelligence*, 135 (1-2):73–123, 2002.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, 2024. arXiv:2407.18901.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems*, 36:38975–38987, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.12045.

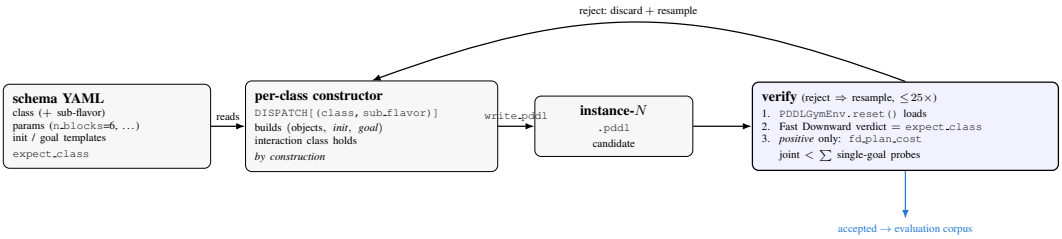


Figure 3: **Offline corpus construction.** Each interaction class is a schema; a per-class constructor builds an instance whose structure makes the class hold *by construction*. Every candidate passes three checks or is discarded and resampled: it must load in `PDDLGymEnv`, Fast Downward's solvability verdict must match the schema's `expect_class`, and for positive-interaction instances the joint optimal plan cost must be strictly below the sum of the two single-goal probe costs.

A Appendix

A.1 Offline Corpus Construction Pipeline

The figure below details the offline pipeline used to construct and verify each task instance before it is admitted to the evaluation corpus described in Section 4.3. Each interaction class is specified by a schema; a per-class constructor builds candidate instances whose structure makes the class hold by construction. Every candidate passes three checks before being admitted: it must load successfully in `PDDLGymEnv`, Fast Downward's solvability verdict must match the schema's expected class, and for positive-interaction instances the joint optimal plan cost must be strictly below the sum of the two single-goal probe costs.

A.2 PROMPTS

Native tool-calling.

```
You are an agent solving a STRIPS planning problem by calling grounded
    actions as tools. You are told the current state and the goal. Call
    exactly one tool per turn to make progress toward the goal. If a
    tool call is rejected, read why and try a different action -- the
    world state does not change on a rejected call. Stop calling tools
    once the goal is achieved, or if you become convinced the goal is
    unreachable.
```

Hybrid ReAct.

```
You are an agent solving a STRIPS planning problem by calling grounded
    actions as tools. You are told the current state and the goal.
    Before each tool call, first reason in a few sentences of plain text
     about which action makes progress toward the goal given the current
     state and anything that was rejected before, then call exactly one
    tool to take that action. If a tool call is rejected, read why and
    try a different action -- the world state does not change on a
    rejected call. Stop calling tools once the goal is achieved, or if
    you become convinced the goal is unreachable.
```

Prompted ReAct. {tools_json}, {objects} and {legend} are filled in per instance.

```
You are an agent solving a STRIPS planning problem. You are told the
    current state and the goal. Here are the actions available to you,
    as a JSON list (each action's parameters must be filled in using
    only the objects listed below):
{tools_json}

Available objects: {objects}
{legend}

At every turn, respond in exactly this format, with nothing before
    Thought: or after the JSON action:
Thought:
<a few sentences of plain text reasoning about which action makes
    progress toward the goal given the current state and anything that
    was rejected before>
Action:
{name: <the action name>, arguments: {<param>: <value>, ...}}

The Action must be valid JSON using exactly one of the action names
    above and its exact parameter names. Do not wrap the JSON in a code
    fence. If a tool call is rejected, read why and try a different
    action in your next turn -- the world state does not change on a
    rejected call. If you become convinced the goal is unreachable,
    respond with Action:
{name: give-up, arguments: {}}
instead of a real action.
```

A.3 Ablation over increasing subgoal count

To understand the relationship between the subgoal count and the interaction class, we sweep the number of subgoals $k \in \{6, 7, 8, 9, 10\}$ under current-state prompting, pooling over all three agent scaffolds.

Table 3: **current-state task completion vs. number of subgoals.** Completion rate (%) pooled over all three scaffolds. Each (k, class) cell is $n = 30$ runs.

k	No interaction	Positive interaction	Negative (solvable)	Overall
Claude Sonnet				
6	100	100	100	100.0
7	100	100	100	100.0
8	100	100	100	100.0
9	100	100	100	100.0
10	100	100	100	100.0
all k	100	100	100	100.0
GPT-5.2				
6	60	33	3	32.2
7	70	43	3	38.9
8	70	30	3	34.4
9	77	77	0	51.1
10	77	77	0	51.1
all k	71	52	2	41.6

Claude Sonnet solves every instance regardless of k. GPT-5.2's completion rate is driven by interaction class rather than subgoal count: performance is comparable for no-interaction and positive-interaction instances across the full range of k, while negative (solvable) completion remains near zero from $k = 6$ through $k = 10$. The effect tracks interaction class, not problem size.

AI USE STATEMENT

In this work, we used generative AI tools for writing assistance specifically:

- Proofreading and Grammar
- Formatting
- Improving the quality of the prose

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.