INTEGRATING TOOL-GUIDED PLANNING SUBAGENTS INTO AGENT HARNESSES AS A TOOL

Anonymous authors
Paper under double-blind review

ABSTRACT

While Language Model (LM) agents show remarkable performance on tasks requiring multiple interactions with the environment, their imperfect textual reasoning can produce plans that fail to satisfy task requirements. When environment interactions cannot be freely undone, acting on such plans can lead to irrecoverable failure under limited execution budgets. To mitigate this problem, we propose **T**ool-guided **P**lanning **S**ubagents (**TPS**), a plug-and-play planning tool for LM agents' harnesses. Specifically, TPS lets an LM agent delegate planning to subagents through a tool call when needed. It deliberately spends additional internal inference before acting to support environment-action-efficient execution. These subagents use an external solver to select a feasible plan. To provide the solver with an appropriate input, the subagents generate diverse candidate plans, simulate their consequences, and merge shared estimated states into an estimated plan graph. The solver then selects a path satisfying the remaining environment constraints within this graph. The selected plan guides the agent's next actions within its original interaction loop. Under the same environment-action budgets, TPS improves success rates by 38.7 and 25.2 percentage points on average across game and text-world benchmarks, respectively, and improves coding benchmark success by 6.2 points on average. In Meta-Harness under the same evaluation budget, TPS reaches a 3.5-point higher best score.

1 INTRODUCTION

Language Model (LM) agents increasingly solve tasks through interactions with external environments, using tools to run code or operate computer interfaces (Xi et al., 2025; Yang et al., 2024; Xie et al., 2024b). These interactions cannot always be undone: actions may irreversibly change the environment, and even reversible actions consume execution time or action budget that cannot be recovered (Zheng et al., 2024; Lin et al., 2026). A harness prepares LM inputs, executes proposed actions, and returns observations for subsequent reasoning (Anthropic, 2026b). For example, ReAct interleaves LM reasoning and actions using feedback from the environment (Yao et al., 2023b).

LM agents use textual reasoning to generate plans and predict action outcomes (Yao et al., 2023b; Hao et al., 2023; Liu et al., 2023b). Imperfect reasoning can produce plans that fail to satisfy the task requirements (Valmeekam et al., 2023). Even when an agent generates a suitable candidate, selecting a plan that satisfies the remaining constraints requires reasoning across predicted states and actions. An incorrect choice can therefore make failure *irrecoverable*: no subsequent action sequence can complete the task within the remaining execution budget, even if the agent recognizes the mistake (Wang et al., 2026b). For example, in Figure 1, an agent holding a dirty apple is asked to put a clean apple in the fridge. Putting it directly in the fridge satisfies the placement condition, but the goal can no longer be achieved within the task step budget.

To reduce such failures, we propose **T**ool-guided **P**lanning **S**ubagents (**TPS**), a plug-and-play planning tool for LM agents' harnesses. TPS uses additional internal inference before acting to support environment-action-efficient execution, helping the agent reach the goal within its remaining action budget. Specifically, the agent invokes the TPS subagents through a tool call when it determines that planning is needed. These subagents use an external solver, such as linear programming (LP) (Schrijver, 1998) and resource-constrained shortest-path search (RCSP) (Desrochers & Soumis, 1988; Beasley & Christofides, 1989), to select a feasible plan under task constraints. The LM for TPS

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.

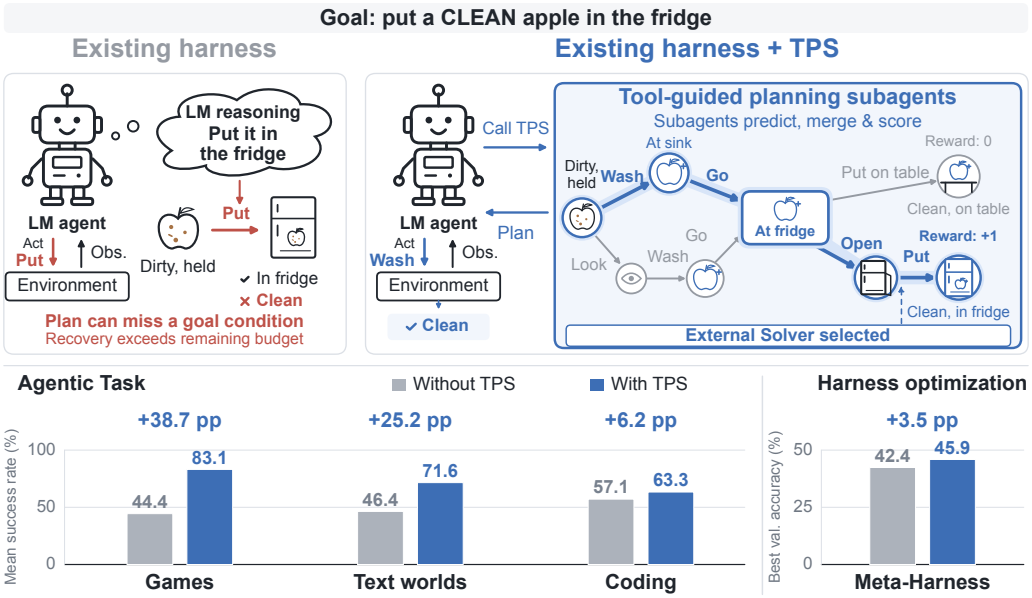


Figure 1: **Overview of TPS. Top:** TPS selects a plan that meets both cleaning and placement requirements within the action budget, while the existing harness fails. **Bottom:** TPS improves mean task success across domains and best mean accuracy in harness optimization.

predicts candidate futures, and the solver handles constrained path selection over those predictions. To provide the solver with an appropriate input, the subagents generate diverse candidate plans, simulate their consequences, estimate action costs, and merge the plans through shared estimated states to form a plan graph. The solver then selects a path on this graph to an estimated goal state under the represented task constraints and remaining budget. The agent uses the selected plan as guidance within its original interaction loop. In Figure 1, the selected plan guides the agent to wash the apple before storing it, satisfying both cleaning and placement requirements within the available action budget.

We evaluate TPS on eight benchmarks spanning game, text-world, and coding tasks. Figure 1 summarizes the success gains under the same environment-execution limits as the base harnesses. Sokoban analyses indicate that TPS uses fewer environment actions and more often preserves enough budget to finish, while the external solver finds feasible paths more reliably than the LM on identical estimated graphs. In Meta-Harness (Lee et al., 2026a), TPS also reaches a higher best mean accuracy under the same evaluation budget.

Our contributions are as follows:

- We propose TPS, a planning tool that lets LM agents invoke planning subagents within their existing harnesses. When TPS is called, the subagents construct a plan graph, and an external solver selects a feasible path to reduce the risk of irrecoverable failure.
- Across eight benchmarks spanning game, text-world, and coding tasks, TPS improves observed success in multiple harnesses. It also reaches a higher best mean accuracy in Meta-Harness under the same evaluation budget.
- Our Sokoban analyses show that TPS uses fewer environment actions and more often preserves enough budget to finish. On identical estimated graphs, the external solver finds feasible paths more reliably than the LM.

2 Related Work

Planning in Language Model Agents. LM agents solve tasks by interacting with external environments through tools (Yao et al., 2023b; Schick et al., 2023). The harness is the software that manages LM agents' inputs, tool execution, and feedback from the environment (Anthropic, 2026b). The representative harness is ReAct (Yao et al., 2023b) that interleaves reasoning and acting, using feedback from the environment to decide the next action. Recently, Recursive Language Models

(RLMs) (Zhang et al., 2025a) let an LM interact with its input context through a code execution environment. The LM can inspect the context and recursively invoke other LM calls for intermediate computation.

Planning within an agent's interaction loop involves reasoning about future actions and their consequences for reaching the goal (Yao et al., 2023b; Hao et al., 2023). To strengthen planning within the loop, several methods search over candidate thoughts and actions through tree-structured reasoning (Yao et al., 2023a) or imagined rollouts and heuristic search (Hao et al., 2023; Liu et al., 2023b; Zhou et al., 2024; Zhuang et al., 2024). Recent work further uses explicit lookahead and reward estimation (Wang et al., 2026b) or adaptive lookahead with learned world models (Liu et al., 2026c) to guide action selection. A complementary line decouples planning from acting by generating a high-level plan before execution (Erdogan et al., 2025; Xu et al., 2023) or by introducing dedicated subagents that an executor agent invokes (Asawa et al., 2026; Dasgupta et al., 2023). Other methods combine information from parallel trajectories (Lee et al., 2026b) or use a procedural graph to generate guidance for the agent's next actions (Lu et al., 2026).

In contrast, our framework exposes planning as a tool that aggregates multiple candidate plans into a graph through shared estimated states. An external solver selects a feasible path under the constraints represented in the graph. Because the tool is invoked through the agent's standard tool-use interface, it plugs into existing harnesses without modifying their reasoning loop. Additional comparisons of candidate verification and plan aggregation are described in Appendix A.

Planning with External Solvers. Several planning methods use LMs to express a task and its constraints in a form that an external solver can use to find a plan (Liu et al., 2023a; Hao et al., 2025a). LMs can act as translators that map natural-language task descriptions into formal planning inputs, such as the Planning Domain Definition Language (PDDL) (Liu et al., 2023a; Guo et al., 2025) or Satisfiability Modulo Theories (SMT) (Hao et al., 2025a;b). Recent work couples information extraction with planner-guided repair (Hua et al., 2026) or uses a PDDL-derived action graph to identify task-relevant actions and objects (Gupta et al., 2026). A complementary line lets the LM emit search components that a classical search procedure consumes (Katz et al., 2024). Structured representations also support task allocation through dependency graphs and linear programming (Obata et al., 2024), or plan verification and repair through graph world models (Wang et al., 2026a).

In contrast, our framework uses a plan graph constructed from the LM's predicted action outcomes. The LM samples concrete candidate plans in the original action language and simulates their consequences as estimated states. An external solver operates on the resulting graph's optimization structure, using predicted rewards and costs to select a feasible path under the remaining budget. This lets our framework use external solvers without a predefined formal model of the environment. Feasibility is checked against the estimated states and costs recorded in the graph.

3 TPS: Tool-guided Planning Subagents

In this section, we introduce Tool-guided Planning Subagents (TPS), a plug-and-play planning tool for LM agents' harnesses. Figure 2 shows the main LM agent calling TPS subagents through a tool call. The example requires fixing an import error and passing tests under a 90-second budget. The main agent uses its harness prompt, while the four planning subagents use the same LM under stage-specific prompts. The subagents construct a plan graph, and an external solver selects a feasible path. The main agent uses the selected plan to guide its actions through tools.

The main agent invokes TPS when it determines that planning is needed. We define the main agent's state as its interaction history h, including actions and observations up to the current step. The call provides h, the task goal g, and remaining budget b. When the task has a budget, b is a vector with one entry per constrained quantity, such as execution time or the number of actions. For action-limited tasks, we denote the total environment-action budget by B_{env}; its remaining value is passed to the planner through b. We denote internal LM and planning computation by C_{infer} and account for it separately from environment actions. TPS uses additional C_{infer} to select actions that preserve feasibility within B_{env}. Algorithm 1 in Appendix B.1 gives the full procedure.

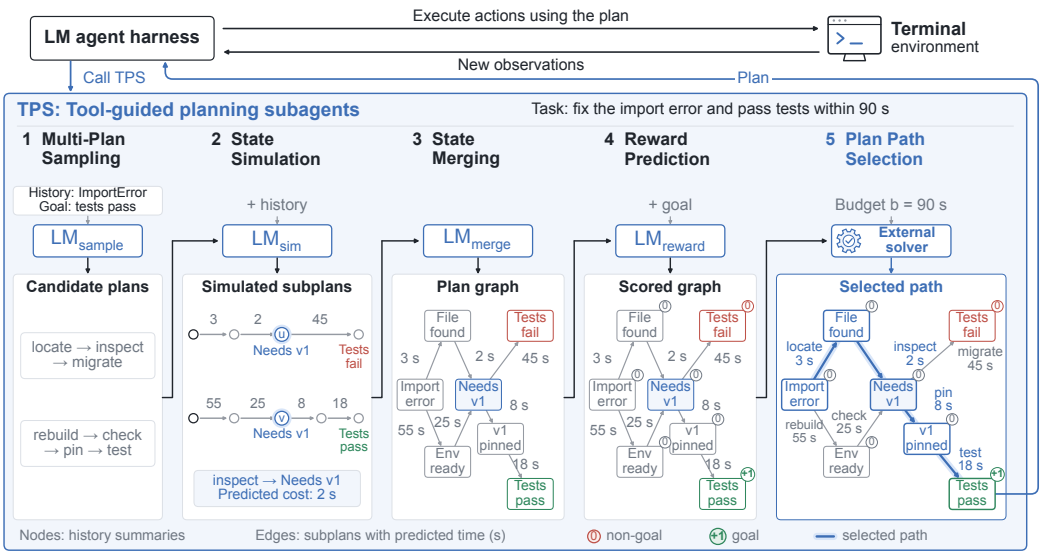


Figure 2: **TPS procedure.** For software repair under a time budget, the agent calls planning subagents to construct and score a plan graph. An external solver selects the blue path, whose subplans guide the main agent's actions.

3.1 Plan Graph Construction

The plan graph represents possible continuations of the interaction history under candidate plans. Let $\mathcal{G} = (V, E)$ denote the directed graph, where each node $v \in V$ stores a history summary and each edge $e \in E$ represents a subplan. A summary records observed or predicted conditions relevant to subsequent subplans and goal satisfaction. These summaries provide the state representations used for simulation and merging. The subscripts on LM identify the prompt used at each stage below.

Multi-Plan Sampling. Given the current history h and goal g, the first subagent generates M diverse candidate plans, denoted by $P_1, \ldots, P_M$:

$$(P_1, \ldots, P_M) \leftarrow \mathrm{LM}_{\mathrm{sample}}(h, g, M).$$

The sampling prompt follows the ReAct format (Yao et al., 2023b).

A candidate plan $P = (p_1, \ldots, p_L)$ is a sequence of L subplans. Each subplan p_i describes a step the main agent can carry out through one or more concrete actions. For example, "Needs v1" in Figure 2 means that a dependency requires version 1. The "pin" subplan specifies updating that dependency to the required version. The main agent determines the commands and code changes during execution.

State Simulation. The second subagent predicts the outcomes and costs along each candidate plan P_m. Its output T_m starts with a summary of h and records the updated summary and cost after each subplan:

$$T_m \leftarrow \mathrm{LM}_{\mathrm{sim}}(h, P_m), \qquad m = 1, \ldots, M.$$

Each corresponding graph edge stores a nonnegative cost vector $c(e)$, with entries matching the constraints in b. Depending on the task, these entries represent resource use or an LM-assigned action penalty, as detailed in Appendix B.2.

State Merging. The third subagent takes the simulated plans and merges history summaries that it judges equivalent for subsequent planning:

$$\mathcal{G} \leftarrow \mathrm{LM}_{\mathrm{merge}}(T_1, \ldots, T_M).$$

Different histories can therefore share a node when these task-relevant conditions agree. Each subplan becomes an edge between its preceding and resulting history summaries, retaining its predicted cost. In Figure 2, "Needs v1" abbreviates the predicted conditions shared by two candidates.

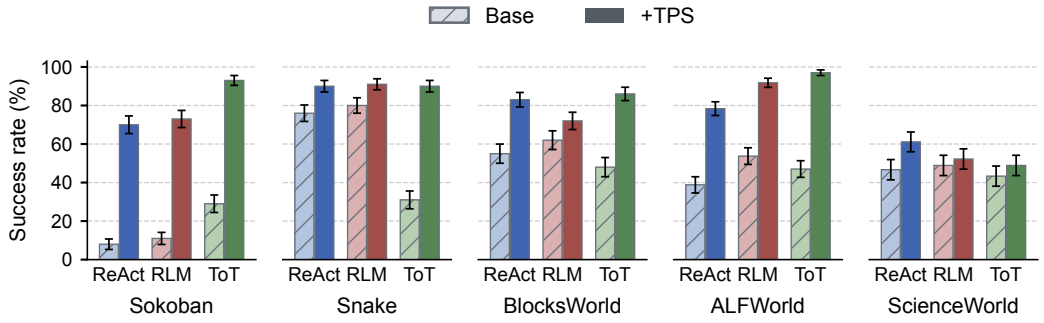


Figure 3: **Success rates across game and text-world tasks.** Success increases in every evaluated benchmark–harness pair, with larger gains on ALFWorld than on ScienceWorld.

Reward Prediction. Given the merged graph and the goal, the fourth subagent predicts one reward $r(v)$ for every node $v \in V$:

$$\{r(v)\}_{v \in V} \leftarrow \mathrm{LM}_{\mathrm{reward}}(\mathcal{G}, g).$$

The reward is 1 when the summary indicates goal satisfaction, -1 when it indicates a dead-end, and 0 otherwise.

3.2 Constrained Path Selection

The solver selects a feasible path using the graph's predicted rewards and costs. The starting node v_0 summarizes the current history h. Let $\mathcal{P}$ denote the set of directed walks on $\mathcal{G}$ from v_0 to a node with $r(v) = 1$, within a finite planning horizon. For each walk π, let $R(\pi)$ denote the cumulative predicted reward under the solver's scoring rule, specified in Appendix B.2. The selected path $\pi^\star$ is obtained by solving

$$\pi^\star \in \arg\max_{\pi \in \mathcal{P}} R(\pi) \quad \text{s.t.} \quad \sum_{e \in \pi} c(e) \preceq b, \tag{1}$$

where $\preceq$ denotes element-wise inequality and the budget constraint is omitted when the task has no budget. The solver enforces feasibility within the estimated graph using its predicted states, rewards, and costs. Whether the selected path succeeds in the environment also depends on the accuracy of those predictions and the agent's execution.

We can use Integer Linear Programming (ILP) (Schrijver, 1998). Other solvers can also search the same graph under its budget constraints, including resource-constrained shortest-path search (RCSP) (Desrochers & Soumis, 1988; Beasley & Christofides, 1989), constrained Monte Carlo Tree Search (MCTS) (Lee et al., 2018), and so on. Solver details are given in Appendix B.2.

4 Experimental Results

We evaluate TPS across game, text-world, and coding tasks. For each harness, we compare **Base**, the harness on its own, against **Base** + **TPS**, which additionally exposes TPS as a planning tool, under the same benchmark-specific environment-execution limits. We report success rate and use ILP as the path-selection solver unless stated otherwise. For game and text-world tasks, we use ReAct (Yao et al., 2023b), RLM (Zhang et al., 2025a) with maximum recursion depth $r = 2$, and Tree-of-Thoughts (ToT) (Yao et al., 2023a), all with `gpt-4.1-mini` (OpenAI, 2025). For additional experimental details, please refer to Appendix C.

Game Tasks. Our game benchmarks cover sequential action planning in Sokoban (Schrader, 2018), Snake (Ouyang et al., 2026), and BlocksWorld (Slaney & Thiébaux, 2001). Figure 3 shows gains for all three harnesses across these games. On Sokoban, success increases from 8% to 70% for ReAct, 11% to 73% for RLM, and 29% to 93% for ToT. ToT also improves from 31% to 90% on Snake, while RLM improves from 62% to 72% on BlocksWorld. The mean gain across benchmark–harness pairs is 38.7 percentage points. The gains across all three harnesses indicate that TPS can support planning across different game tasks and harness designs.

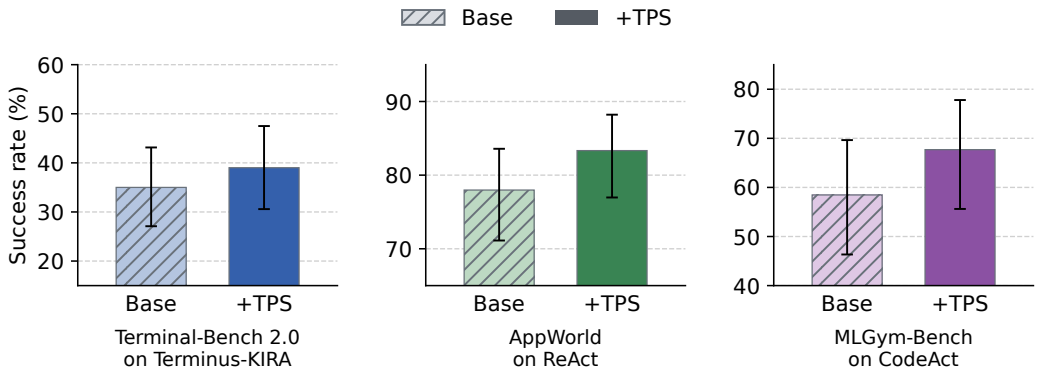


Figure 4: **Success rates on coding agentic tasks.** The evaluated shell and API harnesses achieve higher observed success with planning support.

Text-World Tasks. To assess transfer to household and science tasks through textual interactions, we run the same three harnesses on ALFWorld (Shridhar et al., 2021) and ScienceWorld (Wang et al., 2022). Figure 3 shows that TPS increases ALFWorld success from 38.8% to 78.4% for ReAct, 53.7% to 91.8% for RLM, and 47.0% to 97.0% for ToT. On ScienceWorld, the corresponding gains are 14.4, 3.3, and 5.6 percentage points. The mean gain is 25.2 percentage points. Planning can therefore also benefit text-world performance, although the size of the gain varies across tasks.

Takeaway 1. TPS can improve success across game and text-world harnesses, with the largest observed gains on Sokoban and ALFWorld.

Coding Agentic Tasks. We next assess whether TPS transfers to agents that execute shell commands, interact with application APIs, or carry out coding workflows. The benchmarks are Terminal-Bench 2.0 (Merrill et al., 2026) with Terminus-KIRA (KRAFTON AI & Ludo Robotics, 2026), AppWorld (Trivedi et al., 2024) with ReAct (Yao et al., 2023b), and MLGym-Bench (Nathani et al., 2025) with CodeAct (Wang et al., 2024). We use `claude-haiku-4.5` (Anthropic, 2025) as the backbone for all three benchmarks. For each benchmark, we compare the existing harness with and without TPS as a planning tool. Figure 4 shows higher observed success with TPS: from 35.0% to 39.0% on Terminal-Bench, 78.0% to 83.3% on AppWorld, and 58.5% to 67.7% on MLGym-Bench. For Terminal-Bench, we report task-averaged success. The mean gain across the three benchmarks is 6.2 percentage points. These gains point to a potential benefit for agents operating through shell and API tools. Appendix D.3 examines the task-level changes.

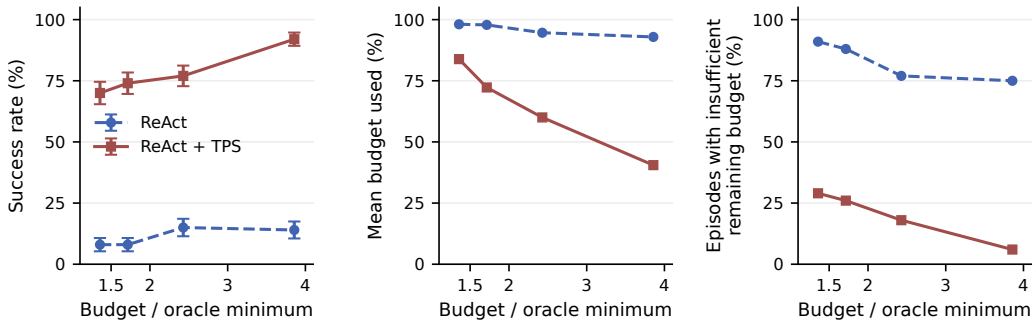
Takeaway 2. TPS may also improve task success when added to existing shell and API harnesses.

5 Analysis

We analyze environment-action use and the role of the external solver to understand how TPS improves success under limited action budgets. Sokoban provides an oracle for checking whether enough actions remain to finish, while identical-graph comparisons isolate feasible-path finding from graph prediction. The budget comparisons use ReAct and ReAct + TPS with `gpt-4.1-mini` and four candidate plans; experimental details are in Appendix E.1.

Success across Action Budgets. To assess how planning uses additional environment actions, we vary $B_{\text{env}} = T^\star + \Delta$ for $\Delta \in \{2, 4, 8, 16\}$, where $T^\star$ is the oracle minimum. The horizontal axis reports the mean ratio $B_{\text{env}}/T^\star$ across episodes. Figure 5a shows TPS success rising from 70% to 92% across this range, while ReAct remains between 8% and 15%. Planning helps the agent benefit from additional actions; a larger action allowance alone does not close the gap.

Action Consumption. We measure the fraction of the action budget consumed, averaged over all episodes including failures. Figure 5b reports lower budget consumption with TPS across the



(a) Success across budgets. (b) Action consumption. (c) Insufficient remaining budget.

Figure 5: **Success and action use across Sokoban budgets.** Across budgets relative to an optimal solution, TPS maintains higher success, consumes a smaller share of the budget, and less often leaves too few actions to finish.

Table 1: **Ablation of planning components on Sokoban.** TPS outperforms candidate selection and graph-based LM selection.

Variant	Planning subagents	Multiple plans	Plan graph	External solver	Success (%)
ReAct					11 ± 3.1
Single Plan	✓				13 ± 3.4
Best-of-N	✓	✓			20 ± 4.0
Graph + LM	✓	✓	✓		29 ± 4.5
TPS	✓	✓	✓	✓	$\mathbf{37 \pm 4.8}$

Table 2: **Feasible-path finding on identical graphs.** ILP finds available paths more reliably than the LM.

Graph nodes	LM (%)	ILP (%)
≤ 10	95.2	100.0
11–15	97.7	100.0
16–20	93.9	100.0
≥ 21	84.6	100.0

evaluated budgets; at $\Delta = 16$, it uses 40.5% on average, compared with 92.9% for ReAct. Higher success with fewer actions indicates a benefit in environment-action efficiency.

Insufficient Remaining Budget. The oracle identifies the first action after which a reachable goal requires more actions than remain. For example, three actions are needed but only two are available. Physical deadlocks, where the goal becomes permanently unreachable, are counted separately. In Figure 5c, TPS reduces the affected episodes from 91 to 29 at $\Delta = 2$ and from 75 to 6 at $\Delta = 16$, out of 100 per configuration. Thus, TPS more often preserves a way to finish within the remaining action allowance. The oracle-based calculation is detailed in Appendix E.3.

Takeaway 3. On Sokoban, TPS can improve success while using fewer environment actions and less often leaving too little budget to finish.

Finding a Feasible Plan on the Same Graph. To isolate the role of path selection, `gpt-4.1-mini` and ILP receive identical recorded nodes, edges, rewards, costs, and remaining budgets. We group the graphs by node count and measure the fraction for which each method finds a feasible path, conditioning on such a path being present. In Table 2, the LM finds feasible paths on 84.6%–97.7% of graphs across size groups, while ILP succeeds in every group. For graphs with at least 21 nodes, the LM returns a feasible path in 22 of 26 cases; ILP returns one in all 26. Even with identical graph information, the LM can miss an available feasible path. The comparison supports using an external solver to find paths satisfying the estimated graph constraints across graph sizes. Appendix E.4 describes the evaluation protocol; Appendix E.5 compares the external solvers.

Ablation Study. Beyond graph-level feasibility, the component ablation tests whether the full procedure improves task success over candidate generation and selection. Using Qwen3-14B as both the executor and planning model, we compare a single plan, Best-of-N (Wang et al., 2023a) selection among four plans, graph-based LM selection, and TPS. Table 1 shows success increasing from 20% with Best-of-N to 29% with Graph + LM and 37% with TPS. In this setting, organizing predicted futures into a graph and applying a solver can yield more useful guidance than choosing among candidate plans alone. For the additional 8B results and configuration details, see Appendix E.6.

Table 3: **Task success and inference cost.** A smaller model with TPS can match a larger advisor's success at lower inference cost.

Base	Planning agent	PFLOPs / episode	Success (%)
4B	4B (TPS)	1.25	23 ± 4.2
14B	32B (Best-of-4)	2.42	22 ± 4.1
14B	32B (single plan)	1.21	18 ± 3.8

Table 4: **Comparison with a larger planning advisor.** The benefit of TPS varies by base model.

Planning agent	8B base Success (%)	14B base Success (%)
32B (single plan)	18 ± 3.8	18 ± 3.8
32B (Best-of-4)	$\mathbf{19 \pm 3.9}$	22 ± 4.1
TPS	16 ± 3.7	$\mathbf{37 \pm 4.8}$

Takeaway 4. On the same estimated graphs, ILP finds feasible paths more reliably than the LM across the evaluated graph sizes.

6 Practical Considerations

We analyze graph accuracy and inference cost to assess the reliability and computational trade-offs of using TPS in practice. We check graph predictions against the real environment and compare TPS with guidance from a larger planning advisor.

Graph Accuracy. We evaluate graph accuracy on Sokoban and ALFWorld through state comparisons and environment replay. On Sokoban, canonical comparison of predicted states gives 99.7% merge precision, and local transition accuracy is 97.1%. On ALFWorld, replay gives 92.4% merge precision, 60.2% goal-label precision, and 78.9% action applicability. High merge precision alone thus does not ensure accurate simulation or goal prediction. Appendix F.1 details the measurements.

Takeaway 5. The reliability of recommended plans depends on how accurately the graph represents the environment.

Inference Cost. As an alternative use of inference compute, we consider guidance from a larger planning advisor (Asawa et al., 2026; Anthropic, 2026a) alongside TPS. The 32B advisor generates either one plan or four plans with LM-based selection (Best-of-4) (Wang et al., 2023a). All models are from the Qwen3 family (Yang et al., 2025), and the accounting includes base-agent, planning-subagent, and advisor calls.

Table 3 compares configurations with similar success or similar estimated inference cost. With TPS, the 4B model achieves 23% success at 1.25 PFLOPs per episode, compared with 22% at 2.42 PFLOPs for a 14B base using the 32B advisor with Best-of-4. The single-plan advisor achieves 18% success at 1.21 PFLOPs with the same 14B base. At fixed base-model sizes (Table 4), TPS achieves 37% success at 14B, compared with 18% and 22% for the two advisor configurations. At 8B, the advisors achieve 18% and 19%, compared with 16% for TPS. TPS can thus offer favorable compute–success trade-offs with a smaller model, although its relative benefit over a larger advisor depends on the base model. Appendix F.3 gives the full compute–success frontier and explains the estimation protocol.

Takeaway 6. TPS can make a smaller model competitive with a larger planning advisor in both task success and inference cost.

7 TPS for Harness Optimization

We also apply TPS to an agent that optimizes another agent's harness. The goal is to improve a fixed task model's performance by revising the code around it. Meta-Harness (Lee et al., 2026a) is a framework that automates this process with an LM proposer. The proposer reads the current harness code, execution traces, and evaluation feedback, then writes a revised candidate harness. The task model runs with the candidate harness on validation examples, and the results guide subsequent revisions. Under a limited evaluation budget, the proposer must anticipate which revisions will improve task performance.

Harness Optimization Setup. We follow the main experimental setup of Meta-Harness (Lee et al., 2026a), with a Claude Sonnet 5 (Anthropic, 2026c) proposer and a fixed `gpt-oss-20b` (Agarwal et al., 2025) task model. The proposer modifies the harness code that stores and retrieves training examples and feedback, constructs prompts, and processes predictions. The task model uses each candidate harness to predict reactants on USPTO, diseases from symptoms on Symptom2Disease, and charges from case descriptions on LawBench. Each candidate is evaluated with three seeds per dataset. We aggregate seed results within each dataset and average accuracy equally across datasets. Base Meta-Harness uses the same proposer without TPS. Both configurations allow 12 iterations with three candidate evaluations per iteration.

Applying TPS to the Proposer. The proposer calls TPS to plan harness revisions. We treat a proposed revision as an action and the resulting candidate harness as a state. Each node represents a candidate harness, and an edge records a revision of its parent. Planning subagents examine training examples to predict how a revision would change the harness's behavior. They can then extend that candidate with further revisions. The solver selects a subset of candidates to implement and evaluate under the per-iteration budget. The resulting validation scores inform subsequent proposals. Appendix G describes how TPS is integrated into the proposer.

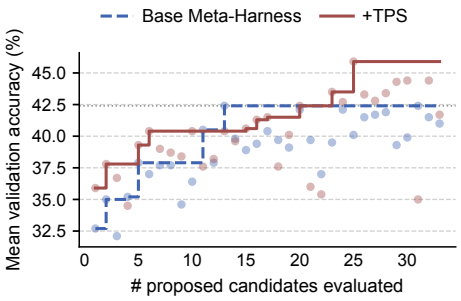


Figure 6: **Harness optimization in Meta-Harness.** TPS reaches a higher final best mean validation accuracy under the same evaluation budget. Lines track the best score so far; translucent points show individual candidate scores.

Results. Under a shared evaluation budget, the comparison tracks the best mean validation accuracy reached by Base Meta-Harness and Meta-Harness + TPS. Figure 6 shows the running best of this score, with translucent points marking individual candidate scores. Base reaches 42.4% and then plateaus, while TPS continues to improve and reaches 45.9%. The curves cross during optimization; the gain is in the final best mean score within the evaluation budget. The final score provides evidence that TPS can guide harness revisions toward better validation performance within a limited number of evaluations.

Takeaway 7. TPS can also guide harness optimization, reaching a higher mean validation accuracy under the same evaluation budget.

8 Conclusion

LM agents can generate plausible plans yet take actions that leave no way to complete a task within the remaining environment budget. We proposed **TPS** to reduce this risk by spending additional internal inference before acting: LM subagents generate and simulate candidate plans, and an external solver selects a path satisfying the constraints in the resulting estimated graph. Experiments across game, text-world, and coding domains show that adding TPS improves observed success in diverse existing harnesses under the same environment-execution limits. Our Sokoban analyses further show that TPS more often preserves enough actions to finish and that, on identical estimated graphs, the external solver finds feasible paths more reliably than the LM.

Future Work & Limitations. Future work could add preferred completion times alongside strict execution limits, allowing the planner to spend more time for higher expected solution quality within those limits. In addition, TPS uses a solver specified in advance. Our experiments show differences in solution quality and runtime across solvers and graph sizes. Automatically selecting a solver based on the task formulation and graph structure is another direction for future work. Further considerations are discussed in Appendix I.

AI USE STATEMENT

Generative AI tools assisted with grammar correction, code refactoring and debugging, LaTeX preparation, literature searches for related work, and offline analyses. The authors are responsible for the final manuscript, analyses, and research artifacts.

ETHICS STATEMENT

Planning assistance can make agents more effective at executing shell commands and application actions, including actions whose consequences extend beyond a benchmark. Feasibility on an LM-predicted graph is not a guarantee of correctness or safety in the real environment. Deployment therefore requires task-appropriate permissions and validation of consequential actions. The additional language-model calls also increase computational and energy costs.

REPRODUCIBILITY STATEMENT

The code is available at https://anonymous.4open.science/r/TPS-ICLR-2027-F210. Appendices B and C describe the method and evaluation setup, and Appendix H collects the harness and planning prompts.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Anthropic. Introducing Claude Haiku 4.5. https://www.anthropic.com/news/claude-haiku-4-5, 2025.

Anthropic. The advisor strategy: Give agents an intelligence boost. https://claude.com/blog/the-advisor-strategy, 2026a.

Anthropic. Demystifying evals for AI agents. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents, 2026b.

Anthropic. Introducing Claude Sonnet 5. https://www.anthropic.com/news/claude-sonnet-5, 2026c.

Parth Asawa, Alan Zhu, Abigail O'Neill, Matei Zaharia, Alex Dimakis, and Joseph E. Gonzalez. How to train your advisor: Steering black-box LLMs with advisor models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026.

John E Beasley and Nicos Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 1989.

Junjie Chen, Haitao Li, Jingli Yang, Yiqun Liu, and Qingyao Ai. Enhancing LLM-based agents via global planning and hierarchical execution. *arXiv preprint arXiv:2504.16563*, 2025.

Zewen Chi, Li Dong, Qingxiu Dong, Yaru Hao, Xun Wu, Shaohan Huang, and Furu Wei. The era of agentic organization: Learning to organize with Language Models. *arXiv preprint arXiv:2510.26658*, 2025.

Soyeon Choi, Kangwook Lee, Oliver Sng, and Joshua M Ackerman. Infected Smallville: How disease threat shapes sociality in LLM agents. *arXiv preprint arXiv:2506.13783*, 2025.

Sanjiban Choudhury. Process reward models for LLM agents: Practical framework and directions. *arXiv preprint arXiv:2502.10325*, 2025.

Ishita Dasgupta, Christine Kaeser-Chen, Kenneth Marino, Arun Ahuja, Sheila Babayan, Felix Hill, and Rob Fergus. Collaborating with Language Models for embodied reasoning. *arXiv preprint arXiv:2302.00763*, 2023.

Martin Desrochers and François Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR: Information Systems and Operational Research*, 1988.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-Act: Improving planning of agents for long-horizon tasks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.

Zeyu Gao, Yao Mu, Jinye Qu, Mengkang Hu, Shijia Peng, Chengkai Hou, Lingyue Guo, Ping Luo, Shanghang Zhang, and Yanfeng Lu. DAG-Plan: Generating directed acyclic dependency graphs for dual-arm cooperative planning. *arXiv preprint arXiv:2406.09953*, 2024.

Weihang Guo, Zachary Kingston, and Lydia E Kavraki. CaStL: Constraints as specifications through LLM translation for long-horizon task and motion planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2025.

Piyush Gupta, Sangjae Bae, Jiachen Li, and David Isele. Scale-Plan: Scalable language-enabled task planning for heterogeneous multi-robot teams. *arXiv preprint arXiv:2603.08814*, 2026.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with Language Model is planning with world model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. Large Language Models can solve real-world planning rigorously with formal verification tools. In *Proceedings of the Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2025a.

Yilun Hao, Yang Zhang, and Chuchu Fan. Planning anything with rigor: General-purpose zero-shot planning with LLM-based formalized programming. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025b.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Yafei Hu, Quanting Xie, Vidhi Jain, Jonathan Francis, Jay Patrikar, Nikhil Keetha, Seungchan Kim, Yaqi Xie, Tianyi Zhang, Hao-Shu Fang, et al. Toward general-purpose robots via foundation models: A survey and meta-analysis. *arXiv preprint arXiv:2312.08782*, 2023.

Keru Hua, Ding Wang, Yaoying Gu, and Xiaoguang Ma. DUPLEX: Agentic dual-system planning via LLM-driven information extraction. *arXiv preprint arXiv:2603.23909*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can Language Models resolve real-world github issues? In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Michael Katz, Harsha Kokel, Kavitha Srinivas, and Shirin Sohrabi. Thought of Search: Planning with Language Models through the lens of efficiency. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Jeonghye Kim, Sojeong Rhee, Minbeom Kim, Dohyung Kim, Sangmook Lee, Youngchul Sung, and Kyomin Jung. ReflAct: World-grounded decision making in LLM agents via goal-state reflection. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Jing Yu Koh, Stephen Marcus McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *Transactions on Machine Learning Research*, 2025.

Yilun Kong, Jingqing Ruan, YiHong Chen, Bin Zhang, Tianpeng Bao, Shi Shiwei, du Guo Qing, Xiaoru Hu, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, and Xueqian Wang. TPTU-v2: Boosting task planning and tool usage of large language model-based agents in real-world industry systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024.

KRAFTON AI and Ludo Robotics. Terminus-KIRA: Boosting frontier model performance on terminal-bench with minimal harness, 2026. URL https://github.com/krafton-ai/kira.

Jongmin Lee, Geon-Hyeong Kim, Pascal Poupart, and Kee-Eung Kim. Monte-carlo tree search for constrained POMDPs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *Proceedings of the Conference on Language Modeling (COLM)*, 2026a.

Yoonsang Lee, Howard Yen, Xi Ye, and Danqi Chen. Agentic aggregation for parallel scaling of long-horizon agentic tasks. In *Proceedings of the Conference on Language Modeling (COLM)*, 2026b.

Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong Jin, and Chenyan Xiong. Benchmark test-time scaling of general llm agents. *arXiv preprint arXiv:2602.18998*, 2026.

Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Jiaju Lin, Haoran Zhao, Aochi Zhang, Yiting Wu, Huqiuyue Ping, and Qin Chen. AgentSims: An open-source sandbox for large language model evaluation. *arXiv preprint arXiv:2308.04026*, 2023.

Yuxiang Lin, Zihan Wang, Mengyang Liu, Yuxuan Shan, Longju Bai, Junyao Zhang, Xing Jin, Boshan Chen, Jinyan Su, Xingyao Wang, et al. BAGEN: Are LLM agents budget-aware? *arXiv preprint arXiv:2606.00198*, 2026.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering Large Language Models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023a.

Tengxiao Liu, Zifeng Wang, Jin Miao, I-Hung Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, Samira Daruki, Yi Liang, William Yang Wang, Tomas Pfister, and Chen-Yu Lee. Budget-aware tool use enables effective agent scaling. In *Proceedings of the Conference on Language Modeling (COLM)*, 2026a.

Xuanzhang Liu, Jianglun Feng, Zhuoran Zhuang, Junzhe Zhao, Maofei Que, Jieting Li, Dianlei Wang, Hao Tong, Ye Chen, and Pan Li. CoDA: A context-decoupled hierarchical agent with reinforcement learning. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, 2026b.

Yanming Liu, Xinyue Peng, Jiannan Cao, Shi Bo, Yuwei Zhang, Xuhong Zhang, Sheng Cheng, Xun Wang, Jianwei Yin, and Tianyu Du. Tool-Planner: Task planning with clusters across multiple tools. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

Youwei Liu, Jian Wang, Hanlin Wang, Beichen Guo, and Wenjie Li. Imagine-then-Plan: Agent learning from adaptive lookahead with world models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2026c.

Zhihan Liu, Hao Hu, Shenao Zhang, Hongyi Guo, Shuqi Ke, Boyi Liu, and Zhaoran Wang. Reason for future, act for now: A principled framework for autonomous llm agents with provable sample efficiency. *arXiv preprint arXiv:2309.17382*, 2023b.

Yuxing Lu, Yicheng Chen, Shanchan Wu, and Sercan Ö Arık. Procedural graphs: Self-evolving execution structures for llm agents. *arXiv preprint arXiv:2609.09153*, 2026.

Zhengdong Lu, Weikai Lu, Yiling Tao, Yun Dai, ZiXuan Chen, Huiping Zhuang, Cen Chen, Hao Peng, and Ziqian Zeng. Decompose, plan in parallel, and merge: A novel paradigm for large language models based planning with multiple constraints. *arXiv preprint arXiv:2506.02683*, 2025.

Yichuan Ma, Linyang Li, Yongkang Chen, Peiji Li, Xiaozhe Li, Qipeng Guo, Dahua Lin, and Kai Chen. Timely machine: Awareness of time makes test-time scaling agentic. In *Proceedings of the Association for Computational Linguistics (ACL)*, 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong (Ryan) Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Mikhail Plekhanov, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ricardo Silveira Cabral, Tatiana Shavrina, Jakob Nicolaus Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. MLGym: A new framework and benchmark for advancing AI research agents. In *Proceedings of the Conference on Language Modeling (COLM)*, 2025.

Kazuma Obata, Tatsuya Aoki, Takato Horii, Tadahiro Taniguchi, and Takayuki Nagai. Lip-LLM: Integrating linear programming and dependency graph with Large Language Models for multi-robot task planning. *IEEE Robotics and Automation Letters*, 2024.

OpenAI. GPT-4.1. https://openai.com/index/gpt-4-1/, 2025.

Mingyu Ouyang, Siyuan Hu, Kevin Qinghong Lin, Hwee Tou Ng, and Mike Zheng Shou. GameWorld: Towards standardized and verifiable evaluation of multimodal game agents. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2026.

Aaron Parisi, Yao Zhao, and Noah Fiedel. TALM: Tool augmented Language Models. *arXiv preprint arXiv:2205.12255*, 2022.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023.

Mihir Parmar, Xin Liu, Palash Goyal, Yanfei Chen, Long Le, Swaroop Mishra, Hossein Mobahi, Jindong Gu, Zifeng Wang, Hootan Nakhost, et al. PlanGen: A multi-agent framework for generating planning and reasoning trajectories for complex problem solving. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025.

Jinghua Piao, Yuwei Yan, Jun Zhang, Nian Li, Junbo Yan, Xiaochong Lan, Zhihong Lu, Zhiheng Zheng, Jing Yi Wang, Di Zhou, et al. AgentSociety: Large-scale simulation of LLM-driven generative agents advances understanding of human behaviors and society. *arXiv preprint arXiv:2502.08691*, 2025.

Raffaele Pisano and Roberto Navigli. Process reward models meet planning: Generating precise and scalable datasets for step-level rewards. In *Proceedings of the Association for Computational Linguistics (ACL)*, 2026.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-needed decomposition and planning with Language Models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, 2024.

Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent Q: Advanced reasoning and learning for autonomous ai agents. *arXiv preprint arXiv:2408.07199*, 2024.

Jingqing Ruan, YiHong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, du qing, shi shiwei, Hangyu Mao, Xingyu Zeng, and Rui Zhao. TPTU: Task planning and tool usage of large language model-based AI agents. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Max-Philipp B. Schrader. gym-sokoban. https://github.com/mpSchrader/gym-sokoban, 2018.

Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for llm reasoning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in hugging face. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Zichao Shen, Chen Gao, Jiaqi Yuan, Tianchen Zhu, Xingcheng Fu, and Qingyun Sun. SDA-PLANNER: State-dependency aware adaptive planner for embodied task planning. *arXiv preprint arXiv:2509.26375*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Simranjit Singh, Andreas Karatzas, Michael Fore, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. An LLM-Tool compiler for fused parallel function calling. *arXiv preprint arXiv:2405.17438*, 2024.

John Slaney and Sylvie Thiébaux. Blocks World revisited. *Artificial Intelligence*, 2001.

Jiwoong Sohn, Tomasz Sternal, Kenneth Styppa, Torsten Hoefler, and Michael Moor. Process reward agents for steering knowledge-intensive reasoning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. AdaPlanner: Adaptive planning from feedback with Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 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 Association for Computational Linguistics (ACL)*, 2024.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of Large Language Models-a critical investigation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Ruiyang Wang, Hao-Lun Hsu, Swarajh Mehta, Jiwoo Kim, Zhihao Dou, and Miroslav Pajic. GAVEL: Graph world models for verified and efficient long-horizon llm task planning. *arXiv preprint arXiv:2609.19315*, 2026a.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023a.

Zehong Wang, Fang Wu, Hongru Wang, Xiangru Tang, Bolian Li, Zhenfei Yin, Yijun Ma, Yiyang Li, Weixiang Sun, Xiusi Chen, et al. Why reasoning fails to plan: A planning-centric analysis of long-horizon decision making in llm agents. *arXiv preprint arXiv:2601.22311*, 2026b.

Zhilin Wang, Yu Ying Chiu, and Yu Cheung Chiu. Humanoid Agents: Platform for simulating human-like generative agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, 2023b.

Duo Wu, Jinghe Wang, Yuan Meng, Yanning Zhang, Le Sun, and Zhi Wang. CATP-LLM: Empowering Large Language Models for cost-aware tool planning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2025.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of Large Language Model based agents: A survey. *Science China Information Sciences*, 2025.

Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, et al. AgentPRM: Process reward models for LLM agents via step-wise promise and progress. In *Proceedings of the ACM Web Conference (WWW)*, 2026.

Yu Xia, Yiran Jenny Shen, Junda Wu, Tong Yu, Sungchul Kim, Ryan A Rossi, Lina Yao, and Julian McAuley. SAND: Boosting LLM agents with self-taught action deliberation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 3062–3077, 2025.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. TravelPlanner: A benchmark for real-world planning with language agents. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024a.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. ReWOO: Decoupling reasoning from observations for efficient augmented Language Models. *arXiv preprint arXiv:2305.18323*, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-Agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree Of Thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in Language Models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023b.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive Language Models. *arXiv preprint arXiv:2512.24601*, 2025a.

Cong Zhang, Xin Deik Goh, Dexun Li, Hao Zhang, and Yong Liu. Planning with multi-constraints via collaborative language agents. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, 2025b.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. ReST-MCTS*: LLM self-training via process reward guided tree search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Shiqi Zhang, Xinbei Ma, Zouying Cao, Zhuosheng Zhang, and Hai Zhao. Plan-over-Graph: Towards parallelable LLM agent schedule. *arXiv preprint arXiv:2502.14563*, 2025c.

Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. WebPilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2025d.

Yuanhang Zheng, Peng Li, Ming Yan, Ji Zhang, Fei Huang, and Yang Liu. Budget-constrained tool learning with planning. In *Findings of the Association for Computational Linguistics (ACL)*, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in Language Models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. ToolChain*: Efficient action space navigation in Large Language Models with A* search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

APPENDIX

Appendices A and B expand the related work and method, and Appendix C defines the common experimental setup. Supplementary results follow the main-paper order: benchmarks (Appendix D), Sokoban analysis (Appendix E), reliability and cost (Appendix F), and harness optimization (Appendix G). Prompt templates are collected in Appendix H, followed by further discussion in Appendix I.

A ADDITIONAL RELATED WORK

A.1 AGENTIC TASKS

Agentic tasks refer to problems in which an agent interacts with the external environment to accomplish the goal (Yao et al., 2023b). Recent advancements have demonstrated the possibility of LMs' agentic ability across diverse domains, including robotic manipulation (Hu et al., 2023), graphical user interface (GUI) navigation (He et al., 2024), and software engineering (Jimenez et al., 2024; Yang et al., 2024). In life simulations, agents typically role-play as Non-Player Characters (NPCs), exhibiting human-like behaviors (Park et al., 2023; Choi et al., 2025). LM agents interact with these environments primarily through tool calls, which has been studied as a learned tool-use interface (Schick et al., 2023; Parisi et al., 2022; Ruan et al., 2023; Kong et al., 2024), as parallel and structured tool composition (Kim et al., 2024; Singh et al., 2024; Liu et al., 2025), and as expert-routing controllers (Shen et al., 2023).

However, the impressive capabilities of these agents often rely on unrestricted resource consumption, which poses a significant barrier to practical deployment. For instance, Park et al. (2023) shows that a two-day simulation of 25 agents incurred thousands of dollars in API costs. Similarly, subsequent studies noted that the latency induced by such complex reasoning loops renders real-time interaction infeasible (Wang et al., 2023b; Lin et al., 2023; Piao et al., 2025). These examples highlight that in real-world scenarios, agents cannot be deployed with unbounded resources.

Instead, agents must operate under constraints, such as monetary budgets, latency limits, or safety protocols (Zheng et al., 2024; Liu et al., 2026a; Wu et al., 2025; Xie et al., 2024a; Zhang et al., 2025b; Ma et al., 2026). Operating under these constraints fundamentally changes the problem. The agent can no longer rely on exhaustive trial-and-error, and small planning errors can compound into constraint violations and irrecoverable states (Valmeekam et al., 2023; Xie et al., 2024a; Hao et al., 2025a). Our work focuses on improving the success rate of LM agents in such constrained settings by mitigating this error compounding.

A.2 INFERENCE-TIME SCALING

Inference-time scaling improves LM outputs by sampling multiple candidates and aggregating their answers (Wang et al., 2023a). Recent work extends this principle to LM agents by training step-level process reward models that score agent trajectories (Xi et al., 2026; Choudhury, 2025; Sohn et al., 2026) and by integrating process rewards with tree search for self-training (Zhang et al., 2024; Pisano & Navigli, 2026). PlanGEN (Parmar et al., 2025) uses constraint-guided iterative verification and selects an inference-time algorithm based on instance complexity. Other work characterizes the verification and context bottlenecks that limit parallel scaling for general agents (Li et al., 2026). Related directions train process verifiers from automated supervision (Setlur et al., 2025) and explore speculative action deliberation (Xia et al., 2025).

Our framework also samples multiple candidates, but uses them to construct a plan graph before the agent takes further actions. The external solver uses LM-predicted rewards and costs to select a feasible path under the constraints represented in the graph. This separates the LM's predictions from constrained path selection, while retaining those predictions as the basis of the graph.

A.3 PLAN STRUCTURE AND AGGREGATION

Beyond the planning methods covered in Section 2, several lines impose explicit structure on the plans that LM agents produce. DAG-style representations encode execution dependencies among tool

calls and sub-tasks (Gao et al., 2024; Zhang et al., 2025c; Chi et al., 2025). PLaG (Lin et al., 2024) combines graph representations with natural-language prompts for asynchronous plan reasoning. Decomposition-and-merge pipelines split a task into sub-plans that are subsequently combined and refined (Lu et al., 2025). AggAgent (Lee et al., 2026b) treats parallel trajectories as an environment that an aggregation agent explores through tools. The Procedural Graph framework (Lu et al., 2026) organizes procedures and their relations into a graph, generates guidance from the current node's neighborhood, and refines the graph using successful and failed trajectories.

Our framework separates prediction from selection: the LM records candidate action outcomes in a plan graph, the external solver selects a path under the represented constraints, and the agent uses it as guidance within its original interaction loop.

A complementary line refines plans adaptively during execution, triggering LM re-planning based on intermediate feedback (Sun et al., 2023; Prasad et al., 2024; Zhang et al., 2025d; Shen et al., 2025), or decoupling the planner and executor through shared context summaries (Liu et al., 2026b; Chen et al., 2025). Other methods refine the reasoning loop through self-critique and reflection on execution traces (Shinn et al., 2023; Kim et al., 2025), or use explicit tree search over the agent's action space (Koh et al., 2025; Putta et al., 2024).

B Method and Solver Details

We provide the full agent–planning procedure and the formulations of the external solvers used in Section 3. Appendix H collects the corresponding LM prompt templates and component interfaces. The implementation is available at https://anonymous.4open.science/r/TPS-ICLR-2027-F210.

B.1 Agent and Planning Procedure

Algorithm 1 gives the procedure in Section 3 with initial planning followed by agent-selected replanning. The harness invokes TPS on the first turn and returns a sequence of subplans as a tool observation; the main LM agent decides whether to call TPS again. It uses this guidance when generating concrete actions in subsequent interactions. If the solver finds no feasible path, the tool returns a NoPlan report.

We use the notation for the history h, goal g, candidate count M, and role-specific LM calls defined in Section 3. Here, P_m is a candidate plan, and T_m is that plan annotated with summaries of conditions relevant to subsequent subplans and goal satisfaction, together with edge costs. The functions r and c assign predicted rewards to nodes and predicted cost vectors to edges, respectively. The main agent and planning subagents use the same LM under the harness prompt and stage-specific planning prompts, respectively. In the main agent's output (u, a), u is the chosen tool and a specifies the concrete action and its tool arguments; o denotes the resulting tool observation.

Algorithm 1 Main LM agent with TPS providing subplans as planning guidance

Require: tool set $\mathcal{T}$ including TPS, initial history h, goal g, budget b, number of candidates M
$o \leftarrow \textsc{Tps}(h, g, b, M)$ ▷ initial planning
Append the planning call and o to h; update b using observed resource consumption
while task interaction continues **do**
  $(u, a) \leftarrow \mathrm{LM}_{\mathrm{agent}}(h, g)$ ▷ agent decides whether to replan
  if u is TPS **then**
    $o \leftarrow \textsc{Tps}(h, g, b, M)$ ▷ subplan sequence or NoPlan
  else
    $o \leftarrow$ execute action a through tool u in the environment
  end if
  Append (u, a, o) to h; update b using observed resource consumption
end while
function $\textsc{Tps}(h, g, b, M)$
  $(P_1, \ldots, P_M) \leftarrow \mathrm{LM}_{\mathrm{sample}}(h, g, M)$ ▷ candidate subplan sequences
  for $m = 1$ **to** M **do**
    $T_m \leftarrow \mathrm{LM}_{\mathrm{sim}}(h, P_m)$ ▷ history summaries and predicted costs
  end for
  $\mathcal{G} \leftarrow \mathrm{LM}_{\mathrm{merge}}(T_1, \ldots, T_M)$ ▷ merge states; retain subplans and costs
  $v_0 \leftarrow$ node summarizing the current history h
  $\{r(v)\}_{v \in V} \leftarrow \mathrm{LM}_{\mathrm{reward}}(\mathcal{G}, g)$ ▷ one reward per node
  $V_{\mathrm{ter}} \leftarrow \{v \in V : r(v) = 1\}$ ▷ predicted goal nodes
  $\pi^\star \leftarrow \textsc{Solve}(\mathcal{G}, v_0, V_{\mathrm{ter}}, r, c, b)$ by Equation (1), or NoPlan
  if $\pi^\star$ is NoPlan **then**
    return NoPlan
  end if
  return the subplan sequence labeling the edges of $\pi^\star$
end function

B.2 Solvers for Plan Path Selection

We detail the solvers used for plan path selection in Section 3.2, including the default ILP formulation and three alternatives compared in Figure 8. All four solvers use the same graph, predicted goal nodes, and budget constraints. We specify their scoring and termination rules below. We denote the starting node by v_0 and the set of predicted goal nodes by $V_{\mathrm{ter}} = \{v \in V : r(v) = 1\}$. Each graph edge corresponds to a subplan, and its cost vector $c(e)$ records the quantities constrained by b. The length limits below count graph transitions; a subplan may require one or more actions during execution.

Resource Costs and Action Penalties. Resource entries in $c(e)$ measure quantities such as action count or predicted execution time. The Sokoban and ALFWorld implementations also constrain the simulator's integer action cost: 1 for a useful action, 2 for an unproductive action, and 5 or more for an action predicted to make the task infeasible. This value is a heuristic penalty, distinct from the number of executed actions. The solver limits its sum by the remaining action budget, alongside the separate action-count limit. Thus, cost enters the feasibility constraints; the solver maximizes predicted reward rather than minimizing this penalty.

B.2.1 Integer Linear Programming

Base Formulation (Without Budget). We optimize over a finite horizon $L_{\max} \geq 1$ using a time-expanded formulation. Each predicted goal node is augmented with a zero-cost `STOP` self-loop, and E^+ denotes the resulting edge set. For each edge $e = (v \to v') \in E^+$ and step $\ell \in \{0, 1, \ldots, L_{\max} - 1\}$, let $x_{e,\ell} \in \{0, 1\}$ indicate whether the walk takes edge e at step ℓ. Let $\mathrm{src}(e) = v$ and $\mathrm{tgt}(e) = v'$. The ILP uses discounted node rewards with $\gamma = 0.9$; the starting node contributes zero to the score. The time-indexed objective below defines $R(\pi)$ for the ILP in

Equation (1):

$$\max_{x} \quad \sum_{\ell=0}^{L_{\max}-1} \sum_{\substack{e\in E^+ \\ \mathrm{tgt}(e)\neq v_0}} \gamma^{\ell} r(\mathrm{tgt}(e))\, x_{e,\ell} \tag{2}$$

$$\text{s.t.} \quad \sum_{e\in E^+} x_{e,\ell} = 1, \quad \forall \ell = 0, \ldots, L_{\max}-1, \tag{3}$$

$$\sum_{e\in E^+:\, \mathrm{src}(e)=v_0} x_{e,0} = 1, \tag{4}$$

$$\sum_{e\in E^+:\, \mathrm{tgt}(e)\in V_{\mathrm{ter}}} x_{e,L_{\max}-1} = 1, \tag{5}$$

$$\sum_{e\in E^+:\, \mathrm{tgt}(e)=v} x_{e,\ell-1} = \sum_{e\in E^+:\, \mathrm{src}(e)=v} x_{e,\ell}, \quad \forall v \in V,\ \forall \ell \in \{1, \ldots, L_{\max}-1\}, \tag{6}$$

$$x_{e,\ell} \in \{0,1\}, \quad \forall e \in E^+,\ \forall \ell. \tag{7}$$

Equation (3) selects one edge per step, and Equation (4) starts the walk at v_0. Equation (5) requires a predicted goal after the final transition, while Equation (6) connects consecutive edges. The `STOP` self-loops pad early arrival to the finite horizon without consuming resources. Their goal rewards remain in Equation (2), so earlier goal arrival receives additional discounted reward. This padding is only used for scoring: the returned subplan sequence ends at the first predicted goal and excludes `STOP` edges.

Budget-Constrained Extension. When the task has a budget, the solver maximizes the same reward score under the cost limits in b. The cost vector $c(e)$ contains the resource entries and any task-specific penalty described above, with $c(e) = \mathbf{0}$ for `STOP` edges. We add a component-wise budget constraint:

$$\max_{x} \quad \sum_{\ell=0}^{L_{\max}-1} \sum_{\substack{e\in E^+ \\ \mathrm{tgt}(e)\neq v_0}} \gamma^{\ell} r\big(\mathrm{tgt}(e)\big)\, x_{e,\ell} \tag{8}$$

$$\text{s.t.} \quad \sum_{\ell=0}^{L_{\max}-1} \sum_{e\in E^+} c(e)\, x_{e,\ell} \preceq b, \tag{9}$$

constraints Equation (3)–Equation (7).

Equation (9) enforces the *budget constraint*: the cumulative predicted cost of the selected walk must not exceed b in any dimension ($\preceq$ denotes componentwise inequality). The remaining structural constraints are unchanged. If the solver finds no feasible path, TPS returns a NoPlan report to the agent, which continues its own reasoning within the existing harness.

Scoring in the Solver Comparison. The alternative solvers below accumulate discounted rewards up to the first predicted goal, without `STOP` padding. The comparison in Figure 8 scores every returned path using this same unpadded return, including the paths returned by ILP. The padded ILP objective and the unpadded return are different scoring rules; agreement on the recorded graphs is an empirical result.

B.2.2 Greedy with Goal-Distance Lookahead

At each node v, the solver picks the outgoing edge whose target maximizes the discounted node-score, restricted to edges that (a) do not violate any remaining budget and (b) keep at least one goal reachable within the remaining length. Among ties, the unweighted BFS distance from the target to the nearest goal is used as a secondary key. This tiebreak is essential in our setting: when intermediate node-scores are zero (the typical case where only goals carry nonzero LM-predicted score), the score lookup alone cannot distinguish detour edges from progress edges, and a naive greedy can drift on score-zero plateaus and incur extra discount. The procedure is summarized in Algorithm 2; per-call runtime is $O(L \cdot (|V| + |E|))$.

Algorithm 2 Greedy path selection on plan graph $\mathcal{G} = (V, E)$

Require: Plan graph $\mathcal{G}$, start node v_0, terminal set V_{ter}, length budget L, resource budget b, edge resources $c(\cdot)$, node scores $r(\cdot)$, discount γ

$v \leftarrow v_0,\ \pi \leftarrow (\,),\ \ell \leftarrow 0$
while $\ell < L$ **and** $v \notin V_{\text{ter}}$ **do**
  $\mathcal{F} \leftarrow \{e = (v \to v') \in E : c(e) \preceq b$ **and** ReachableWithin$(v', V_{\text{ter}}, L - \ell - 1)\}$
  if $\mathcal{F} = \emptyset$ **then**
    return NoPlan
  end if
  $e^\star \leftarrow \arg\max_{e=(v\to v')\in\mathcal{F}} \big(\gamma^\ell r(v'),\ \mathbb{1}[v' \in V_{\text{ter}}],\ -d_{\text{BFS}}(v', V_{\text{ter}})\big)$ ▷▷ lex. tiebreak
  Append $e^\star$ to π; $v \leftarrow \text{tgt}(e^\star)$; $b \leftarrow b - c(e^\star)$; $\ell \leftarrow \ell + 1$
end while
return π if $v \in V_{\text{ter}}$ else NoPlan

B.2.3 RCSP: Label-Setting Dijkstra with Pareto Pruning

The Resource-Constrained Shortest Path solver (Desrochers & Soumis, 1988; Beasley & Christofides, 1989) searches for a goal-reaching path that maximizes the unpadded discounted return under the budget constraints. It maintains, at each node v, a set $\mathcal{L}(v)$ of *labels* of the form $\ell = (\text{length}, \text{time}, \text{cost}, \text{score}, \text{parent})$. Labels are expanded in BFS order from a priority queue. When a new label ℓ reaches node v, the dominance test $\ell \succeq_{\text{Pareto}} \ell'$ holds iff ℓ is weakly better in all four dimensions: smaller-or-equal length / time / cost AND larger-or-equal score. The new label is dropped if any existing label in $\mathcal{L}(v)$ dominates it; conversely, any existing label that the new one dominates is removed. The procedure is given in Algorithm 3.

Pareto pruning retains nondominated partial walks, allowing the solver to find an optimal feasible path under the represented objective and budgets when search completes. Worst-case runtime is exponential in $|V|$ because resource-constrained path selection is NP-hard. On the recorded graphs in Figure 8, RCSP takes less than 0.1 ms per call.

B.2.4 CC-MCTS: Constrained Monte Carlo Tree Search

We adapt cost-constrained POMCP (Lee et al., 2018) to the plan graph. The solver runs UCT-style tree search with outgoing subplan edges scored by a Lagrangian-augmented Q value:

$$e^\star = \arg\max_e \ \big[\, Q_R(v,e) - \lambda_T\, Q_T(v,e) - \lambda_C\, Q_C(v,e) \,\big] + c_{\text{exp}} \sqrt{\log N(v)/N(v,e)},$$

where Q_R, Q_T, Q_C are Monte-Carlo returns for reward, time-cost, and tool-cost respectively, accumulated separately during backup. After each simulation the dual variables are updated by projected gradient ascent on the constraint violation: $\lambda_k \leftarrow \max(0,\ \lambda_k + \eta(\text{cost}_k - b_k))$. To avoid crediting trajectories that fail to reach a goal, we apply a CC-POMCP-style reward-shaping mask: rollout returns contribute to Q_R *only when the trajectory ends at a goal node*. Hyperparameters: 5000 simulations per call, exploration constant c_{exp}=1.4, dual learning rate η=0.5, λ initialized to 0. The full simulation loop is given in Algorithm 4.

Comparison summary. ILP can certify optimality for its finite-horizon objective when solved to optimality, while RCSP searches the unpadded discounted return. Greedy and CC-MCTS are approximate: greedy chooses locally attractive edges, while CC-MCTS uses a finite rollout budget. A failure to find a feasible path with an approximate solver does not certify infeasibility. Figure 8 compares path quality and runtime across graph sizes.

Algorithm 3 RCSP (label-setting Dijkstra with Pareto pruning)

Require: Plan graph $\mathcal{G}$, start v_0, terminals V_{ter}, length budget L, resource budget b, edge resources $c(\cdot)$, node scores $r(\cdot)$, discount γ

Initialize $\mathcal{L}(v) \leftarrow \emptyset$ for all $v \in V$; $\ell_0 \leftarrow (0, \mathbf{0}, 0, \perp)$; insert ℓ_0 into $\mathcal{L}(v_0)$ and into priority queue $\mathcal{Q}$
$\ell^\star \leftarrow \perp,\ r^\star \leftarrow -\infty$
while $\mathcal{Q}$ not empty **do**
  $\ell \leftarrow \mathcal{Q}.\textsc{Pop}()$ (min length first); let v be the node holding ℓ
  if ℓ has been removed by dominance **then**
    continue
  end if
  if $v \in V_{\text{ter}}$ **and** $\ell.\text{score} > r^\star$ **then**
    $\ell^\star \leftarrow \ell;\ r^\star \leftarrow \ell.\text{score};$ **continue** ▷▷ stop-at-goal
  end if
  if $\ell.\text{length} \geq L$ **then**
    continue
  end if
  for all outgoing edges $e = (v \to v') \in E$ **do**
    $\ell' \leftarrow (\ell.\text{length} + 1,\ \ell.\text{cost} + c(e),\ \ell.\text{score} + \gamma^{\ell.\text{length}} r(v'),\ e)$
    if $\ell'.\text{length} > L$ **or** $\ell'.\text{cost} \not\preceq b$ **then**
      continue
    end if
    if $\exists\, \tilde{\ell} \in \mathcal{L}(v')$ s.t. $\tilde{\ell}$ Pareto-dominates ℓ' **then**
      continue
    end if
    Remove from $\mathcal{L}(v')$ every label dominated by ℓ'
    Insert ℓ' into $\mathcal{L}(v')$ and $\mathcal{Q}$
  end for
end while
return walk reconstructed from parent-chain of $\ell^\star$, or NoPlan if $\ell^\star = \perp$

Algorithm 4 CC-MCTS simulation loop (one call to the solver)

Require: Plan graph $\mathcal{G}$, start v_0, terminals V_{ter}, length budget L, resource budget b, #simulations K, exploration c_{exp}, dual rate η

$\lambda_T \leftarrow 0,\ \lambda_C \leftarrow 0;$ initialize $N(v) = N(v,e) = 0,\ Q_R = Q_T = Q_C = 0$ everywhere
for $k = 1, \ldots, K$ **do**
  $v \leftarrow v_0;$ traj $\leftarrow (\,);$ remaining budget vector $\bar{b} \leftarrow b,$ remaining length $\bar{L} \leftarrow L$
  while $v \notin V_{\text{ter}}$ **and** feasible outgoing edge exists **and** $\bar{L} > 0$ **do**
    $\mathcal{F} \leftarrow$ feasible outgoing edges from v under $(\bar{L}, \bar{b})$
    if $\exists$ unexplored $e \in \mathcal{F}$ **then**
      $e \leftarrow$ uniform random over unexplored; **break** ▷▷ expand-then-rollout
    else
      $e \leftarrow \arg\max_{e \in \mathcal{F}} \left[Q_R(v,e) - \lambda_T Q_T(v,e) - \lambda_C Q_C(v,e) \right] + c_{\text{exp}} \sqrt{\log N(v) / N(v,e)}$
    end if
    Append (v,e) to traj; update $v, \bar{b}, \bar{L}$ by traversing e
  end while
  rollout $\leftarrow$ random rollout from v until goal or budget exhausted
  $g \leftarrow \mathbb{1}$[trajectory reached a goal] ▷▷ goal-credit mask
  Backup along traj $\cup$ rollout: update N, Q_R (multiplied by g), Q_T, Q_C via incremental mean
  $\lambda_T \leftarrow \max(0,\ \lambda_T + \eta(\text{traj.time} - b_T));\ \lambda_C \leftarrow \max(0,\ \lambda_C + \eta(\text{traj.cost} - b_C))$
end for
Reconstruct $\pi^\star$ greedily from v_0: at each v pick $\arg\max_e [Q_R(v,e) - \lambda_T Q_T(v,e) - \lambda_C Q_C(v,e)]$ over visited edges
return $\pi^\star$ if it reaches a goal within the budget, else NoPlan

Table 5: **Environment-execution budgets across benchmarks.** Internal planning inference is accounted for separately from the interactions and execution time used to complete the task.

Benchmark	Environment execution
Sokoban	Player moves
Snake	Snake moves
BlocksWorld	Block-manipulation actions
ALFWorld	Household-environment interactions
ScienceWorld	Science-environment interactions
Terminal-Bench	Terminal execution under task timeouts
AppWorld	Code-execution interactions with application APIs
MLGym-Bench	Code/tool execution through CodeAct

C Experimental Setup

This section defines the benchmarks, execution limits, and harness integrations used in Section 4. Analysis-specific protocols accompany their results in Appendices E–G; prompt templates are collected in Appendix H.

C.1 Tasks and Backbones

The main evaluation covers eight benchmarks across game, text-world, and coding tasks. Sokoban, Snake, and BlocksWorld use discrete actions to move boxes, navigate a snake, or rearrange blocks. The Snake environment is our deterministic, step-based implementation inspired by Google Snake in GameWorld (Ouyang et al., 2026), with fixed pellet locations and a goal of consuming all pellets. ALFWorld and ScienceWorld use textual observations and actions for household and science tasks. These five benchmarks use `gpt-4.1-mini`. Terminal-Bench 2.0 tasks, AppWorld, and MLGym-Bench evaluate shell, API, and machine-learning research workflows, respectively, with `claude-haiku-4.5`. MLGym-Bench (Nathani et al., 2025) comprises 13 open-ended AI research tasks involving implementation, training, and evaluation across multiple domains. The Qwen3 model-size and component analyses use ReAct on ten fixed Sokoban maps with actual optimal lengths $T^\star \in \{5, 6, 7\}$, repeated ten times each. The directory label `T6` denotes a difficulty bucket, not an exact solution length. There are 100 episodes but only ten distinct maps per configuration. The game main results also use 100 episodes per configuration. ALFWorld and ScienceWorld use 134 and 90 episodes per configuration, respectively. The Snake instances span task lengths 5–11, and the BlocksWorld instances span 4–10. AppWorld uses 168 tasks; the MLGym-Bench comparison contains 65 evaluated runs per configuration. Terminal-Bench uses 86 tasks. The baseline includes 510 trials, with five or ten repetitions per task; TPS attempts five repetitions per task. Five TPS infrastructure failures are excluded from the success-rate calculation, leaving 425 scored trials; six other TPS exceptions count as failures. We compute success rates within each task and average them with equal task weights.

Domain means weight benchmark–harness pairs equally. Error bars in game, text-world, and Qwen3 comparisons, and uncertainties in planning-comparison tables, denote binomial standard errors. The coding figure uses task-bootstrap 95% intervals for Terminal-Bench and 95% Wilson intervals for AppWorld and MLGym-Bench.

C.2 Environment Execution and Internal Inference

Table 5 identifies the environment interactions and execution resources for each benchmark. Environment-action limits bound moves or interactions, while the Terminal-Bench harness enforces task timeouts. Internal inference is the LM computation used by the agent and its planning subagents; we account for its tokens, estimated FLOPs, and planning calls separately. Elapsed episode time includes this computation as well as tool execution and service waiting.

C.3 Harness Integration

For ReAct, TPS is an additional planning tool in the reasoning–action loop. The default integration calls TPS on the first turn; the agent then decides whether to request further planning and how to use the selected plan. For RLM, we fix the maximum recursion depth to $r = 2$ for both Base and Base + TPS across game and text-world tasks. For text worlds, RLM stores the interaction history in a Python REPL variable. The root LM inspects that history with code and can call fresh sub-LMs; TPS is an additional tool available from the REPL. ToT incorporates TPS as a planning meta-action in its search. The coding experiments expose the same planning-tool interface through Terminus-KIRA, ReAct, and CodeAct on Terminal-Bench, AppWorld, and MLGym-Bench, respectively.

C.4 Planning Comparisons

Single Plan follows the separation of planning and execution in Plan-and-Act (Erdogan et al., 2025), using one planning subagent. Our Best-of-N baseline (Wang et al., 2023a) uses an LM to select one of N candidate plans; we use $N = 4$ in these experiments. Graph + LM and TPS both construct a graph from four candidate plans. They differ in whether an LM or an external solver selects the path. The advisor comparison uses the advisor–executor separation (Asawa et al., 2026; Anthropic, 2026a), with Qwen3-32B for planning and Qwen3-8B or Qwen3-14B for execution. BoN in Table 4 denotes Best-of-N, also evaluated with $N = 4$. All components of TPS use the executor model in the Qwen3 task-agent experiments. For the graph-size diagnostic, `gpt-4.1-mini` and ILP operate on the same recorded graph. We report the fraction for which each method returns a feasible path, using only graphs containing one as the denominator.

Path-Selection Solver. We use ILP for the task-agent experiments. The solver evaluates the requirements represented by the graph's estimated states, rewards, and costs. The formulation and alternative solvers are described in Section B.2. Meta-Harness instead selects a set of candidate revisions under an evaluation budget, as described in Appendix G.

D Additional Benchmark Results

We extend the benchmark evaluation in Section 4 with additional Sokoban backbones, paired coding-task comparisons, and concrete execution examples. Appendix C describes the shared benchmark setup.

D.1 Additional Results with API Models

We extend the ReAct comparison to API models with different baseline performance, using Sokoban instances with $T^\star = 6$. Each configuration has 100 episodes, except `gpt-4.1-nano` + TPS ($n = 99$). Figure 7 shows higher success with TPS for every evaluated model. Base success ranges from 5% for `gpt-4.1-nano` to 90% for `gpt-5-nano`. The gains are 62, 80, and 73 percentage points for `gpt-4.1-mini`, `gpt-4.1`, and `claude-haiku-4.5`, respectively, and 6 points for `gpt-5-nano`. The improvements span widely different Base success rates, although the gain does not increase monotonically as Base performance decreases.

D.2 AppWorld Results and Execution Example

We compare Base and TPS on the same AppWorld tasks to quantify uncertainty in the observed success gain. For AppWorld, we resample the 168 paired task outcomes 10,000 times to obtain a percentile interval for the success difference (seed 25092026). The 5.4-point difference has a 95% interval of $[-0.6, 11.9]$ points.

Playlist Archiving. In task `634f342_3`, the agent must read a song list from the file-system app, remove the listed songs from every Spotify playlist, and move them into a new playlist named "Archived Playlist Songs." Both runs use ReAct with `claude-haiku-4.5` and an environment-action budget of 50. Table 6 compares their recorded execution traces. The TPS run retrieves the complete playlist list before calling the planning tool, then follows the returned sequence to perform

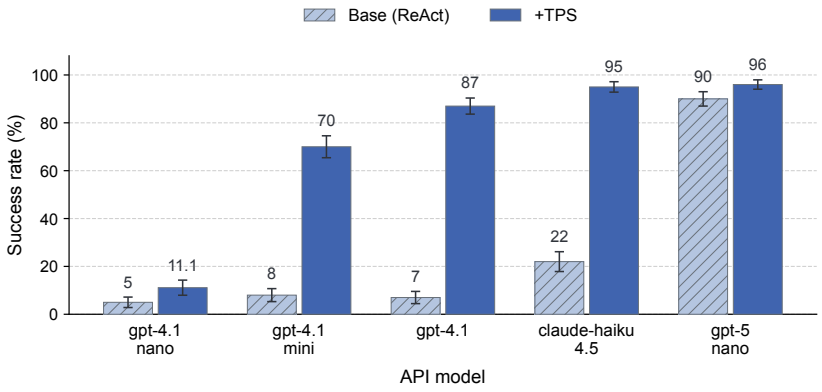


Figure 7: **Success rates across API models on Sokoban.** TPS improves success across models with widely different Base success rates.

Table 6: **Execution traces for an AppWorld playlist-archiving task.** The TPS run completes the requested update across all playlists, while Base leaves part of the task unfinished.

Stage	Base	Base + TPS
Information gathered	Reads the first library page, covering five playlists and 27 distinct songs.	Handles pagination and retrieves all seven playlists before calling TPS with 28 actions remaining.
Plan and execution	Matches the retrieved songs against the file, archives the matches, and declares completion.	The returned plan orders song-detail retrieval, matching and removal, archive creation, and song insertion. Execution follows this sequence.
Observed result	Archives eight songs and passes four of seven checks; the task fails.	Removes 14 song occurrences and archives all 13 distinct target songs; all seven checks pass.

the update. Base submits a partial update as complete. The example illustrates how a plan can organize a cross-app workflow once the agent has gathered the relevant information.

D.3 Terminal-Bench Per-Task Analysis

The evaluation covers all Terminal-Bench 2.0 tasks with a $3\times$ timeout multiplier. Reported scores are the average of 5 runs.

TPS has a higher recorded rate on 23 tasks, an equal rate on 52, and a lower rate on 11. The largest increases are on `caffe-cifar-10` (0% to 80%), `pytorch-model-cli` (40% to 100%), and `custom-memory-heap-crash` (0% to 60%). The largest decreases are on `overfull-hbox` (80% to 20%), `reshard-c4-data` (60% to 0%), and `sqlite-with-gcov` (100% to 60%). The complete per-task rates are retained in the accompanying analysis files. The TPS configuration invokes planning on the first turn and lets the agent choose subsequent planning calls.

Build-Directory Constraints. The `sqlite-with-gcov` task requires compiling SQLite with coverage instrumentation in `/app/sqlite`. In the Base example, the agent builds in that directory, placing the coverage artifacts where required. The TPS plan specifies `./configure --prefix=/app/sqlite` but leaves the build directory unspecified. The executor builds in `/tmp/sqlite`, so the `.gcno` files remain outside the required directory and the coverage check fails. An installation prefix does not determine where compilation artifacts are written. This case illustrates why environment-specific requirements must be represented in the plan: feasibility within an estimated graph cannot enforce a condition omitted from it.

Exploring Alternative Solution Forms. The `distribution-search` task requires a probability distribution satisfying target forward and reverse KL divergences from the uniform distribution. In the Base example, the agent explores two-value and multi-bin distributions before using continuous parameter optimization to find a solution. The TPS execution instead follows an exponential-decay proposal, adjusting its exponent and tail without exploring substantially different distribution families, and fails. This contrast suggests that candidate diversity must extend to different solution approaches; parameter variations within an unsuitable family may leave the solver without a useful alternative.

E Additional Sokoban Analysis

Following Section 5, we first analyze success and action use under different environment-action budgets, then compare feasible-path finding on identical graphs and the components of TPS. Candidate-count and qualitative analyses further examine how planning supports task execution.

E.1 Setup for Action-Budget Analyses

The action-budget analyses in Section 5, and the candidate-count analysis in Appendix E.7, compare ReAct and ReAct + TPS with `gpt-4.1-mini`. We use ten fixed maps with optimal solution lengths $T^\star \in \{5, 6, 7\}$ and repeat each configuration ten times per map, giving 100 episodes per configuration. Both harness prompts omit the immediate-push preference and the instruction to execute a previously committed push. We vary $M \in \{2, 4, 8\}$ at $B = T^\star + 2$, and vary the additional action budget $\Delta \in \{2, 4, 8, 16\}$ at $M = 4$, with $B = T^\star + \Delta$. The two analyses share the $M = 4$, $\Delta = 2$ configuration. Candidate plans are sampled in parallel at temperature 1, with LM-based simulation, merging, and reward prediction, and ILP for path selection. The comparisons match maps and action budgets across methods and allow each method to use its own inference compute. Here $B = B_{\text{env}}$ is the total environment-action budget. Mean budget consumption averages the per-episode ratio of executed actions to B over successful and failed episodes together. The horizontal axis in Figure 5 averages the per-episode ratio $B/T^\star$, using each map's oracle minimum $T^\star$. The four settings have mean ratios 1.36, 1.71, 2.43, and 3.86, respectively.

For ReAct + TPS, the harness invokes TPS on the first turn, and the agent chooses subsequent planning calls. The implementation returned a budget-relaxed fallback plan in two of 766 planning calls; both corresponding episodes failed. We replayed all 1,000 recorded action sequences and verified their success labels and action counts.

For the additional Sokoban comparisons, we also compute bootstrap intervals using 10,000 resamples of paired maps and independent resamples of repetitions within each method and map. Harness prompt examples are collected in Appendix H.1.

E.2 Success and Action Use across Budgets

At $M = 4$ and $\Delta = 2$, ReAct and ReAct + TPS obtain 8% and 70% success, respectively. Figure 5 in the main paper reports success, budget consumption, and failures caused by insufficient remaining actions across $\Delta \in \{2, 4, 8, 16\}$. The corresponding paired-map 95% bootstrap intervals for the success gains are $[45, 79]$, $[50, 82]$, $[47, 77]$, and $[65, 89]$ percentage points. Figure 14 shows the action and latency trade-offs for these configurations and the candidate-count sweep.

E.3 Insufficient Remaining Budget and Physical Deadlocks

To distinguish budget exhaustion from physical deadlock, we locate the first action after which the goal cannot be reached within the remaining action allowance. Let s_t be the true state after t environment actions, and let $d^\star(s_t)$ be its shortest distance to the goal under the true transition function. We compute $d^\star$ by exhaustive Sokoban reachability and identify

$$\tau = \min\{t : d^\star(s_t) > B_{\text{env}} - t\}.$$

If $d^\star(s_\tau)$ is finite, the first loss is classified as insufficient remaining actions; if it is infinite, the first loss is a physical deadlock. The classification uses the first crossing, even if a later action also creates a deadlock. Every failed trajectory has a crossing, and no successful trajectory does. Figure 5c

Table 7: **Loss of budget feasibility across action budgets.** The first loss separates insufficient remaining actions from physical deadlocks; TPS reduces losses of budget feasibility.

	ReAct		ReAct + TPS	
Additional budget Δ	Budget loss	Deadlock	Budget loss	Deadlock
2	91	1	29	1
4	88	4	26	0
8	77	8	18	5
16	75	11	6	2

Table 8: **Feasible-path finding by graph size.** ILP finds feasible paths more consistently than the LM on the same recorded graphs.

Graph nodes	Feasible graphs	LM found	ILP found
≤ 10	42	40	42
11–15	43	42	43
16–20	33	31	33
≥ 21	26	22	26
Total	144	135	144

reports episodes whose first loss is due to insufficient remaining actions, divided by all episodes in the configuration.

Each configuration contains 100 episodes, so counts are also percentages of evaluated episodes.

E.4 Finding Feasible Paths across Graph Sizes

The graph-size comparison uses 335 archived planning calls from 200 Sokoban episodes with `gpt-4.1-mini` and four candidate plans. The annotated-plan Graph + LM runs cover the `T6` and `T10` difficulty buckets, with 100 episodes each. An exact graph search identifies 144 calls whose recorded graph contains a feasible path; these calls form the denominator in Table 2. We group them into ≤ 10, 11–15, 16–20, and ≥ 21 nodes. Table 8 lists the counts underlying each rate.

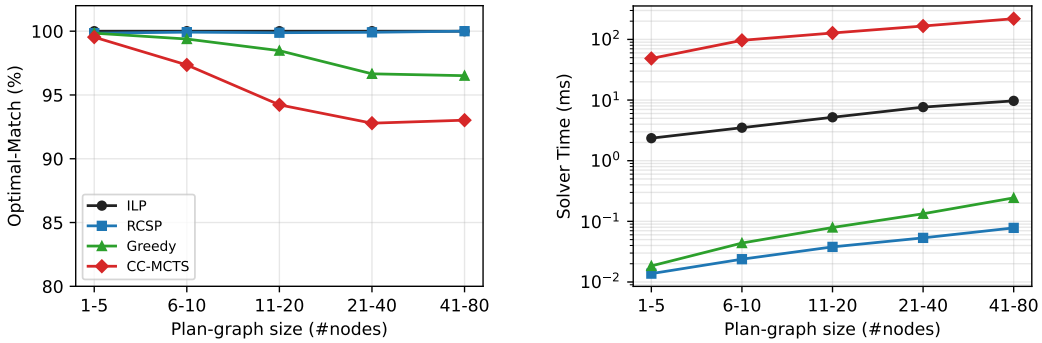
A returned path is feasible if it follows recorded edges from the initial node, ends at a node with predicted reward at least 1, and respects both the remaining action-count limit and the sum of predicted edge costs. LM results use the archived outputs after the original host validation, which follows the first matching outgoing edge for each action; fallback outputs are excluded. ILP is run offline on the same nodes, edges, rewards, costs, and budgets, retaining distinct node and edge identifiers, including parallel edges. No new LM calls or environment interactions are used. The solver uses discount 0.9 and a two-second limit per graph, and its returned paths are checked against the recorded constraints.

ILP finds feasible paths in all 144 eligible graphs, compared with 135 for the LM. The LM's rates vary across size groups and do not decrease monotonically with node count; ILP remains at 100% in each group. These rates concern feasibility within the estimated graph. For comparisons with other external solvers, see Appendix E.5.

E.5 External Solver Comparison across Graph Sizes

To relate graph size to solution quality and runtime, we benchmark the external solvers on 8,461 Sokoban plan graphs pooled over configurations and backbones from an earlier experimental pipeline. Graph sizes range from 1 to 80 nodes, with median 11. This pool is separate from the Qwen3 component experiments. We report Optimal-Match, the fraction of graphs on which a feasible returned path has an unpadded discounted score at least as high as the ILP reference path, and per-call solver time in Figure 8.

Figure 8 shows that RCSP closely matches ILP across graph sizes. The approximate solvers fall below the ILP reference score more often as the graphs grow, with Greedy and CC-MCTS reaching approximately 96.5% and 93.0% Optimal-Match on the largest graphs. ILP takes 2.4–9.7 ms per call



(a) Optimal-Match.

(b) Solver runtime.

Figure 8: **Comparison of external solvers across graph sizes.** (a) RCSP closely matches ILP across graph sizes, while Greedy and CC-MCTS fall below the ILP reference score more often on larger graphs. (b) RCSP achieves this agreement with substantially less solver time than ILP.

Table 9: **Planning-component ablation across Qwen3 sizes.** TPS outperforms graph-based LM selection, while graph construction alone does not consistently improve over candidate selection.

Model	ReAct	Single Plan	Best-of-N	Graph + LM	**TPS**
Qwen3-8B	5 ± 2.2	7 ± 2.6	11 ± 3.1	9 ± 2.9	$\mathbf{16 \pm 3.7}$
Qwen3-14B	11 ± 3.1	13 ± 3.4	20 ± 4.0	29 ± 4.5	$\mathbf{37 \pm 4.8}$

across size groups, RCSP takes less than 0.1 ms, and CC-MCTS takes approximately 70–200 ms. On these recorded graphs, RCSP offers an effective alternative to ILP, while approximate selection can sacrifice path quality as graphs grow.

E.6 Additional Component Ablation

The additional component ablation checks whether the main-paper comparisons hold with a smaller model, evaluating the same planning variants with Qwen3-8B and Qwen3-14B. The planning configurations are defined in Appendix C.4; all use the executor model for planning, and the multi-plan variants use four candidates. Moving from Graph + LM to TPS raises success from 9% to 16% at 8B and from 29% to 37% at 14B (Table 9). Graph + LM performs below Best-of-N at 8B. The gains favor solver-based selection at both sizes; graph construction with LM-based selection does not consistently improve success.

E.7 Sensitivity to the Number of Candidate Plans

To measure how candidate count affects success, we vary $M \in \{2, 4, 8\}$ at a fixed action budget of $B = T^\star + 2$, following the Sokoban protocol in Appendix E.1. Figure 9 reports success rates of 59%, 70%, and 82% for $M = 2, 4, 8$, respectively, compared with 8% for ReAct. The highest observed success rate is at $M = 8$. The 95% bootstrap interval for the difference between $M = 8$ and $M = 4$ is $[-1, 26]$ percentage points. TPS improves over ReAct at every evaluated candidate count, although the interval does not establish an advantage for $M = 8$ over $M = 4$. Varying M changes both candidate coverage and graph-construction compute; it does not isolate an individual planning component.

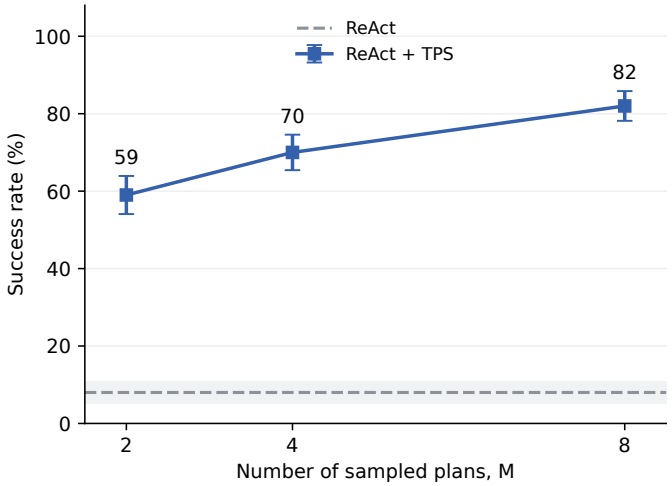


Figure 9: **Sensitivity to the number of candidate plans.** Success increases with candidate count in the evaluated range, and TPS remains above ReAct.

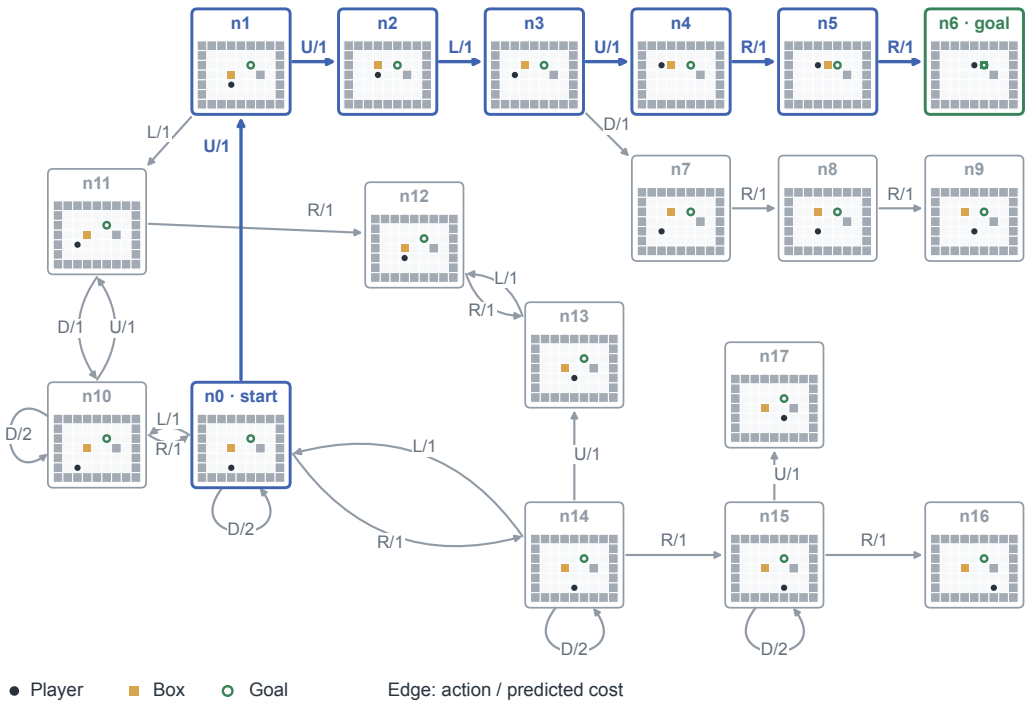
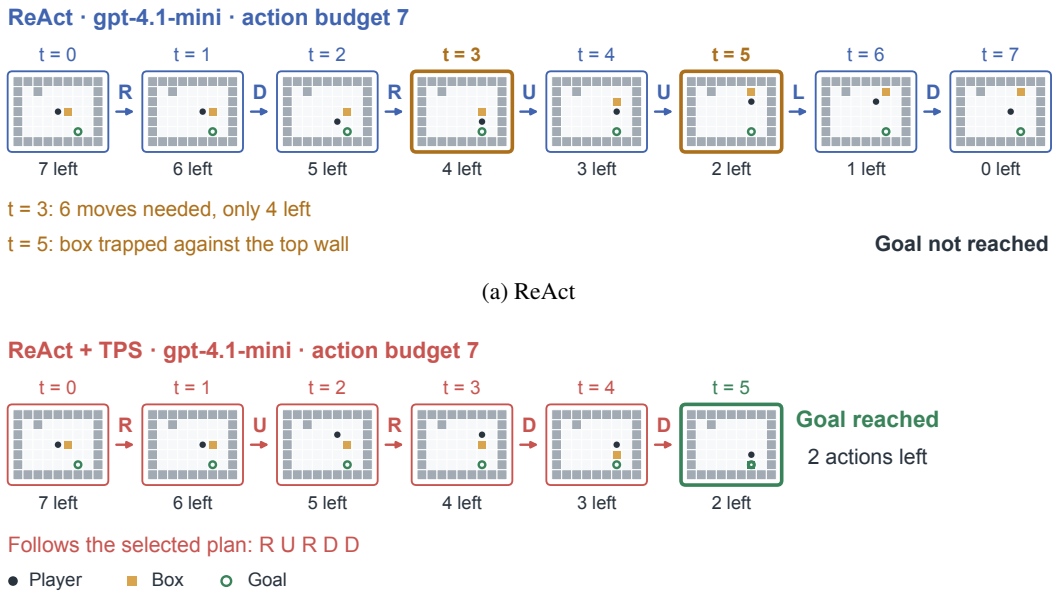


Figure 10: **Recorded TPS plan graph on Sokoban.** The blue path shows the solver's selected plan, while gray branches show other predicted transitions. The agent follows the selected plan and reaches the goal within the remaining action budget.

E.8 Qualitative Analysis

Graph Example. We illustrate the planning process with a recorded ReAct + TPS episode using `gpt-4.1-mini`. Figure 10 shows the first planning call, with 8 environment actions remaining and 8 candidate plans. The recorded graph contains 18 nodes and 55 edge records, corresponding to 27 distinct transitions. Nodes show estimated player and box positions on the original map, and edge labels give the action and predicted cost. Identical parallel transitions are drawn once; self-loops represent blocked moves that leave the state unchanged. ILP selects the blue path `U U L U R R`, which uses 6 actions and has total predicted cost 6. The agent executes this sequence and reaches the goal without another planning call.

Qualitative Comparison. A second recorded Sokoban instance illustrates how the two harnesses use their available actions. Figure 11 compares ReAct and ReAct + TPS with `gpt-4.1-mini`, an action budget of $B = 7$, and 4 candidates for TPS. After ReAct's third action, the goal requires at least 6 more actions, but only 4 remain. At $t = 5$, ReAct pushes the box against the top wall, creating a physical deadlock. TPS instead executes its initial selected plan, `R U R D D`, and reaches the goal in 5 actions with 2 actions remaining. All depicted states are reconstructed from the recorded actions using the Sokoban environment, and shortest-path search verifies the first loss of budget feasibility and the later deadlock. The comparison illustrates how planning before execution can avoid both an unnecessary detour and an irreversible push.



(a) ReAct

(b) ReAct + TPS

Figure 11: **Execution trajectories under the same action budget.** On the same Sokoban instance, ReAct takes a detour and later pushes the box into a deadlock. TPS follows its selected plan and reaches the goal within the budget.

F Graph Accuracy and Computational Cost

We expand the reliability and cost analyses in Section 6 by comparing graph predictions with environment states and accounting for the inference and elapsed time used by planning. The graph analyses distinguish state merging, simulation, and execution; the cost analyses compare model sizes and planning configurations.

F.1 Graph Accuracy

F.1.1 Sokoban State-Merging Accuracy

We evaluate whether $\mathrm{LM}_{\mathrm{merge}}$ (Section 3.1) correctly identifies equivalent states. On Sokoban (easy), canonical state equality serves as ground truth for the merging decisions made by `gpt-4.1-mini`.

Ground truth. Sokoban admits a natural ground truth for state equivalence: each observation can be parsed deterministically into the canonical Sokoban state (player position and box positions), and two states are equivalent iff their canonical forms are identical. We use this canonical equality as the oracle against which the merging decisions of $\mathrm{LM}_{\mathrm{merge}}$ are scored.

Metrics. We score $\mathrm{LM}_{\mathrm{merge}}$ on two complementary quantities:

- **Precision**: of all merges $\mathrm{LM}_{\mathrm{merge}}$ performed during evaluation, the fraction in which the merged nodes are canonically equal.
- **Recall**: of all canonically-equal node pairs that arise across the plan graphs, the fraction that $\mathrm{LM}_{\mathrm{merge}}$ actually folded into a single node.

A merge between non-equivalent estimated states can introduce an invalid path. Equivalent states can also remain represented by separate nodes. These metrics measure agreement with canonical equality of the recorded states; they do not verify the simulated transitions or predicted rewards and costs.

Results. As reported in Table 10, $\mathrm{LM}_{\mathrm{merge}}$ achieves precision 99.7% and recall 75.4%. Most recorded merges satisfy the state-equivalence criterion, while roughly a quarter of equivalent pairs

Table 10: **State-merge accuracy on Sokoban.** Recorded merges have high precision, while some equivalent states remain separate.

Quantity	Value
Merge precision	**99.7%**
Merge recall	**75.4%**

remain separate. This is consistent with the prompt's instruction to merge only clearly equivalent states (Section 3.1). It does not establish that all errors are harmless or that Best-of-N provides a lower bound on TPS's success rate. Even when the graph retains a successful candidate, errors in simulation or reward and cost prediction can cause the solver to select a path that fails during execution.

F.1.2 Sokoban Simulation Accuracy

We analyze the recorded first planning call of each episode in the revised-prompt Sokoban $M = 4$, $\Delta = 2$ evaluation (100 calls on ten maps), comparing each recorded simulated transition before merging against the true Sokoban transition function. The 400 candidate records contain 2,439 action transitions. Of these, 2,432 have physically valid predicted predecessor states, and 2,362 of those predict the correct next state (97.1%). The other seven predecessors are invalid and are reported separately, rather than counted as valid inputs to the simulator. When each candidate's actions are instead replayed from the real initial state, 2,250/2,439 predicted post-action states match the replay (92.3%). The lower rollout agreement captures accumulated prediction errors that a local transition check can miss. These descriptive counts concern repeated transitions within ten maps and do not justify independent-binomial uncertainty over edges.

F.1.3 ALFWorld Graph and Execution Accuracy

We replay all 134 archived ReAct + TPS ALFWorld episodes in TextWorld, reproducing the recorded observations and success labels for 1,135 executed actions. The episodes contain 168 planning calls. For each call, we restore the environment by executing the recorded history, then replay each candidate and the selected recommendation from that state. Recommendations use the original remaining environment-action budget, and replay stops when the task terminates.

Graph-State and Transition Checks. We compare each candidate's graph node with the corresponding state reached during replay. The state projection contains object identities in inventory, their hot/cold status, and the agent's current receptacle location. Projected-state agreement requires all these fields to match. An action is applicable when it belongs to the engine's admissible command set immediately before execution. A positive goal label is correct when the environment's success flag is true at that occurrence. These checks cover 3,751 observed candidate-action outcomes; 59 suffix actions lie after environment termination. The state metric measures rollout agreement of the stored graph representation, combining simulation and merging.

State Equivalence. For pairs of post-action occurrences from different candidates, we compare shared graph-node identity against equality of the replayed engine states. Two engine states are equal when their complete grounded facts, including explicit negative facts, admissible commands, and success flags agree. This exact-state criterion gives 2,736 true-positive, 224 false-positive, and 3,891 false-negative pairs. Merge precision and recall therefore measure the resulting graph representation against exact environment-state equality.

Selected-Plan Replay. We evaluate 126 solver-selected plans returned under the original graph budgets. Each plan is executed unchanged from its recorded planning state within the remaining environment-action budget, without subsequent replanning. Of these, 67 reach the real goal, 64 use only applicable actions until termination, and 62 satisfy both conditions. These per-call rates measure fixed-plan executability; full-agent evaluation additionally includes action choices based on feedback and optional replanning.

Table 11: **Graph accuracy and selected-plan replay in ALFWorld.** Graph checks and replay without replanning reveal errors in predicted states, goals, and plan execution.

Metric	Correct / total	Rate (%)	95% interval
Merge precision (exact state)	2736/2960	92.4	[87.4, 96.7]
Merge recall (exact state)	2736/6627	41.3	[35.4, 47.9]
Goal-label precision	206/342	60.2	[51.3, 68.5]
Goal-label recall	206/288	71.5	[61.7, 80.7]
Projected-state rollout agreement	2933/3751	78.2	[74.4, 81.8]
Candidate-action applicability	2960/3751	78.9	[75.6, 82.3]
Selected path: all actions applicable	64/126	50.8	[41.9, 59.4]
Selected path: reaches real goal	67/126	53.2	[45.0, 61.2]
Selected path: applicable and reaches goal	62/126	49.2	[40.5, 57.6]

Table 11 reports the full metrics. Intervals use 10,000 bootstrap resamples of episodes, retaining all calls, transitions, and state pairs from each sampled episode (seed 25092026).

Table 12: **Episode outcomes by initial planning conditions.** Candidate availability and agreement with the initial recommendation define three groups; final outcomes include subsequent actions and replanning.

Successful candidate	Initial recommendation	Success	Failure	Total
Available	Followed	65	1	66
Available	Not followed	4	6	10
Unavailable	—	1	23	24
All episodes		70	30	100

F.2 Planning and Execution

We follow the initial planning call through execution to distinguish candidate availability, graph correctness, and the agent's use of the selected plan.

F.2.1 Candidate-to-Execution Stage Analysis

We reconstruct the first planning call of each episode in the revised-prompt Sokoban $M = 4$, $\Delta = 2$ evaluation. The cohort contains 100 episodes on ten maps, with the initial call made before any environment action. For the full stage analysis, we apply the following cumulative gates in order:

1. **Candidate coverage**: at least one original candidate prefix reaches the true goal within the original action budget.
2. **Faithful solution simulation**: at least one successful prefix has every predicted predecessor and successor state equal to its true replay state.
3. **Validated graph path**: the graph retains a path with oracle-valid state transitions and a correctly labeled true-goal endpoint, satisfying both the length and predicted-cost budgets. Dynamic programming checks existence.
4. **Solver selection**: the recorded recommendation is a valid graph path to a predicted goal within the same budgets.
5. **Execution agreement**: the actual initial environment-action sequence matches the complete initial recommendation.
6. **Final success**: the episode reaches the real goal, verified by replaying its executed actions.

Each count intersects the current condition with every preceding condition, yielding $100 \to 76 \to 76 \to 76 \to 76 \to 66 \to 65$. The counts summarize attrition within the initial-call cohort; five successes occur outside the final cumulative group, giving 70 online successes overall. Table 12 groups all episodes by candidate coverage and exact agreement with the initial recommendation, and reports final success or failure after all subsequent actions and planning calls. The three groups are disjoint and account for all 100 episodes. Execution agreement measures whether the initial executed action sequence matches the entire recommendation; later replanning can contribute to success in any group. Because all episodes with candidate coverage pass the intervening simulation, graph-path, and solver gates, the 65 successes in the recommendation-following group are also the final cumulative count.

Recommendation Replay. Independently of the cumulative gates, 77 initial estimated graphs contain a budget-feasible path and the recorded solver returns a feasible recommendation for all 77. We reset the environment to the initial state and execute each recommendation literally under the original action budget, without subsequent replanning. The recommended action sequence reaches the true goal in 74 of these 77 cases (96.1%). This distinguishes selection correctness on the graph, the predictions supporting the selected path, and subsequent execution behavior.

F.2.2 Failure Indicators across Planning and Execution

We distinguish observable error indicators from causal attributions. For the first-call Sokoban cohort in Appendix F.2.1, Table 13 reports non-exclusive indicators both across all episodes and within the 30 episodes that ultimately fail. Candidate coverage means that at least one original candidate prefix

Table 13: **Failure analysis across planning and execution stages.** Missing successful candidates and departures from executable advice identify different limitations; overlapping indicators do not assign a unique cause to each failed episode.

Indicator	All episodes	Failed episodes
No successful original candidate prefix	24	23
At least one simulated-transition error	44	14
At least one false-positive goal occurrence	4	2
At least one false-negative goal occurrence	12	0
Executor departs from initial advice before replanning	10	6
Executable initial advice but eventual episode failure	6	6

Table 14: **Task success and inference cost across model sizes.** A smaller model with TPS can outperform a larger ReAct model with less estimated compute.

Configuration	PFLOPs/ episode	Relative compute	Success (%)
32B, ReAct	1.58	1.0×	15 ± 3.6
4B + TPS	1.25	0.8×	23 ± 4.2
14B +32B Advisor (BoN)	2.42	1.5×	22 ± 4.1
14B + TPS	4.09	2.6×	$\mathbf{37 \pm 4.8}$
32B + TPS	11.4	7.2×	30 ± 4.6

reaches the real goal within the action budget. Simulation errors compare predicted states with the transition oracle. Goal-label errors compare graph rewards with actual candidate-prefix outcomes, and therefore can also reflect preceding simulation errors.

Denominators are 100 and 30 episodes, respectively. All six failures despite executable initial advice also contain an action deviation before the next planning call. All first planning calls occur before any environment action, so this sample cannot estimate effects of late invocation. Recorded edge costs include heuristic penalties; disagreement with physical action counts alone is not a cost-prediction error.

F.3 Inference Compute

The Qwen3 comparison accounts for every LM call made during an episode, including planning, selection, and advisor calls. We report estimated PFLOPs per episode and normalize by the 1.58 PFLOPs of Qwen3-32B ReAct. For each call, we estimate compute as $2P(N_{\text{in}} + N_{\text{out}})$, where P is the parameter count of the model making that call. We tokenize cached interaction histories with the Qwen3 tokenizer and reconstruct planning calls from the recorded candidate plans, simulated trajectories, and graphs. Draft-plan lengths are represented by their structured-plan lengths; merge and reward completion lengths use the archived node-based estimates. Every call is charged for its full prompt, including repeated context. Table 3 uses Qwen3-4B with TPS as the reference. The reference uses 4B for both the base agent and the planning subagents. The comparison configurations use a 14B base agent and a 32B planning advisor: Best-of-4 gives similar success, while a single plan gives similar mean estimated PFLOPs. Figure 12 includes all 22 archived configurations, each evaluated on the same ten maps with ten repetitions per map. Its Pareto boundary contains configurations for which no other plotted configuration has both lower or equal compute and higher or equal success, with at least one strict improvement. Each point reports the mean inference compute and success of a complete configuration. Table 14 reports compute and success for additional model and planning configurations.

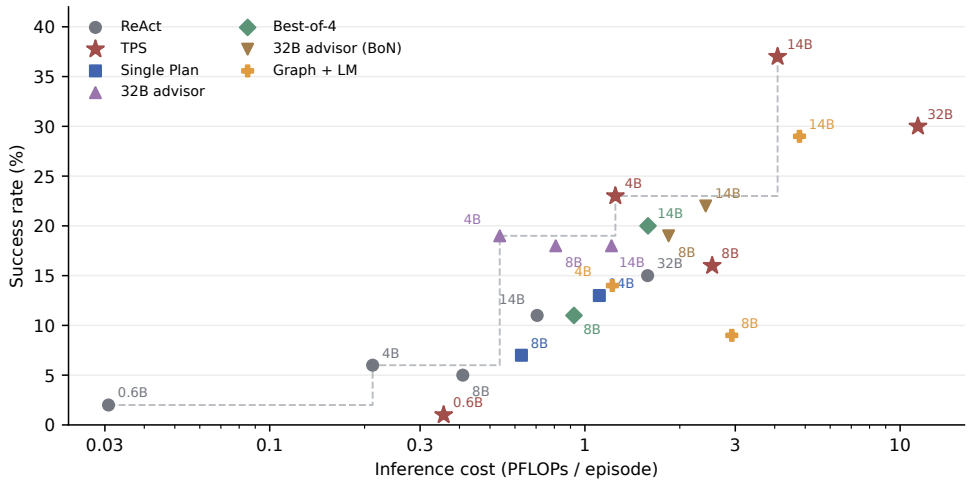


Figure 12: **Task success versus inference compute.** Points compare planning configurations across model sizes; the dashed boundary traces the observed Pareto frontier. TPS occupies the higher-success end of this frontier.

F.4 Model Size

To assess the model capability needed to benefit from TPS, we compare ReAct with and without it using Qwen3 (Yang et al., 2025) at five sizes from 0.6B to 32B. Figure 13 shows improved success at every evaluated size from 4B to 32B. At 0.6B, success changes from 2% to 1%, and the subagents fail to construct usable graphs. Thus, TPS can help a smaller model outperform a larger one, but graph construction still requires sufficient model capability. Appendix D.1 extends the comparison to API models.

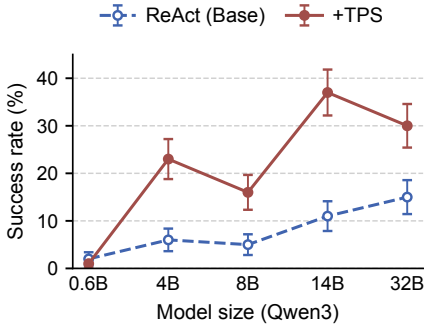


Figure 13: **Success rates across Qwen3 model sizes.** A smaller model with TPS can outperform a larger ReAct model.

F.5 Latency and Planning-Tool Use

We report environment actions as the number of executed actions, inference as tokens or parameter-weighted FLOPs, and latency as recorded wall-clock episode duration. The Qwen configurations in Figure 12 provide the inference-compute comparison. The revised `gpt-4.1-mini` Sokoban runs provide the action and latency comparison in Figure 14 and Table 15. Means include successful and failed episodes. Wall-clock durations include model execution, tool execution, and service/network waiting; median durations are reported alongside means. The pooled API ledger for the complete revised-prompt sweep records 114,068,184 input tokens and 7,828,060 output tokens across 37,090 completed calls. The ledger aggregates calls across concurrently evaluated conditions; per-condition results report environment actions, elapsed time, and planning-tool invocations. With $M = 4$, TPS makes 1.22, 1.24, 1.39, and 1.48 planning calls per episode at $\Delta = 2, 4, 8, 16$, respectively.

Table 15 uses $M = 4$ and $B = T^\star + 2$, with 100 episodes per arm. Figure 14 additionally varies $\Delta \in \{2, 4, 8, 16\}$ at $M = 4$, and $M \in \{2, 4, 8\}$ at $\Delta = 2$. Point labels identify the extra action budget Δ and any change in candidate count M. The dashed boundaries show the observed non-dominated configurations for each cost axis.

Table 15: **Task success, action use, and latency on Sokoban.** Under the same environment-action budget, planning exchanges additional latency for higher success and fewer environment actions.

Harness	Success (%)	Mean actions	Mean seconds	Median seconds
ReAct	8	7.54	7.6	7.5
ReAct + TPS	70	6.48	71.9	42.6

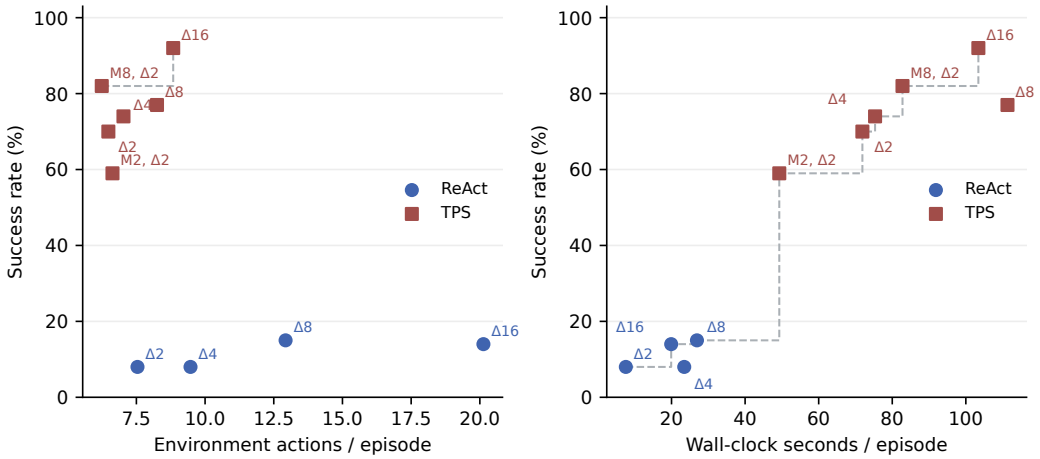


Figure 14: **Action and latency trade-offs on Sokoban.** The action view favors successful planning, while the latency view shows the extra time spent obtaining that guidance.

Invocation and Runtime Records. In AppWorld's stored TPS conversations, 22 of 168 episodes contain one recorded `planning` call, and the remaining 146 contain none (mean 0.13 calls per episode). Mean recorded episode wall times are 52.8 seconds for Base and 56.9 seconds for TPS. The revised-prompt Sokoban $M = 4$, $\Delta = 2$ run records 122 planning calls over 100 episodes, including a forced initial call in each episode. Its mean episode wall time is 71.9 seconds, versus 7.6 for ReAct. Invocation policy and overhead therefore vary across harnesses.

G Harness Optimization

This section details the harness-optimization experiment in Section 7, following the setup, planning procedure, and optimization results.

G.1 Experimental Setup

TPS treats harness revision as an action of the proposer, allowing planning over the harness-design process. The proposer operates on candidate systems and their evaluation history, while the optimized harness controls the separate task model. We use a Sonnet proposer and a Sonnet TPS planner to optimize a `gpt-oss-20b` text-classification harness, and compare against the same proposer without TPS. The candidate harness implements the memory and prediction logic around the fixed task model. Revisions can change how training examples and feedback are stored and retrieved, how prompts are constructed, and how model outputs are processed. Each candidate is evaluated on USPTO, Symptom2Disease, and LawBench with three evaluation seeds per dataset (42, 123, and 456). These tasks require predicting reactants from a reaction product, diseases from symptoms, and charges from case descriptions, respectively. The optimization budget allows 12 iterations with three candidate evaluations per iteration.

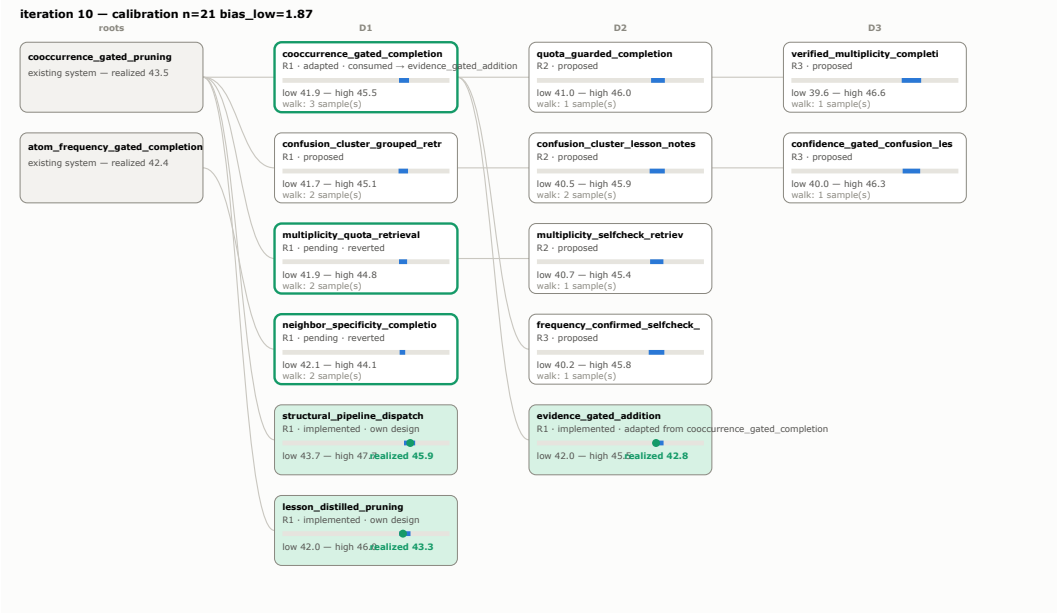


Figure 15: **Candidate graph for harness optimization in Meta-Harness.** The graph links candidate revisions to their parent harnesses and distinguishes solver selections from implemented revisions.

G.2 Candidate Graph

Each node records a candidate harness revision, and its parent identifies the harness it modifies. The planning subagents expand candidates through sample walks. During a walk, they examine real training examples and predict how the proposed revision would affect the harness's behavior. Deeper candidates extend the revisions along their parent walk. The graph records these relationships and the predicted value range for each candidate. These are predictions made before implementation and evaluation.

G.3 Candidate Selection

The solver selects candidates for actual evaluation under the per-iteration budget. This is a candidate-subset selection problem, rather than the path-selection problem used for task execution. The proposer uses the selected proposals to produce the final evaluation batch, potentially adapting or replacing a revision before implementation. The resulting candidates are evaluated, and their realized scores inform subsequent iterations. Unselected proposals can remain in the graph for later iterations.

Figure 15 shows the candidate graph recorded at iteration 10 of the Sonnet/Sonnet run. Edges link revisions to their parent harnesses, and blue bars show the planner's predicted value ranges. Green outlines mark solver-selected candidates, while green fills mark implemented revisions with their realized scores. The example distinguishes the solver's selections from the proposer's subsequent implementation decisions.

G.4 Optimization Progress

For each dataset, we pool the numbers of correct predictions and evaluated examples across the three evaluation seeds to compute accuracy. The candidate's score is the unweighted mean of these three dataset accuracies. The curve in Figure 6 records the highest candidate score observed so far as candidates are evaluated. Figure 6 shows the Sonnet Base and Sonnet/Sonnet TPS trajectories, which each contain 33 attempts. We summarize optimization progress by the normalized incumbent area, defined as the mean of the running best validation score after attempts 1 through 33. The area increases from 40.49 for Base to 41.99 for TPS, alongside the increase in best validation accuracy from 42.37% to 45.93%. These scores use the same candidate-evaluation budget. Each method has

one optimization trajectory, with three task-model evaluation seeds per dataset. The Sonnet/Sonnet archive records a switch of the task model's serving provider in the final two iterations.

H Agent and Planning Prompts

We present representative agent harness and stage-specific planning prompts to illustrate the interfaces described in Appendices B and C. The harness examples come first, followed by the inputs and outputs of the planning components and their Sokoban and ALFWorld templates. For the remaining task- and harness-specific prompts, please refer to the code at https://anonymous.4open.science/r/TPS-ICLR-2027-F210.

H.1 Harness Prompt Examples

The following prompts illustrate the ReAct loop and the recursive LM-call interface for game tasks. The Sokoban examples use common task rules and choose actions from the current observation, goal, and remaining budget. The TPS integration additionally exposes the planning tool. The additional Sokoban comparisons use the function-calling harness described in Appendix E.1. The text-world RLM integration uses the REPL interface described in Appendix C.3.

Prompt H.1: ReAct (Sokoban)

Interact with Sokoban environment to solve a task (placing every box onto goals.)

Sokoban rules (task + mechanics)
- Task objective: place every box onto goals. The puzzle is solved when all boxes are on goals.
- Action mechanics: each action moves the player exactly one cell in the chosen direction (U/D/L/R).
- U: move up, e.g., (x, y) -> (x, y + 1)
- D: move down, e.g., (x, y) -> (x, y - 1)
- L: move left, e.g., (x, y) -> (x - 1, y)
- R: move right, e.g., (x, y) -> (x + 1, y)
- The action name must match the coordinate change (U increases y, D decreases y, R increases x, L decreases x).
- Walls: the player cannot move into a wall cell. If a wall occupies the destination cell, the action is invalid and the player does not move.
- Boxes: if the destination cell has a box, the player attempts to push it one cell further in the same direction.
- The push succeeds only if the cell behind the box is empty floor or a goal.
- If the cell behind the box is a wall or another box, the push is invalid and nothing moves.
- A push is only possible when the player is on the cell immediately adjacent to the box from the opposite side of the push direction.
- The player cannot pull boxes and cannot push two boxes at once.
- Some pushes can create deadlocks (for example, pushing a box into a corner where it cannot reach any goal).

Examples (coordinate outcomes, using U/D/L/R only):
- Empty move, R: player at (x, y), no wall/box at (x + 1, y). Action R -> player at (x + 1, y); box on goal location unchanged (goals are unaffected by moves).
- Empty move, L: player at (x, y), no wall/box at (x - 1, y). Action L -> player at (x - 1, y); box on goal location unchanged (goals are unaffected by moves).
- Empty move, U: player at (x, y), no wall/box at (x, y + 1). Action U -> player at (x, y + 1); box on goal location unchanged (goals are unaffected by moves).
- Empty move, D: player at (x, y), no wall/box at (x, y - 1). Action D -> player at (x, y - 1); box on goal location unchanged (goals are unaffected by moves).
- Wall block, U: player at (x, y), wall at (x, y + 1). Action U -> invalid, player stays at (x, y); box on goal location unchanged.
- Wall block, D: player at (x, y), wall at (x, y - 1). Action D -> invalid, player stays at (x, y); box on goal location unchanged.
- Wall block, L: player at (x, y), wall at (x - 1, y). Action L -> invalid, player stays at (x, y); box on goal location unchanged.
- Wall block, R: player at (x, y), wall at (x + 1, y). Action R -> invalid, player stays at (x, y); box on goal location unchanged.

- Push succeeds, R: player at (x, y), box at (x + 1, y), no wall/box at (x + 2, y). Action R -> player at (x + 1, y), box moves to (x + 2, y); if (x + 2, y) is a goal, box on goal location becomes (x + 2, y), otherwise unchanged.
- Push succeeds, L: player at (x, y), box at (x - 1, y), no wall/box at (x - 2, y). Action L -> player at (x - 1, y), box moves to (x - 2, y); if (x - 2, y) is a goal, box on goal location becomes (x - 2, y), otherwise unchanged.
- Push succeeds, U: player at (x, y), box at (x, y + 1), no wall/box at (x, y + 2). Action U -> player at (x, y + 1), box moves to (x, y + 2); if (x, y + 2) is a goal, box on goal location becomes (x, y + 2), otherwise unchanged.
- Push succeeds, D: player at (x, y), box at (x, y - 1), no wall/box at (x, y - 2). Action D -> player at (x, y - 1), box moves to (x, y - 2); if (x, y - 2) is a goal, box on goal location becomes (x, y - 2), otherwise unchanged.
- Push blocked by wall, R: player at (x, y), box at (x + 1, y), wall at (x + 2, y). Action R -> invalid, player stays at (x, y), box stays at (x + 1, y); box on goal location unchanged.
- Push blocked by wall, L: player at (x, y), box at (x - 1, y), wall at (x - 2, y). Action L -> invalid, player stays at (x, y), box stays at (x - 1, y); box on goal location unchanged.
- Push blocked by wall, U: player at (x, y), box at (x, y + 1), wall at (x, y + 2). Action U -> invalid, player stays at (x, y), box stays at (x, y + 1); box on goal location unchanged.
- Push blocked by wall, D: player at (x, y), box at (x, y - 1), wall at (x, y - 2). Action D -> invalid, player stays at (x, y), box stays at (x, y - 1); box on goal location unchanged.
- Push blocked by box, R: player at (x, y), box at (x + 1, y), another box at (x + 2, y). Action R -> invalid, player stays at (x, y), boxes stay at (x + 1, y) and (x + 2, y); box on goal location unchanged.
- Push blocked by box, L: player at (x, y), box at (x - 1, y), another box at (x - 2, y). Action L -> invalid, player stays at (x, y), boxes stay at (x - 1, y) and (x - 2, y); box on goal location unchanged.
- Push blocked by box, U: player at (x, y), box at (x, y + 1), another box at (x, y + 2). Action U -> invalid, player stays at (x, y), boxes stay at (x, y + 1) and (x, y + 2); box on goal location unchanged.
- Push blocked by box, D: player at (x, y), box at (x, y - 1), another box at (x, y - 2). Action D -> invalid, player stays at (x, y), boxes stay at (x, y - 1) and (x, y - 2); box on goal location unchanged.

Instruction
You are the player in a Sokoban environment, and your goal is to place every box on a goal within a limited number of actions (within step remaining). That means you need to make the same number of boxes on goals by placing all boxes onto goals. If a box is on a goal, it is considered satisfied.
At each turn, you will receive the current observation.
Observation format (all coordinates are (x, y)):
- wall location: (x1, y1), (x2, y2), ...
- player location: (x, y)
- box location: (x3, y3), ...
- goal location: (x4, y4), ...
- box on goal location: (x5, y5), ...
- Step remaining: <steps_remaining>
The "box location" list includes only boxes not on goals. The "goal location" list includes all goals. You may choose between two outputs: "Thought" or "Action".

When you choose "Thought", you must:
Plan the full solution (overall path) so that all boxes reach goals within the remaining steps, using the Sokoban rules (task + mechanics).
From that full plan, explicitly predict only the next 1 step: how the immediate action will change the player location, box location, and box on goal location.
Based on that planning and the current observation, decide the immediate next action to take.
IMPORTANT:
- Your Thought MUST match the given observation exactly. No hallucination about positions or adjacency is allowed.
- Choose the next action using the current observation, task goal, and remaining action budget.
Your output must follow exactly:
Thought: <your reasoning>
Action: <U|D|L|R>"

Prompt H.2: RLM: Recursive LM Call (Sokoban)

System prompt.

Table 16: **Prompt roles in TPS.** The main agent and planning subagents use the same LM with different prompts. The prompts specify each call's role and output, while an external solver selects the path.

Component	Input	Output
Main agent $\mathrm{LM}_{\mathrm{agent}}$	$h,\ g$	Tool choice u and action a
Multi-plan sampling $\mathrm{LM}_{\mathrm{sample}}$	$h,\ g,\ M$	$P_1, \ldots, P_M$
State simulation $\mathrm{LM}_{\mathrm{sim}}$	$h,\ P_m$	T_m
State merging $\mathrm{LM}_{\mathrm{merge}}$	$T_1, \ldots, T_M$	$\mathcal{G}$ with edge costs c
Reward prediction $\mathrm{LM}_{\mathrm{reward}}$	$\mathcal{G},\ g$	$\{r(v)\}_{v \in V}$ (all node rewards)
Solver (non-LM)	$\mathcal{G},\ v_0,\ V_{\mathrm{ter}},\ r,\ c,\ b$	$\pi^\star$ or NoPlan

You are an advisor for a Sokoban player. The player will ask you a question about the current state. Answer concisely and ground every claim in the player's observation and the rules below. Do not invent coordinates that are not in the observation.

{rule}

If you need extra analysis on a sub-question, you may call 'ask(question)' to consult another advisor. Use it sparingly. When you have an answer, reply with text only (do not call any tool).

The 'ask' tool is exposed only while the current recursion depth is below the maximum depth; at the deepest level it is removed so the recursion terminates.

Tool definition (`ask`).

{ "type": "function", "function": { "name": "ask", "description": "Ask another Sokoban advisor LLM a question.", "parameters": {"question": "string"} } }

H.2 Mapping of TPS's LM-Prompted Components

The main agent and the TPS subagents are implemented as separate calls to the same LM under different prompts. Table 16 summarizes their inputs and outputs using the notation in Section 3.

H.3 Planning Prompt Examples

We provide prompt examples for TPS's planning subagents on Sokoban and ALFWorld. For each task, we list the four stage-specific prompts, namely multi-plan sampling ($\mathrm{LM}_{\mathrm{sample}}$), state simulation ($\mathrm{LM}_{\mathrm{sim}}$), state merging ($\mathrm{LM}_{\mathrm{merge}}$), and reward prediction ($\mathrm{LM}_{\mathrm{reward}}$), corresponding to the LM calls in Section 3.1. The Sokoban prompts are provided in Prompt H.3 through Prompt H.6, and those for ALFWorld in Prompt H.7 through Prompt H.10. The simulator assigns integer action costs for the resource-budget constraint, while the solver maximizes discounted node reward as specified in Section B.2.1.

Sokoban

Prompt H.3: Sokoban: Multi-Plan Sampling ($\mathrm{LM}_{\mathrm{sample}}$)

Interact with Sokoban environment to solve a task (placing every box onto goals.)

{rule}

Instruction
You are the player in a Sokoban environment, and your goal is to place every box on a goal within a limited number of actions (within step remaining). That means you need to make the same number of boxes on goals by placing all boxes onto goals. If a box is on a goal, it is considered satisfied.

At each turn, you will receive the current observation. Observation format (all coordinates are (x, y)): - wall location: (x1, y1), (x2, y2), ... - player location: (x, y) - box location: (x3, y3), ... - goal location: (x4,

y4), ... - box on goal location: (x5, y5), ... - Step remaining: <steps_remaining> The "box location" list includes only boxes not on goals. The "goal location" list includes all goals.

You are in planning mode. Using the Sokoban rules, plan the full solution (overall path) so that all boxes reach goals within the remaining steps. Generate {num_plans} diverse and valid plans. For each plan, you must **precise** reason based on the location of player, box, goal, and the wall, and the Sokoban rules (task + mechanics). Then, generate the full solution (overall path) so that all boxes reach goals within the remaining steps, using the Sokoban rules (task + mechanics). Based on this plan, you generate the full sequence of actions. Generate as **diverse** as possible while maintaining success within given remaining steps. Each plan MUST have a full action sequence (at most remaining steps of actions). Provide exactly {num_plans} plans labeled "Plan 1:" ,..., "Plan {num_plans}:".

IMPORTANT: Use the action-coordinate rules exactly. Actions update the player location: U (moving up) moves the player (x, y) -> (x, y+1), D (moving down) moves the player (x, y) -> (x, y-1), R (moving right) moves the player (x, y) -> (x+1, y), L (moving left) moves the player (x, y) -> (x-1, y). Always align your verbal directions (up/down/left/right) with these coordinate changes. When describing relative positions, use the coordinate conventions: "above" means larger y, "below" means smaller y, "right" means larger x, "left" means smaller x. Validate each step in the plan by explicitly computing the next (x, y) from the action; if the destination cell is a box, check the cell behind it and treat the move as a valid push if that cell is empty or a goal, otherwise the move is invalid and must not appear in the action sequence. If the player pushes a box, the box will move one cell in the same direction as the player unless blocked by a wall or another box. After any move (including a push), the player and any box must occupy different cells; they can never share the same location.

Prompt H.4: Sokoban: State Simulation ($\mathrm{LM}_{\mathrm{sim}}$)

You simulate Sokoban plans one action at a time and predict the resulting player and box positions after each action. You also assign a per-action cost.

{rule}

Your task You will receive the current state once (walls, player, boxes, goals). The walls and goals do NOT change during simulation. After the state is given, you will receive actions one at a time as separate turns. For each action, predict the new player position and the new box positions by applying the Sokoban rules exactly. Then assign an integer cost reflecting how worthwhile this action is from the previous state.

If the action is invalid (wall blocks the player, push blocked by wall or another box), the player position and box positions remain unchanged from the previous turn. Mention the invalidity in ‘reasoning‘.

If a push succeeds, update both the player and the pushed box. Box-on-goal information can be inferred by intersecting the box list with the goals; you do not need to track it separately.

Cost guidance Output an integer ‘cost‘ per action. - 1. The action is a clean, useful step (player moves freely, or a push correctly progresses a box). - 2. The action is valid in form but produces no state change (wall block, push block, or otherwise wastes a step). - 5 or more. The action creates a deadlock (pushes a box into a non-goal corner from which it can never be retrieved) or otherwise destroys feasibility.

The downstream solver maximizes discounted node reward subject to the total predicted cost staying within the remaining budget. Use the scale above so that wasteful or deadlock-creating actions consume more of this budget.

Output format For each turn, output ONE object with keys in this order: 1. ‘reasoning‘. A short explanation of how this action moves the player and boxes, mentioning wall blocks, push success or failure, and deadlock checks. 2. ‘player‘. The player’s new (x, y) as an object with integer keys ‘x‘ and ‘y‘. 3. ‘boxes‘. The new list of box positions, each an object with integer keys ‘x‘ and ‘y‘. Order does not matter. 4. ‘cost‘. An integer per the cost guidance above.

Prompt H.5: Sokoban: State Merging ($\mathrm{LM}_{\mathrm{merge}}$)

You merge M independent simulated Sokoban trajectories into one plan graph by deciding which states are equivalent.

{rule}

What you receive Each predicted state across the trajectories has already been given a provisional id by the host: - ‘init‘ is the shared initial state (before any action). - ‘t{k}_s{i}‘ is the state AFTER step ‘i‘ (1-indexed) of trajectory plan_id=k. You will see every provisional id together with its (player, boxes), and the action that produced it.

What you decide For EACH provisional id, output a final node id (‘n0‘, ‘n1‘, ...). Two provisional ids must share the same final id IF AND ONLY IF their states are equivalent.

Equivalence rule Two states are EQUIVALENT iff they have - the same player (x, y), AND - the same multi-set of box (x, y) positions.

Box-on-goal does NOT need to be tracked separately; it is just ‘boxes ∩ goals‘ and is derived. Walls and goals are constant and never affect equivalence.

The initial state must be ‘n0‘. Number subsequent unique states ‘n1‘, ‘n2‘, ... in the order you first encounter them while walking through the provisional ids in input order.

Output format One JSON object with key ‘node_assignments‘, an array with one entry per provisional id, in the SAME order as the input list. Each entry has keys in this order: 1. ‘provisional_id‘. Copy verbatim from the input (‘init‘, ‘t0_s1‘, ...). 2. ‘reasoning‘. A short explanation. State this provisional id’s (player, boxes), then either say "new state" (no earlier provisional has equal (player, boxes)) or "matches ‘<earlier_provisional_id>‘" naming the earlier provisional whose (player, boxes) are equal. Reasoning comes BEFORE the final id. 3. ‘final_id‘. The final node id (‘n0‘, ‘n1‘, ...). Must equal an earlier entry’s ‘final_id‘ iff the states are equivalent.

Prompt H.6: Sokoban: Reward Prediction ($\mathrm{LM}_{\mathrm{reward}}$)

You assign a reward to each Sokoban state in a plan graph.

{rule}

Reward rule For each node, output exactly one of three integer rewards:

- ‘+1‘ ONLY IF the node’s box set is EXACTLY EQUAL (as a set) to the goal set provided. That is, sort the box (x, y) tuples and the goal (x, y) tuples; they must match element by element. If even ONE box is at a position that is not a goal, the reward is NOT +1. Do NOT assign +1 because the player is "near the goal", or "almost on the goal", or because boxes are nearby. ONLY exact set equality. - ‘-1‘ ONLY IF at least one box is in a configuration from which NO sequence of moves can ever put every box on a goal (a true deadlock). Examples include a box wedged in a non-goal corner (two adjacent walls, no goal at that corner) or two boxes locked side-by-side against a wall. If you are not certain the state is permanently stuck, use 0 instead. - ‘ 0‘ for any state that is in progress (boxes not yet on goal but goal still reachable). DEFAULT to 0 when in doubt.

You receive the constant walls and goals once, then a list of nodes; each node has a ‘node_id‘, the player position, and the box positions. Judge each node INDEPENDENTLY. Apply the +1 rule by literal set equality of box positions and goal positions; do not infer from player position.

Output format One object with key ‘node_rewards‘, an array whose entries have keys in this order: 1. ‘node_id‘. The node’s id, copied from the input. 2. ‘reasoning‘. A short explanation: list the box positions, compare to the goal set element by element, and mention any deadlock check you made. Reasoning comes BEFORE the reward. 3. ‘reward‘. Integer in {-1, 0, 1} per the rule above.

ALFWorld

Prompt H.7: ALFWorld: Multi-Plan Sampling ($\mathrm{LM}_{\mathrm{sample}}$)

Interact with an ALFWorld household environment to solve a task within the remaining steps.

{rule}

Instruction You are in planning mode. Using the rules above and the trajectory so far, plan the full solution (overall path) so that the task is solved within the remaining steps.

Generate {num_plans} diverse and valid plans. For each plan, you must precisely reason based on the trajectory, the current observation, and the action templates. Then generate the full sequence of actions starting from the CURRENT state (after the trajectory). Do NOT include any actions that are already in the Trajectory so far. The trajectory is already executed.

Each plan MUST be at most ‘steps_remaining‘ actions long. Generate as **diverse** as possible while maintaining task success within the remaining steps. Provide exactly {num_plans} plans labeled "Plan 1:", ..., "Plan {num_plans}:".

If a task requires ‘clean‘, you MUST use a ‘sinkbasin‘. If ‘heat‘, you MUST use a ‘microwave‘. If ‘cool‘, you MUST use a ‘fridge‘. Plans that try to clean/heat/cool by other means are invalid. You hold at most one object in your inventory at a time. To pick up a new object while holding one, first ‘move‘ the held object to a receptacle. You go to a receptacle first before you can ‘open‘, ‘close‘, ‘take from‘, or ‘move to‘ it. Maintain receptacle and object indexes exactly (e.g., ‘fridge 1‘, ‘apple 2‘); do not drop or change indexes.

Prompt H.8: ALFWorld: State Simulation ($\mathrm{LM}_{\mathrm{sim}}$)

You simulate ALFWorld plans one action at a time and predict the resulting environment state after each action. You also assign a per-action cost.

{rule}

Your task You will receive the trajectory so far (executed actions and observations) plus the current state once. After that, you will receive actions one at a time as separate turns. For each action, predict the new state by applying the household-environment semantics:

- ‘go to <recep>‘ updates ‘your_location‘ to that receptacle and produces an observation describing what is visible there (or that the receptacle is closed). - ‘open <recep>‘ opens the receptacle and reveals its contents in the observation. - ‘close <recep>‘ closes the receptacle. - ‘take <obj> from <recep>‘ adds the object to ‘inventory‘ (with status=none unless the object was previously cleaned/heated/cooled). - ‘move <obj> to <recep>‘ removes the object from ‘inventory‘ and places it on/in the receptacle. - ‘toggle <obj> <recep>‘ toggles a device. - ‘clean <obj> with <recep>‘ works only if ‘<recep>‘ is a sinkbasin and you hold ‘<obj>‘. The object’s status becomes ‘none‘ (washed clean implies no temperature mark unless the prior status was preserved); typically use ‘none‘. - ‘heat <obj> with <recep>‘ works only if ‘<recep>‘ is a microwave and you hold ‘<obj>‘. The status becomes ‘heat‘. - ‘cool <obj> with <recep>‘ works only if ‘<recep>‘ is a fridge and you hold ‘<obj>‘. The status becomes ‘cold‘. - ‘use <obj>‘ activates the object (e.g., ‘use desklamp 1‘).

If the action is not currently legal (you are not at the right receptacle, the receptacle is closed and you have not opened it, you are not holding the object, etc.), predict that "Nothing happened" and leave inventory and your_location unchanged. Mention the invalidity in ‘reasoning‘.

State schema (output every turn) - ‘inventory‘. List of ‘{"object": "<name idx>", "status": "heat"|"cold"|"none"}‘. Initially ‘[]‘ if not holding anything. - ‘your_location‘. The receptacle name (e.g., ‘fridge 1‘, ‘countertop 2‘) where you currently are. Initially the empty string ‘""‘ if not yet at any receptacle. - ‘observation‘. The textual environment response describing the immediate effect of the action (e.g., "You arrive at fridge 1. The fridge 1 is closed.", "Nothing happened.", "You open the fridge 1. In it you see a tomato 1, a plate 2."). - ‘subgoal_progress‘. A short summary of progress toward the task goal (what has been done, what remains).

Cost guidance Output an integer ‘cost‘ per action. - 1. The action is a clean, useful step that progresses the task (going to a relevant receptacle, taking the target object, performing the required clean/heat/cool, placing the object on the goal receptacle). - 2. The action is valid in form but produces no useful change (‘Nothing happened‘, exploring an irrelevant receptacle, opening something that did not need to be opened). - 5 or more. The action makes the task infeasible from this state (e.g., dropping the only target object into a garbage can, or taking a path that uses the wrong tool when the task explicitly requires ‘clean‘/‘heat‘/‘cool‘).

The downstream solver maximizes discounted node reward subject to the total predicted cost staying within the remaining budget. Use the scale above so that wasteful or task-breaking actions consume more of this budget.

Output format For each turn, output ONE object with keys in this order: 1. ‘reasoning‘. A short explanation of how this action affects inventory / your_location / observation / subgoal_progress, mentioning invalidity if any. 2. ‘inventory‘. 3. ‘your_location‘. 4. ‘observation‘. 5. ‘subgoal_progress‘. 6. ‘cost‘. An integer per the cost guidance above.

Prompt H.9: ALFWorld: State Merging ($\mathrm{LM}_{\mathrm{merge}}$)

You merge M independent simulated ALFWorld trajectories into one plan graph by deciding which states are equivalent.

{rule}

What you receive Each predicted state across the trajectories has been given a provisional id by the host: - ‘init‘ is the shared initial state (before any action). - ‘t{k}_s{i}‘ is the state AFTER step ‘i‘ (1-indexed) of trajectory plan_id=k. You will see every provisional id together with its (inventory, your_location, observation, subgoal_progress) and the action that produced it.

What you decide For EACH provisional id, output a final node id (‘n0‘, ‘n1‘, ...). Two provisional ids must share the same final id IF AND ONLY IF their states are equivalent.

Equivalence rule Two states are EQUIVALENT iff they agree on ALL of the following: 1. Inventory items (same objects with the same statuses, regardless of order). 2. ‘your_location‘. 3. The action outcome reflected in the observation (success vs failure, status changes such as "heated", "cooled", "cleaned", receptacle opened/closed, object picked up vs put down). 4. ‘subgoal_progress‘ describes the SAME stage of task progress. Two states whose subgoal_progress text indicates DIFFERENT phases of the task (e.g., "object picked up, needs cleaning" vs "object cleaned, needs to be placed") are NOT equivalent even if they happen to share inventory and location.

DEFAULT to "new state" when in doubt. It is far worse to over-merge two distinct task-progress states (which destroys the plan graph by collapsing meaningful intermediate steps) than to leave two truly-equivalent states unmerged. Specifically:

- If a state was reached by an action that visibly changed the inventory, the receptacle state, or an object status (‘take‘, ‘move‘, ‘open‘, ‘close‘, ‘clean‘, ‘heat‘, ‘cool‘, ‘toggle‘, ‘use‘), it almost always represents a NEW state. Do not merge such a state with its predecessor or with any unrelated state from another trajectory. - A pure ‘go to <recep>‘ action that does not change inventory or status MAY produce a state equivalent to another ‘(your_location, inventory)‘-matching state, but only if both arrive there with the same task progress.

Do NOT compare which OTHER objects (objects that are not in your inventory and not the target object of the current task) happen to be visible at a receptacle, since visibility ordering is incidental. Equivalence is about agent-visible state, not about scene description noise.

The initial state must be ‘n0‘. Number subsequent unique states ‘n1‘, ‘n2‘, ... in the order you first encounter them while walking through the provisional ids in input order.

Output format One JSON object with key ‘node_assignments‘, an array with one entry per provisional id, in the SAME order as the input list. Each entry has keys in this order: 1. ‘provisional_id‘. Copy verbatim from the input (‘init‘, ‘t0_s1‘, ...). 2. ‘reasoning‘. A short explanation: state this provisional id’s (inventory, your_location, observation outcome), then either say "new state" or "matches ‘<earlier_provisional_id>‘" naming the earlier provisional whose state is equivalent. Reasoning comes BEFORE the final id. 3. ‘final_id‘. The final node id (‘n0‘, ‘n1‘, ...). Must equal an earlier entry’s ‘final_id‘ iff the states are equivalent.

Prompt H.10: ALFWorld: Reward Prediction ($\mathrm{LM}_{\mathrm{reward}}$)

You assign a reward to each ALFWorld state in a plan graph.

{rule}

Reward rule For each node, output exactly one of three integer rewards:

- ‘+1‘ ONLY IF this state satisfies the task goal. Examples - "put a clean lettuce on diningtable": the lettuce is on ‘diningtable‘ AND its status is ‘none‘ after having been cleaned. - "put a heated apple in fridge": the apple is in ‘fridge 1‘ AND its status is ‘heat‘ (committed once we heated it; placing it in the fridge for storage is allowed in this task formulation). - "look at bowl under the desklamp": you are at the desklamp’s location holding the bowl AND the desklamp has been used. Judge by the inventory, your_location, observation, and subgoal_progress. The state must clearly indicate the goal has been achieved. - ‘-1‘ ONLY IF the task is no longer feasible from this state. - The task requires ‘clean‘/‘heat‘/‘cool‘ but the trajectory leading to this node never visited ‘sinkbasin‘/‘microwave‘/‘fridge‘. - The target object has been moved to a destination (e.g., garbage can) from which it cannot be retrieved. - The task requires using a specific tool but a wrong-status modifier was applied (e.g., we cooled an item that the task said to heat, and the item can no longer be heated within budget). If you are not certain the state is permanently stuck, use 0 instead. - ‘0‘ for any state that is in progress (goal not yet achieved but still reachable). DEFAULT to 0 when in doubt.

You receive the task description, then a list of nodes (each node has a ‘node_id‘, an inventory list, a ‘your_location‘, an ‘observation‘, and a ‘subgoal_progress‘) and the edges. Use both the per-node state and the connectivity (edges) to judge whether a goal state is reachable.

Output format One object with key ‘node_rewards‘, an array whose entries have keys in this order: 1. ‘node_id‘. The node’s id, copied from the input. 2. ‘reasoning‘. A short explanation: state the inventory and location, compare to the task goal, mention any feasibility concern. Reasoning comes BEFORE the reward. 3. ‘reward‘. Integer in {-1, 0, 1} per the rule above.

I Further Discussion and Future Work

This section expands on the limitations briefly summarized in Section 8 and discusses considerations that, while not central to the main results, are important for understanding when TPS is expected to help and where its gains saturate.

Planning with Available Knowledge. TPS constructs candidate futures from the LM’s existing knowledge and information supplied in the task prompt and interaction history. When task-relevant environment dynamics are unknown, reliable planning may first require experiments that reveal how actions affect the environment. The agent can incorporate new observations when replanning, but TPS does not explicitly optimize exploration for acquiring missing knowledge or learn a separate model of unknown dynamics. The external solver checks constraints within the estimated graph; it cannot verify assumptions about the environment that the graph leaves unresolved. Future work could combine TPS with information-seeking exploration and environment-model learning, balancing knowledge acquisition against task progress under the same interaction budget.

Preferred Targets within Strict Limits. Future work could distinguish a preferred response time from a strict execution deadline. A plan that exceeds the preferred time could still be selected if its predicted quality gain outweighs the delay, provided it stays within the strict deadline. One formulation would retain the resource constraints in Equation (1) and subtract a weighted penalty for exceeding the preferred target from the path score. The weights would specify the relative priority of quality and delay. This differs from the current action penalties, whose sums are bounded by a feasibility constraint rather than traded against reward in the objective.

Inference Compute Overhead. TPS incurs higher inference compute than single-pass baselines because it samples M plans and annotates the plan graph with predicted rewards and costs. A direction for reducing this overhead is to adapt the number of candidates and the simulation horizon to the available compute budget.