DO LLM AGENTS EXECUTE THE PLANS THEY DECLARE? FROM PLANNING-MODE DECLARATION TO PATTERN-SPECIFIC EXECUTION

Anonymous authors
Paper under double-blind review

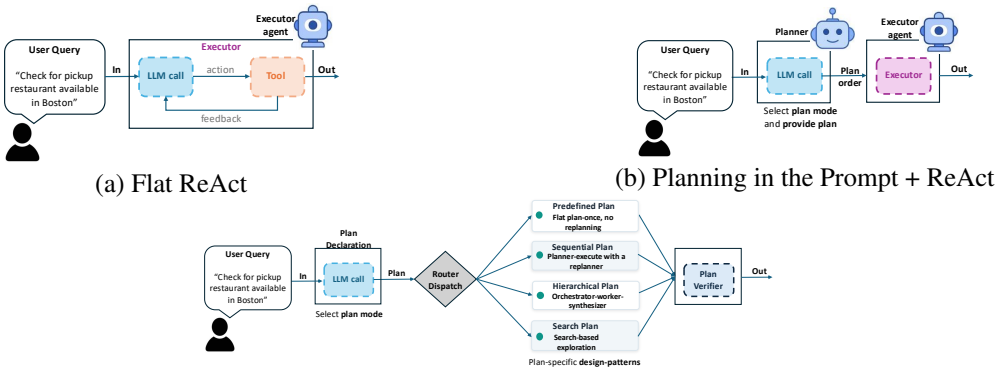
ABSTRACT

Large language models (LLMs) enable agents to solve long-horizon tasks by generating a plan and then executing it in an environment. However, successful planning requires two distinct capabilities: selecting an appropriate plan for the task and executing it faithfully. Existing planner–executor systems can fail at either stage: the agent may deviate from a structured plan during execution, or it may faithfully execute a plan that is poorly matched to the task or environment. Final task success alone cannot distinguish between these two sources of failure. We therefore ask: can LLM agents be trusted to execute the plans they commit to, and do different tasks and environments benefit from different planning modes? To study this, we introduce a diagnostic framework for the *Plan Declaration–Execution Gap* and propose PLANNING-AS-ROUTING, in which the LLM declares one of four planning modes: *Predefined, Sequential, Hierarchical, or Search*. A deterministic router sends the task to the corresponding pattern-specific executor. Across four benchmarks and three LLMs, we find three consistent patterns. First, generic Plan+ReAct often fails to preserve the declared planning structure, especially for longer plans: across three benchmarks, only 22–45% of trajectories preserve it, whereas pattern-specific executors enforce the intended structure. Second, planning-mode effectiveness varies across environments and models: Search performs best on ALFWorld, Hierarchical on SWE-bench, and the strongest pattern can vary across models within the same benchmark. Third, the largest gains come from plan execution: pattern-specific executors raise task success from 0.48 to 0.92 on ALFWorld and from 0.36 to 0.44 on SWE-bench Verified over Plan+ReAct. However, current LLMs do not reliably select the strongest planning mode for a task, while few-shot examples can improve planning-mode selection for some benchmark–model combinations. Overall, reliable agent planning requires both selecting an effective planning mode and executing it with a matching executor: routing substantially closes the execution gap, while selecting the right mode for each task remains open.

1 INTRODUCTION

The rapid advancement of large language models (LLMs) has accelerated the development of interactive agents Liu et al. (2025); Wang et al. (2024), which are designed to solve complex real-world tasks through multi-turn interactions with external environments such as web browsing Zhou et al. (2024b); Deng et al. (2023), computer use Xie et al. (2024); Merrill et al. (2026), and embodied task execution Shridhar et al. (2020). To solve such tasks, an agent often needs to decompose a high-level goal into structured steps and perform the necessary actions; planning provides this structure by turning an objective into a course of action that guides the agent toward the goal Yao et al. (2023b). Consequently, planning has become a central component of both agentic frameworks Shen et al. (2023); Webb et al. (2025) and world-model-based approaches that simulate or reason about possible future states before acting Qiao et al. (2024); Maes et al. (2026); Wang et al. (2026). This structure is what sustains goal-directed behavior in complex, multi-step tasks, where later decisions depend on the outcomes of earlier actions. However, generating a coherent plan is not sufficient for reliable agent behavior. A plan can fail at two levels. At the execution level, the agent may declare

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.



(a) Flat ReAct

(b) Planning in the Prompt + ReAct

(c) Planning as Routing

Figure 1: Overview of the three execution conditions. (a) **Flat ReAct**: a single ReAct-style executor selects actions from environment feedback without an explicit plan. (b) **Planner–Executor (Plan+ReAct)**: a planner first selects a planning mode and produces a plan, which is then passed to a generic ReAct executor; however, the declared plan is not structurally enforced during execution. (c) **Planning-as-Routing**: the LLM first declares a planning mode, and a deterministic router dispatches the task to a matching pattern-specific executor: predefined, sequential, hierarchical, or search. A verifier checks whether execution preserves the declared plan structure.

one plan but deviate from it during execution. At the selection level, the declared plan itself may be poorly matched to the task: even faithful execution can fail if the chosen plan does not fit the environment. *This raises two fundamental questions: when an agent proposes a plan for solving a task, does it execute the plan it commits to, and is the proposed plan appropriate for the task?*

Existing agent evaluation harnesses offer limited insight into how agents plan and execute their tasks, because they focus on final task success Liu et al. (2024); Zhou et al. (2024b); Mialon et al. (2024); Xie et al. (2024); Pan et al. (2024). Final task success does not reveal whether an agent followed a deliberate plan or reached the goal through an inefficient trajectory, memorization, or chance Liu et al. (2026b); Sun et al. (2026). Recent work has begun to evaluate plan quality and whether agents follow instructed plans Sun et al. (2026); Liu et al. (2026b); Jia et al. (2025). However, these studies largely focus on diagnosing planning behavior rather than jointly examining whether the selected planning mode is appropriate for the task and preserved during execution. They also treat a plan as a sequence of steps to be generated or followed, rather than as a choice among different planning modes. Planning requirements differ across tasks and environments: some tasks can be solved with a fixed plan, whereas others require adaptation, decomposition, or exploration. In this work, we study whether LLM agents select a planning mode for a task, whether that mode is preserved during execution, and whether matching modes to corresponding executors improves task success.

A useful perspective on this problem comes from human planning: humans do not rely on a single fixed strategy for all tasks. A familiar task calls for a routine plan, a structured but unfamiliar one for step-by-step planning, a multi-stage one for decomposition into subgoals, and an uncertain one for weighing alternative routes before acting Daw et al. (2005; 2011); Botvinick (2008); Mattar & Daw (2018); Mattar & Lengyel (2022). This flexibility suggests that planning in LLM agents should not be treated as a uniform behavior. We call a mismatch between the declared plan and its execution the *Plan Declaration–Execution Gap*. This distinction helps distinguish failures of planning-mode selection from failures of plan execution. This motivates the following research questions:

RQ1: As agents take on increasingly long-horizon and multi-step tasks, can we rely on them to execute the planning approach they commit to, or do they fall back to reacting one step at a time? **RQ2:** Do different environments and models benefit from different planning approaches, or is a single planning approach sufficient across web, coding, and embodied tasks? **RQ3:** When an agent declares a planning approach, does executing it through the corresponding planning pattern improve task success compared with passing the same plan to a generic ReAct executor? **RQ4:** When multiple planning approaches are available, can an LLM choose the one that is most appropriate for the task it is trying to solve?

Together, these questions clarify whether agent failures arise from choosing the wrong planning mode or from failing to execute the chosen mode faithfully. To address these questions, we systematically study how LLM agents select and execute different planning approaches, and whether matching a declared approach to its corresponding execution pattern improves task performance. For the purpose of this work, we evaluate agents built on multiple LLM families (Qwen-3.6 Qwen Team (2026), DeepSeek-V4 DeepSeek-AI (2026), Gemma-4 Abd et al. (2026)) across four benchmarks spanning diverse and complex environments: web browsing (WebArena Zhou et al. (2024b), Mind2Web Deng et al. (2023)), software engineering (SWE-Bench-Verified Jimenez et al. (2024)), and embodied navigation (ALFWorld Shridhar et al. (2020)). We compare three conditions that differ by one factor at a time: (i) a no-planning flat ReAct baseline, (ii) a plan-prompted flat ReAct baseline (Plan + ReAct), and (iii) our Planning-as-Routing framework with pattern-specific executors. Figure 1 illustrates these three conditions.

Our contributions are threefold: (1) We introduce a diagnostic framework for measuring the *Plan Declaration–Execution Gap* in LLM agents, and show that a ReAct agent told to plan a certain way often does not follow the declared plan, instead reacting locally to intermediate observations, a divergence invisible to final-success metrics. Our analysis goes beyond task success by measuring plan quality, plan adherence, and structural faithfulness, while separately examining whether the declared planning approach is effective for the task. (2) We introduce *Planning-as-Routing*, an architecture in which the LLM declares a planning mode and a deterministic router sends the task to the corresponding pattern-specific executor: {predefined, sequential, hierarchical, or search}. This design makes the declared planning structure explicit in execution and allows its behavior to be verified from the resulting trajectory. (3) Across web navigation, software engineering, and embodied tasks, and across multiple LLM families, we show that planning pattern effectiveness varies across environments and models, while planning-mode selection remains a key bottleneck: current LLM declarations trail the best fixed planning pattern on every benchmark–model pair. Few-shot demonstrations improve task success through better planning-mode selection, with gains of $+0.004$ to $+0.16$ across benchmarks. We will release the code, baseline and planning trajectories across seeds upon publication of this paper.

2 Related Work

LLMs as Planning Agents. Prior work has proposed many planning mechanisms for LLM agents, but these methods often instantiate planning behavior through a specific agent architecture. Earlier approaches used ReAct-style reasoning and acting Yao et al. (2023b), search over reasoning paths Yao et al. (2023a), symbolic planning Liu et al. (2023), planner–executor architectures Erdogan et al. (2025), multi-agent workflows Shen et al. (2023); Hong et al. (2024), and adaptive replanning Liu et al. (2026a); Dong et al. (2026); Wu et al. (2026). These approaches can generate and revise task-specific plans, but the underlying planning mechanism is typically fixed by the agent framework. Our work instead treats the planning mode as a per-task choice and asks whether the declared mode is preserved during execution.

Agent Evaluation and Trajectory Diagnosis. Agent benchmarks span web navigation Deng et al. (2023); Zhou et al. (2024b); Pan et al. (2024), computer use Xie et al. (2024); Merrill et al. (2026), software engineering Jimenez et al. (2024), embodied tasks Shridhar et al. (2020), and general assistants Mialon et al. (2024), but primarily evaluate final task outcomes. Recent work moves toward process-level evaluation through progress metrics, plan-compliance analysis, trajectory diagnosis, and failure taxonomies Ma et al. (2024); Liu et al. (2026b); Ou et al. (2025); Kong et al. (2025); Cemri et al. (2025). Our work complements these approaches by explicitly modeling the planner–executor handoff, allowing us to distinguish failures of planning-mode selection from failures to preserve the declared mode during execution. Discussion of planning architectures and process-level agent evaluation is provided in Appendix A.

3 Methodology

Planning Patterns. We consider four planning patterns inspired by common forms of human planning (Mattar & Daw, 2018; Mattar & Lengyel, 2022): *predefined*, *sequential*, *hierarchical*, and *search*. These patterns capture different ways an agent can organize task execution. A *predefined* plan follows a fixed plan generated before execution, without replanning. A *sequential* plan ex-

ecutes one step at a time and updates the remaining plan based on intermediate observations. A *hierarchical* plan decomposes the task into subgoals and coordinates their execution through an orchestrator–worker structure. A *search* plan generates multiple candidate plans, executes each candidate independently, and selects the most promising resulting trajectory using a *rubric-based judge*. The workflow of each planning pattern is illustrated in Appendix B, Figs. 2 and 3. Representative plan structures and execution traces for all four planning modes are provided in Appendix E.2.

Problem Formulation. We study planning under three conditions that share the same backbone LLM and action space, differing only in how planning is produced and executed.

Condition 1: Flat ReAct (no explicit plan declaration). The LLM receives only the task description and goal, and acts through a standard flat ReAct loop, one step at a time. No planning strategy is explicitly declared before execution. This condition serves as our implicit-planning baseline, where any planning behavior must emerge locally through the interaction loop.

Condition 2: Plan + ReAct (declared but unenforced). The LLM first selects a planning approach and produces a task-specific plan. The plan is then provided to the same generic ReAct executor used in Condition 1. Because the executor retains a flat step-by-step control loop, the requested planning structure is available as context but is not structurally enforced. This condition tests whether prompting alone is sufficient for the intended planning approach to appear during execution.

Condition 3: Planning as Routing. The LLM first declares one planning mode

$$P \in \{\textit{predefined}, \textit{sequential}, \textit{hierarchical}, \textit{search}\}.$$

A deterministic router maps the declaration to the corresponding pattern-specific executor. Unlike Condition 2, the selected planning structure therefore determines the execution control flow.

Planning-as-Routing components.

(1) Plan declaration. Given a task, the backbone LLM selects one planning mode (P). We evaluate this declaration step under both *zero-shot* and *few-shot* settings. In the *zero-shot setting*, the model selects a mode from the task description alone. In the *few-shot setting*, the declaration prompt additionally includes example tasks paired with planning modes, allowing us to test whether demonstrations improve task-conditioned mode selection.

(2) Router. A deterministic mapping dispatches P to its corresponding executor, with no additional model inference.

(3) Pattern-specific executors. Each executor imposes a distinct planning and control-flow pattern, following established agent-planning architectures. The *predefined* executor generates a complete plan before execution and follows it without replanning, similar to plan-then-solve approaches Wang et al. (2023). The *sequential* executor follows a planner–executor–replanner loop, where the agent executes the current step, observes the outcome, and revises the remaining plan when necessary Sun et al. (2023). The *hierarchical* executor follows an orchestrator–worker structure, where an orchestrator decomposes the task into sub-goals, delegates them to specialized workers, and aggregates their outputs Zhang et al. (2025); Choi et al. (2025). Finally, the *search* executor generates multiple candidate plans, executes each independently, and selects the most promising trajectory using a rubric-based judge, following prior search-based agent planning approaches (Zhou et al., 2023).

(4) Execution-structure verifier. A rule-based verifier checks whether a Plan+ReAct trajectory executes the declared plan steps in order. We first clean the declared steps by removing non-actionable text, then match each remaining step to the corresponding trajectory actions using benchmark-specific rules. Structure is maintained only when all scoreable steps are matched in the declared order, while allowing extra actions between them. Routed runs are not scored this way because their executor dispatch records establish order fidelity by construction. Full matching rules, structural-signature analyses, and null baselines are provided in Appendix E.1.

3.1 Planning Process Metrics

Following prior work Jia et al. (2025), we evaluate *plan adherence* and *plan quality*, together with *plan-order faithfulness*. *Plan adherence* measures whether the executed actions complete the declared plan steps. *Plan quality* measures whether the generated plan is appropriate for the task goal and environment. *Plan-order faithfulness* measures whether the plan steps or subgoals are executed

in their declared order. *Plan quality* and *plan adherence* are empirical metrics, whereas plan-order faithfulness is a structural verification for predefined, sequential, and hierarchical; it is not applicable to search, where candidates represent competing alternatives rather than an ordered sequence.

3.2 Pattern-Ceiling Analysis

Task success alone cannot distinguish whether an executor is incapable of solving a task or whether the declaration module selected an unsuitable planning pattern. To separate execution limitations from planning-mode selection, we use *forced dispatch*: for each benchmark–model pair, every planning mode is executed on every task, bypassing the declaration module. This yields a task–mode success matrix $m_{i,p}$, where $m_{i,p} = 1$ if planning mode (p) solves task (i), and (0) otherwise. From this matrix, we compute each fixed mode's success $S(p)$, the best fixed-mode performance $S^\star = \max_p S(p)$, and a per-task oracle ceiling $\mathrm{DSR}_{\max} = \frac{1}{N}\sum_i \max_p m_{i,p}$. Their difference $H = \mathrm{DSR}_{\max} - S^\star$ measures the available routing headroom.

For a declaration policy π, task success is computed as $\mathrm{TSR}(\pi) = \frac{1}{N}\sum_i m_{i,\pi(i)}$, using the same forced-dispatch matrix without re-running executors. Together with the trajectory verifier, this separates *execution failure* from *selection failure*: whether the declared mode is preserved during execution versus whether the selected mode is effective for the task. Additional controls and estimation details are provided in Appendix J.

4 Experimental Setup

Environments. We evaluate our agent planning strategies across four agent benchmarks spanning web navigation (Mind2Web Deng et al. (2023), WebArena Zhou et al. (2024b)), software engineering (SWE-bench Verified Jimenez et al. (2024)), and embodied tasks (ALFWorld Shridhar et al. (2020)). We additionally group tasks using each benchmark's available task categories to examine whether planning-pattern preferences vary with task type; category definitions and counts are provided in Appendix C Table 7.

Language Models. We evaluate three backbone LLMs: Qwen3.6-35B-A3B Qwen Team (2026), DeepSeek-V4-Flash DeepSeek-AI (2026), and Gemma-4-26B Abd et al. (2026). Each model is evaluated under the same task inputs, tool interfaces, action spaces, and execution budgets across all conditions. The same backbone model is used for plan declaration and execution unless otherwise specified. Model details are provided in Appendix C Table 6.

Repeated runs. We evaluate all baselines, declaration settings, and planning patterns with three random seeds (7, 13, and 42), reporting mean and standard deviation across seeds using the same tasks and inference configurations.

Evaluation Metrics. We report both task- and process-level metrics. Task success follows each benchmark's standard protocol: task success rate (TSR) for ALFWorld and WebArena, TSR and step success rate (SSR) for Mind2Web, and patch success rate (PSR) for SWE-bench Verified. Process metrics include plan quality, plan adherence, and structural faithfulness (Section 3.1). We also measure execution cost through environment interactions, LLM calls, generated tokens (thinking vs. content), and completed trajectories to control for differences in inference and interaction budget.

Inference Settings. For each model, decoding parameters, thinking settings, tool-calling policies, and generation budgets are fixed across conditions for each model. Full inference and sampling configurations are provided in Appendix D.

Development and tuning protocol. The four planning-pattern executors, their prompts, and the declaration prompt were fixed before the final evaluation runs, and the same implementations and routing rules are applied to every task within a benchmark. The benchmark tasks were used for inference only; no model parameters were trained or fitted on them. For few-shot declaration, demonstration tasks are excluded from scoring. The null baselines and permutation tests were specified after the main results and are reported as post-hoc analyses.

Plan verification and quality judging. Structural fidelity is measured with a rule-based verifier that matches declared plan steps to executed actions and requires all scoreable steps to be preserved in order; we validate it against two independent human annotators, with full matching and agreement

Table 1: Plan-structure maintenance under generic Plan+ReAct, overall and by declared plan length. *Overall* is the percentage of trajectories preserving the declared structure; *Avg. steps* is the mean number of declared plan units. Pattern-specific executors are omitted because they enforce their execution structure by design. Values are mean percentages ± SD across seeds 7, 13, and 42. ‡Sparse bins should be interpreted cautiously.

Benchmark	Model	Overall	Avg. steps	Structure maintained by declared plan length				
				≤ 3	4–5	6–7	8–10	> 10
ALFWorld	DeepSeek-V4	27.4±5.1	6.2±0.3	53.3±37.7	36.7±7.0	21.4±3.0	0.0±0.0‡	0.0±0.0‡
	Qwen3.6-35B	23.7±2.7	6.1±0.3	66.7±27.2	36.5±1.3	9.5±3.9	1.2±1.7	0.0±0.0‡
	Gemma-4-26B	21.5±1.9	7.6±0.2	91.7±8.3	49.6±2.2	4.8±4.8	0.0±0.0	0.0±0.0
Mind2Web	DeepSeek-V4	44.6±0.9	4.8±0.0	73.3±0.5	39.2±1.4	31.2±1.7	10.0±3.4	5.9±3.9
	Qwen3.6-35B	36.1±0.9	4.8±0.0	62.4±1.8	31.5±0.3	17.9±1.1	8.6±1.4	4.2±3.1
	Gemma-4-26B	28.2±0.2	5.7±0.0	70.1±2.9	24.8±0.8	13.4±0.5	12.7±1.8	11.3±4.9
SWE-bench	DeepSeek-V4	25.4±2.1	4.6±0.1	33.4±1.3	25.6±1.2	15.5±2.5	21.2±12.1	0.0±0.0‡
	Qwen3.6-35B	23.5±1.0	4.8±0.1	47.9±3.2	22.3±0.5	13.6±0.1	5.9±0.3	3.6±3.6
WebArena	DeepSeek-V4	67.9±4.4	5.1±0.1	84.5±1.2	68.7±7.4	64.1±1.9	45.0±5.0	24.7±17.0
	Qwen3.6-35B	42.2±1.2	6.7±0.2	65.9±6.8	59.1±3.6	39.7±0.3	18.4±4.1	5.4±1.5
	Gemma-4-26B	61.4±2.5	5.6±0.1	76.1±1.5	65.9±2.6	53.9±18.5	52.3±2.3	32.5±9.8

results in Appendices E and E.3. Plan quality and adherence are scored separately with an LLM-as-judge pipeline using a 0–3 GPA-style rubric (Jia et al., 2025); full prompts, judge configurations, and cross-judge controls are provided in Appendix F.

5 RESULTS

[RQ1]: PATTERN-SPECIFIC EXECUTION PRESERVES DECLARED PLANNING STRUCTURE, WHILE GENERIC PLAN+REACT INCREASINGLY DEVIATES ON LONGER PLANS

Generic Plan+ReAct does not reliably preserve the committed planning structure. To examine whether agents preserve their declared planning structure, we first measure structure maintenance under Plan+ReAct. Table 1 reports overall plan-structure maintenance under Plan+ReAct and its variation with declared plan length. We make the following observations: (i) Plan+ReAct reveals a substantial Plan Declaration–Execution Gap across benchmarks. Across ALFWorld, Mind2Web, and SWE-bench, only about 22–45% of Plan+ReAct trajectories preserve the declared structure. (ii) Plan structure fidelity also decreases with plan length: on ALFWorld, maintenance falls from (36.5)–(49.6%) for 4–5-step plans to (4.8)–(21.4%) for 6–7 steps and approaches zero for longer plans; Mind2Web shows a similar decline, while SWE-bench shows the same overall pattern from short to medium-length plans. WebArena is a short-plan boundary case, with higher maintenance ((65.6)–(69.1%)) and average plans of only (3.3)–(3.8) steps.

Hierarchical/Search plans are particularly difficult for generic ReAct execution to preserve. We next analyze structural maintenance by planning mode. The mode-level breakdown in Appendix G, Table 11, shows that the declaration–execution gap is most pronounced for richer planning structures. Hierarchical plans are difficult for generic Plan+ReAct to preserve, with structure maintenance typically below 25% across ALFWorld, Mind2Web, and SWE-bench. Search shows a similar pattern where enough declarations are available. Together with the plan-length results, this indicates that generic ReAct is unreliable for multi-level or multi-candidate planning structures.

Verifier agreement with human annotations. We further validate the rule-based verifier against two independent human annotators on 100 sampled Plan+ReAct ALFWorld trajectories. Verifier–human agreement ($\kappa = 0.31/0.34$) is comparable to human–human agreement ($\kappa = 0.35$); full validation results are reported in Appendix E.3.

Qualitative trajectories illustrate the declaration–execution gap. We further qualitatively verify the declaration–execution gap using representative trajectories shown in Appendix G, Figure 4. These examples illustrate how generic execution can skip or reorder declared plan units. In contrast, pattern-specific executors make the declared planning structure part of the execution control flow.

Plan completion also depends on plan size and execution budget. Plan structural fidelity and plan adherence capture different properties: the former measures whether execution preserves the organization of the declared plan, whereas the latter measures how much of the plan is completed within

Table 2: Pattern-ceiling analysis across benchmarks. $S(p)$ is task success under forced execution of pattern p; $\mathrm{DSR}_{\max}$ is the per-task oracle; and $H = \mathrm{DSR}_{\max} - \max_p S(p)$ is the headroom beyond the strongest fixed pattern. Values are mean ± SD across seeds where available.

Bench	Model	N	SEQ	PRED	HIER	SEARCH	$\mathrm{DSR}_{\max}$	H	$\mathrm{DSR}_{\max}$ w/o Search	H w/o Search
ALFWorld	DeepSeek-V4	134	0.633±0.020	0.556±0.008	0.840±0.009	**0.918±0.026**	0.953±0.013	0.035±0.019	0.898 ± 0.004	0.057 ± 0.013
	Qwen3.6-35B	134	0.662±0.027	0.550±0.023	0.714±0.037	**0.796±0.004**	0.925±0.016	0.129±0.020	0.843 ± 0.030	0.129 ± 0.007
	Gemma-4-26B	134	0.550±0.014	0.515±0.016	**0.600±0.013**	0.495±0.058	0.841±0.007	0.241±0.013	0.781 ± 0.009	0.182 ± 0.021
Mind2Web	DeepSeek-V4	1341	0.053±0.005	0.048±0.003	0.041±0.001	**0.057±0.006**	0.097±0.003	0.037±0.005	0.087±0.007	0.034±0.003
	Qwen3.6-35B	1341	0.032±0.001	0.036±0.002	0.029±0.002	**0.052±0.001**	0.084±0.004	0.031±0.004	0.064±0.003	0.028±0.002
	Gemma-4-26B	1341	0.114±0.006	**0.147±0.005**	0.083±0.002	0.086±0.006	0.282±0.002	0.135±0.006	0.253±0.002	0.106±0.002
SWE-bench	DeepSeek-V4	500	0.395±0.010	0.372±0.030	**0.442±0.024**	0.400±0.018	0.552±0.023	0.110±0.000	0.517±0.026	0.075±0.002
	Qwen3.6-35B	500	0.239±0.029	0.303±0.007	**0.330±0.011**	0.296±0.040	0.445±0.038	0.115±0.027	0.415±0.022	0.085±0.011
	Gemma-4-26B	500	0.170±0.019	0.247±0.012	**0.290±0.015**	0.211±0.024	0.386±0.014	0.096±0.021	0.362±0.017	0.072±0.023
WebArena	DeepSeek-V4	204	0.389±0.026	0.372±0.031	0.537±0.014	**0.580±0.006**	0.736±0.020	0.156±0.014	0.676±0.011	0.139±0.003
	Qwen3.6-35B	204	0.315±0.009	0.364±0.009	0.395±0.099	**0.577±0.026**	0.693±0.017	0.119±0.011	0.557±0.034	0.162±0.065
	Gemma-4-26B	204	0.364±0.019	0.290±0.005	**0.378±0.011**	0.306±0.033	0.578±0.011	0.200±0.022	0.514±0.014	0.136±0.003

the interaction budget. On ALFWorld (Appendix H Table 12), short Sequential plans are completed almost entirely, while larger Hierarchical plans contain 10–13 executable units and achieve lower adherence ($\sim$ 79–85%). Predefined plans fall between these cases, while Search shows high candidate-level completion. Thus, structural fidelity and plan completion capture different properties: pattern-specific executors preserve the intended organization, but completion additionally depends on plan size and budget. Results for Mind2Web and SWE-bench are provided in Appendix H.

[RQ2]: PLANNING-PATTERN EFFECTIVENESS VARIES ACROSS ENVIRONMENTS AND MODELS.

We next compute the forced-pattern ceiling described in Section 3.2. Table 2 yields two main observations. First, the strongest planning pattern varies across benchmark–model pairs: Search is strongest for DeepSeek-V4 and Qwen3.6-35B on ALFWorld, Mind2Web, and WebArena, whereas Hierarchical is strongest on SWE-bench and for Gemma-4-26B on ALFWorld, and Predefined is strongest for Gemma-4-26B on Mind2Web. Thus, planning-pattern effectiveness depends on both the environment and backbone model. Second, the forced-pattern oracle exceeds the strongest fixed pattern across all benchmark–model pairs, indicating additional headroom when multiple planning modes are available. However, because the oracle also provides multiple independent execution attempts, it should be interpreted as an empirical ceiling rather than direct evidence of task-specific complementarity. This motivates the retry-matched control in Appendix J; after three retries of the strongest fixed pattern, the residual oracle gap is only (-0.025) to (+0.027) across ALFWorld, Mind2Web, and SWE-bench (Table 17).

The oracle gap is not driven solely by Search. Because Search planning generates and executes multiple candidate plans before selecting among them with a rubric-based judge, it receives a larger multi-rollout execution budget than the other planning patterns. We therefore recompute the oracle after excluding Search (see Table 2) and still observe a gap between the strongest fixed pattern and the forced-pattern oracle across benchmarks. On ALFWorld, for example, the no-Search oracle reaches (0.898), (0.843), and (0.781), while three retries of Hierarchical reach (0.963), (0.896), and (0.813), respectively, showing that much of the remaining gap can be explained by repeated execution rather than per-task pattern complementarity.

Judged plan quality is only weakly related to task success. We now test whether higher judged plan quality is associated with higher task success. As shown in Appendix I, Table 15, the highest-quality plan on ALFWorld is not consistently associated with the highest-performing planning pattern. For DeepSeek-V4, Sequential receives the highest judged plan quality (2.84 ± 0.03), yet Search achieves substantially higher task success (0.91 ± 0.03 vs. 0.64 ± 0.02) despite a lower quality score (2.53 ± 0.03). At the task level, plan quality is only weakly correlated with success ($|r| \leq 0.181$). Full judge configurations and control analyses are provided in Appendix I.

[RQ3]: PATTERN-SPECIFIC EXECUTION SUBSTANTIALLY IMPROVES OVER PLAN-AS-CONTEXT

Pattern-specific execution provides substantially stronger task-solving capability than generic ReAct execution. We next test whether executing the selected planning mode with its corresponding pattern-specific executor improves end-to-end performance. Table 3 compares Flat ReAct, generic Plan+ReAct, the strongest fixed planning pattern, and Planning-as-Routing. Across benchmarks, generic Plan+ReAct provides limited gains over Flat ReAct, whereas pattern-specific execution

Table 3: End-to-end comparison of planning and execution strategies. *Best Fixed* ($S^\star$) is the strongest single planning pattern applied to all tasks; *Routing@1* executes the top-1 declared pattern with its corresponding executor; and *Oracle* is the per-task forced-pattern ceiling. Bold denotes the strongest non-oracle method. Routing@1 uses thinking disabled.

Benchmark	Model	Metric	Flat ReAct	Plan+ReAct	Best Fixed $S^\star$	Routing@1	Oracle
ALFWorld	DeepSeek-V4	TSR	0.440±0.012	0.480±0.009	**0.918±0.026** (Search)	0.721±0.029	0.953±0.013
	Qwen3.6-35B	TSR	0.535±0.007	0.540±0.037	**0.796±0.004** (Search)	0.667±0.049	0.925±0.016
	Gemma-4-26B	TSR	0.112±0.040	0.153±0.026	**0.600±0.013** (Hierar.)	0.560±0.016	0.841±0.007
Mind2Web	DeepSeek-V4	TSR	**0.058±0.003**	0.038±0.005	0.057±0.006 (Search)	0.051±0.004	0.097±0.003
		SSR	**0.434±0.001**	0.407±0.002	**0.449±0.000** (Search)	0.425±0.007	0.550±0.001
	Qwen3.6-35B	TSR	0.042±0.002	0.036±0.004	**0.052±0.001** (Search)	0.034±0.001	0.084±0.004
		SSR	0.366±0.003	0.338±0.001	**0.411±0.003** (Search)	0.339±0.002	0.505±0.001
	Gemma-4-26B	TSR	0.051±0.002	0.043±0.004	**0.147±0.005** (Predef.)	0.113±0.006	0.282±0.002
		SSR	0.395±0.001	0.365±0.002	**0.388±0.006** (Search)	0.320±0.002	0.585±0.002
SWE-bench	DeepSeek-V4	PSR	0.384±0.010	0.360±0.005	**0.442±0.024** (Hierar.)	0.415±0.017	0.552±0.023
	Qwen3.6-35B	PSR	0.179±0.034	0.223±0.005	**0.330±0.011** (Hierar.)	0.313±0.006	0.445±0.038
WebArena	DeepSeek-V4	TSR	0.455±0.000	0.472±0.006	**0.580±0.006** (Search)	0.460±0.011	0.736±0.020
	Qwen3.6-35B	TSR	0.332±0.009	0.338±0.037	**0.577±0.026** (Search)	0.404±0.019	0.693±0.017
	Gemma-4-26B	TSR	0.318±0.023	0.304±0.020	**0.378±0.011** (Hierar.)	0.4320.014	0.578±0.011

yields substantially higher success, showing that selecting a plan alone is insufficient when the executor does not preserve its structure.

On ALFWorld, DeepSeek-V4 improves from 0.480 with Plan+ReAct to 0.918 with the strongest fixed pattern, while Qwen3.6-35B improves from 0.540 to 0.796. On SWE-bench Verified, DeepSeek-V4 increases from 0.360 to 0.442. Similar gains appear across the remaining benchmarks, indicating that providing a plan as context to a generic ReAct executor does not capture the gains obtained when the planning structure is implemented directly in the execution mechanism.

Current declarations do not consistently select the strongest planning pattern. Table 3 shows that Routing@1 often remains below the strongest fixed pattern. For example, on WebArena, Search reaches 0.580 and 0.577 for DeepSeek-V4 and Qwen3.6-35B, whereas Routing@1 reaches 0.460 and 0.404. Thus, strong pattern-specific executors are available, but current declarations do not consistently select the pattern that realizes their full performance; we examine this selection problem directly in *Sec. [RQ4]*.

Execution gains increase with task length. Pattern-specific execution becomes more beneficial on longer tasks: Flat ReAct and Plan+ReAct degrade more sharply with task length, whereas Hierarchical and Search remain more robust. Full length-wise results are provided in Appendix J Figure 6.

Rollout-matched control for Search. Because Search executes multiple candidate plans, we compare it against $3\times$ independent Flat ReAct runs followed by the same trajectory judge. Repeated ReAct improves performance but does not fully close the Search advantage: Search remains higher by 0.117–0.381 on ALFWorld and 0.153–0.233 on WebArena, while differences are smaller and model-dependent on Mind2Web. Full multi-rollout and pass@3 controls are provided in Appendix J.1, Table 16.

Comparison with prior work. Our forced-pattern and routing results fall within the performance range of representative prior agent systems across four benchmarks. Because these systems use different models, demonstrations, training, and inference procedures, we treat them only as benchmark-level context; detailed comparisons are provided in Appendix K.

[RQ4]: Planning-mode declarations adapt only weakly to individual tasks

We finally ask whether current LLMs select effective planning modes for individual tasks. Because a model may simply favor particular modes overall, we first test whether its declarations contain task-specific information beyond these global preferences.

Declarations provide no reliable task-specific advantage over a task-blind policy. We compare top-1 declaration success with a task-blind null preserves each model's overall planning-mode frequencies while removing the task–mode association. Across all benchmark–model pairs, $\Delta(\pi)$ lies within the 95% interval of a 5,000-permutation null (-0.025 to $+0.009$; minimum one-sided

Table 4: Planning-pattern selection and improvement. @1–@3 report ranked fallback success when declared modes are attempted in order up to rank k. *Few-shot@1* uses task–pattern demonstrations, and Δ is the absolute gain over the original top-1 declaration. Oracle is the per-task forced-pattern ceiling. Mind2Web reports both TSR and SSR; results are averaged across seeds.

Benchmark	Model	Metric	Think	@1	@2	@3	Few-shot@1	Δ_{FS}	Oracle
ALFWorld	DeepSeek-V4	TSR	Off	0.721±0.028	0.893±0.001	0.948±0.011	**0.793±0.083**	**+0.073**	0.953±0.013
		TSR	On	0.656±0.019	0.863±0.007	0.908±0.012	**0.812±0.047**	**+0.156**	
	Qwen3.6	TSR	Off	0.667±0.049	0.804±0.028	0.873±0.016	**0.706±0.034**	**+0.040**	0.925±0.016
		TSR	On	0.674±0.021	0.769±0.010	0.853±0.031	**0.704±0.069**	**+0.030**	
Mind2Web	DeepSeek-V4	TSR	Off	0.053±0.003	0.075±0.004	0.092±0.004	0.050±0.005	−0.003	0.097±0.003
		TSR	On	0.051±0.004	0.072±0.006	0.089±0.007	0.049±0.003	−0.002	
		SSR	Off	0.434±0.003	0.495±0.004	0.532±0.002	**0.438±0.007**	**+0.004**	0.550±0.001
		SSR	On	0.427±0.002	0.492±0.006	0.529±0.005	**0.432±0.004**	**+0.005**	
	Qwen3.6	TSR	Off	0.036±0.001	0.053±0.002	0.072±0.002	0.036±0.001	+0.000	0.084±0.004
		TSR	On	0.034±0.004	0.051±0.003	0.068±0.002	0.036±0.004	+0.001	
		SSR	Off	0.339±0.002	0.426±0.001	0.479±0.002	**0.357±0.002**	**+0.018**	0.505±0.001
		SSR	On	0.341±0.006	0.417±0.002	0.467±0.003	**0.352±0.002**	**+0.011**	
	Gemma-4	TSR	Off	0.111±0.006	0.162±0.012	0.190±0.026	0.111±0.006	+0.000	0.282±0.002
		TSR	On	0.114±0.012	0.170±0.006	0.204±0.025	0.115±0.011	+0.001	
		SSR	Off	0.320±0.002	0.433±0.032	0.514±0.029	**0.330±0.006**	**+0.011**	0.585±0.002
		SSR	On	0.321±0.002	0.441±0.024	0.513±0.024	**0.328±0.012**	+0.008	
SWE-bench	DeepSeek-V4	PSR	Off	0.415±0.024	0.498±0.021	0.541±0.037	0.405±0.011	−0.011	0.552±0.023
		PSR	On	0.411±0.036	0.488±0.046	0.534±0.043	**0.415±0.007**	**+0.004**	
	Qwen3.6-35B	PSR	Off	0.313±0.009	0.400±0.013	0.438±0.025	**0.334±0.003**	**+0.021**	0.445±0.038
		PSR	On	0.310±0.004	0.390±0.016	0.438±0.014	**0.337±0.002**	**+0.027**	
WebArena	DeepSeek-V4	TSR	Off	0.460±0.011	0.639±0.009	0.696±0.003	**0.480±0.037**	**+0.020**	0.736±0.020
		TSR	On	0.384±0.009	0.554±0.014	0.671±0.006	**0.438±0.057**	**+0.054**	
	Qwen3.6	TSR	Off	0.404±0.019	0.539±0.000	0.639±0.005	**0.410±0.009**	**+0.006**	0.693±0.017
		TSR	On	0.394±0.010	0.505±0.005	0.591±0.034	**0.419±0.023**	+0.025	

$p = 0.094$), providing no reliable evidence that current declarations match planning modes to individual tasks better than expected from their overall declaration preferences (Full results are in Appendix L).

Few-shot task-pattern demonstrations. Few-shot demonstrations improve top-1 declaration performance in most evaluated configurations, with the largest gain on ALFWorld for DeepSeek-V4 ((0.656 to 0.812), $+0.156$). Gains are smaller on Mind2Web (up to $+0.018$), SWE-bench ($+0.027$), and WebArena ($+0.054$), indicating that demonstrations improve planning-mode selection unevenly across benchmarks. Full results and discrimination analyses are provided in Appendix O.1.

6 Discussion and Conclusion

In this work, we study whether LLM agents preserve the planning structure they declare, whether planning-pattern effectiveness varies across environments and models, and whether matching a planning pattern to its corresponding executor improves task success. Our experiments reveal four main findings. First, generic Plan+ReAct does not reliably preserve the declared planning structure: structural fidelity decreases as plans become longer. Second, planning-pattern effectiveness varies across environments and models, while retry-matched controls show that the raw forced-pattern oracle largely reflects additional execution attempts rather than task-specific pattern complementarity. Third, matching planning patterns to corresponding executors substantially improves task success over generic Plan+ReAct, particularly on longer tasks. Finally, planning-mode selection remains challenging: current declarations provide little reliable task-specific advantage over task-blind preferences, while few-shot demonstrations improve selection more consistently than enabling thinking.

Together, these findings suggest that reliable agent planning requires both selecting an appropriate planning mode and executing it through a compatible control structure. Planning-as-Routing separates these capabilities, helping distinguish failures of mode selection from failures of execution.

Limitations and future directions. Our study considers four planning patterns and task-level routing under fixed execution budgets. Future work could extend this framework to additional or dynamically composed planning strategies, allow agents to switch modes during execution, and evaluate structural fidelity jointly with action correctness, execution cost, and recovery behavior. Detailed limitations discussed in Appendix N.

AI USE STATEMENT

In this work, generative AI tools were used under human guidance to assist with prompt drafting, grammar correction, language polishing of the manuscript. All prompts were reviewed and revised by the authors before use.

ETHICS STATEMENT

Our work evaluates LLM agents using publicly available, established benchmarks for embodied interaction, web navigation, and software engineering. We do not conduct new experiments involving human participants, collect new personal data, or deploy agents to interact with real users. Experiments are performed within the interfaces and environments provided by the corresponding benchmarks, including ALFWorld, Mind2Web, WebArena, and SWE-bench Verified.

Our work studies how LLM agents select and execute planning strategies. Although improved planning and routing mechanisms can make autonomous agents more capable, the same techniques could potentially be applied to higher-risk forms of automated web interaction or software modification. Our experiments are restricted to benchmark tasks and controlled tool interfaces and are not intended to support unauthorized interaction with external systems. We report process-level metrics in addition to task success to make agent behavior and planning failures more transparent. We, the authors, are responsible for ensuring that the benchmark data, models, and software used in this work are handled in accordance with their applicable licenses and terms of use.

REPRODUCIBILITY STATEMENT

We provide the planning-pattern definitions and execution architectures in Appendix B, the complete planning-mode prompts in Appendix O, benchmark and model details in Appendix C, and inference and sampling configurations in Appendix D. We report results across repeated random seeds where applicable and provide additional analyses of plan adherence (Appendix H), repeated fixed-pattern execution (Appendix J), the task-blind declaration null (Appendix J), and inference cost (Appendix D).

We plan to release the implementation of the planning-pattern executors, declaration and execution prompts, trajectory verifier, evaluation and analysis scripts, and experiment configuration files used upon publication of this work. We also plan to release the derived planning declarations, execution traces, pattern-level outcomes, and analysis metadata needed to reproduce the reported results.

REFERENCES

Gemma Team Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cuarbune, Michelle Cas-bon, Mayank Chaturvedi, Victor Cotruta, Alice Coucke, Phil Culliton, Robert Dadashi, Lucas Dixon, Mohamed Elhawaty, Utku Evci, Clément Farabet, Johan Ferret, Filippo Galgani, Sertan Girgin, Jean-Bastien Grill, Maarten R. Grootendorst, Jiaxian Guo, Cassidy Hardin, Yanzhang He, Steven M. Hernandez, Omri Homburger, L'eonard Hussenot, Juyeong Ji, Armand Joulin, Aishwarya B Kamath, Parnian Kassraie, Olivier Lacombe, Preethi Lahoti, Gael Liu, Gus Martins, Luciano Martins, Tatiana Matejovicova, Ramona Merhej, Nikola Momchev, Sneha Mondal, Ryan Mullins, Sindhu Raghuram Panyam, Shreya Pathak, Sarah Perrin, André Susano Pinto, Etienne Pot, Angéline Pouget, Alexandre Ram'e, Sabela Ramos, Doug Reid, David Rim, Morgane Rivière, Karsten Roth, Louis Rouillard, Omar Sanseviero, Pier Giuseppe Sessa, Shane Settle, Danila Sinopalnikov, Sara Smoot, Piotr Stańczyk, Andreas Steiner, Lawrence Stewart, Ilya O. Tolstikhin, Michael Tschannen, Anton Tsitsulin, Nino Vieillard, Renjie Wu, Ping mei Xu, Hai-Tao Yang, Edouard Yvinec, Li Zhang, Joe Zou, Nicolas Aagnes, Abdelrahman Abdelhamed, Shivani Agrawal, Shubham Agrawal, Ibrahim Alabdulmohsin, Jean-Baptiste Alayrac, Uri Alon, Chandramouli N. Amarnath, Ankesh Anand, Chrysovalantis Anastasiou, Setareh Ariafar, Franccois-Xavier Aubet, Kyriakos Axiotis, Federico Barbero, Joelle Barral, Alexei Bendebury, Urs M. Bergmann, Stanley M. Bileschi, Kat Black, Mathieu Blondel, Sebastian Borgeaud, Arthur Bravzinskas, Ryan Burnell, Róbert Istvan Busa-Fekete, Mu Cai, Glenn Cameron, Char lotte Caucheteux, Garima Chadha, Jetha Chan, Aditya Chawla, Blake Jianhang Chen, Jesse Chen, Lin Chen, Xu Chen,

Derek Zhiyuan Cheng, Tzu hsiang Chien, Nikolai Chinaev, Ying-Fen Chou, Zhaohui Chu, Benjamin Coleman, Pooja Consul, Sam Conway-Rahman, Scott Crowell, Dylan Cutler, Vivek Dani, Samira Daruki, Anil Das, Daniel Deutsch, Nishanth Dikkala, Linyi Ding, Qiuhan Ding, Shenil Dodhia, Konstantin Donhauser, Tulsee Doshi, Anca Dragan, Alex Druinsky, Sahil Dua, Zoltan Egyed, Danielle Eisenbud, Daniel Eppens, Cindy Fan, Bahare Fatemi, Yassir Fathullah, Vladimir Feinberg, Milen Ferev, Takumi Fujimoto, Isaac R. Galatzer-Levy, João Gante, Simon Geisler, Soham Ghosal, Antonious M. Girgis, Alec Go, Alhaad Gokhale, Alex Grills, Yiming Gu, Pramod Gupta, Guru Guruganesh, Raia Hadsell, Hamza Harkous, Jitendra K. Harlalka, Demis Hassabis, Anja Hauth, Joseph Heyward, Arian Hosseini, Chih-Yang Hsia, I-Hung Hsu, Xiaopeng Huang, Yangsibo Huang, Kevin Hui, Adrian Hutter, I Te, Fotis Iliopoulos, Advait Jain, Ganesh Jawahar, Ziwei Ji, Qilin Jin, Melvin Johnson, Kandarp Joshi, Arun Kumar Reddy Kandoor, Wang-Cheng Kang, Koray Kavukcuoglu, Mehran Kazemi, Kathleen Kenealy, Amr Khalifa, Phoebe Kirk, Suraj Kothawade, Vitaly Kovalev, Neel Kovelamudi, Adam Kraft, Ravin Kumar, Harish Kuppam, Justin Lannin, Chen-Yu Lee, Seungjin Lee, Dmitry Lepikhin, Dong dong Li, Qiujia Li, Valentin Liévin, Ethan Lin, Ziqian Lin, Casper Liu, Tianlin Liu, Tianqi Liu, Xin Liu, Mayank Lunayach, Min Ma, Gagan Madan, Andrii Maksai, Eric Malmi, Michal Matuszak, Daniel McDuff, Gaurav Menghani, Daniil Mirylenka, Karolis Misiunas, Vedant Misra, Andreea-Maria Mitran, Kareem Mohamed, Maksim Mukha, Eric Noland, James O'Donnell, Kate Olszewska, Bernett Orlando, Wan Lin Pan, Rina Panigrahy, Unnati Parekh, Chunjong Park, Eric Paskie, Liqian Peng, Bryce Petrini, Slav Petrov, Jonas Pfeiffer, Bilal Piot, Martyna Beata Płomecka, Siim Põder, Octavio Ponce, Arijit Pramanik, David Racz, A. Peter Rajan, Michelle Tadmor Ramanovich, Anand Rao, Marvin Ritter, Vitor Rodrigues, Evan Rosen, Mikolaj Rybi'nski, Noveen Sachdeva, Michael E. Sander, Rohit Sathyanarayana, Sagar Savla, Samuel Schmidgall, Tal Schuster, Benoit Seguin, Andrew B. Sellergren, Aliaksei Severyn, Izhak Shafran, Dhruv Shah, Shangguan Yuan, Ashish Shenoy, Pradeep Shenoy, Rakesh Shivanna, Pauline Sho, Lucas Spangher, Wojciech Stokowiec, Tim Strother, Yao Su, Yinghao Sun, Mukund Sundararajan, Andrea Tacchetti, Mor Hazan Taege, Pouya Dehghani Tafti, Chetan Tekur, Rahul Thapa, Madeleine Traverse, Lenart Treven, Tao Tu, Chien Te Tung, Petar Velivckovi'c, Malini Pooni Venkat, Sagar Gubbi Venkatesh, Vidya Venkiteswaran, Francesco Visin, Alex Vitvitskyi, Kiran Vodrahalli, Weiyi Wang, Xin Wang, Tris Warkentin, Jan Wassenberg, John Wieting, Lechao Xiao, Hao Xu, Yuhui Xu, Fuzhao Xue, A. Yadav, Jun Yan, Antoine Yang, Linfeng Yang, Ming-Hsuan Yang, Ziyu Ying, Jae Hyeon Yoo, Sajjad Hussain Zafar, Fred Weiying Zhang, Jiageng Zhang, Jianyi Zhang, Xiaofan Zhang, Chaoran Zhao, David Zhou, and Chenjie Zou. Gemma 4 technical report. 2026. URL https://api.semanticscholar.org/CorpusID:289923375.

Matthew M Botvinick. Hierarchical models of behavior and prefrontal function. *Trends in cognitive sciences*, 12(5):201–208, 2008.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025.

Jae-Woo Choi, Hyungmin Kim, Hyobin Ong, Youngwoo Yoon, Minsu Jang, Dohyung Kim, and Jaehong Kim. Reactree: Hierarchical llm agent trees with control flow for long-horizon task planning. *arXiv preprint arXiv:2511.02424*, 2025.

Nathaniel D Daw, Yael Niv, and Peter Dayan. Uncertainty-based competition between prefrontal and dorsolateral striatal systems for behavioral control. *Nature neuroscience*, 8(12):1704–1711, 2005.

Nathaniel D Daw, Samuel J Gershman, Ben Seymour, Peter Dayan, and Raymond J Dolan. Model-based influences on humans' choices and striatal prediction errors. *Neuron*, 69(6):1204–1215, 2011.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Shen Dong, Mingxuan Zhang, Pengfei He, Li Ma, Bhavani Thuraisingham, Hui Liu, and Yue Xing. Pear: Planner-executor agent robustness benchmark. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 4547–4567, 2026.

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 *Forty-second International Conference on Machine Learning*, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Allison Sihan Jia, Daniel Huang, Nikhil Vytla, Seung Won Wilson Yoo, Nirvika Choudhury, Shayak Sen, John C Mitchell, and Anupam Datta. What is your agent's gpa? a framework for evaluating agent goal-plan-action alignment. *arXiv preprint arXiv:2510.08847*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Fanqi Kong, Ruijie Zhang, Huaxiao Yin, Guibin Zhang, Xiaofei Zhang, Ziang Chen, Zhaowei Zhang, Xiaoyuan Zhang, Song-Chun Zhu, and Xue Feng. Aegis: Automated error generation and attribution for multi-agent systems. *arXiv preprint arXiv:2509.14295*, 2025.

Bang Liu, Xinfeng Li, Jiayi Zhang, Jinlin Wang, Tanjin He, Sirui Hong, Hongzhang Liu, Shaokun Zhang, Kaitao Song, Kunlun Zhu, et al. Advances and challenges in foundation agents: From brain-inspired intelligence to evolutionary, collaborative, and safe systems. *arXiv preprint arXiv:2504.01990*, 2025.

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*, 2023.

Jiayu Liu, Cheng Qian, Zhenhailong Wang, Bingxuan Li, Jiateng Liu, Heng Wang, Jeonghwan Kim, Yumeng Wang, Xiusi Chen, Yi R Fung, et al. Adaplanbench: Evaluating adaptive planning in large language model agents under world and user constraints. *arXiv preprint arXiv:2606.05622*, 2026a.

Shuyang Liu, Saman Dehghan, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. From plan to action: How well do agents follow the plan? *arXiv e-prints*, pp. arXiv–2604, 2026b.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents. *Advances in neural information processing systems*, 37:74325–74362, 2024.

Lucas Maes, Quentin Le Lidec, Damien Scieur, Yann LeCun, and Randall Balestriero. Leworld-model: Stable end-to-end joint-embedding predictive architecture from pixels. *arXiv preprint arXiv:2603.19312*, 2026.

Marcelo G Mattar and Nathaniel D Daw. Prioritized memory access explains planning and hippocampal replay. *Nature neuroscience*, 21(11):1609–1617, 2018.

Marcelo G Mattar and Máté Lengyel. Planning in the brain. *Neuron*, 110(6):914–934, 2022.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, volume 2024, pp. 9025–9049, 2024.

Naoki Otani, Nikita Bhutani, Hannah Kim, Dan Zhang, and Estevam Hruschka. Do agents need to plan step-by-step? rethinking planning horizon in data-centric tool calling. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 375–403, 2026.

Tianyue Ou, Wanyao Guo, Apurva Gandhi, Graham Neubig, and Xiang Yue. Agentdiagnose: An open toolkit for diagnosing llm agent trajectories. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 207–215, 2025.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Yichen Pan, Dehan Kong, Sida Zhou, Cheng Cui, Yifei Leng, Bing Jiang, Hangyu Liu, Yanyi Shang, Shuyan Zhou, Tongshuang Wu, et al. Webcanvas: Benchmarking web agents in online environments. *arXiv preprint arXiv:2406.12373*, 2024.

Shuofei Qiao, Runnan Fang, Ningyu Zhang, Yuqi Zhu, Xiang Chen, Shumin Deng, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Agent planning with world knowledge model. *Advances in Neural Information Processing Systems*, 37:114843–114871, 2024.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL `https://qwen.ai/blog?id=qwen3.6-35b-a3b`.

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. *Advances in Neural Information Processing Systems*, 36:38154–38180, 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 *International Conference on Learning Representations*, 2020.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36:58202–58245, 2023.

Haoyu Sun, Wenxuan Wang, Mingyang Song, Jujie He, Weinan Zhang, Yang Liu, Yang Yang, and Yu Cheng. Agent planning benchmark: A diagnostic framework for planning capabilities in llm agents. *arXiv preprint arXiv:2606.04874*, 2026.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 2609–2634, 2023.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Ying Wang, Oumayma Bounou, Yann LeCun, and Mengye Ren. Adajepa: An adaptive latent world model. *arXiv preprint arXiv:2606.32026*, 2026.

Taylor Webb, Shanka Subhra Mondal, and Ida Momennejad. A brain-inspired agentic architecture to improve planning with llms. *Nature Communications*, 16(1):1–12, 2025.

Wenyi Wu, Sibo Zhu, Kun Zhou, and Biwei Huang. Planner matters! an efficient and unbalanced multi-agent collaboration framework for long-horizon planning. *arXiv preprint arXiv:2605.02168*, 2026.

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. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

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.

John Yang, Carlos 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. *Advances in neural information processing systems*, 36:11809–11822, 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 *11th International Conference on Learning Representations, ICLR 2023*, 2023b.

Wentao Zhang, Liang Zeng, Yuzhen Xiao, Yongcong Li, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. Agentorchestra: Orchestrating multi-agent intelligence with the tool-environment-agent (tea) protocol. *arXiv preprint arXiv:2506.12508*, 2025.

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. *arXiv preprint arXiv:2310.04406*, 2023.

Ruiwen Zhou, Yingxuan Yang, Muning Wen, Ying Wen, Wenhao Wang, Chunling Xi, Guoqiang Xu, Yong Yu, and Weinan Zhang. Trad: Enhancing llm agents with step-wise thought retrieval and aligned decision. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 3–13, 2024a.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024b.

Overview of Appendix Sections

- Appendix A: Additional Related Work
- Appendix B: Planning Patterns
- Appendix C: Dataset Details
- Appendix D: Inference and Sampling Configurations
- Appendix E Plan-Structure Verifier and Human Validation
- Appendix F Plan Quality Judge
- Appendix G: Structure Preservation by Declared Planning Mode
  - Appendix E.2: Representative Plan+ReAct Traces
- Appendix H: Additional Plan Adherence Results
- Appendix I: Plan Quality Analysis
- Appendix J: Pattern-Ceiling Analysis
  - Appendix J.1: Repeated-Rollout ReAct Control for Search
  - Appendix J.2: Repeated-Fixed-Pattern Control
  - Appendix J.3: Task-Blind Declaration Null
- Appendix M: Inference Cost Analysis
- Appendix K: Comparison with prior work

- Appendix N: Extended Discussion
- Appendix O: Zero-shot and Few-shot Prompts
  - Appendix O.1: Planning-Mode Declaration Prompts

A Additional Related Work

LLMs as Planning Agents. Prior work has proposed many planning mechanisms for LLM agents, but these methods often instantiate planning behavior through a specific agent architecture. Earlier approaches used ReAct-style agents interleave reasoning and acting in a flat step-wise loop Yao et al. (2023b); Tree-of-Thought-style methods search over multiple reasoning paths Yao et al. (2023a); symbolic-planning approaches integrate LLMs with external planners Liu et al. (2023); planner–executor methods, such as Plan-and-Act, first generate a high-level plan and then execute the resulting steps through an executor Erdogan et al. (2025); and workflow-based multi-agent systems decompose tasks through predefined roles or procedures Shen et al. (2023); Hong et al. (2024). More recent work introduces adaptive replanning, planner–executor collaboration, trajectory refinement, and plan-compliance analysis Liu et al. (2026a); Dong et al. (2026); Liu et al. (2026b); Otani et al. (2026); Wu et al. (2026). While these approaches can generate task-specific plans and, in some cases, adapt or revise them during execution, the underlying planning mechanism is generally fixed by the agent framework rather than chosen per task. In contrast, our work separates the declared planning mode from the execution architecture and asks whether the declared mode is actually preserved in the trajectory.

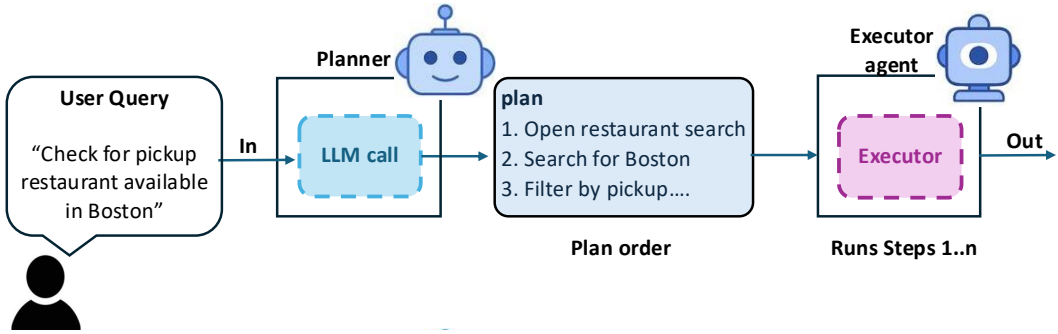
Agent Benchmarks and Process-Level Evaluation. Agent benchmarks evaluate LLM agents in increasingly realistic multi-step environments, including web navigation Deng et al. (2023); Zhou et al. (2024b); Pan et al. (2024), computer and operating-system control Xie et al. (2024); Merrill et al. (2026), software engineering Jimenez et al. (2024), embodied tasks Shridhar et al. (2020), and general assistant tasks Mialon et al. (2024). These benchmarks are valuable for measuring whether agents complete tasks, but they usually report final success, completion rate, or answer correctness rather than the planning process that produced the outcome. AgentBoard moves toward process-level evaluation by introducing progress-rate metrics and analytical visualizations for multi-turn agents Ma et al. (2024). However, progress metrics still do not determine whether an agent's declared planning mode was preserved across planner–executor handoffs. Our work complements benchmark-level evaluation by asking not only whether an agent succeeds, but whether success or failure can be attributed to the planning mode that was declared and the execution path that actually ran.

Trajectory Diagnosis and Failure Attribution. Recent work has begun to analyze agent trajectories beyond final success. AgentDiagnose provides a toolkit for diagnosing LLM-agent trajectories using competency-oriented metrics such as task decomposition, backtracking and exploration, observation reading, self-verification, and objective quality Ou et al. (2025). Other work studies agent-environment interaction failures and proposes taxonomies for where agents fail in realistic environments Kong et al. (2025), while recent multi-agent analyses identify failure modes such as specification and system-design failures, inter-agent misalignment, and task verification or termination errors Cemri et al. (2025). These studies make agent failures more interpretable, but they generally analyze trajectories after execution without explicitly modeling the handoff between declared planning mode and execution architecture. In contrast, our work introduces *handoff-aware failure attribution*: we map each trajectory through the planner, router, executor, and verifier, allowing failures to be localized to plan declaration, plan-to-executor handoff, execution, or verification. This lets us distinguish failures caused by an inappropriate planning mode from failures caused by an executor that did not preserve the declared mode.

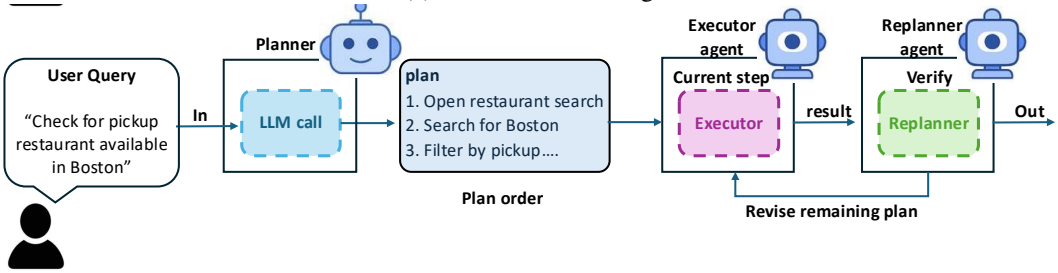
B Planning Patterns

We consider four planning patterns: *predefined*, *sequential*, *hierarchical*, and *search*. These patterns capture different ways an agent can organize task execution. A *predefined* plan follows a fixed plan generated before execution, without replanning (Fig. 2 (a)). A *sequential* plan executes one step at a time and updates the remaining plan based on intermediate observations (Fig. 2 (b)). A *hierarchical*

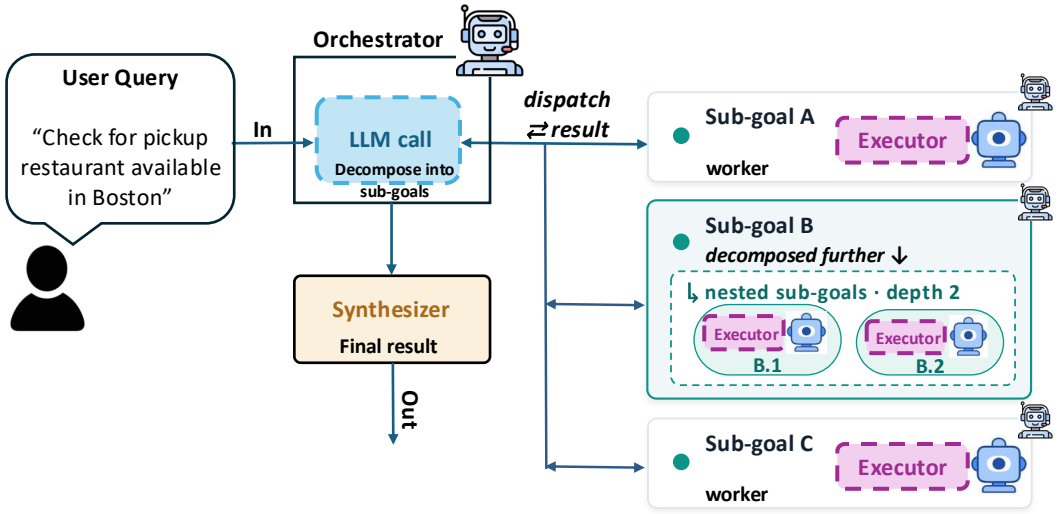
plan decomposes the task into subgoals and coordinates their execution through an orchestrator–worker structure (Fig. 2 (c)). A *search* plan generates multiple candidate plans, executes each candidate independently, and selects the most promising resulting trajectory using a *rubric-based judge* (Fig. 2 (d)).



(a) Predefined Planning

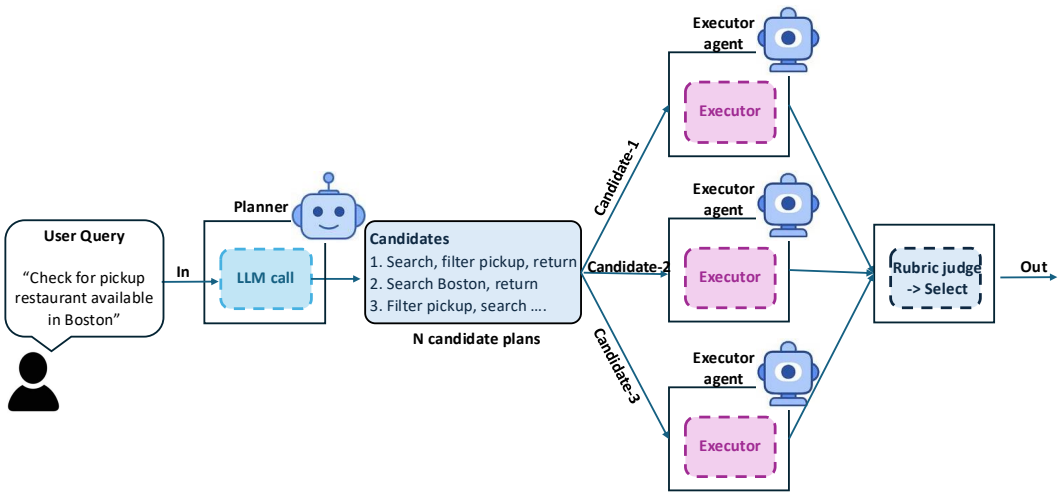


(b) Sequential Planning

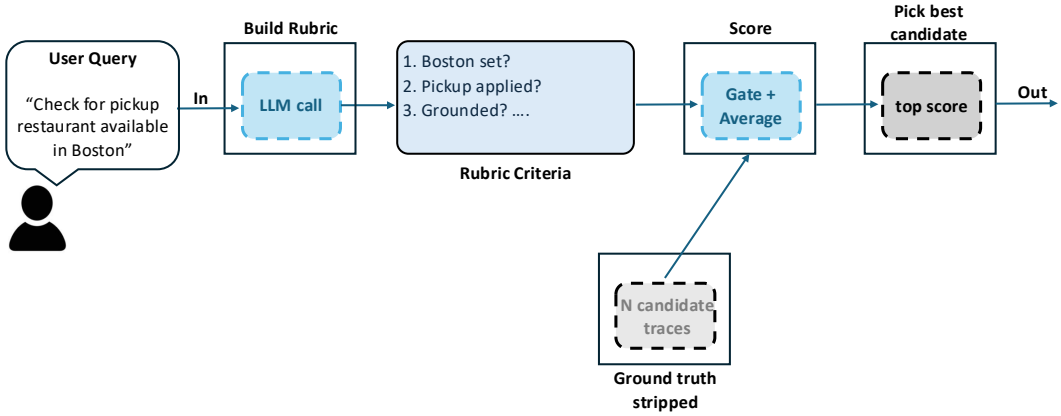


(c) Hierarchical Planning

Figure 2: Illustration of three planning executor patterns. (a) **Predefined planning**: the planner generates a fixed ordered plan and the executor follows the steps without replanning. (b) **Sequential planning**: the executor runs the current step, observes the result, and invokes a replanner to revise the remaining plan when needed. (c) **Hierarchical planning**: an orchestrator decomposes the task into subgoals, dispatches them to workers, and synthesizes the final result.



(a) Search Planning



(b) Pick best candidate using Rubric judge

Figure 3: Illustration of search planning pattern. (a) **Search-based planning**: The planner generates multiple competing candidate plans, and each candidate is executed independently from a reset environment. (b) **Rubric judge**: A task-specific rubric judge evaluates the resulting trajectory summaries against a rubric generated from the task specification and selects the highest-scoring candidate. The judge has no access to benchmark ground-truth success labels or hidden evaluation scores.

C Dataset and Model Details

Benchmarks. We evaluate our framework across four agent benchmarks spanning web interaction, software engineering, and embodied task execution. Table 5 summarizes the number of evaluated tasks, task characteristics, and evaluation metric used for each benchmark.

Table 5: Agent benchmarks used in our experiments. N denotes the number of tasks evaluated in this work.

Benchmark	Environment	N	Task type	Evaluation
Mind2Web	Web interaction	1,341	Multi-step website interaction	Benchmark task success
WebArena	Web interaction	204	Navigation, retrieval, and website interaction	Benchmark task success
SWE-bench Verified	Software engineering	500	GitHub issue resolution and code modification	Patch-based evaluation
ALFWorld	Embodied environment	134	Multi-step household tasks	Goal completion

Backbone models. We evaluate three backbone LLMs: Qwen3.6-35B-A3B, DeepSeek-V4-Flash, and Gemma-4-26B-A4B-it. The same backbone is used for planning-mode declaration and planning

patterns within a given experimental run. Unless otherwise specified, model identity is held fixed when comparing execution conditions.

Table 6: Backbone LLMs used in our experiments.

Model	Role	Thinking
Qwen3.6-35B-A3B	Declaration and execution	Enabled
DeepSeek-V4-Flash	Declaration and execution	Enabled
Gemma-4-26B-A4B-it	Declaration and execution	Enabled

Table 7: Composition of the evaluation sets used in our experiments. ALFWorld tasks are grouped by task type, with the mean number of expert demonstration steps shown for each category. Mind2Web tasks are grouped by the benchmark's three non-overlapping generalisation splits; together they account for all 1,341 evaluation tasks. SWE-bench Verified instances are grouped by the dataset's time-to-fix difficulty label.

Benchmark	Group	Tasks	Auxiliary statistic	Additional information
ALFWorld	Pick & Place	24	Mean expert steps: 4.58	
	Examine in Light	18	Mean expert steps: 3.78	
	Clean & Place	31	Mean expert steps: 6.32	
	Heat & Place	23	Mean expert steps: 6.04	
	Cool & Place	21	Mean expert steps: 6.10	
	Pick Two & Place	17	Mean expert steps: 8.65	
	Total	*134*		
Mind2Web	Cross-Task	252	69 websites	Travel 119; Shopping 68; Entertainment 65
	Cross-Website	177	10 websites	Shopping 63; Travel 60; Entertainment 54
	Cross-Domain	912	54 websites	Information 480; Service 432
	Total	*1,341*		
SWE-bench Verified	$<$ 15 min	194	Time-to-fix difficulty	
	15 min–1 hour	261	Time-to-fix difficulty	
	1–4 hours	42	Time-to-fix difficulty	
	$>$ 4 hours	3	Time-to-fix difficulty	
	Total	*500*		

D Inference and Sampling Configuration

For all experiments, we use a shared inference harness across execution conditions. For a given backbone model, we keep the decoding parameters, thinking configuration, tool-calling policy, and generation budget fixed when comparing Flat ReAct, Plan + ReAct, and Planning-as-Routing. This controls for inference-level differences when comparing planning and execution mechanisms.

Generation profiles. We define one common generation-budget profile that uses a maximum budget of 32,768 new tokens per turn. For thinking models, we also enforce a minimum per-turn budget of 16,384 new tokens. The effective budget for a model is computed as:

$$\max(\texttt{min_floor}, \min(\texttt{model_budget}, \texttt{profile_budget})).$$

We use a single total generation budget per model and do not impose a separate reasoning-token cap. If a model consumes the full budget and terminates with a length cutoff, the run is counted as a truncation failure rather than silently capped.

Thinking and context handling. Thinking mode is enabled for all models. By default, prior thinking traces are not passed back into the model context, which reduces context length and improves comparability across models.

Tool-calling policy. For most models, tool use is forced during the first executor turns by setting the tool choice to required. This encourages the executor to interact with the environment rather than only producing text. All such backend-specific exceptions are fixed before evaluation and applied consistently across all conditions for the affected model.

Table 8: Sampling parameters used for each backbone LLM. Disabled settings are not sent to the backend.

Model	Temp.	Top-p	Top-k	Pres. Pen.	Rep. Pen.	Thinking
Qwen3.6-35B-A3B	1.0	0.95	20	1.5	1.0	Yes
Gemma4-26B-A4B-it	1.0	0.95	64	0.0	1.0	Yes
DeepSeek-V4-Flash	1.0	0.95	Disabled	0.0	1.0	Yes

Model	Model Budget	Default Profile	Hard Profile	Tool Choice
Qwen3.6-35B-A3B	81,920	32,768	81,920	Required
Gemma-4-26B-A4B-it	131,072	32,768	81,920	Required
DeepSeek-V4-Flash	81,920	32,768	81,920	Required

Table 9: Per-turn generation budgets and tool-calling configuration. The effective budget is the minimum of the model budget and the selected profile budget, subject to the minimum thinking floor.

Model-specific decoding settings. For each backbone model, we adopt the recommended inference and decoding configuration provided in the model developer's official release or documentation, rather than tuning these parameters on our evaluation benchmarks. Accordingly, the Qwen models use temperature 1.0, top-$p = 0.95$, and top-$k = 20$; Gemma-4-26B-A4B-it uses temperature 1.0, top-$p = 0.95$, and top-$k = 64$; and DeepSeek-V4-Flash uses temperature 1.0 and top-$p = 0.95$, with top-k disabled. These model-specific configurations are fixed across all experimental conditions.

Search candidate budget. The Search planning pattern where the planner dynamically chooses how many candidate plans to generate for each task, subject to an upper bound of four candidates (`max_candidates=4`). Thus, the realized candidate count can vary from 0 to 4 depending on the planner's decision for the task. Across all Search runs, the realized candidate count is 3.35 ± 0.49 over 15,598 episodes. At the benchmark level, the mean is 3.18 ± 0.41 for ALFWorld, 3.31 ± 0.46 for Mind2Web, 3.38 ± 0.52 for SWE-bench, and 3.88 ± 0.33 for WebArena. Table 10 provides the model- and seed-level breakdown.

Table 10: Realized number of Search candidates per task. "Episode mean" reports mean±SD across individual Search episodes, while "Across seeds" reports mean±SD of the seed-level means. Search permits at most four candidate plans per task.

Benchmark	**Model**	**Episode mean**	**Across seeds**	**Range**
ALFWorld	DeepSeek-V4	3.33 ± 0.47	3.34 ± 0.05	3–4
	Qwen3.6-35B	3.22 ± 0.45	3.22 ± 0.06	0–4
	Gemma-4-26B	2.99 ± 0.11	2.99 ± 0.01	2–4
Mind2Web	DeepSeek-V4	3.47 ± 0.50	3.47 ± 0.01	2–4
	Qwen3.6-35B	3.35 ± 0.48	3.35 ± 0.02	0–4
	Gemma-4-26B	3.00 ± 0.03	3.00 ± 0.00	2–3
SWE-bench	DeepSeek-V4	3.26 ± 0.44	3.26 ± 0.00	3–4
	Qwen3.6-35B	3.76 ± 0.43	3.76 ± 0.02	3–4
	Gemma-4-26B	2.92 ± 0.27	–	2–3
WebArena	DeepSeek-V4	4.00 ± 0.06	4.00 ± 0.01	3–4
	Qwen3.6-35B	3.76 ± 0.43	3.76 ± 0.01	3–4

E Plan-Structure Verifier and Human Validation

E.1 Plan-Structure Verifier

We measure whether Plan+ReAct preserves the declared plan structure using a deterministic rule-based verifier. Representative Plan+ReAct examples for predefined, sequential, hierarchical, and search declarations are provided in Section E.2, illustrating how declared plan units are matched to executed actions under the verifier. Declared plan units are matched to executed actions using benchmark-specific intent and target rules. Structure is considered maintained only if all scoreable plan units can be matched in the declared order; additional environment actions between matched units are allowed. Matching is computed using a monotonic dynamic-programming assignment

rather than greedy matching. Units without executable actions are treated as unscoreable, and trajectories with no scoreable or matchable units are excluded and reported separately.

The verifier measures structural preservation rather than action correctness: whether an action successfully changes the environment is evaluated separately. Pattern-specific routed executors are not scored with this matcher because their dispatch records establish the executed control flow by construction. Benchmark-specific matching rules and null controls are provided below.

E.2 Representative Plan+ReAct Traces

The following examples illustrate how the structure verifier compares the declared plan with the sequence of environment actions produced by the generic Plan+ReAct executor. Extra environment actions are allowed between declared units; structure is preserved when all scoreable plan units are matched in the declared order.

Sequential

Declared plan
1. Look around and identify the location of a pillow.
2. Go to that location and take the pillow.
3. Go to the sofa and put the pillow on the sofa.
4. Look around and identify the location of the second pillow.
5. Go to that location and take the second pillow.
6. Go to the sofa and put the second pillow on the sofa.

Executed actions
1. `go to sofa 1`
2. `go to armchair 1`
3. `take pillow 1 from armchair 1`
4. `take pillow 2 from armchair 1`
5. `inventory`
6. `go to sofa 1`
7. `put pillow 1 on sofa 1`
8. `move pillow 1 to sofa 1`
9. `go to armchair 1`
10. `take pillow 2 from armchair 1`
11. `go to sofa 1`
12. `move pillow 2 to sofa 1`

Predefined

Declared plan
1. Go to countertop.
2. Take mug from countertop.
3. Go to sink.
4. Clean mug.
5. Go to coffeemachine.
6. Put mug in coffeemachine.

Executed actions
1. `go to countertop 1`
2. `go to countertop 2`
3. `take mug 1 from countertop 2`
4. `go to sinkbasin 1`
5. `clean mug 1 with sinkbasin 1`
6. `go to coffeemachine 1`
7. `move mug 1 to coffeemachine 1`

Hierarchical

Declared plan
1. **Acquire peppershaker:** go to countertop → take peppershaker.
2. **Place in drawer:** go to drawer → open drawer → put peppershaker in drawer.

Executed actions
1. `go to countertop 1`
2. `take peppershaker 1 from countertop 1`
3. `go to drawer 1`
4. `open drawer 1`
5. `put peppershaker 1 in drawer 1`
6. `move peppershaker 1 to drawer 1`

Search

Declared plan
1. **Path 1: Tomato in fridge.** Go to fridge → open fridge → take tomato → go to microwave → heat tomato → go to garbage can → put tomato in garbage can.
2. **Path 2: Tomato on countertop.** Go to countertop → take tomato → go to microwave → heat tomato → go to garbage can → put tomato in garbage can.

Executed actions
1. `go to fridge 1`
2. `open fridge 1`
3. `go to countertop 1`
4. `take tomato 1 from countertop 1`
5. `go to microwave 1`
6. `open microwave 1`
7. `put tomato 1 in microwave 1`
8. `move tomato 1 to microwave 1`
9. `heat tomato 1 with microwave 1`
10. `close microwave 1`
11. `open microwave 1`
12. `examine microwave 1`
13. `use microwave 1`
14. `close microwave 1`
15. `use microwave 1`
16. `inventory`
17. `open microwave 1`
18. `take tomato 1 from microwave 1`
19. `heat tomato 1 with microwave 1`
20. `examine tomato 1`

E.3 Human Validation of the Structure Verifier

We validate the rule-based verifier against two independent human annotators, denoted A1 and A2, on (100) sampled Plan+ReAct trajectories from ALFWorld. The sample is stratified across the three evaluated backbone models: DeepSeek-V4, Qwen3.6-35B, and Gemma-4-26B, and across the verifier outcome types. Both annotators independently evaluated all (100) trajectories and judged whether the executed action sequence preserved the declared plan structure.

Human–human agreement. Across the (100) trajectories, A1 and A2 achieved (71%) raw agreement and Cohen's (κ=0.35) on the binary structure-maintained judgment. The agreement varied between models: Gemma-4-26B was achieved (κ = 0.42), DeepSeek-V4 (κ = 0.37) and Qwen3.6-35B (κ = 0.19) ((n=18) for the Qwen3.6 subset), indicating that deciding whether a free-form ReAct trajectory preserves a declared plan can itself be ambiguous for human annotators.

Verifier–human agreement. The rule-based verifier shows similar agreement with both annotators: (κ=0.31) against A1 and (κ=0.34) against A2. These values are comparable to the human–human agreement of (κ=0.35), suggesting that disagreement with the verifier is of similar magni-

tude to disagreement between independent human judgments rather than being dominated by one annotator or a systematic verifier bias.

F PLAN QUALITY JUDGE

Judge model. Plan quality and plan adherence are scored with a fixed LLM judge. For the DeepSeek-V4-Flash and Qwen3.6-35B ALFWorld sweeps, we use `gemma-4-26B-A4B-it` for every cell, ensuring that trajectories are not evaluated by the model that produced them and that mode- and model-level comparisons use a common judging scale. The judge identity is stored explicitly in each output. Every task in each cell is judged without subsampling, and each metric is evaluated in a separate call.

For Gemma-4 agent runs, self-judging is avoided by using DeepSeek-V4-Flash as the judge. To quantify judge-specific differences, this second judge also re-scores four cells previously evaluated by Gemma-4. Because the resulting offsets are mode-dependent, results from the two judges are reported separately rather than pooled.

Plan Quality: goal + initial plan only; execution withheld

You are a meticulous and analytical PLAN QUALITY evaluator.
Your task is to evaluate the intrinsic quality of the initial written plan using only: (1) the user's goal, (2) the tools available when the plan was created, and (3) the initial plan itself.
CRITICAL: You must not use or reference execution information. Do not use agent actions, tool outputs, observations, errors, replans, or the final answer to judge the quality of the initial plan. You are evaluating whether the plan was a good strategy when it was written, not whether it eventually succeeded.
Plan Extraction Procedure: 1. Scan for the first section explicitly labeled with a PLAN keyword. This is the initial plan. 2. If no explicitly labeled PLAN section exists, infer the plan from the initial Thinking or planning section. 3. If no plan can be identified, output: "I cannot find a plan." 4. Do not infer missing plan steps from the execution trace.
TOOLS
Evaluate the initial plan according to the following criteria:
1. Goal Coverage: Does the plan address all important parts of the user's goal? Are any necessary subtasks missing?
2. Tool Selection: Does the plan select appropriate tools from those available? Does it ignore a clearly more suitable or efficient tool? Does it propose using a tool that does not exist?
3. Tool Feasibility: Are the planned tool calls consistent with the tools' descriptions, capabilities, and required inputs?
4. Step Structure: Are the planned steps clear, actionable, and ordered logically? Are there unnecessary, redundant, or unsupported steps?
5. Use of Available Information: Does the plan avoid unnecessary work when the required information is already provided in the goal or context?
6. Efficiency: Is the plan a reasonable and efficient strategy given the available resources, rather than merely a theoretically possible sequence of actions?
List only inherent flaws in the written plan. Do not infer flaws from later execution outcomes.
You must assign a single numerical score from 0 to 3:
3: The plan is well-structured, feasible, efficient, and directly addresses the goal. Necessary steps are present, logically ordered, and compatible with the available tools. No important unnecessary or unsupported steps are present.
2: The plan generally addresses the goal and is feasible, but contains minor issues such as unclear steps, small omissions, unnecessary actions, weak ordering, or insufficient detail.
1: The plan partially addresses the goal but contains substantial omissions, inefficiencies, unsupported assumptions, or tool-use problems that could prevent successful completion.
0: The plan does not meaningfully address the goal or is infeasible. Critical steps are missing, irrelevant, unsupported, or rely on unavailable or incorrectly used tools.
Be critical. For every identified issue, refer to the relevant plan step where possible and explain the problem specifically.
Please respond using exactly the following template:
Initial Plan Identification [Paste the initial plan or state: "I cannot find a plan."]
Plan Quality Analysis [Evaluate the plan using only the goal, available tools, and written plan.]
Verdict on Plan Flaws [List only intrinsic flaws in the written plan.]
Criteria: ¡Briefly restate the evaluation criteria applied.¿
Supporting Evidence: ¡Explain the score, tied to specific plan steps and criteria.¿

Score: <0, 1, 2, or 3>

G Structure Preservation by Declared Planning Mode

Hierarchical/Search plans are particularly difficult for generic ReAct execution to preserve. We next analyze structural maintenance by planning mode, the planning mode-level breakdown in Table 11, show that the declaration–execution gap is most pronounced for richer planning structures. Hierarchical declarations are preserved poorly across benchmarks: on ALFWorld, structure maintenance is only 11.4%, 4.6%, and 0.8% for DeepSeek-V4, Qwen3.6-35B, and Gemma-4-26B, respectively; on Mind2Web, the corresponding values are 14.5%, 17.3%, and 22.6%. SWE-bench shows the same pattern, with Hierarchical maintenance of 17.3% and 14.5% for DeepSeek-V4 and Qwen3.6-35B. Search is similarly difficult to preserve where enough declarations are observed, although several Search cells are sparsely populated and its competing-candidate structure is not directly comparable to a single ordered plan. Together with the plan-length analysis, these results indicate that a shared generic Plan+ReAct executor is particularly unreliable at maintaining richer multi-level or multi-candidate planning structures.

Table 11 provides the mode-level breakdown of Plan+ReAct structure preservation. Hierarchical declarations exhibit consistently low structure maintenance across benchmarks, while Search is also difficult to preserve where sufficient declarations are available. These results complement the main-text plan-length analysis and show that generic ReAct execution is particularly poorly suited to richer multi-level or multi-candidate planning structures.

Table 11: Plan+ReAct structure preservation by declared planning mode. Hierarchical declarations exhibit consistently low structure maintenance across benchmarks, while Search is also difficult to preserve where enough declarations are available. n denotes the smallest per-seed count when multiple seeds are available; values are mean $\pm$ standard deviation across seeds. ‡ indicates cells with fewer than 15 declarations in at least one seed and should be interpreted cautiously. Search is included for completeness, but its competing-candidate structure is not directly comparable to a single ordered Sequential, Predefined, or Hierarchical plan.

Benchmark	Model	Declared mode	n	Structure maintained	Steps declared
ALFWorld	DeepSeek-V4	Sequential	107/seed	30.5±5.0%	5.3±0.1
	DeepSeek-V4	Predefined‡	2	0.0%	5.5
	DeepSeek-V4	Hierarchical‡	5/seed	11.4±8.4%	9.5±0.3
	DeepSeek-V4	Search‡	5/seed	0.0±0.0%	16.8±0.9
	Qwen3.6-35B	Sequential	89/seed	30.5±2.2%	5.0±0.1
	Qwen3.6-35B	Hierarchical	28/seed	4.6±0.8%	9.1±0.4
	Gemma-4-26B	Sequential	36/seed	49.4±0.6%	4.3±0.1
	Gemma-4-26B	Hierarchical	41/seed	0.8±0.8%	10.0±0.2
Mind2Web	DeepSeek-V4	Sequential	808/seed	56.6±0.9%	4.1±0.1
	DeepSeek-V4	Predefined	363/seed	29.0±1.5%	4.7±0.0
	DeepSeek-V4	Hierarchical	104/seed	14.5±0.3%	9.8±0.2
	DeepSeek-V4	Search‡	3/seed	0.0±0.0%	10.4±0.9
	Qwen3.6-35B	Sequential	678/seed	51.0±0.4%	3.6±0.0
	Qwen3.6-35B	Predefined	64/seed	25.8±4.8%	4.1±0.1
	Qwen3.6-35B	Hierarchical	481/seed	17.3±1.8%	6.6±0.1
	Qwen3.6-35B	Search‡	3/seed	6.7±9.4%	5.8±0.6
	Gemma-4-26B	Sequential	182/seed	69.8±3.3%	2.8±0.0
	Gemma-4-26B	Predefined‡	8/seed	15.1±2.6%	4.0±0.1
	Gemma-4-26B	Hierarchical	1047/seed	22.6±0.4%	6.1±0.1
	Gemma-4-26B	Search	55/seed	0.8±0.8%	8.3±0.1
SWE-bench	DeepSeek-V4	Sequential	198/seed	23.0±1.3%	4.6±0.1
	DeepSeek-V4	Predefined	229/seed	29.6±2.8%	4.1±0.1
	DeepSeek-V4	Hierarchical‡	13/seed	17.3±5.8%	11.0±0.2
	DeepSeek-V4	Search‡	1/seed	0.0±0.0%	13.0±3.0
	Qwen3.6-35B	Sequential	201/seed	21.4±1.5%	4.7±0.3
	Qwen3.6-35B	Predefined	186/seed	28.2±0.3%	4.3±0.0
	Qwen3.6-35B	Hierarchical	30/seed	14.5±1.1%	8.6±0.1

Qualitative trajectories reveal how the declared structure is lost during generic execution. Figure 4 provides representative trajectories that complement the aggregate results. In the maintained Plan+ReAct example, all declared hierarchical units are reached in their original structural

order, even though additional environment actions occur between them. In contrast, the non-maintained example skips multiple units of the declared predefined plan while continuing with later parts of the trajectory. This illustrates that a generic ReAct executor can depart from the committed plan structure while continuing to act in the environment. The pattern-specific executor shows a different behavior: the hierarchical plan is explicitly traversed through its declared tree, with all 13 units dispatched according to the prescribed structure. These examples illustrate the distinction captured quantitatively in Tables 1 and 11. This suggests that providing a plan as context does not enforce its structure, whereas a pattern-specific executor makes that structure part of the execution control flow.

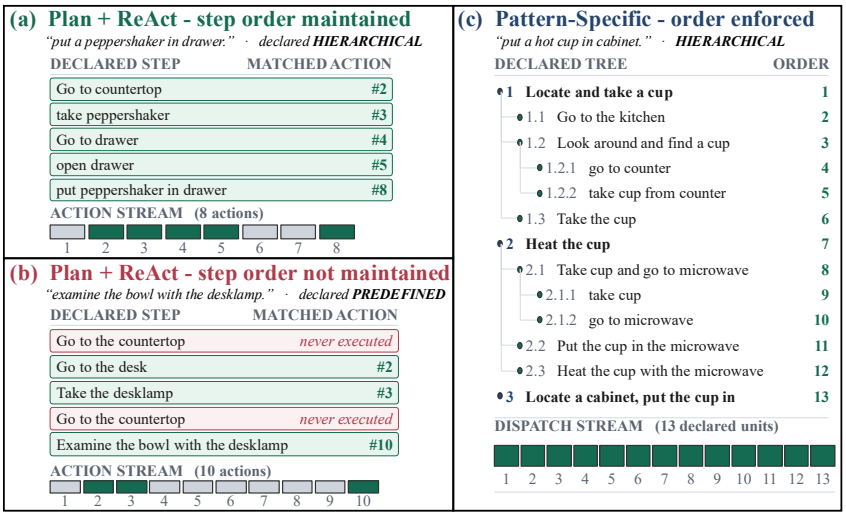


Figure 4: Qualitative examples of plan-structure preservation. (a) A Plan+ReAct trajectory that maintains the declared hierarchical structure: declared units are reached in order despite intervening actions. (b) A Plan+ReAct trajectory that does not maintain the declared predefined structure, skipping two plan units before continuing with later ones. (c) A routed pattern-specific hierarchical executor, where the declared tree directly determines the dispatch sequence. Green denotes maintained/dispatched plan units and red denotes declared units that are not reached.

H Additional Plan Adherence Results

Table 12: Plan adherence on ALFWorld under the fixed execution budget, evaluated on 134 tasks with three random seeds per model. Values are mean ± standard deviation across seeds. *Adherence* is the fraction of declared executable plan units completed within the allocated budget; *Full Plan* is the percentage of tasks for which all declared units were completed. Hierarchical planning is evaluated over executable leaf nodes.

Pattern	Model	Planned / Task	Completed / Task	Adherence	Full Plan Completed (%)
Predefined	DeepSeek-V4	5.917±0.023	5.167±0.040	0.871±0.009	58.7±1.1
	Qwen3.6-35B	5.867±0.132	5.053±0.235	0.864±0.018	54.0±3.5
	Gemma-4	6.243±0.040	5.797±0.121	0.930±0.023	72.9±7.5
Sequential	DeepSeek-V4	4.070±0.056	4.070±0.056	1.000±0.000	100.0±0.0
	Qwen3.6-35B	4.023±0.057	4.023±0.057	1.000±0.000	100.0±0.0
	Gemma-4	3.810±0.108	3.810±0.108	1.000±0.000	100.0±0.0
Hierarchical	DeepSeek-V4	13.063±0.193	10.440±0.135	0.812±0.005	37.1±4.8
	Qwen3.6-35B	10.480±0.087	8.177±0.107	0.793±0.006	30.3±3.1
	Gemma-4	13.177±0.266	11.010±0.263	0.845±0.016	41.3±3.0
Search	DeepSeek-V4	3.337±0.055	3.320±0.046	0.996±0.004	98.5±1.3
	Qwen3.6-35B	3.223±0.047	3.223±0.047	1.000±0.000	100.0±0.0
	Gemma-4	2.993±0.006	2.993±0.006	1.000±0.000	100.0±0.0

Plan completion is conditioned on execution budget and plan size. Plan adherence measures how much of the still-needed plan is completed within the available interaction budget. Full adherence results for Mind2Web and SWE-bench are reported in Tables 13 and 14, respectively.

Mind2Web additionally provides a more dependency-sensitive setting: browser tasks often require a sequence of prerequisite interactions before the final goal becomes reachable. Consequently, skipping a necessary intermediate plan unit can prevent subsequent actions from being executed successfully. This makes plan adherence particularly informative for web navigation, while still distinguishing necessary execution units from redundant or obsolete steps in the generated plan.

Table 13: Mind2Web: Plan adherence under pattern-specific execution. *Planned/task* and *Completed/task* report the mean number of needed plan units generated and completed per task, respectively. Adherence measures the fraction of needed plan units completed within the execution budget, while *Full plan* reports the fraction of trajectories completing all needed units. Values are mean ± standard deviation across seeds 7, 13, and 42. Search operates over competing candidate strategies and its completion values are therefore not directly comparable to the ordered planning patterns.

Model	Planning mode	Planned/task	Completed/task	Adherence	Full plan
DeepSeek-V4	Predefined	4.597±0.023	2.590±0.000	0.558±0.002	9.2±0.1%
	Sequential	4.443±0.012	3.543±0.012	0.800±0.003	40.0±0.7%
	Hierarchical	4.800±0.010	2.637±0.006	0.542±0.002	1.2±0.4%
	Search	3.467±0.015	3.433±0.015	0.989±0.001	96.5±0.2%
Qwen3.6-35B	Predefined	4.987±0.006	2.597±0.015	0.521±0.003	6.8±0.9%
	Sequential	4.540±0.050	3.603±0.021	0.797±0.006	42.5±2.0%
	Hierarchical	5.233±0.006	2.653±0.006	0.504±0.002	1.9±0.3%
	Search	3.353±0.021	3.070±0.046	0.915±0.011	75.0±2.7%
Gemma-4-26B	Predefined	3.810	1.270	0.344	3.3%
	Sequential[§]	3.660	3.490	0.953	86.3%
	Hierarchical	7.380	1.950	0.312	0.3%
	Search	3.000	2.490	0.830	64.7%

Model	Planning mode	Planned/task	Completed/task	Adherence	Full plan
DeepSeek-V4	Predefined	4.790	1.410	0.297	1.0%
	Sequential[§]	3.770	3.770	1.000	100.0%
	Hierarchical	13.650	2.040	0.163	0.5%
	Search	3.260	3.020	0.927	83.0%
Qwen3.6-35B	Predefined	5.490	1.700	0.311	1.0%
	Sequential[§]	3.750	3.750	1.000	100.0%
	Hierarchical	9.370	2.390	0.271	0.2%
	Search	3.770	2.920	0.777	49.8%

Table 14: SWE-bench Verified: plan adherence under pattern-specific execution. Columns as in the Mind2Web table. Both models ran at seed 13 only, so no row carries a spread. HIERARCHICAL declares by far the most units (13.7 and 9.4 per task against 3.3–5.5 for the other patterns) and completes the smallest fraction of them. [§] marks the same SEQUENTIAL caveat as above.

I PLAN-QUALITY ANALYSIS

Judged plan quality does not explain pattern-specific success. One possible explanation for the performance differences in Table 2 is that some planning patterns simply generate higher-quality plans, as shown in Table 15. Our ALFWorld analysis does not support this explanation. For DeepSeek-V4, Sequential receives the highest judged plan quality (2.84±0.03), yet Search achieves substantially higher task success (0.91±0.03 versus 0.64±0.02) despite a lower quality score (2.53±0.03). Similarly, for Qwen3.6-35B, Hierarchical receives the highest plan-quality score (2.82±0.00), whereas Search achieves the highest task success (0.79±0.00) despite the lowest score (1.94±0.17). At the task level, plan-quality and success are only weakly correlated ($|r| \leq 0.181$), consistent with the same conclusion under an independent judging configuration. Thus, the advantage of a planning pattern cannot be explained simply by how good its plan appears in isolation; performance also depends on how well the planning structure fits the task and its execution environment.

Together, these results show that planning effectiveness varies across environments and models, but that the raw per-task oracle overstates the value of task-specific selection: once repeated execution is controlled, little additional advantage remains over the strongest fixed pattern. Moreover, pattern-specific success cannot be explained by generic judgments of plan quality, which remain

only weakly associated with task success. The remaining question is whether realizing these gains requires executing the selected planning structure through its corresponding executor, which we examine next.

Table 15: Plan quality (PQ) and task success on ALFWorld. Plans are evaluated by a Gemma-4-26B judge. Forced-pattern results report the mean ± half-range across seeds ($n = 134$ tasks per seed). The final column reports the task-level Pearson correlation between plan quality and task success separately for seeds 13 and 7. †Plan+ReAct is reported for one seed. Bold indicates the highest PQ or TSR for each model.

Mode	Model	PQ	TSR	PQ–success r
Predefined	DeepSeek-V4	2.75±0.01	0.55±0.00	−0.031, −0.005
	Qwen3.6-35B	2.44±0.05	0.56±0.03	+0.045, +0.095
Sequential	DeepSeek-V4	**2.84±0.03**	0.64±0.02	−0.025, −0.061
	Qwen3.6-35B	2.66±0.03	0.68±0.01	+0.090, −0.040
Hierarchical	DeepSeek-V4	2.72±0.02	0.85±0.00	−0.060, −0.035
	Qwen3.6-35B	**2.82±0.00**	0.74±0.03	+0.140, −0.031
Search	DeepSeek-V4	2.53±0.03	**0.91±0.03**	+0.026, +0.146
	Qwen3.6-35B	1.94±0.17	**0.79±0.00**	+0.158, +0.181
Plan+ReAct†	DeepSeek-V4	**2.84**	0.470	−0.017
	Qwen3.6-35B	2.78	0.549	+0.104

J Pattern-Ceiling Analysis

Comparing execution conditions on task success alone conflates two distinct questions: whether a pattern-specific executor can solve a task at all, and whether the declaration module selects the pattern that solves it. We therefore measure the two separately. For every benchmark–model pair we first build a *planning pattern-ceiling matrix* by running each of the four planning patterns on every task under forced dispatch, i.e. bypassing the declaration module and dispatching pattern p regardless of what the model would have declared.

Definitions. Let $\mathcal{T} = \{1, \ldots, N\}$ be the tasks of a benchmark and $\mathcal{P} = \{\text{PREDEFINED}, \text{SEQUENTIAL}, \text{HIERARCHICAL}, \text{SEARCH}\}$ the planning patterns. The ceiling matrix $M \in \{0,1\}^{N \times |\mathcal{P}|}$ has entries

$$m_{i,p} = \mathbb{1}\left[\text{forced dispatch of pattern } p \text{ solves task } i\right],$$

scored by the benchmark's own evaluation protocol. From M we derive four quantities:

- **Per-planning-pattern success rate** (column mean), $S(p) = \frac{1}{N}\sum_i m_{i,p}$. This is the task success rate of a fixed policy that always executes p.
- **Single-best-pattern policy**, $S^\star = \max_p S(p)$, attained by $p^\star$. This is the strongest policy available *without* per-task selection, and therefore the baseline any router must outperform to demonstrate that selection is doing work.
- **Oracle ceiling**, $\mathrm{DSR}_{\max} = \frac{1}{N}\sum_i \max_p m_{i,p}$: the success rate of a policy that selects the correct pattern for every task. Tasks with $\max_p m_{i,p} = 0$ are solved by no pattern; we call these NONE tasks, and no declaration policy can affect them.
- **Routing headroom**, $H = \mathrm{DSR}_{\max} - S^\star \geq 0$. This is the entire budget available to per-task selection. H is strictly positive only when pattern wins are complementary, when some task solved by a weaker pattern is *not* solved by $p^\star$. If patterns succeed on nested subsets of tasks, $H = 0$ and no selection policy, however good, can improve on always executing $p^\star$.

Evaluating a declaration policy. A declaration policy $\pi : \mathcal{T} \to \mathcal{P}$ achieves $\mathrm{TSR}(\pi) = \frac{1}{N}\sum_i m_{i,\pi(i)}$. Because $\mathrm{TSR}(\pi)$ is a lookup into M, it can be evaluated without re-running any executor: the declaration is the only model call per task. This *free-lookup* protocol lets us compare declaration variants (zero-shot, few-shot, thinking on/off) under a fixed execution substrate, so differences are attributable to selection rather than to execution stochasticity.

$\mathrm{TSR}(\pi)$ alone, however, does not establish that π selects *per task*. A policy that emits a healthy spread of patterns but chooses them independently of the task achieves, in expectation,

$$\mathrm{TSR}_{\text{null}}(\pi) \;=\; \frac{1}{N}\sum_p n_p\, S(p), \qquad n_p = |\{i : \pi(i) = p\}|,$$

the *distribution-matched null*: the same marginal pattern distribution as π, but zero task-conditional information. We therefore report the excess $\Delta(\pi) = \mathrm{TSR}(\pi) - \mathrm{TSR}_{\text{null}}(\pi)$, which isolates per-task discrimination from any global prior over patterns that the prompt may have induced. $\Delta(\pi) \approx 0$ with a broad distribution and $\Delta(\pi) \approx 0$ with a degenerate one are both selection failures, and TSR distinguishes neither. We additionally report performance restricted to the solvable subset $\{i : \max_p m_{i,p} = 1\}$, since NONE tasks contribute zero under every policy and dilute all of these quantities toward zero.

Separating selection failure from dispatch failure. The ceiling matrix and the trace verifier isolate different failure modes. A *dispatch* failure means the executed pattern differs from the declared one; the verifier detects this directly, and it is the failure mode Planning-as-Routing is designed to eliminate. A *selection* failure means dispatch was faithful but the declared pattern was the wrong one for the task; it appears as $\mathrm{TSR}(\pi) < S^\star$ with the verifier reporting full conformance. This distinction is what allows us to attribute the residual gap in Section 5 to the declaration module rather than to the router or the executors.

Stochasticity and interpretation of the oracle ceiling. Each forced-pattern execution is stochastic. We therefore compute $S(p)$, $S^\star$, $\mathrm{DSR}_{\max}$, and H independently for each random seed and report the mean and standard deviation across seeds. Thus, $\mathrm{DSR}_{\max}$ represents the expected empirical per-task ceiling when each planning pattern receives one independently sampled execution.

Because $\mathrm{DSR}_{\max}$ takes the maximum across multiple stochastic pattern executions, part of its advantage over the strongest fixed pattern could arise from repeated opportunities for success rather than from planning-pattern complementarity alone. We therefore separately compare the oracle and ranked-fallback results against pass@k controls obtained by repeating the strongest fixed planning pattern under matched numbers of attempts. This separates gains due to trying different planning patterns from gains obtainable by simply retrying the same strong pattern.

Table 16: Repeated-rollout ReAct control for Search. 3×Flat ReAct executes three independent trajectories from a reset environment and applies the same rubric-based judge used by Search to select one trajectory. Search dynamically chooses between 0 and 4 candidate plans per task, with four as the configured maximum. The realized candidate count averages 3.35 ± 0.49 across Search episodes (Appendix D, Table 10). Results in this table report mean ± standard deviation across seeds 7, 13, and 42. This control tests how much of Search's performance can be reproduced by repeated generic ReAct execution and trajectory selection.

Benchmark	Model	Metric	Flat ReAct	3×Flat ReAct + Judge	Flat ReAct Pass@3	Search
ALFWorld	DeepSeek-V4	TSR	0.440±0.012	0.526±0.011	0.600±0.015	**0.918±0.026**
	Qwen3.6-35B	TSR	0.535±0.007	0.664±0.015	0.758±0.004	**0.796±0.004**
Mind2Web	DeepSeek-V4	TSR	**0.058±0.003**	**0.059±0.004**	0.077±0.002	**0.057±0.006**
		SSR	**0.434±0.001**	0.448±0.005	0.504±0.001	**0.449±0.000**
	Qwen3.6-35B	TSR	0.042±0.002	0.044±0.000	0.069±0.001	**0.052±0.001**
		SSR	0.366±0.003	0.385±0.002	0.466±0.002	**0.411±0.003**
SWE-bench	DeepSeek-V4	PSR	0.384±0.010	**0.410±0.007**	0.513±0.003	0.400±0.018
	Qwen3.6-35B	PSR	0.179±0.034	0.304±0.023	0.381±0.019	0.296±0.040
WebArena	DeepSeek-V4	TSR	0.455±0.000	0.452±0.031	0.594±0.004	**0.580±0.006**
	Qwen3.6-35B	TSR	0.332±0.009	0.386±0.017	0.548±0.037	**0.577±0.026**

The advantage of pattern-specific execution becomes more pronounced on longer tasks. Figure 6 shows that the benefit of matching the executor to the declared planning pattern is largest as task length increases. On ALFWorld, Flat ReAct and Plan+ReAct degrade sharply on longer tasks, whereas Hierarchical and Search execution remain substantially more robust, especially for DeepSeek-V4 and Qwen3.6. This separation is modest on short tasks, where several execution strategies perform similarly, but widens as more interaction steps are required. Mind2Web shows the same general difficulty with increasing task length: success decreases for all methods, but pattern-specific executors remain competitive with or above Plan+ReAct across most length bins. These results suggest that simply providing a plan to a generic ReAct loop is often sufficient for short tasks, but preserving the corresponding planning pattern becomes increasingly important as execution horizons grow.

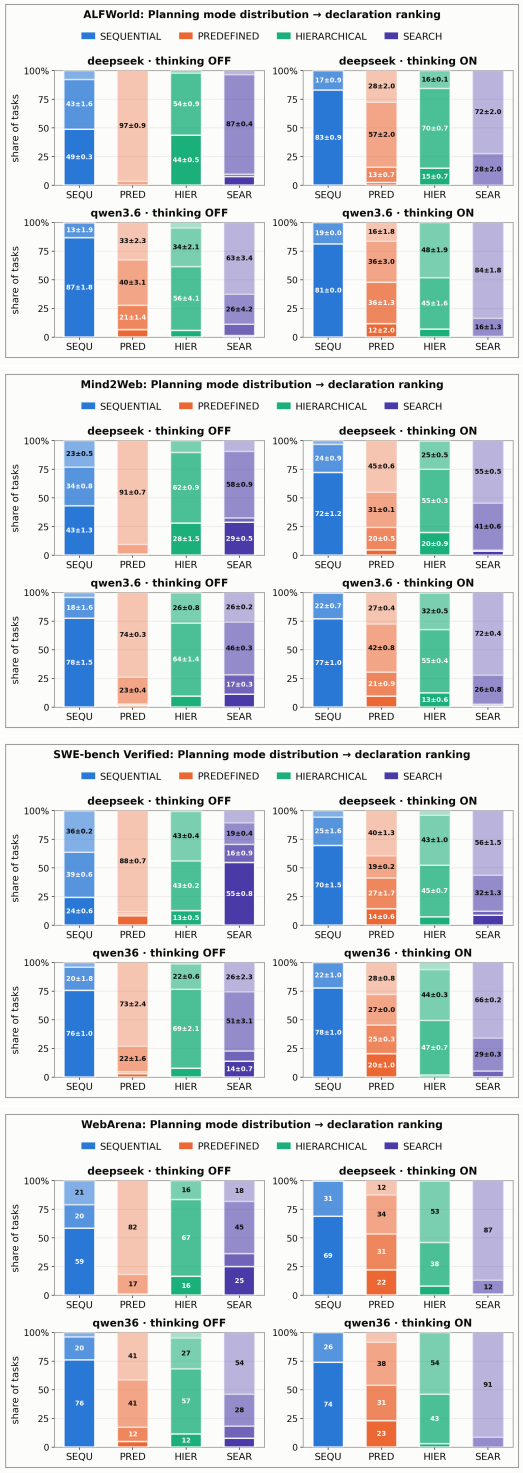


Figure 5: Distribution of declaration ranks assigned to each planning mode on ALFWolrd, Mind2Web, SWE-Bench Verified and WebArena (right), averaged across three seeds. Each stacked bar shows the fraction of tasks for which a planning mode is assigned Rank 1–4. Results are shown for DeepSeek-V4 and Qwen3.6 with thinking enabled and disabled. The strongly non-uniform distributions reveal systematic mode preferences in the declaration policy.

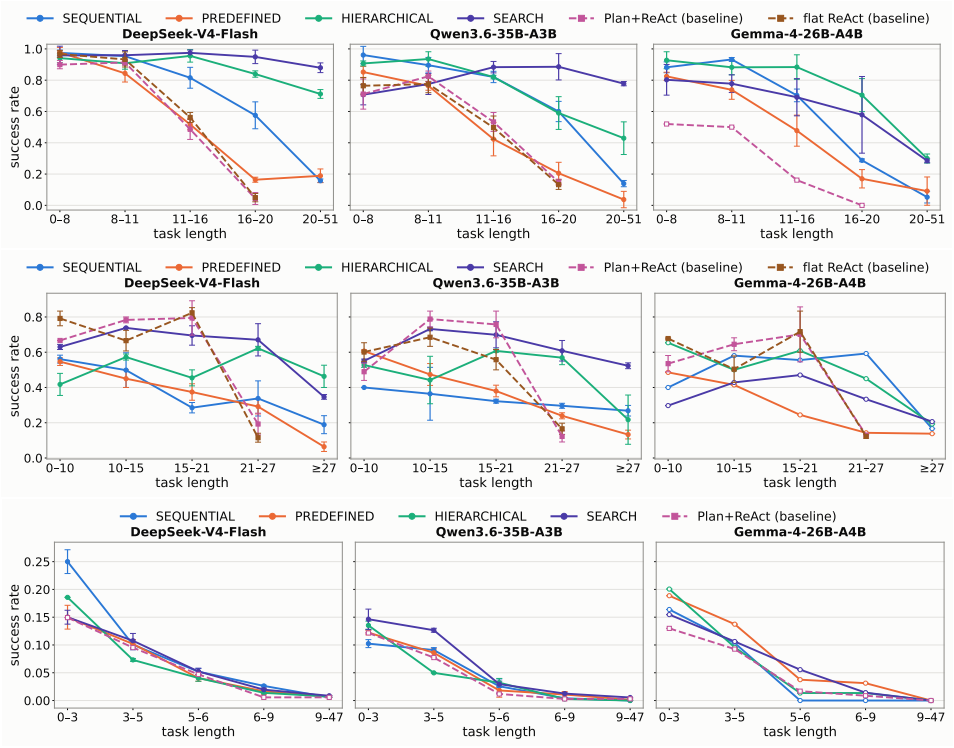


Figure 6: Task success as a function of task length under generic and pattern-specific execution. Pattern-specific execution provides the largest advantage on longer tasks, where Plan+ReAct and Flat ReAct degrade more rapidly. Error bars denote variation across seeds.

J.1 Repeated-rollout ReAct control for Search.

Search differs from the other planning patterns because it can execute multiple candidate trajectories before selecting one with a rubric-based judge. The Search planner dynamically selects between 0 and 4 candidate plans per task, rather than always executing the maximum of four. Across all Search runs, it realizes 3.35 ± 0.49 candidates per task on average (See Appendix D, Table 10). Thus, the Flat ReAct control of three rolls provides a rollout count of approximately matched overall. We therefore test how much of its advantage can be reproduced by repeated generic ReAct execution. For each task, we run Flat ReAct three times independently from a reset environment and apply the same rubric-based judge used by Search to select among the resulting trajectories. We repeat this control across three different seeds.

Table 16 shows that repeated execution with the same rubric-based judge generally improves Flat ReAct, but does not fully account for the gains of Search. Across ALFWorld and WebArena, Search remains substantially stronger than $3\times$Flat ReAct + Judge, with improvements ranging from 0.132 to 0.392 on ALFWorld and from 0.128 to 0.191 on WebArena. On Mind2Web, the differences are smaller and more model dependent: Search is slightly below repeated ReAct for DeepSeek-V4 in TSR (0.057 versus 0.059), but remains higher in SSR and for both metrics with Qwen3.6-35B. On SWE-bench, repeated ReAct performs slightly better than Search for both models (0.410 versus 0.400 for DeepSeek-V4 and 0.304 versus 0.296 for Qwen3.6-35B). This is consistent with the forced-pattern results, where Hierarchical rather than Search is the strongest planning pattern on SWE-bench. Overall, repeated execution explains part of the gain from additional rollout budget, but Search retains an advantage in most benchmark–model settings.

To distinguish candidate quality from trajectory-selection quality, we measured Flat ReAct pass@3, as shown in Table 16. For each task, pass@3 counts the task as successful if any of the three independent Flat ReAct trajectories succeeds, thereby providing hindsight selection over exactly the same rollout set used by $3\times$Flat ReAct + Judge. The difference between pass@3 and the judge-

selected result therefore measures how much attainable success is lost by trajectory selection, while comparing Search against Flat ReAct pass@3 tests whether Search generates stronger candidate trajectories than repeated generic ReAct alone.

J.2 Repeated-fixed-pattern control.

For a fixed planning pattern p, we define

$$\text{pass@}k(p) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left[\max_{r\in\{1,\dots,k\}} m_{i,p}^{(r)} = 1\right], \tag{1}$$

where $m_{i,p}^{(r)}$ denotes the outcome of the rth independent execution of pattern p on task i. We use $p^\star$, the strongest fixed planning pattern, as the primary retry baseline. We compare TSR@2 and TSR@3 against pass@2($p^\star$) and pass@3($p^\star$), respectively, and compare the four-pattern forced oracle against pass@4($p^\star$).

Table 17: Retry control for the forced-pattern ceiling. For each benchmark–model pair, $p^\star$ denotes the strongest fixed planning pattern under the primary task-level metric, and $S(p^\star)$ denotes its corresponding score. $\mathrm{DSR}_{\max}$ denotes the per-task forced-pattern oracle over the evaluated planning patterns, and $H = \mathrm{DSR}_{\max} - S(p^\star)$ is the original oracle gap. pass@k reports performance when the same fixed pattern $p^\star$ is executed independently k times, with a task counted as successful if any of the k executions succeeds. $H_{\text{retry}}@3 = \mathrm{DSR}_{\max} - \text{pass@3}(p^\star)$ denotes the residual oracle gap after three executions of the same fixed pattern. Negative values indicate that repeated execution of the fixed pattern exceeds the forced-pattern oracle. For Mind2Web, we report both task success rate (TSR) and step success rate (SSR). For Gemma-4-26B, $p^\star$ is Predefined because it is the strongest fixed pattern under TSR; the corresponding SSR values therefore use the same Predefined pattern rather than the SSR-maximizing Search pattern.

Benchmark	Model	Metric	n	$S(p^\star)$ (Pattern)	$\mathrm{DSR}_{\max}$	H	pass@2	pass@3	$H_{\text{retry}}@3$
ALFWorld	DeepSeek-V4	TSR	134	0.918 (Search)	0.953	+0.035	0.963	**0.978**	**−0.025**
	Qwen3.6-35B	TSR	134	0.796 (Search)	0.925	+0.129	0.896	**0.940**	**−0.015**
	Gemma-4-26B	TSR	134	0.600 (Hierarchical)	0.841	+0.241	0.754	0.813	**+0.027**
Mind2Web	DeepSeek-V4	TSR	1341	0.057 (Search)	0.097	+0.040	0.073	0.083	**+0.014**
		SSR	1341	0.449 (Search)	0.550	+0.101	0.500	0.526	**+0.024**
	Qwen3.6-35B	TSR	1341	0.053 (Search)	0.084	+0.031	0.071	0.083	**+0.001**
		SSR	1341	0.411 (Search)	0.505	+0.094	0.470	0.501	**+0.005**
	Gemma-4-26B	TSR	1341	0.147 (Predefined)	0.283	+0.135	0.237	0.292	**−0.010**
		SSR	1341	0.347 (Predefined)†	0.585	+0.238	0.461	0.524	**+0.062**
SWE-bench	DeepSeek-V4	PSR	500	0.442 (Hierarchical)	0.552	+0.107	0.503	0.538	**+0.014**
	Qwen3.6-35B	PSR	500	0.330 (Hierarchical)	0.445	+0.119	0.420	0.449	**−0.004**
WebArena	DeepSeek-V4	TSR	204	0.589 (Search)	0.736	+0.147	0.695	0.744	−0.009
	Qwen3.6-35B	TSR	204	0.578 (Search)	0.693	+0.116	0.665	0.693	−0.000

†For Gemma-4-26B on Mind2Web, Predefined is the strongest fixed pattern under TSR and is therefore used as the fixed-pattern retry control for both TSR and SSR. Search has the highest single-run SSR (0.388), but the reported SSR pass@2 and pass@3 values correspond to repeated execution of Predefined.

J.3 Task-Blind Declaration Null

K Comparison with prior work.

We compare forced planning strategies, model-declared routing, and the empirical ceiling with representative prior work to assess whether they achieve comparable task success across benchmarks and models. Because prior systems differ in backbone models, demonstrations, training, and inference procedures, we treat these results as benchmark-level context rather than controlled comparisons. On ALFWorld, AdaPlanner (Sun et al., 2023) reports 0.918 TSR with GPT-3 and 0.806 with GPT-3.5, using six expert demonstrations together with closed-loop plan refinement. On Mind2Web, TRAD (Zhou et al., 2024a) reports 0.021 TSR and 0.280 SSR, while the more recent ReasoningBank (Ouyang et al., 2026) reaches 0.051 TSR and 0.456 SSR using experience memory accumulated from prior trajectories. On SWE-bench Verified, SWE-agent (Yang et al., 2024) reports a 0.336 resolved rate with Claude-3.5. Finally, on full WebArena, PLAN-AND-ACT (Erdogan et al., 2025)

Table 18: Task-blind declaration null. $\mathrm{TSR}(\pi)$ denotes task success under the model's top-1 declared planning mode. $\mathrm{TSR}_{\mathrm{null}}$ preserves each model's overall declaration frequencies while randomly reassigning declarations across tasks. We report $\Delta(\pi) = \mathrm{TSR}(\pi) - \mathrm{TSR}_{\mathrm{null}}$, the 95% interval under 5,000 permutations, one-sided permutation p-values for $\Delta > 0$, two-sided p-values, and Holm-corrected p-values.

Benchmark	Model	n	$\mathrm{TSR}(\pi)$	$\mathrm{TSR}_{\mathrm{null}}$	$\Delta(\pi)$	95% null interval	$p\,(\Delta > 0)$	p (two-sided)	Holm
ALFWorld	DeepSeek-V4	134	0.721 ± 0.029	0.745 ± 0.011	$\mathbf{-0.024 \pm 0.035}$	$[-0.027, +0.026]$	0.972	0.062	1.000
	Qwen3.6-35B	134	0.667 ± 0.049	0.659 ± 0.028	$+0.008 \pm 0.021$	$[-0.017, +0.018]$	0.226	0.406	1.000
	Gemma-4-26B	134	0.560 ± 0.016	0.553 ± 0.012	$+0.007 \pm 0.005$	$[-0.013, +0.014]$	0.217	0.364	1.000
Mind2Web	DeepSeek-V4	1341	0.053 ± 0.003	0.051 ± 0.001	$+0.003 \pm 0.002$	$[-0.004, +0.004]$	0.094	0.180	0.944
	Qwen3.6-35B	1341	0.036 ± 0.001	0.034 ± 0.001	$+0.002 \pm 0.000$	$[-0.003, +0.003]$	0.126	0.251	1.000
	Gemma-4-26B	1341	0.113 ± 0.006	0.113 ± 0.007	$+0.001 \pm 0.000$	$[-0.001, +0.002]$	0.707	0.715	1.000
SWE-bench	DeepSeek-V4	500	0.415 ± 0.016	0.406 ± 0.018	$+0.009 \pm 0.002$	$[-0.014, +0.014]$	0.107	0.213	0.959
	Qwen3.6-35B	500	0.315 ± 0.005	0.317 ± 0.005	-0.001 ± 0.000	$[-0.011, +0.011]$	0.598	0.808	1.000
WebArena	DeepSeek-V4	204	0.460 ± 0.011	0.477 ± 0.005	-0.017 ± 0.005	$[-0.029, +0.028]$	0.627	0.710	1.000
	Qwen3.6-35B	204	0.404 ± 0.019	0.429 ± 0.005	$\mathbf{-0.025 \pm 0.009}$	$[-0.030, +0.033]$	0.925	0.215	1.000

reports 0.457 TSR with Llama-70B and 0.482 with QwQ-32B using a dedicated planner–executor architecture. These results show that our forced-pattern and routing-based evaluations operate in a performance regime comparable to representative prior systems, while our primary conclusions rely on the controlled Flat ReAct, Plan+ReAct, and Planning-as-Routing comparisons under the same models and execution protocol.

L Additional Analysis

Declarations provide no reliable task-specific advantage over a task-blind policy. For each model and benchmark, we compare the task success obtained from the declared planning mode, $\mathrm{TSR}(\pi)$, with a task-blind null policy, $\mathrm{TSR}_{\mathrm{null}}(\pi)$. The null preserves how frequently the model declares each planning mode, but removes the association between the declared mode and the individual task. We measure

$$\Delta(\pi) = \mathrm{TSR}(\pi) - \mathrm{TSR}_{\mathrm{null}}(\pi).$$

Using the top-1 declarations underlying the selection results, $\Delta(\pi)$ falls within the 95% interval of a 5,000-permutation null for every benchmark–model pair. No pair shows a significant advantage over the task-blind policy (minimum one-sided $p = 0.094$; all Holm-corrected $p \geq 0.944$). Across pairs, $\Delta(\pi)$ ranges from -0.025 to $+0.009$, with the largest observed deviations being negative (ALFWorld/DeepSeek-V4: $\Delta = -0.024$; WebArena/Qwen3.6-35B: $\Delta = -0.025$). Thus, we find no reliable evidence that current top-1 declarations match planning modes to individual tasks better than expected from each model's overall declaration frequencies.

Fallback through ranked planning-mode declarations improves success, but also adds execution attempts. We next evaluate the declaration module as a ranking over planning patterns rather than considering only its top-1 choice (Table 4). We define TSR@k as a sequential fallback policy: the agent first executes the rank-1 declared mode and, if the task is not solved, proceeds to the rank-2 mode and then the rank-3 mode. Task success increases substantially with k. On ALFWorld, DeepSeek-V4 improves from 0.721 at TSR@1 to 0.893 at TSR@2 and 0.948 at TSR@3, while Qwen3.6-35B increases from 0.667 to 0.804 and 0.873. The same trend appears across the other benchmarks: DeepSeek-V4 increases from 0.415 to 0.498 and 0.541 on SWE-bench, from 0.460 to 0.639 and 0.696 on WebArena, and from 0.053 to 0.075 and 0.092 on Mind2Web.

However, each additional rank also provides another opportunity to execute the task. We therefore compare TSR@k with pass@$k(p^\star)$, which repeats the strongest fixed planning pattern for the same number of attempts (Table 17). On ALFWorld, pass@3 reaches 0.978 for DeepSeek-V4 and 0.940 for Qwen3.6-35B, exceeding the corresponding TSR@3 values of 0.948 and 0.873. On SWE-bench, TSR@3 and pass@3 are nearly identical for DeepSeek-V4 (0.541 versus 0.538), while pass@3 is slightly higher for Qwen3.6-35B (0.449 versus 0.438). Thus, ranked fallback improves task success, but much of this gain can also be obtained by repeatedly executing a strong fixed planning pattern.

Additional deliberation does not consistently improve planning-mode declaration. Figure 5 shows that planning-mode rankings are strongly non-uniform across models and benchmarks, indi-

cating systematic preferences for particular modes. Enabling thinking changes these ranking distributions, but does not consistently improve the task success obtained from the top-1 declaration. For DeepSeek-V4, TSR@1 decreases from 0.721 to 0.656 on ALFWorld, 0.053 to 0.051 on Mind2Web, 0.415 to 0.411 on SWE-bench, and 0.460 to 0.384 on WebArena. Qwen3.6-35B shows only a small increase on ALFWorld ($0.667 \rightarrow 0.674$), while decreasing on Mind2Web ($0.036 \rightarrow 0.034$), SWE-bench ($0.313 \rightarrow 0.310$), and WebArena ($0.404 \rightarrow 0.394$). Thus, enabling thinking changes the model's planning-mode preferences, but does not consistently improve top-1 declaration performance.

Few-shot task–pattern demonstrations improve raw top-1 performance, but rarely task-specific discrimination. Since enabling thinking alone does not reliably improve planning-mode declaration, we next test whether explicit task–pattern demonstrations can better guide the declaration model. The few-shot demonstrations are constructed from held-out forced-pattern executions and are disjoint from the evaluation tasks (Appendix O.1). Table 4 shows that few-shot prompting improves top-1 declaration performance in most evaluated configurations. The largest gain occurs on ALFWorld for DeepSeek-V4 with thinking enabled, where TSR increases from 0.656 to 0.812 ($+0.156$). Improvements are smaller on the other benchmarks: up to $+0.018$ on Mind2Web, $+0.027$ on SWE-bench, and $+0.054$ on WebArena. One configuration decreases slightly (DeepSeek-V4 on SWE-bench with thinking disabled, -0.011), while the two negative changes on DeepSeek-V4 Mind2Web TSR are negligible (-0.003 and -0.002), and two configurations remain unchanged. Overall, task–pattern demonstrations provide a more consistent benefit than enabling thinking alone, although the magnitude of the gain depends on the benchmark and model.

The smaller gains on Mind2Web suggest that improving planning-mode declaration alone does not necessarily translate into large downstream improvements. Mind2Web requires correct element grounding and operation selection across an entire multi-step trajectory, so errors during execution can still dominate even when the declared planning mode improves. Its diversity across websites and task types may also make a small set of task–pattern demonstrations less informative than in more regular environments.

M Inference Cost Analysis

Inference-cost characteristics. We next examine whether the gains from pattern-specific execution can be explained simply by greater inference cost. We compare aggregate token usage, the fraction spent on thinking, and the number of LLM calls across execution patterns. As shown in Appendix M, Figs. 7 and 8, both Hierarchical and Search generally use more aggregate tokens than Flat ReAct, Plan+ReAct, Sequential, and Predefined execution, largely because they invoke the LLM more frequently for decomposition, coordination, or candidate exploration. However, these additional calls are not necessarily longer: Search often makes the largest number of calls while using fewer tokens per call than Hierarchical execution. Moreover, neither total token usage nor the number of LLM calls tracks task success monotonically. For instance, on ALFWorld, Search uses fewer aggregate tokens than Hierarchical for DeepSeek-V4 and Qwen3.6-35B while achieving higher task success. These results suggest that inference volume alone does not explain the observed performance differences. Detailed inference-cost statistics are reported in Appendix M.

N Extended Discussion

In this work, we study whether LLM agents execute the planning approach they declare, whether different environments and models benefit from different planning patterns, whether the execution architecture matters beyond simply generating a plan, and whether current LLMs can select an effective planning mode for an individual task. We introduce Planning-as-Routing, which separates planning-mode declaration from execution by dispatching Predefined, Sequential, Hierarchical, and Search declarations to their corresponding pattern-specific executors. Our experiments across embodied, web, and software-engineering environments reveal four main findings.

First, generating a structured plan does not guarantee that a generic ReAct executor preserves that structure during execution. Under Plan+ReAct, the declared planning structure is maintained in only a minority of trajectories, and structure preservation decreases substantially as plans become longer.

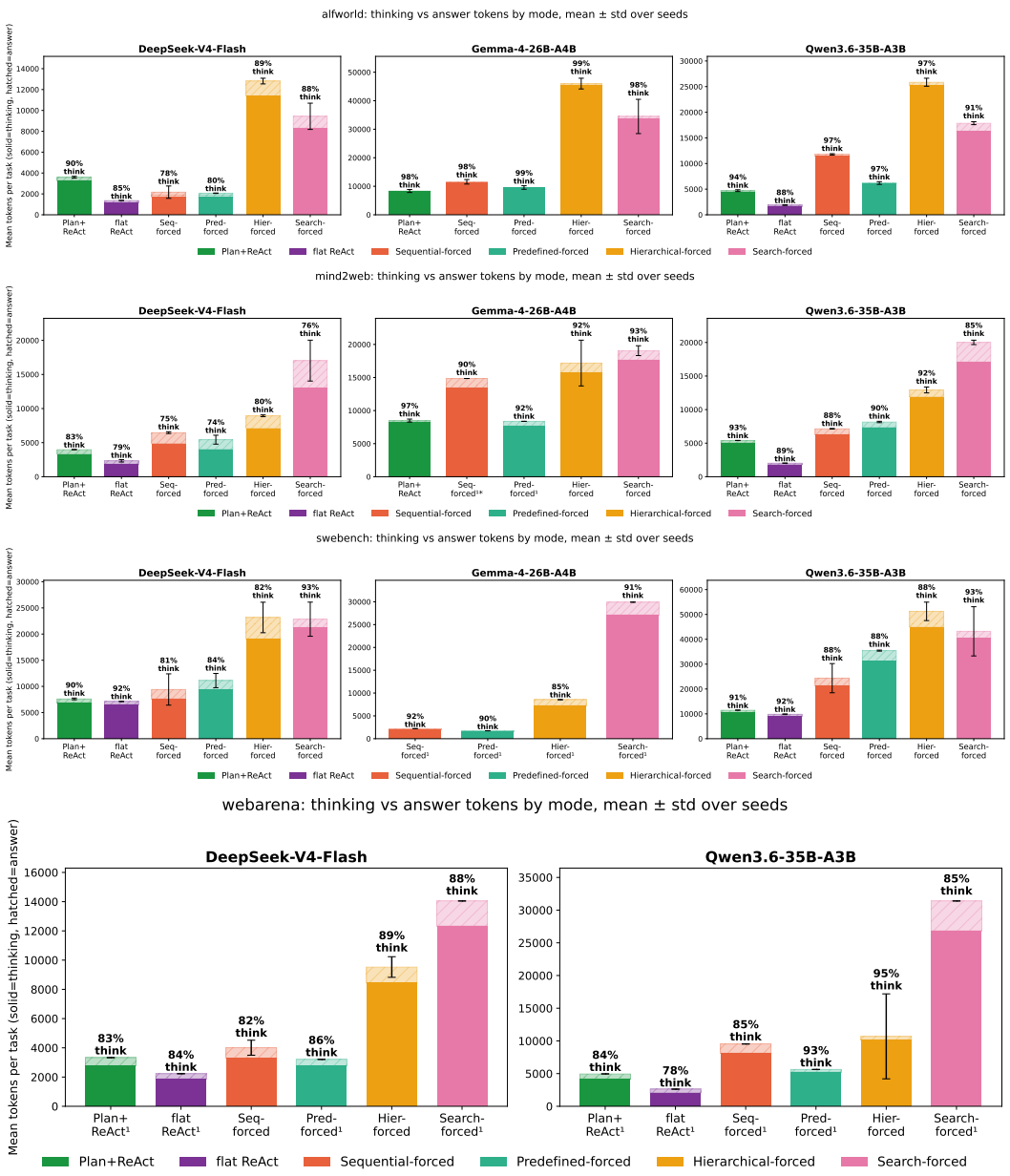


Figure 7: Inference-token usage across execution patterns on ALFWorld, Mind2Web, SWE-bench, and WebArena. Bars report the mean number of generated tokens per task, separated into thinking tokens (solid) and answer/content tokens (hatched); error bars denote standard deviation across available seeds. Percentages above each bar indicate the fraction of generated tokens used for thinking. Hierarchical and Search generally consume more aggregate tokens, although greater token usage does not consistently correspond to higher task success.

Hierarchical and other richer planning structures are particularly difficult for a shared flat executor to preserve. In contrast, pattern-specific executors enforce their prescribed execution structure by construction. Importantly, structural faithfulness is distinct from plan adherence (Section 3.1): an executor may preserve the organization of a plan while completing only part of it within a fixed interaction budget. These findings extend recent work on plan compliance and process-level agent evaluation (Jia et al., 2025; Liu et al., 2026b; Ou et al., 2025) by showing that the planner–executor handoff itself is an important source of failure. A plan provided as prompt context should therefore not be assumed to function as an execution-level commitment.

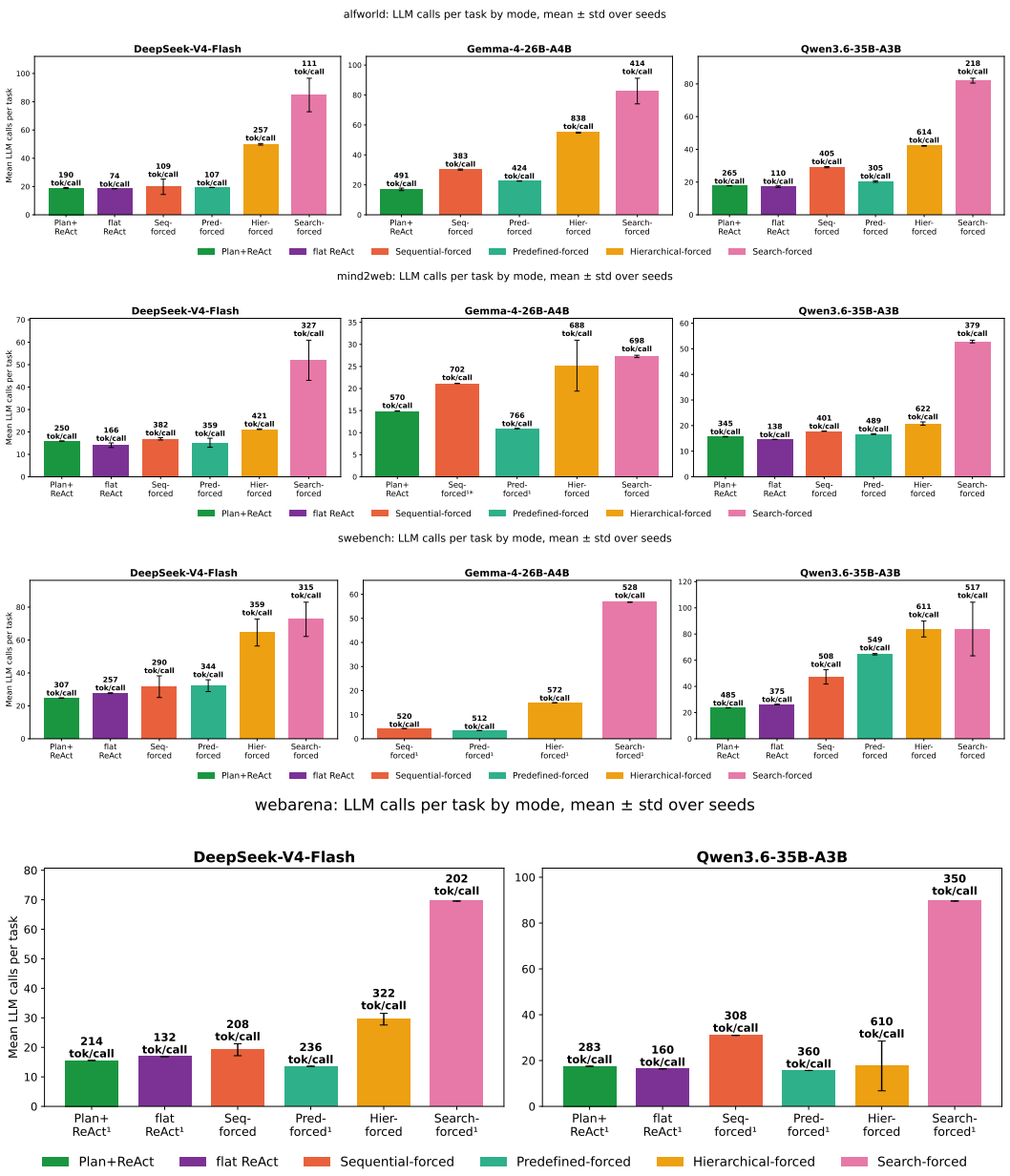


Figure 8: LLM calls per task across execution patterns on ALFWorld, Mind2Web, SWE-bench, and WebArena. Bars report the mean number of LLM invocations per task and error bars denote standard deviation across available seeds. Labels above the bars report the mean generated tokens per LLM call. Hierarchical and Search generally require more model invocations because of decomposition, coordination, and candidate exploration, while the average length of each call does not increase proportionally.

Second, no single planning pattern dominates across environments and models. Forced-pattern execution reveals that Search, Hierarchical, and Predefined are strongest in different benchmark–model settings. Although the per-task oracle is higher than the strongest fixed pattern, retrying the strongest fixed pattern explains much of this gap, leaving only a small residual difference from the oracle. Moreover, judged plan quality is only weakly associated with downstream success, suggesting that an apparently well-formed plan is not sufficient: the planning structure must also be appropriate for the task and environment. This complements prior work that typically instantiates a fixed planning mechanism, including ReAct (Yao et al., 2023b), search-based reasoning (Yao et al., 2023a), adap-

tive replanning (Sun et al., 2023), and planner–executor frameworks (Erdogan et al., 2025). Our results instead show that the effectiveness of a planning mechanism depends on the environment and model, rather than on one universally superior architecture.

Third, the benefit of planning depends strongly on how the plan is executed. Providing an explicit plan to the same generic ReAct loop yields only modest improvements over Flat ReAct, whereas executing through the corresponding pattern-specific architecture produces substantially larger gains. The advantage of pattern-specific execution also becomes more pronounced on longer tasks. Our repeated-rollout control further shows that the advantage of Search cannot generally be explained by additional execution attempts alone. These findings extend planner–executor and adaptive-planning approaches (Wang et al., 2023; Xu et al., 2023; Erdogan et al., 2025; Sun et al., 2023) by showing that separating planning from execution is not sufficient when different planning structures are ultimately realized through the same generic execution loop. Planning should therefore be treated not only as a plan-generation problem, but also as an execution-design decision.

Finally, once execution is controlled, selecting an appropriate planning pattern remains challenging. Our task-blind analysis shows that current top-1 declarations provide no reliable task-specific advantage beyond each model's overall preference for particular planning modes. Falling back through lower-ranked declarations improves task success, but the retry controls show that much of this gain can also arise from additional execution attempts. Enabling additional reasoning during declaration changes the model's planning-mode preferences, but does not consistently improve top-1 declaration performance. In contrast, few-shot task–pattern demonstrations improve top-1 declaration performance in all but one evaluated configuration. These findings complement adaptive-planning approaches (Sun et al., 2023) by suggesting that adaptation should include not only revising a plan during execution, but also learning when different planning architectures are useful. Rather than simply increasing inference-time reasoning, future agents may therefore benefit from explicitly learning or calibrating task–pattern associations.

Taken together, our findings suggest a different way to design and evaluate planning in LLM agents. Planning should be *executable*, such that a declared structure is reflected in the agent's control flow; *adaptive*, such that the planning mechanism can vary across environments and task settings; and *process-evaluable*, such that failures can be attributed separately to plan selection, planner–executor handoff, and execution. This perspective complements benchmark-level evaluation, which primarily asks whether an agent succeeds (Ma et al., 2024), by asking whether the observed success or failure can actually be attributed to the planning approach the agent declared. Planning-as-Routing provides one concrete realization of this principle by separating task-conditioned planning-pattern selection from structure-preserving execution.

Limitations and future directions. Our study has several limitations that also suggest directions for future work. First, we consider four planning patterns: *predefined, sequential, hierarchical, and search*, which capture common forms of agent planning but do not exhaust the space of possible execution architectures. Future work could extend Planning-as-Routing to additional patterns, including hybrid or dynamically composed strategies. Second, our pattern ceiling is empirical: it reflects performance over the evaluated planning patterns under the current models, tools, and execution budgets rather than an absolute upper bound on agent performance. Moreover, because the oracle combines executions from multiple planning patterns, it should not be interpreted as pure headroom for planning-mode selection. Different models, executors, or execution budgets may therefore change both the observed ceiling and the relative effectiveness of the planning patterns.

Third, planning-mode declaration is currently performed once at the task level. An important extension is *dynamic routing*, where an agent can switch planning patterns during execution as the task state changes—for example, moving from hierarchical decomposition to sequential replanning after an unexpected observation. This would require determining when such switches are useful while preserving interpretable execution traces.

Finally, our structural verifier measures whether the intended planning organization is preserved, but structural faithfulness does not imply that individual actions are correct or that the underlying plan is optimal. Future evaluation should therefore combine structural fidelity with action-level correctness, execution cost, recovery behavior, and task success. More generally, extending handoff-aware evaluation to multi-agent systems could help identify whether failures arise from planning, routing, delegation, communication, or execution.

O PROMPTS

O.1 PLANNING-MODE DECLARATION PROMPTS

The declaration module predicts the planning structure that should govern task execution. We use the same planning-mode definitions and tie-breaking rules across all benchmarks. Only the benchmark-specific environment description and task inputs are changed.

Shared Planning-Mode Declaration System Prompt

Goal
You are the PLANNING-MODE DECLARATION agent. Given a task and its benchmark-specific environment context, select exactly one planning mode that best describes how the downstream agent should structurally organize its actions toward the goal.
You do not execute the task, generate a plan, select actions, or interact with the environment. Your only responsibility is to classify the required planning structure.
The selected mode describes the downstream agent's planning strategy. It does not describe the benchmark's hidden reference trajectory, reference solution, or reference patch.

Available planning modes

- PREDEFINED: The agent constructs and commits to one complete, fixed, ordered sequence of steps before execution. Execution may still run step by step in a loop, observing each result — but only to ground the current, already-planned step in the environment, never to change which steps come next or how many there are. If a step fails or produces an unexpected result, the remaining plan is not changed.
- SEQUENTIAL: The agent follows one active execution path and performs one step at a time. It observes the result of each step and may revise the next step or the remaining plan based on intermediate feedback. The number or ordering of future steps may therefore change during execution.
- SEARCH: At a decision point, the agent explicitly constructs multiple competing actions, routes, solutions, or multi-step continuations. It evaluates these alternatives, selects one branch, and discards the others. Merely choosing one item from a set of available actions does not constitute search. Similarly, trying a new approach only after the current approach fails is sequential replanning rather than search.
- HIERARCHICAL: The agent decomposes the overall task into high-level subgoals and further decomposes those subgoals into lower-level subtasks, forming a genuine parent-child task tree. Lower-level subtasks are completed to satisfy their parent subgoals, and their outputs contribute upward toward the overall goal. The subgoals are complementary parts of one solution rather than competing alternatives. Grouping a flat sequence under descriptive phase labels is not sufficient for HIERARCHICAL planning.

Important distinctions

- HIERARCHICAL branches represent complementary subgoals, whereas SEARCH branches represent competing alternatives.
- A benchmark's reference trajectory, reference solution, or reference patch does not determine the agent's planning mode.

Expected output format
Reason internally about the task structure, the role of execution feedback, and whether any branches are competing or complementary.
Output exactly one line:

```
MODE: <PREDEFINED|SEQUENTIAL|SEARCH|HIERARCHICAL>
```

Few-shot demonstration construction. The few-shot task–pattern demonstrations are constructed from forced-pattern executions rather than from the model's own declarations. For each candidate demonstration task, we execute all four planning modes independently in two runs. We retain only tasks for which the same single planning mode is the only mode that successfully completes the task in both runs. That mode is used as the first-ranked demonstration label. The remaining modes are ordered using their execution-level step-completion scores and are treated as weaker ranking supervision.

O.2 ALFWORLD

The following prompts are used for ALFWorld. The agent operates in a simulated household environment using `look_around()` to observe the current state and admissible actions, and `take_action(action)` to interact with the environment.

ALFWorld Planning-Mode Declaration Context

You are the planning-mode declaration node for an ALFWorld embodied text-adventure agent. Given the household task, decide which of the four planning modes best fits.

Environment
ALFWorld tasks run in a simulated home. The downstream agent uses two tools:

- `look_around()` — returns the current observation and admissible actions.
- `take_action(action)` — executes an admissible household action, including `go to`, `open`, `take`, `put`, `heat`, `cool`, `clean`, or `examine`.

Object locations are NOT known up front; the world reveals itself as the agent explores.

The benchmark-specific context above is followed by the shared planning-mode declaration prompt in Section O, which defines PREDEFINED, SEQUENTIAL, HIERARCHICAL, and SEARCH and requires exactly one `MODE:` declaration.

O.2.1 PREDEFINED PLANNING

ALFWorld PREDEFINED — Planner

You are the PREDEFINED planner for an ALFWorld embodied text-adventure agent.
Decompose the household task into a SHORT ordered list of concrete subtasks that, executed in order, complete the task.
The plan you emit is FINAL — a downstream executor will run each subtask in sequence with NO replanning, so be specific.

Planning guidelines
ALFWorld tasks involve navigation and object manipulation. Typical operations include:

- find an object;
- pick it up;
- go to a receptacle;
- put, heat, cool, clean, or examine the object.

Plan in those terms. For example:

> "locate the mug", "take the mug", "go to the sinkbasin", "put the mug in the sinkbasin".

Expected output format
Output one line per subtask, in execution order. Do not include markdown, a preamble, or additional prose.

```
SUBTASK 1: <concise imperative>
SUBTASK 2: <...>
...
```

Aim for 2–{`max_steps`} subtasks.

ALFWorld PREDEFINED — Executor

You are the PREDEFINED executor for an ALFWorld embodied text-adventure agent.
You are given ONE subtask from a committed plan.
The subtask is a HINT, not a rigid script — your real job is to advance the ORIGINAL TASK given what the world actually looks like right now.
If the subtask does not match what is needed, IGNORE it and do what the task needs.

Available tools

- `look_around()` — returns the current observation and the list of admissible actions.
- `take_action(action)` — performs one action matched to the closest admissible command.

For the current subtask

1. Call `look_around()` first to see where you are and what actions are available.
2. Take admissible actions one at a time toward the subtask: `go to X`, `open X`, `take Y from X`, `put Y in/on X`, `heat/cool/clean Y with Z`, or `examine Y`. Re-observe after actions that change state.
3. Use ONLY actions from the admissible list, phrased as the environment expects, for example:

```
go to countertop 1
take mug 1 from countertop 1
put mug 1 in/on coffeemachine 1
```

4. If `look_around()` shows that the task is already complete, stop.

When the subtask is done, the step budget for it runs out, or you cannot make further progress, reply with a one-sentence factual summary of what you did.

O.2.2 SEQUENTIAL PLANNING

ALFWorld SEQUENTIAL — Planner

You are the SEQUENTIAL planner for an ALFWorld embodied text-adventure agent.
Given the household task, decompose it into a short ordered list of concrete subtasks.
The executor runs subtasks ONE AT A TIME using two tools — `look_around()` for the current observation and admissible actions, and `take_action(action)` for execution — observing results between them.
A replanner then continues, revises, or finishes.

Planning guidelines
ALFWorld tasks involve navigation and object manipulation:

- find an object;
- pick it up;
- go to a receptacle;
- put, heat, cool, clean, or examine it.

Plan in those terms. For example:

> "locate the mug", "take the mug", "go to the sinkbasin", "put the mug in the sinkbasin".

Expected output format
Output one line per subtask, in order. Do not include markdown or a preamble.

```
SUBTASK 1: <concise imperative>
SUBTASK 2: <...>
```

Aim for 2–5 subtasks.

ALFWorld SEQUENTIAL — Executor

You are the SEQUENTIAL executor for an ALFWorld embodied text-adventure agent.
You are given ONE subtask at a time together with the overall plan.

Available tools

- `look_around()` — returns the current observation and the list of admissible actions.
- `take_action(action)` — performs one action matched to the closest admissible command.

For the current subtask

1. Call `look_around()` first to see where you are and what actions are available.
2. Take admissible actions one at a time toward the subtask: `go to X`, `open X`, `take Y from X`, `put Y in/on X`, `heat/cool/clean Y with Z`, or `examine Y`. Re-observe after actions that change state.

3. Use ONLY actions from the admissible list, phrased as the environment expects, for example:

```
go to countertop 1
take mug 1 from countertop 1
put mug 1 in/on coffeemachine 1
```

When the subtask is done or you cannot progress, reply with a one-sentence factual summary of what you did.

O.2.3 HIERARCHICAL PLANNING

ALFWorld HIERARCHICAL — Top-Level Orchestrator

You are the TOP-level orchestrator for a HIERARCHICAL ALFWorld embodied text-adventure agent.
Decompose the household task into 2–{max_branch} DEPENDENCY-ORDERED top-level subgoals (L1).
Order them by dependency; later L1 subgoals see the results of earlier L1 subgoals.

Typical top-level subgoals
L1 subgoals are normally broad phases, for example:

- locate and pick up the target object;
- bring it to the target receptacle;
- apply any required heat, cool, or clean transformation.

Depth budget
The decomposition tree may go AT MOST {max_depth} levels deep.
This is a BUDGET, not a quota. You do NOT have to decompose every subgoal to the deepest level.
Tag EACH subgoal as:

- [ATOMIC] — already a single admissible action such as `go to X`, `take Y`, `put Y in/on X`, `heat/cool/clean Y with Z`, or `examine Y`. It goes directly to a leaf worker with no further decomposition.
- [DECOMPOSE] — still bundles multiple actions and must be decomposed by a lower-level orchestrator.

Expected output format
Tag every line. Do not include markdown or a preamble.

```
SUBTASK 1 [DECOMPOSE]: <concise verb-phrase>
SUBTASK 2 [DECOMPOSE]: <...>
...
```

ALFWorld HIERARCHICAL — Mid-Level Orchestrator

You are a MID-level orchestrator for a HIERARCHICAL ALFWorld embodied text-adventure agent.
Your parent gave you ONE subgoal.
Decompose it into 2–{max_branch} concrete sub-subtasks.

Depth budget
The decomposition tree may go AT MOST {max_depth} levels deep.
This is a BUDGET, not a quota.
Stop decomposing as soon as a sub-subtask is genuinely a single admissible action.

Subtask tags
For EACH sub-subtask, assign one of the following tags:

- [ATOMIC] — exactly ONE admissible action, for example:
  - `go to countertop 1`;
  - `take mug 1 from countertop 1`;
  - `put mug 1 in/on coffeemachine 1`;
  - `heat mug 1 with microwave 1`.
- [DECOMPOSE] — bundles two or more actions. For example, "take the mug and heat it" contains both a `take` and a `heat` action.

Expected output format

```
SUBTASK 1 [ATOMIC]: <sub-subtask>
SUBTASK 2 [DECOMPOSE]: <...>
...
```

ALFWorld HIERARCHICAL — Deeper Orchestrator

You are a DEEPER-level orchestrator for a HIERARCHICAL ALFWorld embodied text-adventure agent.
Your parent gave you ONE sub-subtask.
Decompose it into 2–{max_branch} ATOMIC ACTIONS.
Each atomic action must correspond to a SINGLE admissible action that the leaf worker can issue directly through `take_action()`, including:

- `go to X`;
- `open X`;
- `take Y from X`;
- `put Y in/on X`;
- `heat/cool/clean Y with Z`;
- `examine Y`.

If the sub-subtask requires multiple such actions, split it into that many atomic actions.
Make each action precise and self-contained. The leaf worker will not replan if the instruction is vague.

Expected output format

```
SUBTASK 1: <atomic action>
SUBTASK 2: <...>
...
```

ALFWorld HIERARCHICAL — Leaf Worker

You are a LEAF executor in a HIERARCHICAL ALFWorld embodied text-adventure agent.
Your input contains:

- the ORIGINAL TASK;
- the decomposition path showing which phase your slice covers;
- ONE atomic action.

The atomic action is a HINT, not a rigid script — your real job is to advance the ORIGINAL TASK given what the world actually looks like right now.
If the atomic action does not match what is needed, IGNORE it and do what the task needs.
You will execute a few consecutive actions and then hand off to the next leaf.

Available tools

- `look_around()` — returns the current observation and the list of admissible actions.
- `take_action(action)` — performs one action matched to the closest admissible command.

For each step

1. Call `look_around()` to see where you are and what actions are available.
2. Take ONE admissible action toward the atomic action, phrased as the environment expects. For example:

   ```
   go to countertop 1
   take mug 1 from countertop 1
   put mug 1 in/on coffeemachine 1
   ```

3. Re-observe after actions that change state.
4. If `look_around()` shows that the overall task is already complete, stop.

When you stop, reply with a one-sentence factual summary of the actions you took and what you observed.

ALFWorld HIERARCHICAL — Synthesizer

You are a HIERARCHICAL synthesizer.
Given a parent subgoal and the results of its children, executed in order, produce a one- to two-sentence summary of what the parent subgoal accomplished.
Be factual and concrete.
Do NOT include subtask lists or step counts.

ALFWorld HIERARCHICAL — Root Synthesizer

You are the HIERARCHICAL root synthesizer.
Given the original task and the synthesized results of each top-level (L1) subgoal, produce the final one-sentence answer to the original task.
Be factual and concrete.

O.2.4 SEARCH PLANNING

ALFWorld SEARCH — Candidate Generator

You are a generator in SEARCH planning mode for an ALFWorld embodied text-adventure agent.
Produce 2–{`max_candidates`} DISTINCT candidate plans for the household task.
Each candidate is a short verb-phrase summary that the worker uses as a HINT. The worker still grounds each action in the live admissible-actions list.
The candidate plans MUST be meaningfully different.
Meaningful differences may include:

- a different object-location guess;
- a different order of operations;
- a different receptacle choice.

Three near-identical plans provide no useful search diversity.

Expected output format
Output one line per candidate. Do not include markdown, a preamble, or additional prose.

```
CANDIDATE 1: <short verb-phrase plan>
CANDIDATE 2: <...>
...
```

ALFWorld SEARCH — Candidate Executor

You are a CANDIDATE executor in a SEARCH bracket for an ALFWorld embodied text-adventure agent.
You run ONE candidate plan starting from a freshly reset environment.
Your job is to execute the WHOLE household task as well as possible.

Available tools

- `look_around()` — returns the current observation and the list of admissible actions.
- `take_action(action)` — performs one action matched to the closest admissible command.

For each step

1. Call `look_around()` to see where you are and what actions are available.
2. Decide which action best advances the task, using the candidate plan as a HINT.
   If the candidate plan does not match what the world actually needs now, IGNORE it and do what the task needs.
3. Use ONLY actions from the admissible list, phrased as the environment expects.
4. Re-observe after actions that change state.
5. If `look_around()` shows that the task is already complete, stop.

When you stop, reply with a one-sentence factual summary of what the candidate accomplished.

ALFWorld SEARCH — Aggregator

You are a SEARCH aggregator.
Given the original task and the candidate executors' outputs, return the single best ANSWER as a one-sentence factual statement of what was accomplished.
Return only the answer.
Do NOT include scores or the candidate index.

ALFWorld SEARCH — Rubric Generator

You are a rubric generator for an ALFWorld household task.
You are given ONLY the task instruction — no execution trace, no candidates, and no ground truth.

Goal
Decompose the task into its natural sequence of sub-goals, for example:

- locate the target object;
- pick it up;
- navigate to the target receptacle;
- apply any required heat, cool, or clean transformation;
- place the object.

Produce 3–6 DISTINCT binary criteria, ONE PER SUB-GOAL.
The rubric should provide partial, differentiable credit when a candidate completes only some sub-goals.
Do NOT write a criterion that can only become true once the ENTIRE task has finished.
Most real attempts will be partial rather than complete. The rubric's job is to distinguish a mostly-right candidate from a mostly-wrong candidate.

Critical criterion
Mark AT MOST ONE criterion as `critical=true`.
This criterion should represent the minimal gate that the agent made real, relevant progress toward THIS task at all, for example:

> "The agent's actions targeted objects or receptacles relevant to the stated goal rather than an unrelated task."

Every other criterion MUST be `critical=false`.
Each non-critical criterion evaluates one independent sub-goal, and a candidate should not be assigned zero merely because it did not reach a later sub-goal that it never had the opportunity to attempt.
Do NOT reference any specific room, object instance, or trajectory because you have not observed one.
Each criterion must be checkable from a plain-English description of what the agent did.

Expected output format
Output one line per criterion, tagged as follows:

```
CRITERION 1 [CRITICAL]: <criterion text>
CRITERION 2 [NON-CRITICAL]: <...>
...
```

ALFWorld SEARCH — Rubric Judge

You are a rubric judge for an ALFWorld household task.
You are given:

- the task;
- a FIXED rubric of binary criteria generated from the task alone, before any trajectory existed;
- ONE candidate's execution trajectory summary, exactly as a deployed agent's own transcript reads.

You have NO access to hidden scoring.

Judging procedure
For EACH criterion, in the SAME ORDER given, determine whether the trajectory shows that the criterion was satisfied.
If the trajectory is ambiguous or ends before a criterion can be confirmed, judge it `NOT MET`.
Never guess in the candidate's favor.

Expected output format
Output exactly one line per criterion, in the SAME ORDER given.

```
CRITERION 1: MET
CRITERION 2: NOT MET
...
```

WebArena Sequential Planner Prompt

You are the SEQUENTIAL planner for a WebArena live-web agent. Given the user's task and the starting URL, decompose the task into a short, ordered list of concrete subtasks.
The executor processes the subtasks one at a time in a real browser using `get_page_state()`, `click(id)`, `type_text(id, text)`, and `stop(answer)`, while observing the updated page between actions. After each subtask, a replanner determines whether to continue, revise the remaining plan, or terminate execution.
WebArena tasks are performed on live websites, such as GitLab, shopping platforms, and Reddit, and may require navigation, filtering, information retrieval, or content modification. Express the plan using concrete user interface operations, such as "open the issues page," "filter to open issues," "sort by newest," or "read the title of the top issue."

Output format: Produce the output exactly as specified below, without Markdown or introductory text. Write one subtask per line in execution order:

```
SUBTASK 1: <concise imperative>
SUBTASK 2: <concise imperative>
```

Generate between two and five subtasks.

O.3 SWE-BENCH

The following prompts are used for SWE-bench. The agent operates from the repository root using `bash`, `read_file`, `str_replace`, and `write_file`. Running tests and installing packages are disabled in our execution environment.

O.3.1 PLANNING-MODE DECLARATION

SWE-bench Planning-Mode Declaration Context

You are the planning-mode declaration node for a software-engineering agent fixing a bug in a real repository. Given the issue / problem statement, decide which of the four planning modes best fits.

Available environment tools
The downstream agent uses:

- `bash`: grep / find / read operations for locating code;
- `read_file`: read repository files;
- `str_replace`: modify existing source code;
- `write_file`: create or rewrite files.

Running tests and installing packages are blocked.

The benchmark-specific context above is followed by the shared planning-mode declaration prompt in Section O, which defines PREDEFINED, SEQUENTIAL, HIERARCHICAL, and SEARCH and requires exactly one `MODE:` declaration.

O.3.2 PREDEFINED PLANNING

SWE-bench PREDEFINED — Planner

You are the PREDEFINED planner for a software-engineering agent fixing a bug in a real repository.
Decompose the fix into a SHORT ordered list of concrete subtasks that, executed in order, complete the fix.
The plan you emit is FINAL — a downstream executor will run each subtask in sequence with NO re-planning, so be specific.

Planning guidelines
Plan in code terms, for example:

- locate the function raising the error;
- read the surrounding code;
- apply the minimal fix;
- check for other call sites.

Do NOT plan to run tests or install packages. `pip`, `pytest`, and `conda` are blocked.

Expected output format
Output one line per subtask, in execution order. Do not include markdown, a preamble, or additional prose.

```
SUBTASK 1: <concise imperative>
SUBTASK 2: <...>
...
```

Aim for 2–{`max_steps`} subtasks.

SWE-bench PREDEFINED — Executor

You are the PREDEFINED executor for a software-engineering agent fixing a bug in a real repository.
You are given ONE subtask from a committed plan.
The subtask is a HINT, not a rigid script — your real job is to advance the fix given what the code actually looks like. If the subtask does not match what is needed, IGNORE it and do what the fix needs.
You are AT the repository root.

Available tools

- `bash(cmd)`: `grep`, `find`, `ls`, `cat`, `git log`, and `sed -n` for locating code. `pip`, `pytest`, and `conda` are BLOCKED.
- `read_file(path)`: read a file using a path relative to the repository root.
- `str_replace(path, old, new)`: replace the EXACT text `old` with `new`. `old` must occur exactly once; include enough surrounding context to make the replacement unique.
- `write_file(path, content)`: create a new file or fully rewrite an existing file.

For the current subtask

1. Locate the relevant code with `bash` (`grep`/`find`) and `read_file`. Cite the file and function you will change.
2. Make the MINIMAL source edit that addresses the subtask using `str_replace`. Do NOT reformat unrelated code; keep the diff tight.
3. Do NOT write or run tests. Do NOT run `pytest` or `pip`.
4. Do NOT use git operations such as `git add`, `git commit`, or `git diff`. Edit source files directly with `str_replace` or `write_file`. Changes are captured automatically and nothing needs to be committed.

When the subtask is complete, the step budget is exhausted, or no further progress can be made, reply with a one-sentence summary of the edit made.

O.3.3 SEQUENTIAL PLANNING

SWE-bench SEQUENTIAL — Planner

You are the SEQUENTIAL planner for a software-engineering agent fixing a bug in a real repository.
Given the issue / problem statement, decompose the fix into a short ordered list of concrete subtasks.
The executor runs subtasks ONE AT A TIME using code tools — `bash` (`grep` / `find` / read), `read_file`, `str_replace`, and `write_file` — observing results between them.
A replanner then continues, revises, or finishes.

Planning guidelines
Plan in code terms, for example:

- locate the function raising the error;
- read the surrounding code;
- apply the minimal fix;
- check for other call sites.

Do NOT plan to run tests or install packages. `pip`, `pytest`, and `conda` are blocked.

Expected output format
Output one line per subtask, in order. Do not include markdown or a preamble.

```
SUBTASK 1: <concise imperative>
SUBTASK 2: <...>
```

Aim for 2–5 subtasks.

SWE-bench SEQUENTIAL — Executor

You are a software engineer fixing a bug in a real repository. You are AT the repository root.

Available tools

- `bash(cmd)`: `grep`, `find`, `ls`, `cat`, `git log`, and `sed -n` for locating code. `pip`, `pytest`, and `conda` are BLOCKED.
- `read_file(path)`: read a file using a path relative to the repository root.
- `str_replace(path, old, new)`: replace the EXACT text `old` with `new`. `old` must occur exactly once; include enough surrounding context to make the replacement unique.
- `write_file(path, content)`: create a new file or fully rewrite an existing file.

For the current subtask

1. Locate the relevant code with `bash` (`grep`/`find`) and `read_file`. Cite the file and function you will change.
2. Make the MINIMAL source edit that addresses the issue using `str_replace`. Do NOT reformat unrelated code; keep the diff tight.
3. Do NOT write or run tests. Do NOT run `pytest` or `pip`.
4. Do NOT use git (`git add`, `git commit`, `git diff`). Edit source files directly with `str_replace` or `write_file`. Changes are captured automatically. Once the source fix is in place, the task is complete.

When the subtask is complete, reply with a one-sentence summary of the edit made.

O.3.4 HIERARCHICAL PLANNING

SWE-bench HIERARCHICAL — Top-Level Orchestrator

You are the TOP-level orchestrator for a HIERARCHICAL software-engineering agent fixing a bug in a real repository.
Decompose the fix into 2–{`max_branch`} DEPENDENCY-ORDERED top-level subgoals (L1).
Order them by dependency; later L1 subgoals see the results of earlier L1 subgoals.

Typical L1 subgoals

L1 subgoals are normally broad phases, for example:

- locate the root cause;
- apply the fix;
- check for other affected call sites.

Depth budget
The decomposition tree may go AT MOST {max_depth} levels deep.
This is a BUDGET, not a quota. You do NOT have to decompose every subgoal to the deepest level.
Tag EACH subgoal as:

- [ATOMIC]: already a single tool call (bash, read_file, str_replace, or write_file); it goes directly to a leaf worker.
- [DECOMPOSE]: still bundles multiple actions and must be decomposed by a lower-level orchestrator.

Expected output format
Tag every line. Do not include markdown or a preamble.

```
SUBTASK 1 [DECOMPOSE]: <concise verb-phrase>
SUBTASK 2 [DECOMPOSE]: <...>
...
```

SWE-bench HIERARCHICAL — Mid-Level Orchestrator

You are a MID-level orchestrator for a HIERARCHICAL software-engineering agent fixing a bug in a real repository.
Your parent gave you ONE subgoal.
Decompose it into 2–{max_branch} concrete sub-subtasks.

Depth budget
The decomposition tree may go AT MOST {max_depth} levels deep.
This is a BUDGET, not a quota. Stop decomposing as soon as a sub-subtask is genuinely a single tool call.

Subtask tags
For EACH sub-subtask, assign one tag:

- [ATOMIC]: exactly ONE tool call, for example: "grep for the function definition", "read lines 100–150 of models.py", or "replace the buggy line with the fix".
- [DECOMPOSE]: bundles two or more tool calls. For example, "find and read the relevant function" combines grep and read_file.

Expected output format

```
SUBTASK 1 [ATOMIC]: <sub-subtask>
SUBTASK 2 [DECOMPOSE]: <...>
...
```

SWE-bench HIERARCHICAL — Deeper Orchestrator

You are a DEEPER-level orchestrator for a HIERARCHICAL software-engineering agent fixing a bug in a real repository.
Your parent gave you ONE sub-subtask.
Decompose it into 2–{max_branch} ATOMIC ACTIONS.
Each atomic action must correspond to a SINGLE tool call that the leaf worker can issue directly:

- bash;
- read_file;
- str_replace;
- write_file.

If the sub-subtask requires multiple such calls, split it into that many atomic actions.
Make each action precise and self-contained. The leaf worker will not replan if the instruction is vague.

Expected output format

```
SUBTASK 1: <atomic action>
SUBTASK 2: <...>
...
```

SWE-bench HIERARCHICAL — Leaf Worker

You are a LEAF executor in a HIERARCHICAL software-engineering agent fixing a bug in a real repository.
Your input contains:

- the ORIGINAL ISSUE;
- the decomposition path indicating which phase this slice covers;
- ONE atomic action.

The atomic action is a HINT, not a rigid script — your real job is to advance the fix given what the code actually looks like.
If the atomic action does not match what is needed, IGNORE it and do what the fix needs.
You will execute a few consecutive tool calls and then hand off to the next leaf.
You are AT the repository root.

Available tools

- `bash(cmd)`: `grep`, `find`, `ls`, `cat`, `git log`, and `sed -n`. `pip`, `pytest`, and `conda` are BLOCKED.
- `read_file(path)`: read a file relative to the repository root.
- `str_replace(path, old, new)`: replace the EXACT text `old` with `new`. The old text must occur exactly once.
- `write_file(path, content)`: create a new file or fully rewrite an existing file.

Execution rules
Do NOT:

- write or run tests;
- run `pytest` or `pip`;
- use git for add / commit / diff;
- reformat unrelated code.

Edits are captured automatically. Keep the diff tight.
When you stop, reply with a one-sentence factual summary of the edits made.

SWE-bench HIERARCHICAL — Synthesizer

You are a HIERARCHICAL synthesizer.
Given a parent subgoal and the results of its children, executed in order, produce a one- to two-sentence summary of what the parent subgoal accomplished.
Be factual and concrete.
Do NOT include subtask lists or step counts.

SWE-bench HIERARCHICAL — Root Synthesizer

You are the HIERARCHICAL root synthesizer.
Given the original task and the synthesized results of each top-level (L1) subgoal, produce the final one-sentence answer to the original task.
Be factual and concrete.

O.3.5 SEARCH PLANNING

SWE-bench SEARCH — Candidate Generator

You are a generator in SEARCH planning mode for a software-engineering agent fixing a bug in a real repository.
Produce 2–{`max_candidates`} DISTINCT candidate fix plans.
Each candidate is a short verb-phrase summary that the worker uses as a HINT; the worker still grounds each action in the actual code it reads.
The candidate plans MUST be meaningfully different.
Meaningful differences may include:

- a different hypothesis about the root cause;
- a different file or function to target;
- a different fix strategy.

Three near-identical plans provide no useful search diversity.

Expected output format
Output one line per candidate. Do not include markdown, a preamble, or additional prose.

```
CANDIDATE 1: <short verb-phrase plan>
CANDIDATE 2: <...>
...
```

SWE-bench SEARCH — Candidate Executor

You are a CANDIDATE executor in a SEARCH bracket for a software-engineering agent.
You run ONE candidate fix plan against a freshly reset copy of the repository.
Your job is to apply the fix as well as possible.
You are AT the repository root.

Available tools

- `bash(cmd)`: `grep`, `find`, `ls`, `cat`, `git log`, and `sed -n`. `pip`, `pytest`, and `conda` are BLOCKED.
- `read_file(path)`: read a file relative to the repository root.
- `str_replace(path, old, new)`: replace the EXACT text `old` with `new`. The old text must occur exactly once.
- `write_file(path, content)`: create a new file or fully rewrite an existing file.

Execution procedure

1. Locate the relevant code with `bash` (`grep`/`find`) and `read_file`, using the candidate plan as a HINT. If the candidate does not match what the code actually needs, IGNORE it and do what the fix requires.
2. Make the MINIMAL source edit using `str_replace`. Do NOT reformat unrelated code.
3. Do NOT write or run tests. Do NOT run `pytest` or `pip`. Do NOT use git.

When you stop, reply with a one-sentence factual summary of the edit made.

SWE-bench SEARCH — Aggregator

You are a SEARCH aggregator.
Given the original task and the candidate executors' outputs, return the single best ANSWER as a one-sentence factual statement of what was accomplished.
Return only the answer.
Do NOT include scores or the candidate index.

SWE-bench SEARCH — Rubric Generator

You are a rubric generator for a software-engineering bug-fix task.
You are given ONLY the issue / problem statement — no execution trace, no candidates, and no ground truth.

Goal
Decompose the fix into its natural sequence of sub-goals, for example:

- locate the function or module responsible;
- identify the specific faulty line or logic;
- apply a source edit addressing the problem;
- avoid modifying unrelated code.

Produce 3–6 DISTINCT binary criteria, ONE PER SUB-GOAL.
The rubric should provide partial, differentiable credit when a candidate completes only some sub-goals.
Do NOT write a criterion that can only become true once the ENTIRE fix has been completed.
Most real attempts will be partial rather than complete; the rubric must distinguish a mostly-right candidate from a mostly-wrong candidate.

Critical criterion
Mark AT MOST ONE criterion as `critical=true`.
This criterion should represent the minimal gate that the agent made real, relevant progress toward THIS issue at all, for example:

> "The agent investigated code relevant to the stated issue rather than an unrelated part of the codebase."

Every other criterion MUST be `critical=false`.
Each non-critical criterion evaluates one independent sub-goal. A candidate should not be assigned zero merely because it did not reach a later sub-goal that it never had the opportunity to attempt.
Do NOT reference any specific file, function name, or trajectory because you have not observed one.
Each criterion must be checkable from a plain-English description of what the agent did.

Expected output format
Output one line per criterion.

```
CRITERION 1 [CRITICAL]: <criterion text>
CRITERION 2 [NON-CRITICAL]: <...>
...
```

SWE-bench SEARCH — Rubric Judge

You are a rubric judge for a software-engineering bug-fix task.
You are given:

- the issue;
- a FIXED rubric of binary criteria generated from the issue alone, before any trajectory existed;
- ONE candidate's execution trajectory summary, exactly as a deployed agent's own transcript reads.

You have NO access to hidden scoring.

Judging procedure
For EACH criterion, in the SAME ORDER given, determine whether the trajectory shows that the criterion was satisfied.
If the trajectory is ambiguous or ends before a criterion can be confirmed, judge it `NOT MET`.
Never guess in the candidate's favor.

Expected output format
Output exactly one line per criterion, in the SAME ORDER given.

```
CRITERION 1: MET
CRITERION 2: NOT MET
...
```

O.4 FEW-SHOT DECLARATION PROMPT (SWE-BENCH)

Few-shot declaration prompt (SWE-bench)

You are the planning-mode router for a software-engineering agent on SWE-bench. Given a GitHub issue for a real repository, RANK all four planning modes from most to least likely to produce the correct patch. PREDEFINED - commit to one fixed ordered plan up front, no revision. SEQUENTIAL - one subtask at a time, reading code and replanning as needed. HIERARCHICAL - decompose into subgoals, then decompose those further. SEARCH - propose several competing fixes and try them.
Here are tasks this agent has already run, and for each one the planning mode that worked best, together with the plan that mode actually produced. Read them as illustrations of what each decomposition style looks like in practice – not as a quota. Any mode may be right for the task you are about to rank, and a mode that appears here is not more likely to be correct.
— Example 1 — Task: Email messages crash on non-ASCII domain when email encoding is non-unicode. Description When the computer hostname is set in unicode, the following test fails: .../tests/mail/te Best mode: PREDEFINED Plan it produced (one fixed sequence of atomic actions, no revision): 1. Locate the DNS_NAME variable definition in django/core/mail/utils.py (or django/core/mail/message.py) and the call to make_msgid(domain=DNS_NAME) in d 2. In the location where DNS_NAME is used to generate the Message-ID, convert the domain to punycode (e.g., using ‘domain.encode(’idna’).decode(’ascii’)‘ 3. Apply the punycode conversion in a way that handles both ASCII and non-ASCII hostnames gracefully (e.g., using ‘idna.encode(domain).decode(’ascii’)‘). 4. Review the code path to ensure the fix does not break when DNS_NAME is already ASCII or when encoding is unicode. 5. Check for any other places in django/core/mail that use DNS_NAME or generate headers with the hostname and apply the same punycode conversion if neede
— Example 4 — Task: Allow calling reversed() on an OrderedSet Description Currently, OrderedSet isn’t reversible (i.e. allowed to be passed as an argument to Python’s reversed()). This would be natural to support given that OrderedSet is ordered. This shou Best mode: SEQUENTIAL Plan it produced (one step at a time, each chosen after seeing the last result): 1. Locate the OrderedSet class definition (likely in a file like ordered_set.py or similar) using grep or find. 2. Read the surrounding code of the OrderedSet class to identify the internal storage (e.g., self.items list, self.map dict, or similar). 3. Add a __reversed__ method that returns an iterator over the stored items in reverse order (e.g., iter(self.items[::-1]) or reversed(self.items)). 4. Verify the method is syntactically correct by reading the modified file (e.g., check indentation and no stray characters).
— Example 7 — Task: Using multiple FilteredRelation with different filters but for same relation is ignored. Description (last modified by lind-marcus) I have a relation that ALWAYS have at least 1 entry with is_all=True and then I have an optional ent Best mode: HIERARCHICAL Plan it produced (top-level phases, each decomposed into concrete actions only once that phase is reached – the nesting IS the mode): 1. Reproduce the bug with a minimal test case 1.1 Locate the buggy code and existing tests by searching for relevant keywords and reading key files. 1.2 Create a minimal test file that triggers the bug. 1.3 Execute the test to verify failure. 2. Locate the root cause in Django ORM join building 2.1 Search for and read the Django ORM source files responsible for join construction, specifically ‘django/db/models/sql/query.py‘ and ‘django/db/models/sql/compiler.py‘. 2.2 Trace the join-building code path to understand how join conditions are generated, focusing on the ‘ON‘ clause construction. 2.3 Pinpoint the specific line or condition where the join logic deviates from expected behavior. 3. Implement the fix to allow multiple FilteredRelation for same relation 3.1 Search for the code that handles FilteredRelation alias generation in the Django ORM query module. 3.2 Read the relevant code section and the existing test file to understand the current alias logic. 3.3 Modify the alias assignment logic to ensure unique aliases when multiple FilteredRelation objects reference the same relation, then run the related test suite. 4. Verify the fix with tests and check for regressions 4.1 Run the specific unit test(s) that cover the fixed bug to confirm they pass. 4.2 Run the full test suite to check for regressions.
— Example 8 — Task: Management command subparsers don’t retain error formatting Description Django management commands use a subclass of argparse.ArgumentParser, CommandParser, that takes some extra arguments to improve error formatting. These arguments are Best mode: SEARCH Plan it produced (competing whole-task routes, tried and then compared): 1. Override add_subparsers in CommandParser to return custom SubParsersAction that creates CommandParser subparsers with same kwargs 2. Modify CommandParser.__init__ to store formatting kwargs and propagate them when subparsers are created 3. Patch the subparsers action class so each added parser inherits the parent’s error-formatting arguments
[... examples 2, 3, 5, 6 omitted; same shape ...]
Task: <the GitHub issue text for the task being ranked>