Decompose and Depend: Hierarchical Dependency-Aware Planning for Tool-Augmented LLM Agents

Anonymous ACL submission

Abstract

Tool-augmented LLM agents are increasingly used for complex tasks that require multiple external APIs, long-horizon coordination, and adaptation to intermediate results. Existing planning methods, however, either proceed as flat sequential trajectories or construct global dependency graphs whose edges mainly encode mandatory precedence. Such representations struggle to express alternative routes, result-conditioned execution, and local recovery; moreover, they require expanding the entire task into low-level tool calls before any result is observed. We propose **HDAP**, a hierarchical dependency-aware planning framework inspired by Hierarchical Task Network planning. HDAP decomposes compound tasks into subtasks with natural-language preconditions, using one mechanism to represent parallelizable AND dependencies, alternative OR routes, and conditional execution. A dynamic hierarchical expansion policy keeps each planning layer focused on a task-relevant context and tool subset, while allowing later decisions to condition on real execution results. We evaluate HDAP on multiple benchmarks, where it consistently improves over sequential, search-based, and graph-based baselines, with the largest gains on the most complex multi-tool and long-context settings.

1 Introduction

Large language models (LLMs) are increasingly used as agents that solve tasks by interacting with external tools, functions, and web APIs (Schick et al., 2023; Yao et al., 2023b; Patil et al., 2023; Song et al., 2023; Qin et al., 2024). This line of work is useful because many realistic requests require live information or external actions rather than pure text generation. Recent benchmarks therefore move beyond isolated API invocation toward multi-step workflows, multi-turn function calling, and real-world constraint satisfaction (Li et al., 2023; Guo et al., 2024; Yan, 2025; Xie et al., 2024).

For such tasks, the main challenge is no longer merely selecting a tool, but making a *plan*. A plan must decompose a complex goal into intermediate subtasks and arrange them under ordering, parallel, and conditional constraints. For example, a travel assistant may search flights and hotels independently, compute total cost only after both results are available, and search an alternative route only if the cheapest flight is unavailable or too expensive. Without an explicit plan, an agent can only react step by step, making it easy to miss parallel opportunities, repeat completed work, or fail when observations change the feasible path.

This view highlights two requirements that current tool planners only partially satisfy. First, planning should support **dynamic hierarchical decomposition**. Existing global planners often expand a request into a fine-grained tool-level plan before execution, even though later steps may depend on unknown intermediate results. This is especially brittle in large tool environments, such as ToolBench with 16,464 real-world REST APIs across 49 categories (Qin et al., 2024). A more reliable agent should plan coarse stages first, execute the current stage, and refine later stages only when needed.

Second, planning should support **complete dependency modeling**. Sequential methods such as Chain-of-Thought and ReAct are adaptive but linear (Wei et al., 2022; Yao et al., 2023b). Search-based methods explore alternatives (Yao et al., 2023a; Hu et al., 2023; Qin et al., 2024), but treat them as trajectories rather than explicit runtime conditions. Graph-based planners introduce dependency graphs and parallel execution (Ma et al., 2025; Zhang et al., 2025; Wu et al., 2025; Wei et al., 2025), but flat DAG edges mainly express mandatory precedence. They do not naturally distinguish "execute B after A succeeds," "execute B only if

A fails," and "execute B only if A returns a value satisfying a constraint."

We propose **HDAP**, a **H**ierarchical **D**ependency-**A**ware **P**lanning framework for tool-augmented LLM agents. HDAP adapts Hierarchical Task Network planning (Nau et al., 2003; Höller et al., 2020) to open-domain tool use by replacing hand-authored formal Methods with LLM-generated Methods. Each Method decomposes a compound task into subtasks with a type, description, allowed tool subset, input/output requirements, and a natural-language precondition. These preconditions serve as a unified dependency interface: a subtask may run immediately, wait for multiple siblings, act as a fallback after failure, or trigger only when an observed result satisfies a condition.

HDAP performs **dynamic hierarchical execution**. At each layer, the agent evaluates subtask preconditions against the current state and executes all ready subtasks. Atomic subtasks call tools directly; compound subtasks open a new planning layer only when their preconditions are satisfied. This keeps each planning step local, lets downstream decisions use real intermediate results, and supports local recovery without discarding unrelated completed work.

We evaluate HDAP on StableToolBench, BFCL Multi-Turn, and BrowseComp-Plus, where it consistently outperforms strong baselines with gains that scale with task complexity. On the hardest cross-category setting, HDAP reaches 90.2% SoPR, outperforming DFSDT by 6.3 points. On BrowseComp-Plus, hierarchical decomposition reduces context length by 76% at equivalent step counts while retaining substantially higher accuracy under constrained context windows.

Our contributions are as follows:

- We propose a planning language with a unified precondition mechanism supporting AND, OR, and Conditional dependencies, enabling parallel execution and robust handling of unexpected runtime outcomes.
- We propose hierarchical expansion, which decomposes compound tasks only at the execution frontier, reducing context pollution and allowing later planning decisions to condition on actual intermediate results.
- We instantiate these ideas in HDAP and show that richer dependency modeling and hierarchical decomposition yield complementary gains: larger accuracy improvements on complex multi-tool tasks and substantial context efficiency in long-horizon settings.

2 Related Work

2.1 Tool-Augmented Agents and Benchmarks

Tool augmentation connects LLMs to external computation, retrieval, and action modules. Prior work studies modular tool routing (Karpas et al., 2022), self-supervised tool-use data construction (Schick et al., 2023), API call generation with documentation retrieval (Patil et al., 2023), reasoning-action loops (Yao et al., 2023b), REST API control (Song et al., 2023), multimodal model orchestration (Shen et al., 2023), and large-scale API learning (Qin et al., 2024). Evaluation has also shifted from isolated tool calls to multi-step agent tasks through API-Bank, StableToolBench, BFCL, and TravelPlanner (Li et al., 2023; Guo et al., 2024; Yan, 2025; Xie et al., 2024). These benchmarks motivate our focus on planning because agents must preserve state, coordinate tools, adapt to failures, and reason about dependencies among subtasks.

2.2 Sequential, Search-Based, and Graph Planning

Prompting-based planners make intermediate computation explicit. Chain-of-Thought elicits reasoning traces (Wei et al., 2022), Plan-and-Solve separates planning from solving (Wang et al., 2023), and ReAct interleaves reasoning, action, and observation (Yao et al., 2023b). These methods are simple and adaptive, but their control flow is fundamentally sequential: even when a plan is written before execution, it is usually a list of steps rather than a structured task network.

Search-based planning adds alternatives and recovery. Tree-of-Thought treats reasoning as tree search (Yao et al., 2023a), Tree-Planner merges sampled plans into an action tree (Hu et al., 2023), and ToolLLM's DFSDT backtracks over tool-call decision trees (Qin et al., 2024). Graph-based planners go further by making dependencies explicit: DTA-Llama, Plan-over-Graph, GAP, and ComplexTool-Plan generate or learn DAG-style structures for parallel tool execution (Ma et al., 2025; Zhang et al., 2025; Wu et al., 2025; Wei et al., 2025). These approaches improve over linear execution, but their alternatives are usually candidate trajectories and their graph edges mainly encode precedence. HDAP instead stores runtime

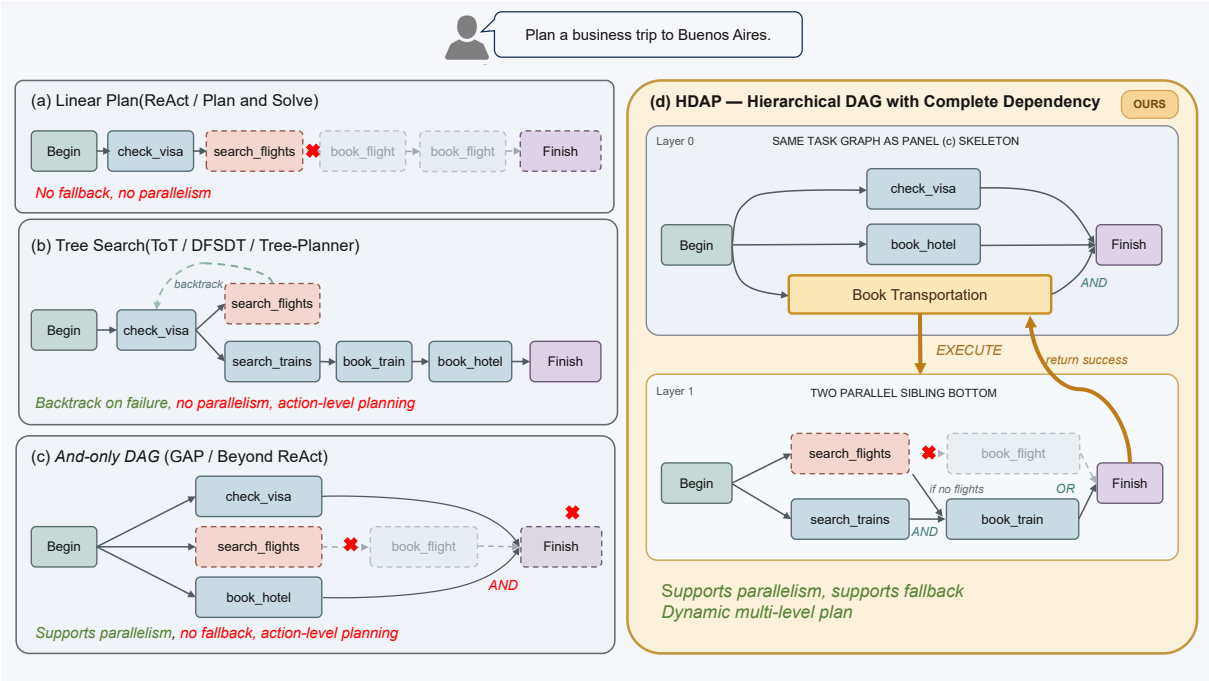

Figure 1: Four planning paradigms. (a) Chain: linear, no parallelism (ReAct). (b) Tree: multi-path search, sequential traversal (ToT, Tree-planner). (c) Flat DAG: fork/join parallelism but only AND dependencies (GAP, ComplexToolPlan). (d) HDAP: each layer is a dependency graph supporting AND, OR, and Conditional nodes; compound nodes expand recursively.

branch conditions as preconditions and expands compound subtasks only when they become ready.

Table 1 compares the methods reviewed above along the two axes that motivate HDAP: planning architecture and dependency expressiveness.

2.3 Hierarchical and Symbolic Planning

Symbolic planning offers a complementary perspective. Classical planners represent actions with formal preconditions and effects, while HTN planning decomposes compound tasks into primitive actions via Methods (Nau et al., 2003); HDDL extends PDDL-style formalisms to hierarchical planning domains (Höller et al., 2020). These ideas match complex tool use, where a user goal can be decomposed into intermediate objectives and only some leaves become concrete API calls. The challenge is domain engineering: classical HTN systems require hand-written Methods, predicates, and effects, while LLM+P-style systems still depend on reliable formalization (Liu et al., 2023). HDAP keeps hierarchical decomposition but treats natural language as the planning interface, generating Methods and evaluating preconditions on demand.

3 HDAP: Hierarchical Dependency-Aware Planning

HDAP draws from two established design patterns. From Hierarchical Task Network (HTN) planning (Nau et al., 2003; Höller et al., 2020), it inherits recursive decomposition through Methods, with preconditions controlling subtask eligibility. From functional programming, it adopts the declaration-then-call discipline: the PLAN step declares each subtask's interface—analogous to a typed function signature—while the EXECUTE step binds concrete values, analogous to a function call. A single LLM handles all roles (planning, reasoning, executing, returning) within one unified multi-turn conversation. Section 3.1 defines the planning language. Section 3.2 describes the per-layer execution algorithm. Section 3.3 discusses the resulting design properties.

3.1 Planning Language

HDAP organizes work into *subtasks*. A PLAN step produces a set of subtasks. Each subtask carries a name, a type (`atomic` for a single API call or `compound` for recursive decomposition), a natural-language description of the goal, a scoped subset of allowed tools, typed input/output requirements, and a precondition as shown in Table 2.

Method	Planning Architecture			Dependency Expression		
	Global Planning	Hierarchical	Re-Plan	AND	OR	Conditional
ReAct(Yao et al., 2023b)	✗	—	—	—	—	—
ToT(Yao et al., 2023a)	✗	—	—	—	—	—
Plan-and-Solve(Wang et al., 2023)	✓	fixed-2	✓	✗	✗	✗
Tree-Planner(Hu et al., 2023)	✓	✗	✓	✗		✗
DTA-Llama(Ma et al., 2025)	✗	fixed-2	weak	✓	✗	✗
GAP(Wu et al., 2025)	✓	✗	weak	✓	✗	✗
ComplexTool-Plan(Wei et al., 2025)	✓	✗	✗	✓	✗	✗
HDAP (ours)	✓	**adaptive**	✓	✓	✓	✓

Table 1: Comparison of planning methods. *Hierarchical*: hierarchical decomposition depth; *Recovery*: local failure recovery. "path-level" denotes OR modeled only over complete solution paths, not intermediate nodes. ReAct and ToT operate without a initial global planning; dependency columns are not applicable (—).

Plan–Execute separation. The `input requirements` and `output requirements` fields declare a subtask's interface at plan time—analogous to a function signature. At execution time, concrete values are bound: `actual_inputs` are extracted from the execution context according to the input requirements; `actual_outputs` capture real API responses or sub-layer results. For compound subtasks, its `name`, `description`, allowed tool subset, `actual_outputs`, and `output_requirements` are propagated to the child layer as its starting context, recursively enforcing the declaration-then-call contract.

Precondition as complete dependency interface. The `precondition` field is a single natural-language slot that provides a *complete dependency interface*: it covers the full spectrum from join conditions (all required predecessors must complete) to alternative routes (any sufficient predecessor suffices) to value-conditioned triggers (a predicate over upstream output must hold). To enable more robust dependency handling, we deliberately forgo structured predicate logic in favor of natural language for expressing preconditions, allowing the LLM to determine at runtime whether each subtask's precondition has been satisfied.

3.2 Dynamic Hierarchical Execution

HDAP uses a single LLM within one multi-turn conversation; all action types (REASON, PLAN, EXECUTE, RETURN) flow through the same message history, naturally compatible with standard chat training formats and prompt caching.

Per-layer execution. As shown in algorithm 1, each layer begins with a PLAN step that declares the full set of subtasks for that layer, then enters the execution loop (Algorithm 1). Within the loop, the LLM continuously evaluates which subtasks are ready given the current context, dispatches them with their bound actual inputs, and updates the context with their outputs—repeating until the layer's goal is achieved or a failure condition is triggered.

Precondition-driven dispatch. There is no separate precondition checker. At each round, the LLM sees the current execution context and determines which pending subtasks' preconditions are satisfied, outputting EXECUTE actions for ready subtasks. Multiple EXECUTE actions in one response trigger parallel dispatch.

Dynamic hierarchical decomposition. Compound subtasks are declared in the plan but expanded only at the execution frontier. The child layer's PLAN step occurs after upstream results are known, conditioning on concrete `actual_inputs` rather than predictions—addressing the limitation that global upfront planners admit only "one chance to generate a correct global plan upfront" (Wei et al., 2025). Each layer's prompt contains only the current task description and the pruned `allowed_tools` subset, avoiding the context pollution and full-inventory tool-selection burden of global planning.

Cross-layer context inherits. By default, a child layer inherits only the task briefing and the pruned tool subset—not the parent's full message history. The `actual_inputs` field carries compact information from parent to child (e.g., a document ID or a price value), but when a subtask genuinely depends on the parent's full accumulated context—pages of search results, opened documents, prior reasoning—the agent can set an `inherit_context` flag on the compound subtask, which passes the entire parent conversation history into the child layer. This avoids forcing the LLM to copy large context into `actual_inputs` by hand. The ablation (Table 4) confirms that this mechanism improves

Algorithm 1 Per-layer execution loop of HDAP.

```
function HDAP-EXECUTE(task, context, depth)
    plan ← PLAN(task, allowed_tools)
    while plan has pending subtasks and goal not achieved do
        ready ← {t ∈ plan.pending | satisfied(t.precondition, context)}
        if ready is empty then
            deadlock → rePlan() or return (failed, error)
        end if
        for all t ∈ ready in parallel do
            t.actual_inputs ← bind(t.input_requirements, context)
            if t.type = atomic then
                t.actual_outputs ← call_api(t)
            else                                                    ▷ compound
                t.actual_outputs ← HDAP-EXECUTE(t, child_context, depth + 1)
            end if
            context[t.name] ← t.actual_outputs
        end for
        if all ready succeeded and goal achieved then
            return (success, aggregate(context))
        else if any failed then
            choose among: try_alternative(plan) | rePlan() | return (failed, error)
        end if
    end while
end function
```

Phase	Field	Role
PLAN	`name`	Unique subtask identifier
	`type`	`atomic` (one API call) or `compound` (recursively decomposable)
	`description`	Natural-language description of what this subtask accomplishes
	`allowed_tools`	Tool subset scoped to this subtask (pruned downward across layers)
	`precondition`	Dependency specifier: AND, OR, or Conditional readiness
	`input_requirements`	Declares expected inputs: content, format
	`output_requirements`	Declares expected outputs: content, format
EXECUTE	`subtask_name`	References a planned subtask (must match `name`)
	`actual_inputs`	Concrete parameter values bound from the execution context
	`inherit_context`	If `true`, passes parent history into child layer (compound only; default `false`)
RETURN	`status`	`success` or `failed`
	`final_result`	Synthesized answer (success) or error analysis (failed)

Table 2: Subtask field definitions across the three action phases.

accuracy: the model selectively uses it for subtasks that require rich contextual grounding.

Local failure recovery. Failure handling is scoped to the current layer. When a subtask fails, the LLM can: (i) re-execute with corrected parameters; (ii) switch to an OR alternative still pending in the plan; (iii) re-plan the current layer; or (iv) return failure upward. A compound subtask returning failure is treated identically—the parent layer decides the next step without discarding completed sibling work.

3.3 Design Properties

Complete dependency modeling. The `precondition` field provides a complete dependency interface (Section 3.1), covering the full spectrum from AND dependencies (all required predecessors must complete) to OR alternatives (any sufficient predecessor suffices) to value-conditioned triggers (a predicate over upstream output must hold). The execution loop evaluates preconditions against the evolving context at each round (Section 3.2), making readiness a runtime judgment over real intermediate results rather than a static graph property. HDAP is consequently the first LLM planning framework to handle the full dependency spectrum within a single mechanism. This completeness has direct practical consequences: parallel execution emerges naturally from simultaneous precondition satisfaction, fallback routes are declared in the plan rather than discovered by exhaustive search, and conditional branches are triggered by actual observed values rather than predicted ones.

Dynamic hierarchical decomposition. Compound subtasks are declared in the plan but ex-

panded only at the execution frontier (Section 3.2). This lazy expansion yields three complementary benefits. First, downstream planning always conditions on real upstream outputs rather than predictions, directly addressing the limitation of global upfront planners. Second, each layer's prompt contains only the current task description and a pruned `allowed_tools` subset, reducing both the number of tools the LLM must reason over and the volume of intermediate results it must attend to. Third, child layers receive only a compact task briefing from their parent, so accumulated search results, opened documents, and prior reasoning from sibling branches do not pollute the active context—a property that becomes increasingly important as task length grows, as the BrowseComp-Plus results confirm (Section 4.4).

Unified single-model design. A single LLM performs all roles—planning, reasoning, executing, and returning—within one standard multi-turn conversation. The format is identical to standard chat training, eliminating train–inference format mismatch, and naturally compatible with prompt caching because the system prompt and early conversation turns remain stable prefixes across layers and similar tasks.

4 Experiments

4.1 Setup

Benchmarks. We evaluate on three benchmarks. **StableToolBench** (Guo et al., 2024) is built on ToolBench's 16,464 real-world REST APIs and partitions tasks by complexity. We report Solvable Pass Rate (SoPR) on the full-set. To further validate the effectiveness of our method on multi-step, multi-tool tasks, we additionally evaluate on the Multi-Turn subset of **BFCL Multi-Turn** benchmark (Yan, 2025). To validate the advantage of dynamic hierarchical decomposition in long-context, multi-step settings, we further evaluate on **BrowseComp-Plus** (Chen et al., 2025b), a dataset of 150 complex multi-hop web research questions requiring information synthesis across multiple sources and 5–100 search steps. (Section 4.4).

Baselines. We focus on comparisons with planning-based methods: (1) *training-based* methods—ToolLLaMA (Qin et al., 2024) with ReAct (Yao et al., 2023b) and DFSDT, and BeyondReAct (Wei et al., 2025); (2) *training-free* methods—LLMCompiler (Kim et al., 2024), and ReAct, DFSDT, and Smurfs run on GPT-4 Turbo and Qwen3.5-35B-A3B. On BFCL Multi-Turn we compare HDAP against the native function-calling baseline on Qwen3.5-35B-A3B.

4.2 Results on multi-tool benchmarks

Table 3 reports SoPR across all six splits on StableToolBench. The central prediction of our framework is that richer dependency modeling and hierarchical decomposition should matter most when tasks require coordinating multiple tools across categories—precisely the G3 setting. This prediction holds: on G3-Inst., HDAP reaches 73.8% (GPT-4 Turbo) and 90.2% (Qwen3.5-35B-A3B), compared to 61.5% and 83.8% for DFSDT. The gap widens monotonically from G1 to G3, confirming that the benefit of explicit dependency structure scales with task complexity rather than reflecting a uniform prompt-engineering advantage.By incorporating alternative planning and hierarchical planning, we achieve significantly greater robustness in handling unexpected situations compared to BeyondReact, which generates a complete, action-level DAG in a single pass.

Figure 2 shows BFCL Multi-Turn results. HDAP's per-layer context isolation bounds the effective context length at each decision point by limiting each layer's prompt to the current subtask and its pruned tool subset, confirming that hierarchical decomposition actively manages attention over long-horizon interactions.

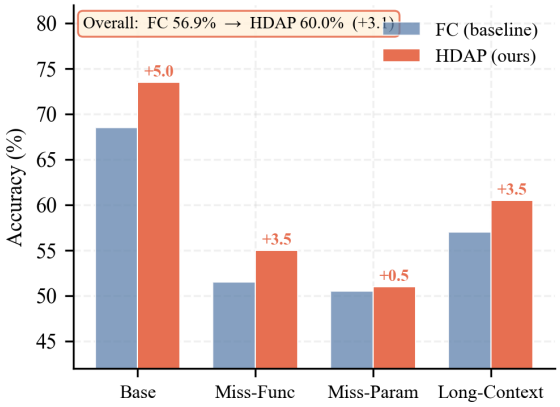

Figure 2: BFCL Multi-Turn accuracy (%) on Qwen3.5-35B-A3B. Miss-F: missing function; Miss-P: missing parameter; Long-C: long context.

4.3 Ablation Study

Table 4 shows that every component contributes to accuracy, with drops ranging from 3.4 to 7.0

Method	LLM	G1-Inst.	G1-Cat.	G1-Tool.	G2-Cat.	G2-Inst.	G3-Inst.	Avg
Training-based								
ToolLLaMA (ReAct)	ToolLLaMA	42.7	38.6	35.4	40.9	39.9	29.8	37.9
ToolLLaMA (DFSDT)		56.6	56.5	55.5	53.4	49.7	53.6	54.2
BeyondReAct	Qwen3-8B + GPT-4o	67.0	59.3	56.6	54.7	60.1	61.0	59.8
Training-free								
LLM Compiler	GPT-3.5	39.2	39.8	35.1	38.4	37.5	27.0	36.2
ReAct	GPT-4 Turbo	41.1	53.2	42.2	50.0	38.7	37.7	43.8
DFSDT		52.7	58.2	59.7	59.3	52.2	61.5	57.3
Smurfs		59.3	73.3	67.4	66.7	55.5	70.5	65.5
HDAP		**67.8**	**65.4**	**70.9**	**61.3**	**74.5**	**73.8**	**69.0**
CoT	Qwen3.5-35B-A3B	71.8	76.1	77.5	65.7	64.2	78.7	72.3
DFSDT		75.8	75.8	84.2	69.1	67.5	83.9	76.0
Smurfs†		65.3	82.9	76.9	63.0	54.2	54.1	66.1
HDAP		**82.5**	**73.9**	**86.7**	**86.3**	**83.5**	**90.2**	**83.8**

Table 3: SoPR (%) on StableToolBench. G1/G2/G3: single-tool / intra-category / cross-category. Baseline results on GPT-4 Turbo are sourced from Smurfs (Chen et al., 2025a). †Smurfs on Qwen3.5-35B-A3B is our reproduction from the released code.

Variant	Avg SoPR	AvgQ	AvgTok
HDAP (full)	**83.8%**	3.9	15,636
w/o re-planning	80.4%	7.4	31,063
w/o precondition	79.3%	4.1	15,681
w/o graph (linear)	79.3%	8.5	28,971
w/o context inherit	79.0%	4.2	18,349
w/o parallel	78.8%	13.2	53,156
w/o planning	78.4%	4.6	15,300
w/o hierarchy	76.8%	3.8	13,203

Table 4: Ablation on StableToolBench (Qwen3.5-35B-A3B). Each row removes one component from the full HDAP system.

points. Beyond accuracy, removing certain components incurs substantial computational overhead: disabling parallelism triples token consumption and quadruples query count, while removing re-planning nearly doubles both——suggesting these components do not merely improve quality but actively prevent redundant computation. Notably, the contrast between hierarchy and parallelism is particularly revealing: removing hierarchy produces the largest accuracy drop (-7.0 points) while slightly *reducing* cost, whereas removing parallelism hurts both accuracy and efficiency severely. This confirmings that hierarchical decomposition earns its keep through focused per-layer context rather than added LLM callsthe two components address distinct bottlenecks—hierarchy controls context focus and commitment granularity, while parallelism converts simultaneous precondition satisfaction into concurrent execution. Finally, the symmetric accuracy drops from removing preconditions and the graph (-4.5 points each) reflect their complementary roles: preconditions encode dependency semantics, the graph provides the topology over which they are evaluated, and neither alone is sufficient.

4.4 Analysis: Context Management via Dynamic Hierarchical Decomposition

To quantify the context management benefit of hierarchical decomposition (Section 3.2), we evaluate on BrowseComp-Plus, a dataset of 150 multi-hop web research questions requiring 5–100 search steps per query. Both HDAP and ReAct use Qwen3.5-35B-A3B with the same search tool.

Context growth. Figure 3 compares per-step context size and cumulative token consumption. ReAct's append-only history grows monotonically, reaching 309K tokens at step 30 and exceeding the 256K window for many queries. Under HDAP, each child layer inherits only a task briefing, so the active context—measured as the maximum across the pending layer stack—averages 80K tokens (3.9× reduction). Cumulative consumption at step 30 is 0.86M tokens versus ReAct's 5.13M (6.0× reduction).

Robustness under constrained windows. We conduct experiments comparing performance across varying context window lengths. Figure 4 reports accuracy. At 8K, HDAP retains 55.3% while ReAct collapses to 2.0%. At 32K—representative of deployed settings—HDAP achieves 65.3% versus ReAct's 22.7%. With the full 256K window, HDAP leads 68.7% to 60.0%, with a null-answer rate of

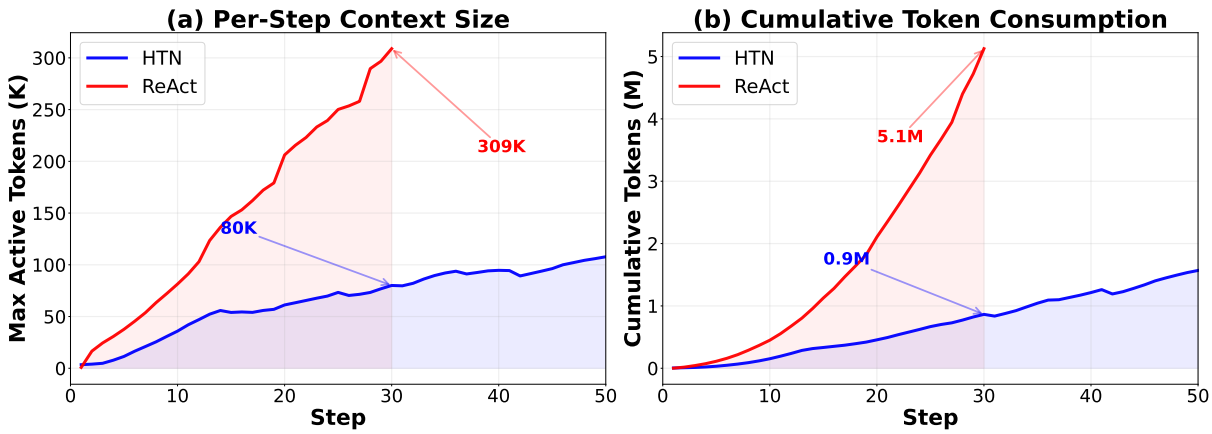

Figure 3: Context growth on BrowseComp-Plus. (a) Per-step prompt tokens. (b) Cumulative token consumption.

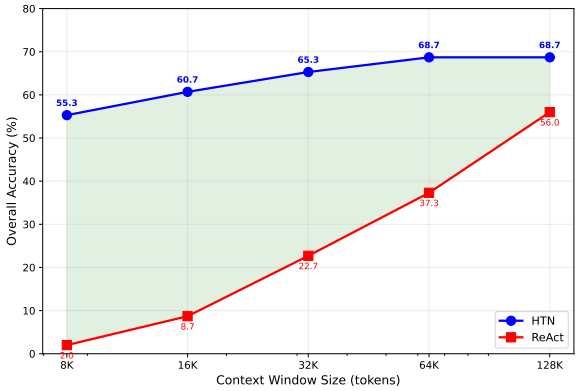

Figure 4: Accuracy under simulated context window sizes on BrowseComp-Plus. HDAP retains substantial accuracy under tight budgets (8k); ReAct requires $\geq$100K to reach comparable performance.

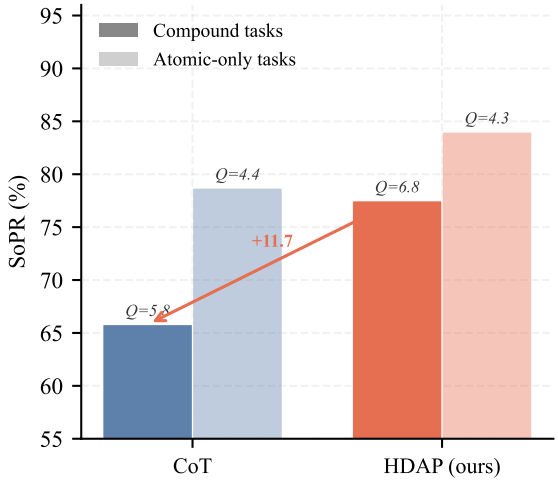

Figure 5: Performance split by task complexity (Qwen3.5-35B-A3B). Compound: tasks where HDAP triggered at least one recursive expansion.

1.3% versus ReAct's 31.3%. The low null rate is structural: even when a sub-layer fails, the RETURN mechanism allows the parent to deliver a partial answer from completed sibling work, whereas ReAct has no fallback when its context fills up.

4.5 Analysis: Hierarchical Decomposition on Complex Tasks

To isolate the effect of hierarchical decomposition, we partition the test set by whether HDAP triggered at least one compound expansion or remained single-layer. Table 5 confirms two things. First, both methods require more queries on the compound partition than on atomic-only tasks (CoT: 5.8 vs. 4.4; HDAP: 6.8 vs. 4.3), validating that HDAP's expansion decisions track genuine task complexity rather than firing indiscriminately. Second, HDAP's accuracy advantage over CoT is larger on compound tasks (+11.7 points) than on atomic-only tasks (+5.3 points), confirming that hierarchical decomposition contributes most precisely where task structure demands it.

5 Conclusion

We presented HDAP, a hierarchical dependency-aware planning framework for tool-augmented LLM agents. HDAP introduces a unified precondition mechanism that expresses AND, OR, and conditional dependencies, and a dynamic hierarchical execution policy that decomposes compound tasks only at the execution frontier. Together, these enable parallel dispatch, result-conditioned planning, and local failure recovery without global upfront commitment. Experiments show consistent improvements over sequential, search-based, and graph-based baselines, with gains that scale with task complexity and context length.

Limitations

HDAP introduces planning overhead that may not be justified for simple single-tool tasks, where a direct ReAct-style loop is likely sufficient. The reliance on natural-language preconditions, while flexible, sacrifices formal verifiability: precondition evaluation depends on the LLM's reasoning, and subtle ambiguities may cause incorrect readiness judgments. Parallel dispatch, though supported in our framework, requires an execution environment that can handle concurrent API calls; in strictly sequential settings, this benefit is unrealized.

References

Muhao Chen, Mian Liu, Zhizhen Wan, and Benyou Yin. 2025a. SMURFS: Leveraging multiple proficiency agents with context-efficiency for tool planning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3333–3349.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, and 1 others. 2025b. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*.

Zhicheng Guo and 1 others. 2024. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*.

Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. 2020. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Mengkang Hu, Yao Mu, Xinmiao Yu, Mingyu Ding, Shiguang Wu, Wenqi Shao, Qiguang Chen, Bin Wang, Yu Qiao, and Ping Luo. 2023. Tree-planner: Efficient close-loop task planning with large language models. *arXiv preprint arXiv:2310.08582*.

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Nevo Rozen, Elad Schwartz, Gal Shachaf, Shai Shalev-Shwartz, and Amnon Shashua. 2022. MRKL systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An llm compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, pages 24370–24391.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116.

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

Xinbei Ma, Zhuosheng Zhang, and Hai Zhao. 2025. Divide-then-aggregate: An efficient tool learning method via parallel tool invocation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*.

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

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

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. 2023. RestGPT: Connecting large language models with real-world RESTful APIs. *arXiv preprint arXiv:2306.06624*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Xiaolong Wei, Yuehu Dong, Xingliang Wang, Xingyu Zhang, Zhejun Zhao, Dongdong Shen, Long Xia, and Dawei Yin. 2025. Beyond ReAct: A planner-centric framework for complex tool-augmented LLM reasoning. *arXiv preprint arXiv:2511.10037*.

Jiaqi Wu and 1 others. 2025. GAP: Graph-based agent planning with parallel tool use and reinforcement learning. *arXiv preprint arXiv:2510.25320*.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. TravelPlanner: A benchmark for real-world planning with language agents. In *Proceedings of the 41st International Conference on Machine Learning*, pages 54590–54613.

Fanjia Yan. 2025. A function calling perspective on scalable large language model agent evaluation. Master's thesis, EECS Department, University of California, Berkeley.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*.

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

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

A Case Study: Hierarchical Execution on a Multi-Clue Research Task

Table A1 illustrates HDAP's execution on a multi-clue short film identification task. Three aspects of the trace are worth noting. First, the root layer identifies four independent search angles and dispatches all four as compound subtasks in a single LLM turn, with no ordering constraints among them—parallelism emerges naturally from empty preconditions rather than from an explicit scheduling module. Second, the four child layers exhibit substantially different execution depths: `search_architect_director` converges in one search call (Step 12), while `search_sound_designer` requires nine sequential tool calls (Steps 14) to pin down an ambiguous clue. This adaptive depth is a direct consequence of lazy expansion: each child layer plans independently given its own task briefing, without inheriting the complexity or context of its siblings. Third, all four subtasks independently converge on the same answer, allowing the root layer to merge results with high confidence (Step 16–17) and deliver a null-free final answer—a structural advantage over flat agents, which have no fallback when any single search path fails.

#	Layer	Action	
Question: Identify the short film matching 6 clues—director is also an architect (2022), produced 2018 / 13 min, one-shot Alexa mini, 3 months rehearsals, about a mother heading to a theatre audition discovering her son is unattended, sound designer skipped half a year of music education before 2010. *Tools:* search, open_page.			
1	Root	**PLAN** → 4 compound subtasks, all preconditions empty (no dependencies):	
		▷ `search_plot_details`	[C, tools: search + open_page]
		▷ `search_technical_specs`	[C, tools: search + open_page]
		▷ `search_architect_director`	[C, tools: search + open_page]
		▷ `search_sound_designer`	[C, tools: search + open_page]
2	Root	**EXECUTE** (parallel) — all 4 dispatched in **one LLM turn**	
	▷ root-1	`search_plot_details`	[depth 1, compound]
3		search → "Patision Avenue" found	[docid 51334]
4		open_page → verify director, year, runtime, one-shot, Alexa Mini	
5		search → verify sound designer Alejandro Cabrera	
6		**RETURN** → "Patision Avenue (Leoforos Patision)," confidence 95%	
	▷ root-2	`search_technical_specs`	[depth 1, compound]
7		search → "Patision Avenue" found	[docid 51334]
8		open_page → verify: 2018, 13 min, one-shot, Alexa Mini	
9		search → verify plot details	[docid 66663]
10		search → verify sound designer education	[docid 47975]
11		**RETURN** → "Patision Avenue," 6/6 clues confirmed	
	▷ root-3	`search_architect_director`	[depth 1, compound]
12		search → Thanasis Neofotistos, architect + film director	
13		**RETURN** → "Patision Avenue," director=architect confirmed	
	▷ root-4	`search_sound_designer`	[depth 1, compound]
14		9 sequential search + open_page calls, probing the sound designer clue from multiple angles before converging on the same film	
15		**RETURN** → "Patision Avenue," sound designer confirmed	
16	Root	**REASON** — merge 4 child results. All independently confirm same answer.	
17	Root	**RETURN** → "Patision Avenue (Leoforos Patision)," confidence 99%.	
		15 LLM queries, 18 tool calls, 5 layers total.	

Table A1: End-to-end execution of HDAP. The root layer PLANs four independent compound subtasks (Step 1), dispatches them in parallel (Step 2), and merges their results (Step 16–17). Each child layer executes at its own depth—from 1 tool call (root-3) to 9 (root-4)—returning only the verified answer to the parent.