TaReD: Tool-Aware Recursive Decomposition for Long-Horizon Tasks

Anonymous authors
Paper under double-blind review

Abstract

Agents combine reasoning with tools to interact with external systems and complete real-world tasks. Early agents typically interleave reasoning and actions along a single execution chain. On complex tasks, this chain becomes unreliable because growing histories obscure intermediate dependencies and allow early planning errors to propagate. Recursively decomposing a complex task into smaller subtasks offers a natural solution, yet effective decomposition must account for the system's capabilities so that each subtask can be executed by the available tools. In realistic systems, however, tool libraries can be too large to expose in full. Injecting every tool description consumes substantial context while making relevant tools harder to retrieve and useful task boundaries harder to identify. We propose tool-aware recursive decomposition, which organizes tools by functional relationships into a hierarchy of capabilities. During execution, the agent discovers tools on demand and uses the hierarchy to recursively decompose a complex task into a subtask tree whose levels are aligned with the capabilities required at each stage. Experiments on complex real-world tasks show that the proposed method improves end-to-end task success rate by up to 40 percentage points over the compared baselines.

1 Introduction

Agents combine language-based reasoning with executable tools to retrieve information and change state in external environments. Tool interfaces extend an agent's capabilities beyond its language context, but also introduce additional planning requirements: the agent must identify which tool is relevant, construct valid arguments, order calls whose results and state changes may influence one another, and maintain those dependencies across a long sequence of operations (Karpas et al., 2022; Yao et al., 2023; Schick et al., 2023; Patil et al., 2023; Qin et al., 2023; Wang et al., 2025).

Early agents typically interleave reasoning and actions along a single execution chain. In this chain-based style, the agent repeatedly reasons about the current state, selects and invokes a tool, observes its result, and then continues from the updated state. As tasks become more complex, the chain must carry more intermediate states and dependencies, making it harder to maintain and recover; an early planning or execution error can propagate through later steps and reduce end-to-end success (Wei et al., 2022; Yao et al., 2023; Liu et al., 2024).

Task decomposition addresses the limitations by breaking a complex task into an ordered sequence of subtasks with explicit dependencies (Zhou et al., 2023; Wang et al., 2023; Khot et al., 2022). Early decomposition methods generally derive these subtask boundaries from task semantics, such as goals, entities, and procedural structure. However, semantic decomposition alone does not ensure that each subtask corresponds to an operation the current system can execute: it may introduce a subtask with no directly executable capability or omit a necessary stage.

A high-quality task decomposition should align with the execution environment's capabilities, so that every subtask can be carried out by corresponding operations available in that environment. Achieving this requires information about system capabilities (Ichter et al., 2023; Karpas et al., 2022; Li et al., 2023; Song et al., 2023). The challenge is to expose enough capability information to make subtasks executable without overwhelming the planner. As tool libraries continue to grow in realistic tasks, this challenge becomes increasingly pronounced: injecting the descriptions and contracts of all tools consumes substantial context, while the volume of tool information can distract the planner from selecting tools relevant to the task. Predicting a small, task-relevant tool subset and exposing only that subset to the planner partially alleviates this burden (Qin et al., 2023; Shi et al., 2025). However,

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

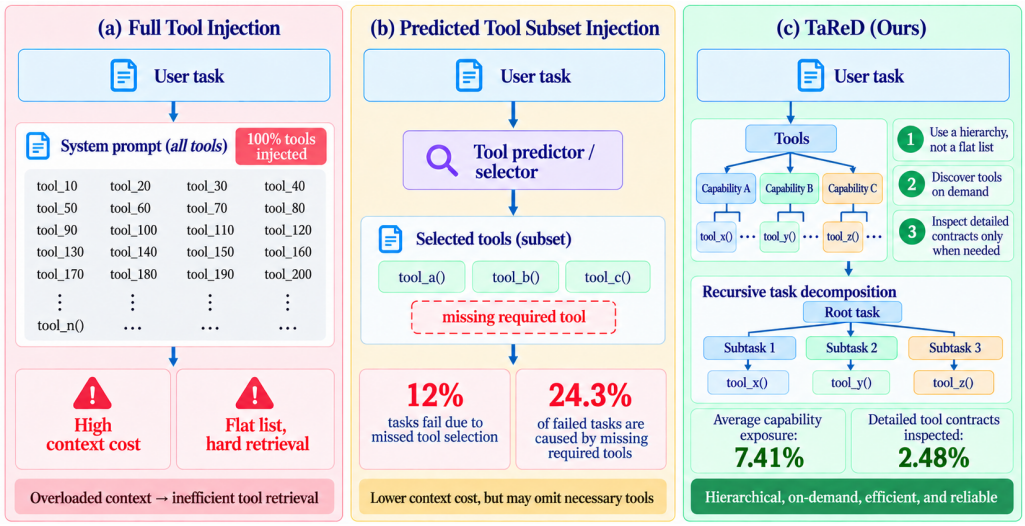


Figure 1: Motivation for tool-aware recursive decomposition. (a) Injecting descriptions and contracts of the full library consumes substantial context and distracts selection of task-relevant tools. (b) A fixed predicted subset can omit necessary tools and offers little structural guidance for dividing a task into capability-aligned subtasks. (c) TaReD navigates a capability hierarchy to retrieve relevant information on demand, guide recursive decomposition toward executable stages, and inspect detailed contracts only when needed.

this approach may omit necessary tools and still presents tool information as a flat list, providing little structural guidance for task decomposition.

We propose *TaReD* (Tool-Aware Recursive Decomposition), which organizes tools by functional relationships into a hierarchy of capabilities. When performing a task, TaReD navigates the hierarchy on demand, uses the discovered capability region to recursively split a task into functional stages, and binds an exact tool contract only when a stage is ready to execute. TaReD only retrieves the tool information needed for each task, avoiding both full-library context injection and the omission of necessary tools caused by a fixed subset. The hierarchy's functional relations also provide structural guidance for dividing the task into capability-aligned stages (Figure 1).

To assess the effectiveness of our method, we evaluate TaReD on complex interactive tasks in AppWorld (Trivedi et al., 2024), comparing it with representative recursive, linear, and plan-then-execute strategies (Prasad et al., 2024; Yao et al., 2023; Wang et al., 2023). All implementations use matched models, budgets, verifiers, and service conditions. Experimental results show that TaReD improves end-to-end task success rate by up to 40 percentage points over the compared baselines. Our contributions are summarized as follows:

- We propose TaReD (Tool-Aware Recursive Decomposition), which organizes a large tool library into a hierarchy and tightly couples tool discovery with task decomposition.
- We implement an architecture and execution protocol for tool-aware recursive decomposition that fully realizes the proposed algorithm and achieves strong performance in task execution.
- We validate TaReD on the AppWorld (Trivedi et al., 2024) test set and show a clear improvement in end-to-end performance over the compared baselines.

2 Related Work

Task decomposition and hierarchical planning. Chain-of-thought agents execute a task as a single sequence of interleaved reasoning and actions (Yao et al., 2023). But as task complexity grows, long action chains strain state tracking and allow early errors to propagate. Task-decomposition methods address this limitation by breaking complex tasks into interdependent subproblems and solving them in an order that respects their dependencies (Zhou et al., 2023; Prasad et al., 2024; Wang et al., 2023;

Khot et al., 2022). Early semantic-driven studies decomposed tasks based on their goals, entities, or procedural structure (Press et al., 2022). However, semantic decomposition alone does not account for which operations the environment can actually execute. A subtask may therefore fail to specify an executable sequence, motivating the use of tool information to ground decomposition.

Tool use and large tool libraries. Early tool-using research studied how agents decide when to invoke tools and how to incorporate their results into ongoing reasoning (Karpas et al., 2022; Yao et al., 2023; Schick et al., 2023). In these settings, the available tools and their descriptions were typically provided in the agent's context. However, as the scale of tools increases, it is impractical to include the full tool information in the context. To avoid overwhelming the planner with lengthy tool information, some methods first predict or retrieve a task-relevant subset and plan over that subset. Although selection based on prediction or retrieval reduces irrelevant choices, it can omit a necessary tool even when the full library contains a solution. Moreover, the selected tools are typically presented as a flat list, which provides little guidance about capability relations or how to organize subtasks around capability boundaries.

Our work organizes tools by functional relationships into a hierarchy of capabilities and queries this hierarchy on demand during execution. This avoids placing the full tool library in the context while retaining on-demand access to every tool in the library, reducing the risk of omitting a necessary tool. The hierarchy also exposes relations among capabilities and provides structural guidance for decomposing tasks into subtasks aligned with the operations available in the environment.

3 PRELIMINARY

We study how an agent can leverage the extensive tool information available in its execution environment to decompose complex tasks effectively and improve long-horizon task execution. A successful execution should achieve the task objective while driving the environment from its initial state to a desired final state.

3.1 TASK AND ENVIRONMENT

We consider a long-horizon task specified by an instruction x and an initial environment state s_0. The agent interacts with the environment through a library of executable tools. Each tool T_i is described by a contract

$$T_i = (\mathsf{In}_i, \mathsf{Out}_i, \mathsf{Pre}_i, \mathsf{Eff}_i), \tag{1}$$

where In_i and Out_i specify its input and output schema, while Pre_i and Eff_i describe its preconditions and state-changing effects. Calling a tool with arguments z_t produces an observation o_t and updates the environment according to

$$(o_t, s_{t+1}) = \mathsf{Execute}(T_i, z_t, s_t). \tag{2}$$

An execution trajectory is a sequence $\tau = ((T_{i_t}, z_t, o_t))_{t=1}^{L}$. The task is completed when a goal predicate $\mathsf{Done}(x, s_L)$ is satisfied.

3.2 TOOL LIBRARY AND CAPABILITY HIERARCHY

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ denote the complete tool library. The library may be too large to expose in the context at once, so we organize tools by functional category into a tree:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \tag{3}$$

where internal nodes summarize capability regions and leaf nodes correspond to tools with complete execution contracts. Each node $v \in \mathcal{V}$ is associated with a capability description $\mathsf{cap}(v)$ and, when v is a leaf, a tool contract T_i. A path from the root to a leaf therefore represents progressively more specific information about an executable capability.

3.3 RECURSIVE TASK DECOMPOSITION

We represent a decomposition as a rooted, ordered task tree whose nodes correspond to functional stages. Specifically, the tree and a stage h are represented as

$$\mathcal{H} = (\mathcal{N}, \mathcal{E}_h), \qquad h = (q_h, c_h, r_h), \quad h \in \mathcal{N}. \tag{4}$$

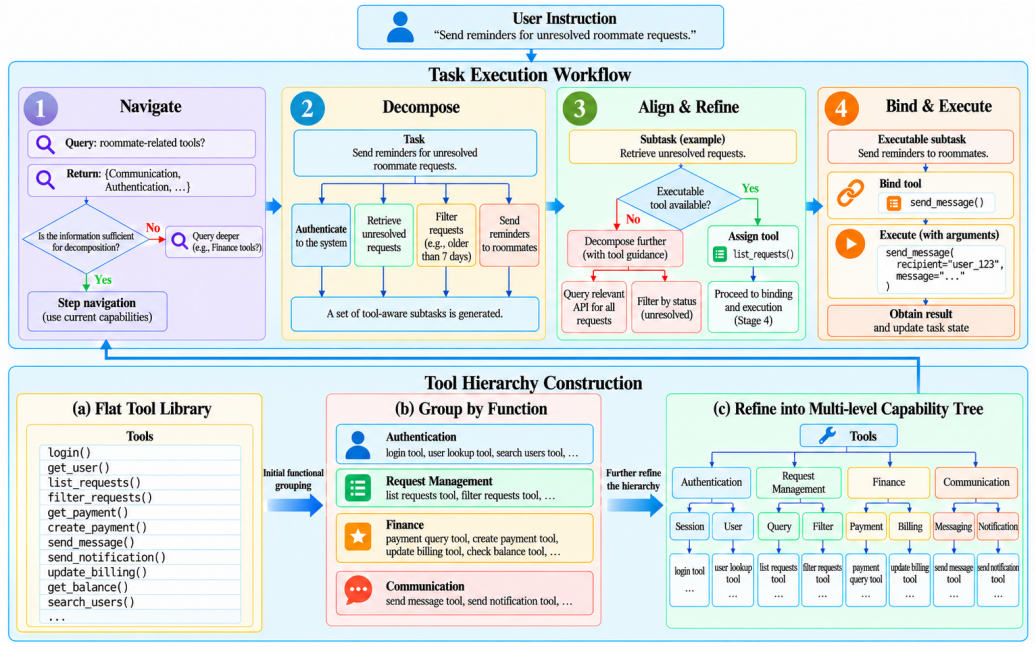


Figure 2: Overview of TaReD. Bottom: a flat tool library is grouped by function and refined into a multilevel capability tree. Top: the agent navigates this hierarchy, decomposes and refines subtasks using capability information, then binds arguments and executes tools when a stage is ready to execute. Returned results update the task state for subsequent stages.

Here, q_h is the stage goal, c_h contains the relevant task context, and $r_h \in \mathcal{V}$ denotes the capability region expected to carry out the stage. An internal node is recursively decomposed into an ordered sequence of child stages, $h \to (h_1, \ldots, h_k)$, while a leaf stage is ready to bind one or more concrete tool contracts and execute.

3.4 Problem Objective

Let $\mathsf{Align}(\mathcal{H}, \mathcal{T})$ denote the condition that every leaf in the recursive task tree has a corresponding tool in the execution environment and that each tool call satisfies its precondition at the time of execution. Given $(x, s_0, \mathcal{G})$, we seek a task tree $\mathcal{H}$ such that $\mathsf{Align}(\mathcal{H}, \mathcal{T})$ holds and the resulting trajectory satisfies $\mathsf{Done}(x, s_L)$. We therefore formulate the planning objective as

$$\max_{\mathcal{H}, \tau} \Pr[\mathsf{Done}(x, s_L)] \quad \text{s.t.} \quad \mathsf{Align}(\mathcal{H}, \mathcal{T}) \tag{5}$$

The objective maximizes the probability that the final environment state satisfies the task goal, while the alignment constraint ensures that the recursive decomposition is consistent with the tool library available in the execution environment.

4 TaReD: Tool-Aware Recursive Decomposition

Existing research on agents still faces challenges when executing long-horizon tasks in environments with large tool libraries. Injecting the full tool library into the context can overwhelm the planner, whereas predicting or retrieving a smaller subset may omit necessary tools and offers little guidance for structuring subtasks. TaReD addresses these limitations by organizing tools into a capability hierarchy and discovering them on demand, keeping the context focused while preserving access to the full library and guiding decomposition around capability relations.

4.1 OVERALL WORKFLOW

TaReD organizes a large tool library into a tree according to functional relationships among tools. During execution, the agent starts at the general root and navigates downward as needed to discover increasingly specific capabilities and tool information; information about irrelevant branches is not added to the context. Starting from the original task, the agent performs this tool-awareness process until it has enough capability information to identify functional stages, then decomposes the task into an ordered set of subtasks and processes them depth first. For each subtask, the agent first decides whether it is directly executable. If it remains complex, TaReD repeats tool awareness and decomposition recursively. If it is simple enough to execute, the agent plans the required tool calls, retrieves their detailed contracts, binds the necessary arguments, and invokes the tools. The resulting values and environment state are stored for use by subsequent dependent subtasks. Execution terminates when no pending task nodes remain or when the agent determines that the available capabilities are insufficient to continue the task.

4.2 HIERARCHICAL CAPABILITY REPRESENTATION

TaReD organizes the complete tool library into a rooted hierarchy $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The hierarchy contains a root node for the complete library, internal capability nodes at different levels of granularity, and leaf nodes corresponding to executable tools. Internal nodes represent functionally related capability regions and provide navigation rather than execution. Each internal node is associated with a compact capability card

$$v = (\mathsf{name}, \mathsf{summary}, \mathsf{children}, \mathsf{scope}), \tag{6}$$

which summarizes the operations supported by that region, the kind of information it consumes and produces, and the boundary of the capability it covers. Child links expose progressively more specific functional regions. A leaf represents a concrete executable operation and provides the exact contract needed when that operation is ready to run.

The hierarchy is therefore more than a flat index of tool names. Its internal relations expose functional boundaries that can be used to scope a subtask: a high-level node identifies what kind of operation is needed, while progressively deeper nodes provide the specificity required to choose an executable tool. By making these relations explicit, the hierarchy represents the execution environment's capability structure and provides guidance for task decomposition.

4.3 ON-DEMAND CAPABILITY NAVIGATION

For a task node $h = (q_h, c_h, r_h)$, where q_h is the current goal, c_h is the available context, and r_h is its expected capability region, the controller navigates the capability hierarchy using the goal and the state of the task:

$$C = \mathsf{Navigate}(q_h, c_h, r_h; \mathcal{G}). \tag{7}$$

Starting from the expected region r_h, the controller queries the capability hierarchy level by level, gathering information relevant to the current goal q_h and context c_h. It continues until the retrieved information is sufficient to plan the current task node. If the expected region does not provide all the capability information needed for planning, the controller searches other regions of the hierarchy and uses their information to fill the gaps. Only information from visited nodes is added to the context.

On-demand navigation keeps irrelevant tool information out of the context and focuses the agent on capabilities needed for the current task. Because the controller can search the complete hierarchy, necessary tools are not omitted by an initially predicted subset.

4.4 CAPABILITY-ALIGNED RECURSIVE DECOMPOSITION

Once the available capability information is sufficient to reason about the current goal, TaReD decides whether that goal can be handled as a compact execution stage. If so, it proceeds directly to the corresponding tool calls, without adding another task-decomposition level. When further decomposition is needed, TaReD uses the acquired capability information to represent the current task as an ordered sequence of child tasks. Let $h = (q_h, c_h, r_h)$ denote the current task node. Its

recursive refinement is

$$\mathsf{Refine}(h) = \begin{cases} \mathsf{Exec}(h), & \mathsf{Direct}(h) = \mathrm{True}, \\ (\mathsf{Refine}(h_j))_{j=1}^{k}, & \mathsf{Direct}(h) = \mathrm{False}, \end{cases} \qquad h_j = (q_j, c_j, r_j), \quad k > 1, \tag{8}$$

where $\mathsf{Direct}(h) \in \{\mathrm{True}, \mathrm{False}\}$ indicates whether the current goal and context, together with the observed capability information, support a compact executable stage. When $\mathsf{Direct}(h) = \mathrm{False}$, TaReD forms $k > 1$ child tasks $h_1, \ldots, h_k$ whose goals $q_1, \ldots, q_k$ capture the capability-grounded functional parts of h. Let $\mathcal{D}_h$ denote the dependency relation among these children, where $(h_i, h_j) \in \mathcal{D}_h$ means that h_j needs an output produced by h_i. For each dependency $(h_i, h_j) \in \mathcal{D}_h$, let B_{ij} denote the input bindings that pass only the values required from h_i to h_j:

$$\mathcal{D}_h \subseteq \{(h_i, h_j) \mid 1 \le i < j \le k\}, \qquad c_j = c_h \cup \{B_{ij} \mid (h_i, h_j) \in \mathcal{D}_h\}. \tag{9}$$

TaReD processes the ordered children depth first: it fully refines and handles h_1 before moving to h_2, and so on. A child that is itself decomposed is handled by the same rule before its parent advances to the next sibling. Each B_{ij} references only the output value required from h_i, rather than merging that predecessor's full context or output record into c_j. The scheduler dispatches a child only after its declared predecessors are closed and its bindings can be resolved to established values. After a child returns, the parent receives its status, summary, and outputs. If the child completed and the committed layer permits automatic advancement, the scheduler dispatches the next pending sibling in order; otherwise, or when a child fails, is uncertain, or requires review, control returns to the parent for review, repair, or replanning. The parent judges whether its goal is satisfied from the actual results and may declare completion when no child obligations remain open and required outputs are available.

The capability hierarchy makes the execution environment's functional structure explicit and guides task decomposition toward capabilities the environment can support. Because navigation proceeds from broad functions to specific operations, TaReD can divide a task according to the capability represented by an intermediate node, without requiring every decomposition decision to select individual tools. This allows the planner to control subtask granularity at the level of functional stages and refine toward concrete operations only when needed. On-demand navigation also keeps each planning context focused on tool information relevant to the task, helping the agent devote its attention to decomposition rather than unrelated capabilities.

4.5 Delayed contract binding and execution

TaReD delays exact contract binding until the current task is judged directly executable. The controller then plans an ordered sequence of one or more tool calls, $P_h = (a_1, \ldots, a_m)$. Before each call, it retrieves the corresponding contract d_i and binds arguments from that contract, the stage context, and the values already stored in $\mathcal{M}$. Calls execute in the planned order, with each result and updated environment state available to subsequent calls. For a stage entered with state s_0 and store $\mathcal{M}_0$, where these denote the current state and value store at the start of this stage rather than the task's initial state or an empty store, the execution and result recording at step i are

$$z_i = \mathsf{Bind}(d_i, c_h, \mathcal{M}_{i-1}), \quad (o_i, s_i) = \mathsf{Execute}(a_i, z_i, s_{i-1}), \tag{10}$$

$$\mathcal{M}_i = \mathcal{M}_{i-1} \cup \{(k_i, o_i, \mathsf{prov}(o_i))\}, \qquad i = 1, \ldots, m. \tag{11}$$

Here, z_i is a contract-valid argument assignment, o_i is the returned observation, s_i is the resulting environment state, and k_i is a stable reference to the stored value and its available provenance. Later calls in the same stage and dependent subtasks can use these references to retrieve required values.

This design keeps detailed parameter schemas out of earlier decomposition decisions while grounding every executed call in its exact contract. It also lets one task stage coordinate multiple sequential operations without forcing each operation into a separate recursive subtask. The referenced value store supports reliable handoffs between calls and subtasks without copying full earlier responses into the context; provenance remains available for tracing, while values can still be used when provenance is unavailable. Together, delayed binding and reference-based handoffs preserve state and dependencies while keeping execution focused on the information each call needs.

4.6 Algorithmic summary

Algorithm 1 expresses TaReD as a recursive procedure over task nodes. At each call, h denotes the current task, and Q is the ordered set of subtasks produced by decomposing it; $\mathcal{D}$ records their

dependencies, and B contains their input bindings. Sufficiency and direct-execution checks are planner judgments grounded in observed capability information. Each recursive call handles one task, and its subtask results return to that current task for review and further planning.

Algorithm 1 TaReD: Tool-Aware Recursive Decomposition

Require: Original task x, initial context c_0, environment state s_0, capability hierarchy $\mathcal{G}$
Ensure: Root status y_0, final environment state s, referenced value store $\mathcal{M}$

Initialize $h_0 = (x, c_0, \text{root}(\mathcal{G}))$, $\mathcal{M} \leftarrow \emptyset$, and $s \leftarrow s_0$
$(y_0, s, \mathcal{M}) \leftarrow \text{HANDLENODE}(h_0, s, \mathcal{M}, \mathcal{G})$
return $(y_0, s, \mathcal{M})$

function $\text{HANDLENODE}(h, s, \mathcal{M}, \mathcal{G})$
    $C \leftarrow \text{NAVIGATE}(q_h, c_h, r_h, \mathcal{G})$
    if $\text{SUFFICIENT}(h, s, \mathcal{M}, C) = \text{False}$ **then**
        return $(\mathsf{insufficient}, s, \mathcal{M})$
    end if
    if $\text{DIRECT}(q_h, c_h, s, \mathcal{M}, C) = \text{True}$ **then**
        $P_h \leftarrow \text{PLANCALLS}(q_h, c_h, s, C)$
        for all $a \in P_h$ in the planned order **do**
            $d \leftarrow \text{INSPECTCONTRACT}(a)$; bind arguments z from d, c_h, and $\mathcal{M}$
            $(o, s) \leftarrow \text{EXECUTE}(a, z, s)$
            Assign fresh reference k to o and set $\mathcal{M}[k] \leftarrow (o, \mathsf{prov}(o))$
            Expose k as an output reference for dependent subtasks
        end for
        $y_h \leftarrow \text{ASSESSSTAGE}(h, s, \mathcal{M})$
        return $(y_h, s, \mathcal{M})$
    end if
    $(Q, \mathcal{D}, B) \leftarrow \text{DECOMPOSE}(q_h, c_h, s, C)$
    $\mathsf{review_required} \leftarrow \text{False}$
    while pending subtasks remain or the current task requires review **do**
        $h_j \leftarrow$ first pending depth-first subtask with completed $\mathcal{D}$-predecessors and resolvable B
        if $h_j = \varnothing$ or $\mathsf{review_required} = \text{True}$ **then**
            $(Q, \mathcal{D}, B, \rho) \leftarrow \text{REVIEWORREVISE}(h, Q, \mathcal{D}, B, s, C)$
            if $\rho = \mathsf{stop}$ **then**
                return $(\mathsf{insufficient}, s, \mathcal{M})$
            end if
            $\mathsf{review_required} \leftarrow \text{False}$
        else
            Resolve h_j's inputs using B, retrieving referenced predecessor values from $\mathcal{M}$
            $(y_j, s, \mathcal{M}) \leftarrow \text{HANDLENODE}(h_j, s, \mathcal{M}, \mathcal{G})$
            Record y_j and its output references in Q
            if $y_j \neq \mathsf{completed}$ **then**
                $\mathsf{review_required} \leftarrow \text{True}$
            end if
        end if
    end while
    $y_h \leftarrow \text{ASSESSTASK}(h, s, \mathcal{M})$
    return $(y_h, s, \mathcal{M})$
end function

Appendix B walks through an illustrative task to show how capability discovery, recursive subtask dependencies, and delayed contract binding interact during execution.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Environment and evaluation. AppWorld is an interactive environment with long-horizon tasks and a large tool library of 457 tools spanning nine applications (Trivedi et al., 2024). Its tasks are organized

into scenarios, each containing several variants with a shared underlying objective and consistent task type. It provides an official programmatic verifier for checking the resulting environment state. We evaluate the most challenging tasks in its *Challenge set (Test-C)*, using GPT-5.6 Luna for all methods and a 600-second limit per episode. Other shared run settings include a fixed random seed of 42, a 100-step episode cap and temperature-0 generation.

Compared methods. We compare four methods: *TaReD*, *ADAPT*, *ReAct*, and *Plan-and-Solve*. We use the official AppWorld implementations of ReAct and Plan-and-Solve, while retaining ADAPT's official architecture and adapting it to the AppWorld Challenge set. TaReD combines hierarchical tool awareness with recursive task decomposition and delays exact tool-contract binding until execution. ADAPT follows an executor-first recursive strategy, decomposing a task when direct execution cannot complete it (Prasad et al., 2024). ReAct interleaves reasoning and tool calls in a single linear trajectory without recursive task decomposition (Yao et al., 2023). Plan-and-Solve follows the two-stage design of first generating a complete natural-language plan and then executing the plan sequentially (Wang et al., 2023). All three baselines use a one-shot Predictor to select a task-specific subset of tools.

Metrics. We report AppWorld's Task Goal Completion (TGC), Scenario Goal Completion (SGC), and an Average Task Verification Score (ATVS). TGC is the fraction of individual tasks for which all official verifier checks pass. SGC is the fraction of scenarios for which all task variants in the scenario pass, measuring consistency across related task formulations. Let p_i and m_i denote the number of verifier checks passed and performed for task i, respectively. The ATVS is

$$S = \frac{100}{N} \sum_{i=1}^{N} \begin{cases} p_i/m_i, & \text{if the verifier ran, } m_i > 0, \text{ and } p_i > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{12}$$

where N is the number of evaluated tasks. This normalization sets the maximum ATVS to 100.

5.2 Main Results and Analysis

Table 1: Results on the AppWorld Challenge set (Test-C). Task Goal Completion (TGC) requires all official verifier checks to pass for an individual task, Scenario Goal Completion (SGC) requires all variants in a scenario to pass, and Average Task Verification Score (ATVS) reports normalized verifier progress with partial credit. TaReD achieves the highest value on all three metrics in the adopted run set.

Method	TGC	SGC	ATVS
TaReD (ours)	**69.3%**	**48%**	**80.28**
ADAPT (Prasad et al., 2024)	46.7%	22%	66.01
ReAct (Yao et al., 2023)	51.3%	26%	66.81
Plan-and-Solve (Wang et al., 2023)	29.3%	16%	60.60

Overall, TaReD performs best among the four methods in Table 1 across individual-task completion, scenario-level consistency, and partial-credit verification. TaReD's advantage over ADAPT follows from the interaction between recursive control and tool awareness. ADAPT recursively plans inside a one-shot, flat tool subset, so omitting a necessary inspection, lookup, or verification tool can leave the task impossible to complete. Moreover, the flat tool list provides little guidance about capability relations or boundaries, making it harder to organize the task into well-grounded functional stages. TaReD instead uses the capability hierarchy to organize the task into functional stages, expands discovery when the current region is insufficient, and postpones exact contract binding until a stage is ready to execute. The combination of effective tool awareness and a well-designed recursive framework enables higher-quality task decomposition and reduces issues that may arise during recursive execution.

ReAct follows a single linear reasoning process with the same flat tool-selection setting. It can avoid child scheduling, dependency handoff, and recursive-plan validation errors that affect ADAPT. However, its linear execution continually accumulates interaction history while providing little structure for organizing the task into distinct stages. Like ADAPT, ReAct also faces the risk of missing necessary tools. TaReD combines the staging benefits of recursion with capability-guided,

on-demand discovery, which is consistent with its better performance in this evaluation. Plan-and-Solve provides another non-recursive planning comparison. It has the lowest TGC, SGC, and ATVS. In addition to potentially missing necessary tools under the same flat Predictor-selected view, Plan-and-Solve commits to a complete plan before execution, leaving limited scope to revise it in response to execution-time problems; consequently, an early error can disrupt subsequent steps.

5.3 Ablation Studies

Tool Awareness within the Recursive Framework. We compare four tool-awareness configurations built from the TaReD recursive execution framework. *Predictor subset* selects a fixed flat tool subset once before execution. *Full tool injection* places all 457 short tool descriptions in the context and retrieves detailed contracts on demand. *Expandable tool subset* starts with the same Predictor-selected subset, but it allows missing tools to be found through AppWorld's official documentation and added to the subset during execution. *Hierarchical tool awareness* uses TaReD's capability hierarchy to guide tool discovery and recursive decomposition (Table 2).

Table 2: Tool-awareness ablations on the adopted AppWorld Test-C run set. TGC measures task completion, SGC completion of all scenario variants, and ATVS partial verifier progress.

Tool awareness	TGC	SGC	ATVS
Predictor subset	51.3%	32%	66.52
Full tool injection	52.0%	22%	71.81
Expandable tool subset	58.7%	42%	77.78
Hierarchical tool awareness	**69.3%**	**48%**	**80.28**

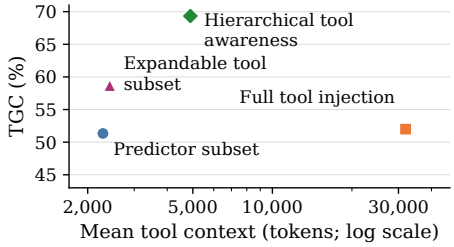


Figure 3: Mean tool-context tokens per saved decision (log scale) versus TGC for the four settings in Table 2.

Full tool injection yields only a slight improvement in task completion over the Predictor subset, suggesting that exposing all tool information does not adequately address the problem: lengthy tool descriptions can distract the agent and complicate tool selection and execution. The expandable tool subset achieves a clearer improvement, supporting on-demand discovery as a way to recover missing tools without placing the entire library in context. Hierarchical tool awareness further improves performance, suggesting that capability hierarchies provide better guidance for task decomposition and execution than flat tool lists. Figure 3 further shows that hierarchical tool awareness substantially improves task execution performance with only a modest increase in tool context.

Tool Exposure and Omission Analysis. We further examine whether flat Predictor selection can hide necessary tools. 12% of the tasks showed strong evidence of a blocking omission, accounting for 24.3% of the total failed cases. This illustrates a failure mode of committing to a flat subset before execution: a capability that was never exposed cannot be selected later.

For TaReD, we count the distinct tools whose capability information appears in a recorded model context and those for which the detailed executable contract is inspected. Across all runs, capability information was exposed for 7.41% of the library on average, while detailed contracts were inspected for 2.48%. The measurements indicate selective exposure in the audited TaReD runs.

6 Conclusion

TaReD addresses the difficulty of decomposing long-horizon tasks when large tool libraries cannot be exposed in full. It combines on-demand hierarchical tool discovery with capability-guided recursive decomposition. This design aligns subtask boundaries with available capabilities without requiring full-library context. A remaining limitation is the overhead of hierarchical tool discovery: navigating several capability levels can require additional model calls and increase execution time relative to one-shot tool selection. Our current implementation uses a tool cache so that previously queried capability cards and tool information can be reused instead of being discovered repeatedly. This reduces redundant queries but increases the amount of information retained in the working context. The resulting trade-off between discovery overhead and context usage remains open.

AI USE STATEMENT

We used generative AI tools as research assistants for code inspection, experiment bookkeeping, and language editing. We also used AI to curate the capability taxonomy for our tool tree; Appendix A.4 details the AI-assisted curation and deterministic construction process. All experimental claims, data processing, figures, and final text are reviewed by the authors against the project records. Any future use beyond these categories will be disclosed here and in the OpenReview submission form.

REFERENCES

Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 287–318. PMLR, 2023. URL https://proceedings.mlr.press/v205/ichter23a.html.

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

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, 2024.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-naacl.264. URL https://aclanthology.org/2024.findings-naacl.264/.

Ofir Press, Muru Zhang, et al. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022.

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. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24497–24524. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1258. URL https://aclanthology.org/2025.findings-acl.1258/.

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. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*, 2023.

Harsh Trivedi, Tushar Khot, et al. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.

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. *arXiv preprint arXiv:2305.04091*, 2023.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. ToolGen: Unified tool retrieval and calling via generation. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/b646bdebeb87dfafe2c6f77a63b564e1-Abstract-Conference.html.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WZH7099tgfM.

A Capability-Tree Construction

This appendix describes how we transform a conventional flat tool library into the hierarchical capability representation used by TaReD. The construction is performed before task execution; it is therefore a static organization of capabilities rather than a tree inferred separately for each task. Its internal nodes summarize capability categories at progressively finer levels of granularity, while its leaves preserve the original executable tool contracts.

A.1 From a flat catalogue to a hierarchy

The input is the public tool catalogue, whose records contain tool identifiers, descriptions, parameter schemas, response information, and any documented preconditions or side effects. We remove the excluded canary field and validate each record's documented identity and required contract fields while preserving its parameters and response schemas. Tools with distinct application semantics or contracts remain separate.

We then assign every tool a single primary path in a capability taxonomy. The path is constructed from progressively more specific functional distinctions: a broad capability domain, an application or business object when needed, an operation scenario, and finally the concrete operation. For example, shopping tools can be organized under a commerce domain, then catalog, cart, or order capabilities, before reaching individual product-search or checkout operations. A new intermediate level is introduced only when it separates meaningful capabilities or execution boundaries; different branches may consequently have different depths. The taxonomy is explicit and versioned, so rebuilding the tree with the same catalogue and taxonomy yields the same primary hierarchy.

The construction creates a root node for the complete library, creates internal capability nodes for the taxonomy paths, and attaches each normalized tool as a leaf under its deepest applicable category. The edges encode parent–child containment and progressively finer functional scope. A tool has one primary parent, which prevents duplicate execution identities and makes descendant counts well-defined. Some capabilities are related across branches, for example authentication may be relevant to several application domains. We represent such relationships with non-tree navigation links rather than copying a tool or treating the link as an execution order. This preserves a tree for hierarchical browsing while allowing the agent to discover a relevant capability outside its current branch when the available evidence is insufficient.

A.2 Node types and stored information

The generated catalogue contains three structural node types. The root is a non-executable entry point for the entire library. Category nodes are non-executable internal nodes that summarize a capability region and point to more specific categories or tools. Native-tool nodes are executable leaves that retain one complete tool contract. Composite tools are not created implicitly by the builder: a category is never treated as callable, and a composite operation would require a separately implemented node type and contract.

Every node has a stable identifier, a display name, a node kind, a parent reference when applicable, child references, and a concise semantic summary. Category nodes additionally store the information needed for planning and navigation: the scope or boundary of the region, input and output hints, relevant preconditions, known limitations, representative capability facets, related-node links, source references, and statistics over descendant contracts. These summaries describe what the region can support; they are not a unioned executable schema, and they do not expose the concrete values or current availability of accounts, tokens, or object identifiers.

Each native-tool node stores the exact source identity and executable contract, including the operation path or method, parameter names and types, required and default values, constraints, natural-language description, and structured success or failure response information. Source references and catalogue checksums allow the leaf description to be traced back to the original tool documentation. Runtime state is deliberately excluded from the static tree: values such as access tokens, product identifiers, and current records are obtained from the environment only when the corresponding stage executes.

A.3 Construction checks and runtime exposure

After the tree is assembled, we validate that every tool has exactly one reachable primary parent, every category reference resolves, node kinds agree with their executable status, and descendant counts match the attached contracts. We also check that related-node links do not duplicate tools or silently define an execution order, and that each leaf contract agrees with its source record. These checks make catalogue changes fail explicitly instead of silently changing a tool's capability region.

The complete tree is not placed in the initial model context. At runtime, a discovery session begins at the root and exposes category summaries and direct children through bounded browsing or expansion. A category description can provide planning-level evidence without revealing all descendant schemas; a native-tool description reveals the exact contract only when the agent has navigated to that leaf and is ready to bind arguments. Pagination and response budgets bound each observation, while the session records only nodes that have actually been discovered. Thus, the same static tree supports reproducible construction and selective, on-demand visibility during recursive planning.

A.4 AI-Assisted Curation and Deterministic Construction

The capability tree is prepared in two stages. First, an AI agent (Codex) curates a versioned taxonomy from the public AppWorld API documentation. The taxonomy explicitly specifies each category's parent, functional summary, input and output hints, preconditions, limitations, related-category links, and member API identifiers. Categories are organized from broad capability domains toward application or business-object groups and more specific operation types. Each API is assigned to exactly one primary category; cross-cutting relations are recorded as links rather than duplicate assignments. This curation is task-independent: it does not use task instructions, reference solutions, live environment values, or verifier feedback.

Second, a deterministic builder converts the taxonomy and the complete public API-documentation set into the serialized catalogue. It checks that the expected application files are present, verifies each API's documented identity and required contract fields, and removes only the public loader's excluded canary field. It then instantiates non-executable category nodes and executable native-tool leaves, retaining the source API descriptions and exact parameter and response contracts. A traversal from the root assigns depths and derives descendant API counts, application coverage, representative source references, capability facets, and summaries of descendant contract properties. The current catalogue contains 137 category nodes and 457 native-tool leaves.

Finally, validation checks that every documented API is assigned once, each category is nonempty and non-callable, every node is reachable exactly once from the root, parent and depth references agree, and leaf contracts and source pointers match their documented records. The builder also records source and taxonomy hashes in the catalogue version and verifies an integrity hash over the generated artifact. Rebuilding from the same public documentation and taxonomy therefore produces the same tree; changes to either source require a new reviewed taxonomy or a regenerated catalogue. The AI agent curates the static organization, while this deterministic construction and validation process creates the artifact used by the runtime agent.

B Illustrative Execution Trace

This appendix gives a schematic example of how capability discovery, recursive decomposition, and execution fit together. It is illustrative rather than a trace from an evaluated episode; the capability regions and tool identifiers correspond to the catalogue, while the task and returned values are hypothetical.

Suppose the agent receives the goal: "Read the project deadline from an email and add it as a task in Todoist." The goal spans two functional stages: obtaining the deadline from communication data and creating a task in a productivity application. The agent first navigates from the root toward email capabilities. It inspects relevant summaries and children, then refines the goal into (i) locating and reading the relevant message and (ii) creating a Todoist task using the extracted deadline. The second child depends on information produced by the first, so the parent records that dependency instead of asking the second child to infer its inputs.

The first child is handled depth first. Once capability information identifies a suitable operation, the agent retrieves its exact contract, binds the message identifier, and executes it. The returned message content is stored under a stable reference, such as k_{mail}. The parent extracts the deadline and makes it available as a structured value associated with that reference, then closes the first child. The scheduler resumes the second child. Its expected region points toward Todoist task capabilities; if needed, the agent searches that region and selects the concrete `todoist.create_task` operation. Only when this stage is ready to execute does the agent inspect the call contract and bind the task title and due date from the task context and stored email result. The returned task identifier and status are saved in the value store, after which the parent checks whether the original goal has been met.

The example highlights two distinctions. Capability summaries guide the choice and granularity of task stages, but they are not executable contracts. The exact contract is bound only after a stage is ready to run. The reference from the email stage preserves the dependency between children without copying the full earlier response into every later planning context. If discovery in the expected region were insufficient, the controller could inspect another relevant capability region before deciding whether to refine or execute the current node.