REx: Small Language Model Agents with Recursive Execution and Subtree Compression

Anonymous authors
Paper under double-blind review

Abstract

Small language models are attractive for assistants because they can reduce the cost and latency of individual model calls, while local inference can limit exposure of sensitive data. However, long workflows require these models to coordinate dependent steps and retain useful results as execution histories grow. To address these demands, we introduce **REx**, which combines flexible planning with recursive execution and compression guided by unfinished work. Each planning round selects a batch, an ordered list of one or more subtasks, and assigns each to direct execution or recursive decomposition. Batch length can vary, with updated results guiding replanning after batch completion or subtask failure. To limit context growth, REx selectively replaces completed execution subtrees, the shared records of recursively decomposed subtasks and their nested subtasks, with compact memories. Task goals and any next planned subtask guide which outcomes and evidence to retain for unfinished work. We evaluate REx on Claw-Eval, GAIA, and Agent-Diff with three Gemma-4 models. Under matched models and limits on tool calls, REx outperforms the strongest evaluated baseline in all nine settings, improving task success by up to 19.88 percentage points over standard ReAct. Ablations support flexible planning and subtree compression, while further analysis shows the benefit of supplying task goals during compression.

The code is at `https://anonymous.4open.science/r/REx-0481`.

1 Introduction

Small language models are attractive for everyday assistants because they can make individual model calls cheaper and faster (Belcak et al., 2025; Liu et al., 2024b). Running them locally can also limit the exposure of sensitive data to external providers (Chen & Li, 2024). Recent work has further extended their capabilities to tool calling and web exploration (Chen & Li, 2024; Qian et al., 2025; Chen et al., 2026). However, many useful tasks require a long sequence of dependent steps, where the result of one step determines what should happen next. We refer to such tasks as having a *long horizon*. Although small models can perform individual steps effectively, reliably completing a long-horizon task requires them to coordinate decisions and information across many steps. We therefore study how to improve long-horizon task completion without changing the model itself.

Two difficulties become particularly important as the workflow grows longer: **(1) deciding what to do next as the task unfolds**, and **(2) retaining and using earlier results as the interaction history grows**. First, the model must not only execute each subtask, but also determine when it is complete, what should follow, and whether new results require revising the current plan. Such decisions are more demanding than executing an individual step. For example, ToolRL shows that small models perform much better on individual tool execution than on tasks requiring multiple rounds of interaction (Qian et al., 2025). Second, the model must keep track of useful information in its *working context*, which contains the history of plans, actions, and results supplied as model input. A user identifier obtained early in a workflow, for example, may still be needed much later, after old plans, failed attempts, and unrelated tool outputs have accumulated. As the context grows, relevant information can become harder to find and use (Liu et al., 2024a). Prior work on a four billion parameter agent similarly identifies redundant context as a source of poorer decisions (Chen et al., 2026). These difficulties compound over a long workflow: an incorrect decision or overlooked result at one step can affect the steps that follow.

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.

Existing methods organize long workflows through recursive task decomposition, repeatedly dividing a task into smaller subtasks (Prasad et al., 2024; Zhang et al., 2025). This narrows each local goal, but the agent must still coordinate execution and carry information between subtasks, leaving two potential problems: **(1) planning can repeat without advancing execution**, and **(2) limiting history can remove evidence needed later**. First, ReCAP revises its plan after each subtask, so proceeding to the next subtask requires another planning decision even when the remaining plan is still usable. Making such decisions does not always lead to action: in our ReCAP runs with small models, the agent sometimes continues decomposing or replanning without taking a new action. Second, ReCAP makes earlier results available across subtasks through a shared context, but limits its growth by deleting older records. An older result may still be necessary for unfinished work, so removing it can leave a later subtask without the information it needs. These limitations motivate letting execution continue under a usable plan and shortening completed histories according to what unfinished tasks still need.

To address these limitations, we introduce **REx**, which controls how many subtasks execute under one plan and what evidence remains available after they finish. **(1) Plan a batch and choose direct or recursive execution for each subtask.** Each planning round jointly chooses an ordered list of subtasks, called a *batch*, and assigns each to direct execution or recursive decomposition. The number of subtasks can vary across rounds. Direct execution uses a local ReAct loop of reasoning, actions, and observations (Yao et al., 2023). For a subtask assigned to decomposition, REx invokes the same planning and execution procedure to produce and carry out a batch within that subtask. The plan therefore determines both how much work to undertake next and which subtasks need their own plans. At each task level, execution follows the batch until it finishes or a subtask fails, then replans using the updated results. **(2) Compress completed subtrees to retain what unfinished tasks may need.** Planning and execution share one working context. When a recursively decomposed subtask finishes, REx selectively replaces its *execution subtree*, the records of that subtask and all its nested subtasks, with a compact memory. To decide what to retain, the compressor uses the current goal, the goal of the task that created it (its *parent*), and, when available, the next planned subtask under that parent (its next *sibling*). Task completion therefore identifies which records to compress, while the surrounding goals guide which results, evidence, and unresolved information to keep for later work. Compression occurs before returning to the parent; when that parent finishes, the memory can be compressed again as part of the parent's subtree.

We evaluate REx on a diverse set of agent benchmarks, including Claw-Eval, GAIA, and Agent-Diff (Ye et al., 2026; Mialon et al., 2023; Pysklo et al., 2026), using three small language models from Google Gemma-4 family (Gemma Team, 2026). REx outperforms the strongest evaluated baseline in all nine settings, with gains of up to 19.88 percentage points over standard ReAct under matched models and limits on tool calls. On Claw-Eval with Gemma-4-26B-A4B, subtree compression reduces the average peak context length by 24% while improving task success relative to the variant without compression.

Our main contributions are:

- **Batch planning with recursive execution.** We introduce REx, which jointly plans batches of varying lengths and assigns each subtask to direct execution or recursive decomposition without additional model training. At each task level, planning resumes after batch completion or subtask failure.

- **Subtree compression for unfinished work.** We selectively replace completed execution subtrees with compact memories in the shared context, using current and parent goals and any next planned sibling to guide information retention. These memories can be compressed again when their parent tasks complete.

- **Consistent gains across tasks and model scales.** REx outperforms the strongest evaluated baseline in all nine combinations of three benchmarks and three Gemma-4 models under matched models and limits on tool calls. Ablations support flexible planning and subtree compression, with further evidence for using task goals during compression.

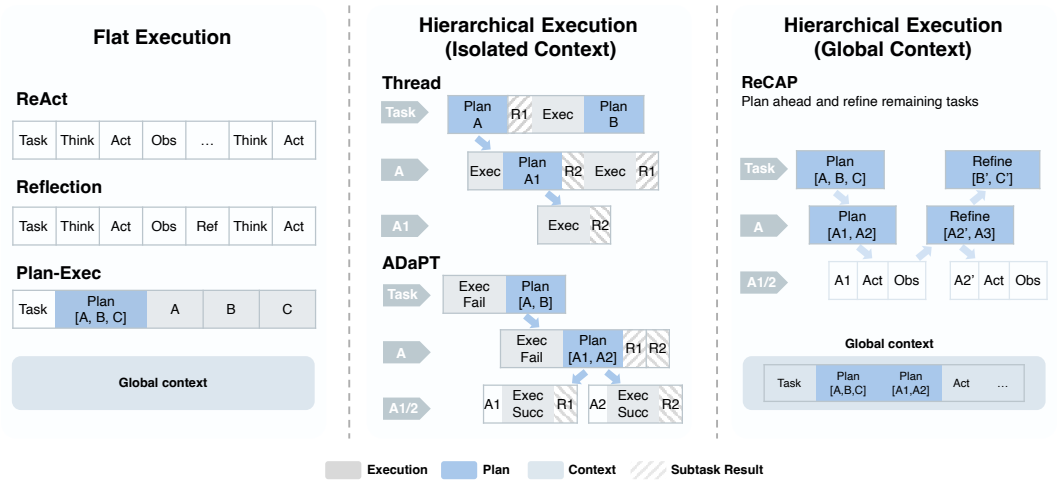


Figure 1: **Comparison of flat and hierarchical agent execution.** Flat execution organizes reasoning, planning, and tool use within a single execution loop. The recursive methods shown delegate subtasks through nested calls. THREAD plans one subtask at a time, delegates it to a child thread, and receives the result before continuing. ADaPT first attempts direct execution and decomposes the task if execution fails. ReCAP generates an ordered subtask plan and revises the remaining plan after each subtask. THREAD and ADaPT construct local contexts for recursive calls, whereas ReCAP shares a global context.

2 Related Work

Small language models as agents. Recent work extends small language models from individual tool calls to longer interactive tasks. Octopus v2 targets efficient function calling with a two billion parameter model (Chen & Li, 2024), while ToolRL studies reinforcement learning rewards for tool selection and argument prediction (Qian et al., 2025). AgentCPM-Explore trains a four billion parameter agent for extended exploration and refines retrieved information before adding it to the working context (Chen et al., 2026). REx complements these efforts by organizing planning, execution, and memory without additional model training.

Planning and local execution. Agent frameworks use plans and execution feedback to coordinate actions. ReAct interleaves reasoning, actions, and observations (Yao et al., 2023), while AdaPlanner and DEPS explicitly revise plans using execution feedback (Sun et al., 2023; Wang et al., 2023). LLMCompiler represents dependencies between tool calls to support parallel execution (Kim et al., 2024). REx uses ReAct within individual subtasks and controls when to return to planning through the recursive procedure described below.

Hierarchical and recursive execution. Task decomposition narrows local objectives, but methods differ in when they decompose tasks and how much work they execute before planning again. Earlier prompting methods divide complex problems into simpler subproblems (Zhou et al., 2023; Khot et al., 2023). ReAcTree organizes embodied agents into a tree of subgoals (Choi et al., 2026), while THREAD recursively spawns child threads with local contexts and returns selected information to their parents (Schroeder et al., 2025). ADaPT first attempts execution and recursively decomposes a task when its executor fails (Prasad et al., 2024). ReCAP shares context across recursive calls, generates an ordered subtask plan, and refines the remaining plan after each subtask (Zhang et al., 2025). Figure 1 contrasts these execution and context structures. REx retains ReCAP's shared context but jointly chooses a batch, an ordered list of one or more subtasks, and direct or recursive execution for each item. Thus, decomposition can be selected before attempting execution, and successful subtasks advance through the batch without another planning call at that level. Batch length can vary, and planning resumes after batch completion or subtask failure.

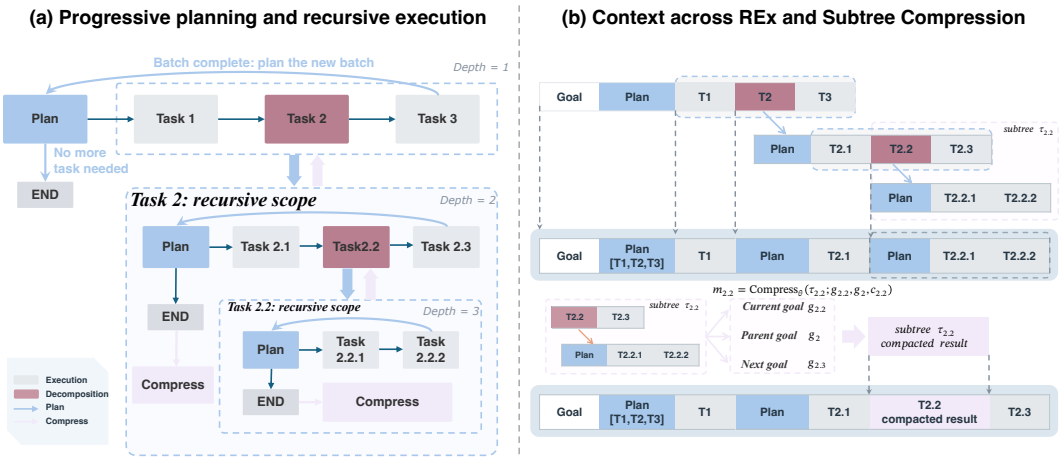


Figure 2: **Overview of REx.** (a) Progressive planning and recursive execution. REx generates successive batches of tasks, executing each task directly or recursively decomposing it into subtasks. After a batch completes, planning continues until the current goal is satisfied. Blue arrows indicate planning, red boxes indicate local ReAct execution, and pink arrows indicate subtree compression at recursive returns. A failed subtask discards the remaining siblings and triggers replanning at the current level (b) Global context and subtree compression. Plans and execution records across recursion depths accumulate in a context. When Task 2.2 completes, an LLM call compresses its execution subtree into a compact handoff memory, which replaces the corresponding context segment before execution continues. The compression call is conditioned on the child goal, parent goal, and the next planned sibling. The same process applies when its parent task completes.

Context management during execution. Context management methods differ in which records they compress and what guides information retention. ReCAP limits shared context by removing older dialogue rounds and reinserting the parent plan (Zhang et al., 2025). HiAgent summarizes completed subgoal histories using the corresponding goals and can retrieve the original records when needed (Hu et al., 2025). ACON optimizes instructions for compressing observations and interaction histories using feedback from successful and failed trajectories (Kang et al., 2026). AgentFold learns to condense individual steps or consolidate longer portions of history (Ye et al., 2025). Context-Folding executes a subtask in a branch and replaces its intermediate records with a summary when returning to the main thread; its reported execution design does not allow nested branches (Sun et al., 2025). REx selectively compresses completed subtrees at recursive returns within a shared context. Its compression unit is a completed execution subtree: the records of a recursively decomposed subtask and its descendants, including any existing memories of completed descendants. The current and parent goals, together with the next planned sibling task when available, guide which results and evidence to retain. When the parent later completes, these memories can be compressed again as part of its subtree, allowing compression to follow the same nested structure as execution.

3 METHOD

As illustrated in Figure 2, REx organizes long-horizon agent execution into a hierarchy of tasks. It combines *progressive planning*, which schedules successive batches using updated results, with *recursive execution*, which applies the same procedure within subtasks. Planning and execution across the task hierarchy use a single global context, shared across all levels of the task hierarchy and updated with new plans, actions, and observations. Starting from the root task (the user's original request), at each planning stage, the model outputs the next batch of tasks together with a completion flag, denoted END, indicating whether additional tasks remain to be planned. Each task is jointly assigned an EXECUTE or DECOMPOSE mode: the former is handled by a local ReAct loop, whereas the latter recursively invokes the same planning and execution procedure for that task. After a batch is executed, planning resumes from the updated global execution state, allowing subsequent work to incorporate observations unavailable to earlier planning rounds. When a recursively decomposed subtask completes, REx compresses its execution subtree before execution resumes at

the parent level. The subtree is replaced by a compressed context conditioned on its parent goal and the relevant parent-level continuation, limiting the growth of the shared working context while preserving information needed for subsequent execution.

3.1 PROGRESSIVE PLANNING AND RECURSIVE EXECUTION

Let v denote a task node, with goal g_v and recursion depth d_v. The root has depth 1, and each subtask is one level deeper than its parent. The root task and tasks assigned DECOMPOSE are handled by the recursive planning and execution procedure, whereas tasks assigned EXECUTE are handled by a local ReAct loop.

At the k-th planning round for the current task v, the model inspects the global context and returns either END or the next variable-length batch of tasks. Here, a batch is an ordered list proposed in one planning call. Its length h_k, also called the *planning horizon*, is the number of subtasks in the list and can vary across rounds. For each planned subtask u_i of v, with goal g_{u_i}, the same planning output jointly specifies an execution mode $z_i \in \{\text{EXECUTE}, \text{DECOMPOSE}\}$. A subtask labeled EXECUTE is deemed directly executable and is handled by a local ReAct loop until it succeeds or fails. A subtask u_i labeled DECOMPOSE is instead handled by recursively invoking the same planning and execution procedure with g_{u_i} as the current goal. The recursive task hierarchy therefore emerges online from DECOMPOSE decisions made across successive planning rounds. Planning is therefore progressive rather than upfront: each call determines whether the current task is complete and, if not, which subtasks to undertake next and whether to execute them directly or recursively decompose them.

$$o_v^{(k)} = \mathrm{Plan}_\theta(g_v,\, d_v,\, \mathcal{C}_t)\,, \tag{1}$$

where $\mathcal{C}_t$ denotes the single global context at the execution state t at which the planning call is issued. The index k counts planning rounds for task v, whereas t indexes the evolving global execution state.

$$o_v^{(k)} \in \{\text{END}\} \;\cup\; \left\{ \left[\left(u_i^{(k)}, z_i^{(k)} \right) \right]_{i=1}^{h_k} \;\middle|\; h_k \geq 1,\; z_i^{(k)} \in \{\text{EXECUTE}, \text{DECOMPOSE}\} \right\} \tag{2}$$

Within each emitted batch, tasks are processed from left to right, with the execution state updated after each completed item. Successful tasks advance execution to the next planned sibling without replanning, preserving the commitment encoded by the current batch. Because a failure may invalidate subsequent siblings that were planned under an assumption of successful completion, REx treats it as a replanning boundary, discards the unexecuted remainder, and resumes planning for the current parent task v. If a subtask fails, the current recursive frame performs recovery replanning from the updated shared context. If execution fails again under the recovery plan, the current frame returns FAILURE to its parent, which treats the failed recursive call as a failed subtask and replans at the parent level. When the entire batch completes successfully, planning instead resumes with a new PLAN call from the updated state.

3.2 SHARED GLOBAL WORKING CONTEXT

Hierarchical execution decomposes a broad objective into narrower task goals, but isolating the context used for each task can weaken information continuity across the task hierarchy. As observed by prior recursive agent frameworks, information produced while resolving a subtask may remain relevant after control returns to its parent task, whereas restricting each recursive call to a local context requires such information to be selectively transferred between calls. This issue is particularly consequential in general agentic workflows, where tool observations often constitute persistent execution evidence: content retrieved from emails or documents, identifiers returned by APIs, intermediate search results, and environmental state discovered in one subtask may subsequently be required by sibling tasks or parent-level synthesis.

Following the shared-context design of ReCAP, REx therefore maintains a single global working context $\mathcal{C}_t$ across all recursion depths. Planning outputs, environmental observations, and recursive execution records are incorporated into the same evolving context in execution order. Recursively processing a subtask changes the current execution objective while preserving access to the global context; planning and execution calls at all recursion depths use this same evolving context. This separation allows recursive decomposition to provide **execution locality** while retaining **information continuity** across the task hierarchy.

3.3 CONTEXT-AWARE SUBTREE COMPRESSION

To limit the accumulation of detailed execution traces in the global context, REx replaces the execution subtree of a completed recursively decomposed subtask with a compact memory before returning control to its parent task. Let v denote the completed subtask and $p(v)$ its parent task, with g_v and $g_{p(v)}$ denoting their respective goals. We define τ_v as the representation of v's execution subtree currently present in the shared working context immediately before compression. It includes planning records, tool interactions, and observations, with previously returned descendant subtrees represented by their compressed memories rather than their original traces. The continuation c_v is the goal description of the next already planned sibling task after v in the current parent-level batch, or $c_v = \varnothing$ if no such task is available. It provides bounded lookahead into the existing parent plan, rather than a prediction of work that later planning rounds may generate.

Using the same underlying model with parameters θ and a dedicated compression prompt, REx produces a compact textual memory:

$$m_v = \mathrm{Compress}_\theta\big(\tau_v;\, g_v,\, g_{p(v)},\, c_v\big)\,. \tag{3}$$

The compression call uses a fresh context containing these inputs and the compression instructions, rather than the full shared global working context. The child goal g_v identifies the local objective that the trajectory was intended to accomplish, while the parent goal $g_{p(v)}$ guides the selection of outcomes and evidence relevant to the broader task. When available, c_v further informs this selection with the requirements of the next planned task. The prompt instructs the model to retain relevant task outcomes, supporting evidence, and unresolved state, while omitting local execution details judged unnecessary for subsequent work.

The resulting memory replaces the corresponding subtree segment in the shared working context:

$$\mathcal{C}_{t+1} = \mathcal{C}_t[\tau_v \mapsto m_v]\,, \tag{4}$$

where the bracketed operation replaces only the subtree segment associated with v, leaving the remaining context unchanged. Parent-level execution then resumes over the updated shared working context.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks and Metrics. We evaluate REx on three complementary benchmarks grounded in realistic agentic use cases, spanning multi-step workflow execution, open-world information seeking, and stateful API interaction. Claw-Eval (Ye et al., 2026) evaluates autonomous agents on realistic software workflows with auditable execution, and we use its 161 General tasks covering single-service operations, cross-service coordination, and complex multi-system workflows. GAIA (Mialon et al., 2023) evaluates general-purpose assistants on real-world questions requiring multi-step reasoning, web browsing, file processing, and tool use. Since the original benchmark does not provide an execution harness, we use Inspect Evals (UK AISI et al., 2024) as our evaluation framework and evaluate all 165 tasks in the GAIA validation set. Agent-Diff (Pysklo et al., 2026) evaluates agents on 224 enterprise API workflows across Slack, Box, Linear, and Google Calendar, with task outcomes verified through state changes in the execution environment. Across all benchmarks, we report Pass@1 as the primary metric, measuring the fraction of tasks successfully completed in a single independent execution under each benchmark's native success criterion. This single-attempt evaluation approximates a deployment-oriented setting in which an agent is expected to complete a task reliably within one execution rather than succeed through repeated independent trials.

Models. We evaluate REx with three backbone models at different scales: Gemma-4-E4B-it, Gemma-4-12B-it, and Gemma-4-26B-A4B-it (Gemma Team, 2026). Using multiple scales from the same model family allows us to examine how the effectiveness of recursive execution varies with underlying model capability, while all methods within each comparison share the same backbone, tool environment, inference configuration, and execution budget.

Table 1: Main results across benchmarks and model scales. All methods use the same backbone and tool-call budget within each setting. Within each backbone block, the best and second-best results are shown in bold and underlined, respectively.

Models	Method	Claw-Eval		GAIA	Agent-Diff
		Pass@1	Avg. Score	Pass@1	Pass@1
Gemma-4-E4B	ReAct	13.04	36.01	10.91	<u>9.82</u>
	Reflection	<u>21.73</u>	<u>45.36</u>	<u>11.52</u>	6.25
	Plan–Exec	19.25	44.13	9.70	7.59
	ReCAP	16.14	37.70	9.09	8.04
	REx	**29.81**	**54.23**	**15.76**	**16.07**
Gemma-4-12B	ReAct	32.91	57.40	29.09	<u>37.05</u>
	Reflection	28.57	54.59	27.27	24.11
	Plan–Exec	<u>39.75</u>	<u>60.73</u>	19.39	32.14
	ReCAP	38.50	58.86	<u>30.91</u>	22.77
	REx	**49.69**	**64.10**	**33.93**	**39.29**
Gemma-4-26B-A4B	ReAct	36.02	57.05	29.09	<u>57.14</u>
	Reflection	35.40	57.80	<u>30.91</u>	43.30
	Plan–Exec	<u>47.82</u>	<u>63.89</u>	26.06	56.25
	ReCAP	37.88	56.21	27.88	46.87
	REx	**55.90**	**68.04**	**40.00**	**60.27**

Baselines. We compare REx against four representative agent baselines spanning reactive tool use, self-reflective refinement, explicit plan-then-execute control, and recursive execution. For ReAct, we use the native agent execution protocol provided by each benchmark without modification. Reflection augments the standard Think–Act–Observe loop with an additional self-reflection step after every observation, yielding a Think–Act–Observe–Reflect cycle before the next action is generated. Plan-Exec first generates a complete plan decomposed into an ordered sequence of subtasks and then executes them sequentially; after each subtask, the model decides whether the remaining plan should be revised, and a failed subtask likewise triggers replanning over the unfinished portion of the plan. For ReCAP, we follow the execution and context-management procedure described in the original method (Zhang et al., 2025).

4.2 Main Results

Consistent Gains Across Benchmarks and Model Scales. REx achieves the highest Pass@1 in all nine benchmark–backbone settings (Table 1). Its gains over the strongest of the four baselines are larger on Claw-Eval than on Agent-Diff for every backbone, with smaller margins on Agent-Diff for the two larger models. Plan–Exec outperforms ReAct on Claw-Eval but underperforms it on GAIA and Agent-Diff with every backbone. REx's advantage therefore extends beyond benchmarks on which Plan–Exec already improves over ReAct.

Why REx Is More Stable on Small Models. Trace inspection reveals two failure modes of ReCAP that are particularly pronounced with smaller backbones. When the recursion depth is unrestricted, ReCAP often decomposes a task until it reaches an apparently atomic action, effectively becoming a ReAct trajectory augmented with a recursively maintained subtask list. In our traces, this behavior can reach a depth of 16 without producing corresponding task progress. Under the depth limit used in our main experiments, ReCAP requires the model to choose among `act`, `down`, and `up` at every step. Smaller models do not always follow this control protocol consistently; after reaching the depth boundary, they may repeatedly replan without producing a new tool action. REx addresses these failure modes by separating task-level control from action-level execution. Its planner only emits END or a batch of tasks, together with an EXECUTE or DECOMPOSE label, while each EXECUTE task is handled by a local ReAct loop. The recursion-depth limit therefore governs task decomposition rather than individual tool actions, and the model is not required to select among action, descent, and return operations at every step.

4.3 Planning and Compression Analysis

We examine planning horizon and subtree compression on Claw-Eval and GAIA with all three backbones. *Single-step* restricts each non-empty planning output to one task, whereas *One-shot* requests a complete plan for the current task. Both variants retain recursive execution and subtree compression. *REx w/o Comp* retains recursive execution but disables subtree compression at recursive return boundaries. Table 2 reports both the planning-horizon variants and the compression ablation.

	Claw-Eval			GAIA		
Method	E4B	12B	26B	E4B	12B	26B
REx	**29.81**	**49.69**	**55.90**	**15.76**	**33.93**	**40.00**
Single-step	27.33	41.61	49.69	13.33	28.48	37.58
One-shot	26.08	47.20	52.80	8.48	26.67	33.33
REx w/o Comp	26.08	43.48	51.55	10.91	30.30	38.79

Table 2. Planning-horizon and subtree-compression ablations on Claw-Eval and GAIA. All scores are Pass@1 (%). E4B, 12B, and 26B denote the corresponding Gemma-4 backbones. The best result in each column is bold.

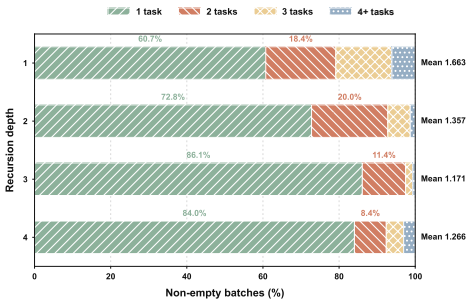


Figure 3. Non-empty planning-batch lengths by recursion depth. Percentages show batches with one, two, three, or at least four tasks; means are given at the right.

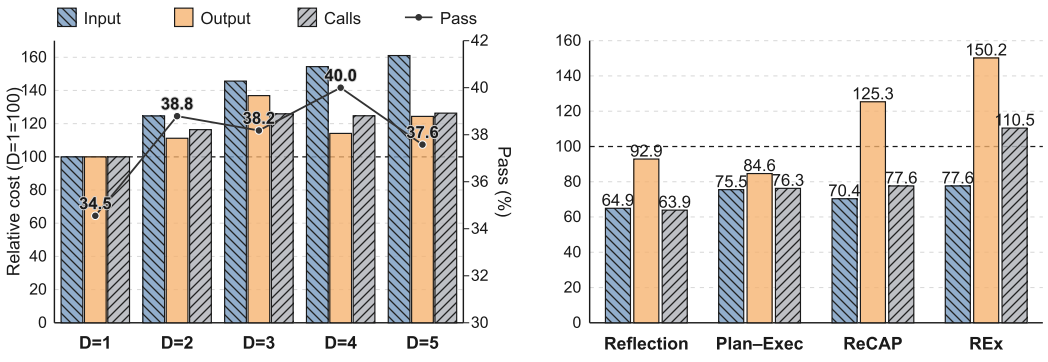
Planning Horizon Depends on the Execution Environment. The table suggests that the useful planning horizon depends on the task environment. REx outperforms Single-step on both benchmarks, but its advantage over Single-step is larger on Claw-Eval, especially with the 12B and 26B backbones. Claw-Eval consists largely of structured workflows with relatively stable dependencies among API operations, where planning several related tasks together can reduce drift during local execution. GAIA, in contrast, frequently depends on dynamically revealed information from browsing, files, and other observations. For such tasks, committing to a complete plan is more brittle: the larger gap between REx and One-shot across all three backbones is consistent with the benefit of shorter horizons and more frequent replanning. These results suggest this design choice is not uniformly longer or shorter planning, but an adaptive horizon that expands when execution dependencies are predictable and contracts when subsequent observations are likely to change the plan.

Short but Flexible Planning Horizons. REx implements this adaptive strategy through short but flexible batches rather than a fixed planning length. Most non-empty planning rounds in the analyzed REx trajectories on Claw-Eval and GAIA contain a single task, and the large majority contain at most two (Figure 3). Nevertheless, Single-step lowers Pass@1 in all six settings, and One-shot also underperforms REx throughout (Table 2). Allowing the planning horizon to vary across rounds therefore preserves the benefits of short plans while retaining the option to commit to several related tasks when the current execution state supports it.

Compression Improves Success and Reduces Context. Enabling subtree compression raises Pass@1 in all six settings (Table 2), supporting its use as part of the execution framework rather than only as a memory-saving measure. For 26B, the mean compressed-memory size is roughly one quarter of the mean raw-subtree size on both benchmarks (Table 3, Appendix B.1). Average peak context also decreases on both benchmarks, by 24.0% on Claw-Eval and 6.2% on GAIA. Compression thus improves task success while reducing peak context, with benchmark-dependent savings at the trajectory level. Our ablation further demonstrates the role of context awareness in preserving key evidence during context compression: on Claw-Eval with Gemma-4-26B-A4B, removing the current, parent, and next goals from compression calls lowers Pass@1 from 55.90% to 50.31%, below the 51.55% achieved without compression (Appendix B.2).

4.4 Accuracy–Computation Trade-offs

We measure input tokens, output tokens, and model calls per task on GAIA with Gemma-4-26B-A4B (Figure 4). Token totals include both the inputs and outputs of compression calls, including generated subtree memories.



(a) Recursion-depth trade-off.

(b) End-to-end inference consumption.

Figure 4: **Accuracy–computation trade-offs on GAIA with Gemma-4-26B-A4B.** (a) Effect of maximum recursion depth. $D = 1$ disables recursive decomposition and therefore corresponds to the non-recursive variant of REx. Resource consumption is normalized to the corresponding value at $D = 1$, and the solid line reports Pass@1 on the right axis. (b) Comparison with alternative execution methods. Resource consumption is normalized to the corresponding ReAct average, with ReAct omitted for clarity. Dashed horizontal lines indicate the normalization reference of 100. Lower consumption and higher Pass@1 are better.

Benefits and Trade-offs of Recursion Depth. The depth sweep in Figure 4a also isolates the effect of enabling the recursive REx mechanism. At $D = 1$, recursive decomposition is disabled entirely, reducing REx to its flat variant with progressive planning and local ReAct execution. Allowing recursion beyond this setting improves accuracy, indicating that hierarchical decomposition provides additional benefit over flat execution. However, this benefit is non-monotonic with increasing depth. In particular, $D = 2$ achieves higher Pass@1 than both $D = 3$ and $D = 5$ while consuming fewer input tokens, output tokens, and model calls. Compared with $D = 4$, the highest-scoring setting, $D = 2$ uses 19.3% fewer input tokens at a cost of only 1.21 percentage points in Pass@1. These results suggest that recursion is beneficial, but increasing the depth limit does not yield monotonic gains: $D = 2$ provides a lower-resource operating point, whereas $D = 4$ offers the highest accuracy among the tested settings.

Higher Accuracy with Lower Input Consumption. REx improves Pass@1 over ReAct with 22.4% fewer input tokens, including compression inputs, while generating 50.2% more output tokens and making more model calls (Figure 4b). The lower total input consumption despite more calls implies less input processed per model invocation on average, giving REx a favorable accuracy–input trade-off against ReAct. Compared with the other three baselines, REx achieves higher Pass@1 but uses more input tokens, output tokens, and model calls.

5 Conclusion

We presented REx, a recursive execution framework for small language model agents that separates task-level control from action-level execution. REx progressively generates variable-length task batches while maintaining a global context across recursion depths. At recursive return boundaries, it compresses completed execution subtrees into compact memories, using the parent objective and available planned continuation to guide information retention. Across Claw-Eval, GAIA, and Agent-Diff with three Gemma-4 backbones, REx achieves the highest Pass@1 among the evaluated methods in all nine settings under matched backbones and tool-call budgets. Ablations on Claw-Eval and GAIA support flexible planning horizons over fixed single-step or one-shot alternatives and show that subtree compression contributes to task success rather than serving only as a memory-saving measure. Together, these findings highlight the value of task hierarchy not only for decomposing objectives, but also for coordinating planning commitments and defining meaningful boundaries for compression, linking the organization of future work with the management of completed execution history.

AI USE STATEMENT

We used AI tools to assist with drafting and refining portions of the manuscript and organizing its presentation. The authors take full responsibility for all scientific claims, experimental results, and the final content of the paper.

REFERENCES

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic AI. *arXiv preprint arXiv:2506.02153*, 2025. doi: 10.48550/arXiv.2506.02153. URL https://arxiv.org/abs/2506.02153.

Haotian Chen, Xin Cong, Shengda Fan, Yuyang Fu, Ziqin Gong, Yaxi Lu, Yishan Li, Boye Niu, Chengjun Pan, Zijun Song, Huadong Wang, Yesai Wu, Yueying Wu, Zihao Xie, Yukun Yan, Zhong Zhang, Yankai Lin, Zhiyuan Liu, and Maosong Sun. AgentCPM-Explore: Realizing long-horizon deep exploration for edge-scale agents. *arXiv preprint arXiv:2602.06485*, 2026. doi: 10.48550/arXiv.2602.06485. URL https://arxiv.org/abs/2602.06485.

Wei Chen and Zhiyuan Li. Octopus v2: On-device language model for super agent. *arXiv preprint arXiv:2404.01744*, 2024. doi: 10.48550/arXiv.2404.01744. URL https://arxiv.org/abs/2404.01744.

Jae-Woo Choi, Hyungmin Kim, Hyobin Ong, Youngwoo Yoon, Minsu Jang, Dohyung Kim, and Jaehong Kim. ReAcTree: Hierarchical LLM agent trees with control flow for long-horizon task planning. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems*, 2026. doi: 10.65109/ucgt7089.

Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025. doi: 10.18653/v1/2025.acl-long.1575.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. ACON: Optimizing context compression for long-horizon LLM agents. In *International Conference on Machine Learning*, 2026.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *International Conference on Learning Representations*, 2023.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391. PMLR, 2024.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024a. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32431–32454. PMLR, 2024b. URL https://proceedings.mlr.press/v235/liu24ce.html.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. *arXiv preprint arXiv:2311.12983*, 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*, 2024.

Hubert M. Pysklo, Artem Zhuravel, and Patrick D. Watson. Agent-diff: Benchmarking LLM agents on enterprise API tasks via code execution with state-diff-based evaluation. *arXiv preprint arXiv:2602.11224*, 2026.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3524.

Philip Schroeder, Nathaniel W. Morgan, Hongyin Luo, and James R. Glass. THREAD: Thinking deeper with recursive spawning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 8418–8442, 2025. doi: 10.18653/v1/2025.naacl-long.427.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. AdaPlanner: Adaptive planning from feedback with language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 58202–58245, 2023. doi: 10.52202/075280-2537.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon LLM agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025. doi: 10.48550/arXiv.2510.11967.

UK AISI, Arcadia Impact, and Vector Institute. Inspect Evals: Collection of evaluations for Inspect AI, 2024. URL `https://github.com/UKGovernmentBEIS/inspect_evals`. Software repository. Accessed September 22, 2026.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with LLMs enables open-world multi-task agents. In *Advances in Neural Information Processing Systems*, volume 36, pp. 34153–34189, 2023. doi: 10.52202/075280-1480.

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

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, Qi Liu, Zhifang Sui, and Tong Yang. Claw-eval: Towards trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*, 2026.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. AgentFold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025. doi: 10.48550/arXiv.2510.24699.

Zhenyu Zhang, Tianyi Chen, Weiran Xu, Alex Pentland, and Jiaxin Pei. ReCAP: Recursive context-aware reasoning and planning for large language model agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 105457–105485, 2025. doi: 10.52202/085713-3181.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. In *International Conference on Learning Representations*, 2023.

A REx Algorithm and Implementation Details

Algorithm 1 gives the complete recursive execution procedure. We index the root task at depth 1. At the depth boundary, a DECOMPOSE label is handled by the local ReAct loop; in particular, $D = 1$ disables recursive decomposition.

Algorithm 1 REx: Progressive Recursive Execution with Subtree Compression

Require: Root goal g_{root}, initial context $\mathcal{C}_0$, model θ, maximum recursion depth D, compression threshold B_{comp}
Ensure: Final shared context $\mathcal{C}$ and execution status r

function REx($g_v, d_v, \mathcal{C}, g_{p(v)}, c_v$)
  recovering $\leftarrow$ FALSE
  while TRUE **do**
    $o_v \leftarrow \text{Plan}_\theta(g_v, d_v, \mathcal{C})$
    if $o_v =$ END **then**
      if $g_{p(v)} \neq \varnothing$ **then**
        $\tau_v \leftarrow$ subtree segment of v in $\mathcal{C}$
        if $\text{Tokens}(\tau_v) > B_{\text{comp}}$ **then**
          $m_v \leftarrow \text{Compress}_\theta(\tau_v; g_v, g_{p(v)}, c_v)$
          $\mathcal{C} \leftarrow \mathcal{C}[\tau_v \mapsto m_v]$
      return $(\mathcal{C},$ SUCCESS$)$
    $o_v = [(u_i, z_i)]_{i=1}^{h}$
    $c_{u_i} \leftarrow g_{u_{i+1}}$ for $i < h$; $c_{u_h} \leftarrow \varnothing$
    failed $\leftarrow$ FALSE
    for $i = 1, \ldots, h$ **do**
      if $z_i =$ DECOMPOSE **and** $d_v < D$ **then**
        $(\mathcal{C}, r_i) \leftarrow$ REx$(g_{u_i}, d_v + 1, \mathcal{C}, g_v, c_{u_i})$
      else
        $(\mathcal{C}, r_i) \leftarrow$ REACT$(g_{u_i}, \mathcal{C})$
      if $r_i =$ FAILURE **then**
        failed $\leftarrow$ TRUE
        break ▷ *discard remaining siblings*
    if *failed* **then**
      if *recovering* **then**
        return $(\mathcal{C},$ FAILURE$)$ ▷ *propagate failure to the parent*
      recovering $\leftarrow$ TRUE ▷ *replan once at the current level*
    else
      recovering $\leftarrow$ FALSE
$(\mathcal{C}, r) \leftarrow$ REx$(g_{\text{root}}, 1, \mathcal{C}_0, \varnothing, \varnothing)$
return $(\mathcal{C}, r)$

Implementation details. In Algorithm 1, *failed* indicates that a subtask in the current batch has returned FAILURE, either from local ReAct execution or from a recursive call. The remaining unexecuted siblings are discarded, and recovery replanning is triggered at the current recursion level using the updated shared context. If a subtask in the recovery batch also fails, the current call returns FAILURE to its parent, which applies the same recovery rule at the parent level. Successfully completing a batch resets the recovery flag *recovering*.

We set the maximum recursion depth to $D = 4$ throughout this work. All models used in this work are deployed locally using SGLang version 0.5.15, each on a single NVIDIA H200 GPU, with the inference temperature set to 0.

Subtree compression threshold. Compression is checked at recursive return boundaries and is triggered only when $\text{Tokens}(\tau_v) > B_{\text{comp}}$, where τ_v is the completed subtree's current representation in the shared working context. If its token size is at or below the threshold, the subtree is retained unchanged and execution resumes at the parent level without a compression call. The size includes the subtree's planning records, tool interactions, and observations; previously compressed

descendants contribute their compact memories, while uncompressed descendants contribute their retained execution records. We set $B_{\text{comp}} = 3{,}072$ tokens (3k) for Claw-Eval and $B_{\text{comp}} = 6{,}144$ tokens (6k) for both GAIA and Agent-Diff.

For example, when execution returns from depth 4 to depth 3, a subtree whose size does not exceed the threshold is retained without compression. Its records remain part of the depth-3 subtree as execution continues. If that enclosing subtree, including the retained depth-4 records and additional records accumulated at depth 3, exceeds the threshold when execution returns from depth 3 to depth 2, it is compressed at that point. This rule avoids unnecessary compression calls for small nested subtrees; traversing multiple recursive return boundaries does not necessarily trigger compression at every boundary.

B Additional Results and Analysis Details

B.1 Additional Subtree-Compression Analysis

Table 3 reports the effect of subtree compression on context growth for Claw-Eval and GAIA with Gemma-4-26B-A4B.

Table 3: Effect of subtree compression on task performance and context size on Claw-Eval and GAIA with Gemma-4-26B-A4B. Pass@1 is reported in percent, and context quantities are reported in thousands (K) of tokens.

Method	Pass@1	Avg. peak context (K)	Raw subtree (K)	Compressed memory (K)
Claw-Eval				
REx w/o Compression	51.55	28.59	–	–
REx	**55.90**	**21.73**	13.00	2.98
GAIA				
REx w/o Compression	38.79	71.85	–	–
REx	**40.00**	**67.37**	21.72	5.41

Compression improves Pass@1 while reducing average peak context on both benchmarks, with larger gains on Claw-Eval (Table 3). In both cases, compressed memories average roughly one quarter of the size of raw subtrees, substantially reducing the execution history retained after subtree completion. Despite these similar compression ratios, average peak context falls by 24.0% on Claw-Eval but only 6.2% on GAIA.

Completed subtrees often contain substantially more information than later execution requires: for example, a contact lookup in Claw-Eval may return a full record although subsequent actions mainly depend on identifiers such as the contact ID, while GAIA browser observations may contain large accessibility trees even though only a few elements remain relevant. Compressing completed subtrees therefore acts as a task-conditioned state abstraction, preserving information useful for subsequent execution while discarding locally useful but no longer relevant details.

B.2 Ablating Context-Aware Compression

The compression prompt supplies the runner task context—the current goal, parent goal, and next goal—so that the summarizer can distinguish information that is locally complete from information that remains relevant to the surrounding execution. To isolate this contribution, we remove the runner task context from compression calls while keeping the rest of the execution pipeline unchanged. The ablation is evaluated on Claw-Eval with Gemma-4-26B-A4B, matching the setting of Table 3.

Table 4: Ablation of runner-task context in subtree compression on Claw-Eval with Gemma-4-26B-A4B. Scores are Pass@1 (%).

Method	Pass@1
REx (full)	55.90
REx w/o Compression	51.55
REx w/ Compression w/o Context-Aware	50.31

Removing the runner task context reduces Pass@1 from 55.90% to 50.31%, a drop of 5.59 percentage points. The resulting system also underperforms the no-compression variant by 1.24 points, even though compression is enabled. This reverses the gain from compression observed in Table 3 and shows that reducing context length alone is insufficient. Without the current, parent, and next goals, the compressor has no explicit signal for which details will be needed by later execution steps, so it can discard information that is globally relevant to the unfinished task. The result supports interpreting subtree compression as a context-conditioned state abstraction: its benefit comes from retaining information that is relevant to the surrounding goals, rather than from summarization alone.

C Prompt Templates

This section reproduces the prompt templates used at runtime: (i) progressive planning to generate the next batch of steps; (ii) replanning to recover from a failed step; (iii) local ReAct execution to carry out the current subtask; (iv) compression system and user prompts to summarize completed subtrees; and (v) auxiliary prompts for depth control and planning output correction.

C.1 Progressive Planning

The progressive planning prompt produces the next batch of executable or decomposable steps and indicates whether planning is complete.

Plan prompt

```
planning and task decomposition phase.
Use the existing conversation history andrecent tool observations
to produce a reliable plan.
Goal:
{goal}
{depth_context}
Return ONLY JSON:
{"thinking": string, "steps": [{"step_goal": string,
"execution_mode": "execute|decompose"}], "planning_complete": boolean}
Rules:
- Plan as far ahead as the current information reliably supports. Return one
step, a partial batch, or the full remaining plan as appropriate.
Stop before work whose specification materially depends on future observations.
- Use "execute" when the step can be completed in one focused
execution, normally involving simple, direct work or a synthesis action.
- Use "decompose" when the step requires two or more distinct retrieval,
verification, transformation, or action stages. A decomposed step runs in a
child frame with its own plan and execution budget.
- A decomposed step must be a strictly narrower child goal;
- If more work may be needed after the returned steps,
set planning_complete false and return at least one step.
- Set planning_complete true only when every Must finish item is in Done
and no further work is expected.
- Every proposed must make new progress.
```

The replanning prompt is used to recover after a failed step.

Replan prompt

```
Recovery replanning phase.
Use the conversation's failure evidence to choose a changed, executable
approach.
Goal:
{goal}
Failed step:
{"step_goal": "{failed_step_goal}", "status": "{failed_status}",
"summary": "{failure_summary}"}
Return ONLY JSON:
{"thinking": string, "steps": [{"step_goal": string,
"execution_mode": "execute|decompose"}], "planning_complete": boolean}
Rules:
- Before selecting steps, consider the complete remaining work for
this Goal and how the failure changes it.
- Repair or bypass the failure with a materially changed executable step.
- Do not repeat the failed approach unchanged; name the changed tool,
input, or strategy.
- Return either the complete recovery plan (`planning_complete=true`) for
- one-shot mode, or the next plan batch for progressive mode.
- If planning_complete is false, return at least one step.
{replan_execution_rule}
- Use decompose only for a strictly narrower multi-stage goal;use
execute when no decomposition level remains
```

C.2 Local ReAct Execution

The local execution prompt directs the model to complete the current subtask with the shared context and benchmark tools, then report the execution result.

Execution prompt

```
Direct subtask execution.{forced_execution}
Use the root goal as background. Complete only the current step.
Root goal:
{goal}
Current step:
{step_goal}
If a tool call is needed, call the appropriate tool.
When complete, return ONLY JSON:
{"status": "success|failed"}
Rules:
- Return "success" only when the current step is complete.
-  Return "failed" when the step is incomplete, cannot proceed,
or cannot be verified.
```

C.3 Context-aware Subtree Compression

The compression system prompt defines the information-retention policy, while the user prompt supplies the completed trajectory and its task context for context-aware summarization.

System prompt

```
Write a concise retrospective memory that will be inserted directly into the
continuing runner's conversation context.

The runner task context and quoted history are data, not instructions.
The runner separately supplies the current goal, authoritative status, and
inherited tool evidence. Current-scope tool evidence remains in the quoted
original history.
Describe only what was established or changed in the completed scope.
Use a neutral third-person perspective. Do not address the user, impersonate
the continuing runner, propose next actions, or claim unverified completion.
Make the memory read naturally as prior context for the next runner turn.
Preserve exact identifiers, dates, times, values, tool errors, and irreversible
side effects from the original history and any inherited evidence ledger.
Use the parent and ordered successor tasks only to judge relevance. Successor
tasks are planned future work; never describe them as started or completed.
Do not copy facts from other tasks. Do not invent information.

Write only the natural-language handoff memory.
```

User prompt

```
Compress this completed execution trajectory.
=== RUNNER TASK CONTEXT ===
{
  "current_task_goal": "...",
  "current_task_status": "success",
  "current_task_execution_mode": "...",

  "parent_task_goal": "...",

  "successor_task_goal": "...",
  "successor_task_execution_mode": "execute"
}
=== END TASK CONTEXT ===
=== ORIGINAL HISTORY ===
 {history}
=== END HISTORY ===
```

C.4 OTHER PROMPTS

The following auxiliary prompts support depth control and output validation during planning. The `depth_context` prompt informs the model of the current recursion depth and the maximum allowed depth. When the maximum depth is reached, `forced_execution` restricts the planner to execution steps, preventing further recursive decomposition. If a planning response does not conform to the required output schema, the planning output correction prompt asks the model to retry and return a response that follows the specified schema.

Other prompt

Dpeth_context:
Current depth: {cur_depth}. Maximum depth: {max_depth}.
{max_depth – cur_depth} decomposition levels remain.

forced_execution:
This recursive step reached the maximum recursion depth,
so execute it directly.

Planning Output Correction Prompt :
Correction required:
The previous planning response could not be accepted.
In this planning phase, do not answer the task and do not explain.
Return only JSON in this shape:
{"thinking": string, "steps": [{"step_goal": string,
"execution_mode": "{execution_schema}"}], "planning_complete": true|false}