Learning Agent-Compatible Context Management for Long-Horizon Tasks

Lu Yi[1*], Runlin Lei[1*], Liuyi Yao[2], Yuexiang Xie[2],
Yuyang Li[3], Wenhao Zhang[2], Zhewei Wei[1†], Yaliang Li[2†], Jian-Yun Nie[4]

[1]Renmin University of China, [2]Tongyi Lab, Alibaba Group
[3]Beijing University of Posts and Telecommunications, [4]Université de Montréal

Abstract

LLM agents increasingly face long-horizon tasks such as web search and deep research in real-world applications, where accumulated context can cause long-context degradation and reasoning failures. Prior work mitigates this through context management with agent-side context control or fixed strategies such as summarization, which require training the agent itself for adaptation — making it impractical for closed-source agents and ignoring that different agents may require different strategies. We introduce Adaptive Context Management (AdaCoM), which trains an external LLM to manage the context of a frozen agent through flexible modification actions and end-to-end reinforcement learning. Across diverse agents on web search and deep research benchmarks, AdaCoM substantially improves performance by preserving task constraints and progress while pruning stale content. The learned strategies reveal a *Fidelity–Reliability Trade-off*: agents with higher vanilla ReAct performance benefit from higher-fidelity context preservation, whereas lower-performing agents require more aggressive compression to stay within a reliable reasoning regime. Transfer experiments show that AdaCoM generalizes most effectively across agents with similar capability (measured by vanilla ReAct performance), suggesting a practical path toward reusable context managers for agent systems.

1 Introduction

With advances in semantic understanding, tool use, and interactive decision making, general-purpose LLM agent applications such as OpenClaw and Hermes Agent have emerged (Steinberger and contributors, 2025; Nous Research, 2025). Such applications often involve long-horizon reasoning, where tasks such as answering multi-constraint search queries (Wei et al., 2025; Li et al., 2025) or producing deep research reports (Du et al., 2025; Wang et al., 2025) require many interdependent steps over a growing context. A central bottleneck for LLMs in such long-horizon tasks is long-context degradation. As tool results and intermediate reasoning accumulate, stale or irrelevant content can obscure salient evidence, amplify positional bias, and make subsequent decisions less reliable (Xiao et al., 2024; Liu et al., 2024; Shi et al., 2023).

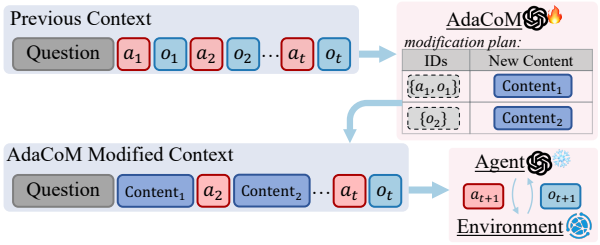

Figure 1: Overview of Adaptive Context Management (AdaCoM). Before each agent step, an external LLM manages the context presented to the frozen agent. Task feedback updates only the manager, enabling AdaCoM to discover agent-compatible context management strategies without training the underlying agent.

Prior work addresses this issue through *context management*, but typically places the burden of managing context on the agent itself. The agent is prompted to summarize its trajectory before acting (Zhou et al., 2025; Chen et al., 2025a), or to invoke context-management tools (Zhang et al., 2025). Because these mechanisms require the agent to follow the new context patterns or tool-use protocols that it may not have encountered during training, prior work often couples them with agent training to realize performance gains (Wu et al., 2025). This training-dependent design is poorly matched to deployment, where widely used agents are often closed-source, and training every user-selected model is infeasible. Moreover, predefined operations such as summarization impose a one-size-fits-

*Lu Yi and Runlin Lei are co-first authors. Work done during internship at Tongyi Lab, Alibaba Group.
†Zhewei Wei and Yaliang Li are corresponding authors.

all strategy despite substantial variation in agents' architectures, training data, and reasoning styles. This motivates a natural question: *can we discover the preferred context management strategy for each agent without training the agent itself?*

We introduce **Adaptive Context Management (AdaCoM)**, a framework based on two principles. First, *architectural decoupling*: context management is handled by an external manager, typically a smaller LLM, while the agent itself remains unchanged. This decoupling makes AdaCoM applicable even when the agent cannot be trained. Second, *operation-level flexibility*: instead of committing to predefined operations such as summarization, the manager can freely modify any part of the context, allowing it to discover agent-compatible management strategies. To learn such strategies, we train the manager with reinforcement learning while keeping the agent frozen. Figure 1 illustrates AdaCoM's process: before each agent step, the manager updates the running context, and the agent then acts on the resulting managed context.

Across diverse agents, AdaCoM substantially improves performance on web search and deep research tasks, reducing constraint forgetting, premature abandonment, and redundant exploration by retaining task-relevant context. Further analysis of AdaCoM's learned strategies reveals a consistent **Fidelity–Reliability Trade-off**. Using vanilla ReAct (Yao et al., 2022) performance as a measure of agent capability, we find that managers for stronger agents preserve more raw trajectory context to maintain fidelity, whereas managers for weaker agents compress more aggressively to keep reasoning reliable. This suggests that each agent has an effective context length beyond which additional raw context becomes harmful, and desirable context management must balance context fidelity against reasoning reliability. Further transfer experiments show that trained managers transfer most effectively between agents with comparable capability, suggesting that practitioners can reuse a trained manager across capability-similar agents without retraining the manager.

Overall, our main contributions are:

- We propose AdaCoM, an adaptive context management framework that decouples context management from the agent and learns agent-compatible strategies without retraining the agent itself.
- AdaCoM substantially improves diverse agents on web search and deep research tasks, mitigating constraint forgetting, premature abandonment, and redundant exploration.
- We identify the Fidelity–Reliability Trade-off and show that AdaCoM transfers most effectively across capability-similar agents, providing practical guidance for deploying context management in real-world agent applications.

2 Related Work

Context management for long-horizon tasks. Existing context management methods typically reduce long trajectories through predefined operations. Summarization methods such as IterResearch (Chen et al., 2025a) and MEM1 (Zhou et al., 2025) maintain a compact progress summary during task solving. Upon each observation, the agent conditions on the previous summary and the latest action-observation pair, then updates the summary and issues the next action in the same generation. ReSum (Wu et al., 2025) uses a separate summarization tool when the context approaches the length limit, compressing previous interaction history before the agent continues. Tool-based methods such as MemAct (Zhang et al., 2025) expose context management to the agent through a pruning tool, allowing it to summarize selected historical messages, remove the originals, and append the summary. These methods prescribe the management operation in advance and/or require the agent to learn when and how to manage its own context, often coupling context management with agent training. In contrast, AdaCoM trains an external manager to manage context while keeping the underlying agent fixed.

Long-term memory for LLM agents. A related but distinct line of work studies long-term memory for LLM agents (Hu et al., 2025). These methods focus on storing and retrieving information across sessions, tasks, or users, including factual knowledge and experiential memory. Representative systems include Mem0 (Chhikara et al., 2025), A-Mem (Xu et al., 2026), Memory-R1 (Yan et al., 2025), and G-Memory (Zhang et al., 2026). Our work is complementary to this direction. Instead of building persistent memory across conversations, we study *working-memory management* within a single long-horizon task, aiming to keep agents' active context useful for task completion.

3 Adaptive Context Management

This section presents AdaCoM, an adaptive context management framework for long-horizon agentic tasks. We first describe its workflow and flexible modification action space (Section 3.1), and then introduce the reinforcement learning procedure for training the context manager (Section 3.2).

3.1 Flexible Context Management Paradigm

Agent's vanilla workflow. Given a task query q, a ReAct-style agent $\mathcal{A}$ maintains an accumulated context $c_t^{\text{vanilla}} = (q, a_1, o_1, \ldots, a_t, o_t)$ at turn t, where a_i contains the agent's reasoning and tool invocation and o_i is the corresponding environment observation. The agent generates the next action by conditioning on the entire context: $a_{t+1} \sim \mathcal{A}(c_t^{\text{vanilla}})$. This append-only workflow is common in long-horizon tasks, but suffers from long-context degradation as context accumulates.

AdaCoM-augmented workflow. AdaCoM uses an external manager to modify the agent's running context, leaving the underlying agent $\mathcal{A}$ frozen. We denote the context manager's policy by $\pi_\theta(\cdot \mid p)$, which selects structured modification actions given a management prompt p.

Let $\tilde{c}_{t-1}$ denote the managed context after turn $t-1$, with $\tilde{c}_0 = (q)$. At turn t, the agent first acts on the managed context, $a_t \sim \mathcal{A}(\tilde{c}_{t-1})$, and the environment returns an observation o_t. We append the new action and observation to obtain the pre-management context $c_t = \text{Append}(\tilde{c}_{t-1}, a_t, o_t)$. The manager samples a structured modification action $m_t \sim \pi_\theta(\cdot \mid p_t)$, with $p_t = \mathcal{P}(c_t)$, where $\mathcal{P}$ constructs the prompt with the management instruction and output schema. Applying m_t to c_t produces the next managed context $\tilde{c}_t$. The next agent action is then generated from $\tilde{c}_t$.

AdaCoM's action space. A central design goal of AdaCoM is operation-level flexibility. Rather than committing to a predefined operation such as summarization, we formulate context management as general-purpose modification over a message sequence. At each turn, the pre-management context c_t is represented as an ordered list of messages with unique message IDs.

The manager action m_t is a structured list of modification operations, expressed in JSON as

$$m_t = \left[\delta_t^{(1)}, \delta_t^{(2)}, \ldots, \delta_t^{(n_t)}\right],$$

where each operation $\delta_t^{(j)}$ selects one or more messages and specifies how they should be rewritten, deleted, or merged; it can also specify the role of the resulting message. Each operation contains four fields: (1) $\text{ids}^{(j)}$, the IDs of the targeted messages; (2) $\text{role}^{(j)} \in \{\text{SYSTEM}, \text{USER}, \text{ASSISTANT}\}$, the role of the resulting message; (3) $\text{justification}^{(j)}$, a short rationale that elicits the manager's reasoning about the modification, encouraging higher-quality edits; it is removed before the managed context is shown to the agent; and (4) $\text{new\_content}^{(j)}$, the resulting content, where empty content deletes the targeted messages and non-empty content rewrites or merges them. Messages not targeted by any operation are copied unchanged, and an empty action list leaves the context unchanged. The full manager prompt is provided in Appendix H.

This action space supports diverse context-management operations beyond a fixed strategy. AdaCoM can remove stale information, condense verbose evidence, merge related messages, or leave the context unchanged when fidelity is important.

3.2 Training the Context Manager

Problem formulation. We formulate manager learning as a Markov decision process (MDP) induced by the frozen agent and the environment. At each manager step t, the current pre-management context c_t is formatted into a prompt $p_t = \mathcal{P}(c_t)$. The manager policy emits a structured modification action $m_t \sim \pi_\theta(\cdot \mid p_t)$, which is then applied to c_t to update the managed context. A rollout induces a sequence of manager decision points $\tau = ((p_1, m_1), \ldots, (p_T, m_T))$. The rollout terminates when the frozen agent emits a final answer or exceeds the maximum number of interaction steps. The trajectory-level outcome reward is the task reward of the agent's final answer.

Training overview. We first use supervised fine-tuning (SFT) to initialize the manager with the required output format, and then optimize it with Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For each query, we sample multiple rollouts from the current manager policy and evaluate the final answer produced by the frozen agent. For tasks with deterministic answers, an LLM judge compares the final answer against the ground truth and produces a binary score; for open-ended tasks, it scores the answer according to a task-specific rubric.

Process reward design. We additionally introduce process rewards to improve credit assignment in long-horizon interactions. We use rule-based pro-

cess rewards computed directly from trajectories, without invoking an additional LLM judge. If the managed context exceeds the given context length limit, we apply a *token penalty* to the manager step that produced the over-length context. When the agent issues two consecutive tool calls with the same tool name and parameters, we apply a *redundant-action penalty* to the intervening manager action, treating repetition as a proxy for insufficient preservation of useful information. We also introduce *format penalties* for structurally invalid manager outputs, including JSON parsing failures, nonexistent message IDs, and missing fields. Finally, when a benchmark provides task-specific intermediate signals, we incorporate them as positive process rewards, as detailed in Section 4.

Two-level advantage estimation. In our setting, outcome rewards and process rewards are defined at different granularities and may have different numerical scales, so directly adding their raw values can make optimization sensitive to reward scaling. Motivated by prior work on reward normalization in RL-based LLM training (Ding et al., 2025; Liu et al., 2026), we first convert them into separately normalized advantages before combining them to improve training stability. Specifically, for each query q, we sample a group of G rollouts $\{\tau_i\}_{i=1}^{G}$ from the old manager policy $\pi_{\theta_{\text{old}}}$. To index rollouts within the group, we add an index i to the notation. Thus, $p_{i,t}$ and $m_{i,t}$ denote the management prompt and manager action at step t of rollout i, respectively. The i-th rollout is $\tau_i = ((p_{i,1}, m_{i,1}), \ldots, (p_{i,T_i}, m_{i,T_i}))$, where T_i is the number of manager steps in rollout i, and each action satisfies $m_{i,t} \sim \pi_{\theta_{\text{old}}}(\cdot \mid p_{i,t})$. Let R_i denote the terminal outcome reward of rollout τ_i, and let $Q_{i,t}$ denote the aggregate process reward assigned to manager step t in rollout i, obtained by summing all applicable process-reward components described above.

We compute advantages at two granularities. The *task-level* advantage assigns the same normalized outcome reward to each step in rollout τ_i:

$$A_i^R = \frac{R_i - \mu_R}{\sigma_R + \varepsilon},$$

where μ_R and σ_R are the mean and standard deviation of $\{R_j\}_{j=1}^{G}$. The *step-level* advantage normalizes process rewards over all steps in the group:

$$A_{i,t}^Q = \frac{Q_{i,t} - \mu_Q}{\sigma_Q + \varepsilon},$$

where μ_Q and σ_Q are computed over $\{Q_{i,t} : i = 1, \ldots, G,\ t = 1, \ldots, T_i\}$.

We combine the two levels as $A_{i,t} = A_i^R + \alpha A_{i,t}^Q$, where α controls the balance between outcome and process supervision. We then renormalize $A_{i,t}$ over all steps in the group: $\hat{A}_{i,t} = (A_{i,t} - \mu_A)/(\sigma_A + \varepsilon)$, where μ_A and σ_A are computed over $\{A_{i,t} : i = 1, \ldots, G,\ t = 1, \ldots, T_i\}$. When $\sigma_R = 0$, all rollouts in the group receive the same outcome reward. The task-level term then vanishes, and the step-level term provides the only learning signal within such same-outcome groups.

Policy optimization. We optimize π_θ with a PPO-style clipped surrogate over manager-emitted tokens. Let u index the tokens in $m_{i,t}$, and let $r_{i,t,u}(\theta)$ be the token-level importance ratio between π_θ and $\pi_{\theta_{\text{old}}}$. The objective is

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{Z}\sum_{i,t,u}\min\Big(r_{i,t,u}(\theta)\hat{A}_{i,t}, \bar{r}_{i,t,u}(\theta)\hat{A}_{i,t}\Big)\right],$$

where $\bar{r}_{i,t,u}(\theta) = \text{clip}(r_{i,t,u}(\theta), 1-\epsilon, 1+\epsilon)$, the expectation is over $q \sim \mathcal{D}$ and rollouts sampled from $\pi_{\theta_{\text{old}}}$, and $Z = \sum_{i=1}^{G}\sum_{t=1}^{T_i}|m_{i,t}|$ is the total number of manager-emitted tokens.

4 Experiments and Analysis

4.1 Experimental Setup

Benchmarks. We evaluate on two representative long-horizon tasks: web search and deep research. For web search, we use BrowseComp-Plus (Chen et al., 2025b), a BrowseComp-derived benchmark (Wei et al., 2025) with a verified corpus for controlled evaluation. The agent is provided with two tools: `search(query, top_k)`, which retrieves the top-k relevant documents, and `get_document(doc_id)`, which returns the corresponding document content. We create a disjoint training/test split of 680/150 instances and use Qwen3-Embed-8B as the retriever, following Sun et al. (2025). The maximum number of iterations is set to 35, and GPT-4o (2024-11-20) is used as a judge to produce a binary correctness score.

For deep research, we follow the data construction procedure of MCP-Bench (Wang et al., 2025) and build a Wikipedia-MCP-based benchmark, denoted as MCP-Bench-Wiki. The agent can use nine Wikipedia MCP tools, covering search and content retrieval. Following the original benchmark protocol, we use Claude Opus 4.6 as the judge. While

Agent / Setting	ReAct	SumAgent	MemAct	SumCoM	AdaCoM w/o train.	AdaCoM
CM Model	–	agent itself	agent itself	Qwen3-4B-Inst.	Qwen3-4B-Inst.	Qwen3-4B-Inst.
CM Trained	–	×	×	✓	×	✓
Qwen3-Max	27.78	26.67 ↓4.0%	**37.33** ↑34.4%	32.22 ↑16.0%	21.11 ↓24.0%	<u>36.67</u> ↑32.0%
Kimi-K2-Instruct	18.56	30.44 ↑64.0%	16.89 ↓9.0%	<u>33.78</u> ↑82.0%	28.02 ↑51.0%	**36.20** ↑95.0%
GLM-4.5-Air	<u>32.56</u>	11.56 ↓64.5%	32.00 ↓1.7%	26.44 ↓18.8%	21.33 ↓34.5%	**35.33** ↑8.5%
DeepSeek-V3	17.78	19.56 ↑10.0%	16.67 ↓6.2%	<u>25.11</u> ↑41.2%	17.57 ↓1.2%	**26.19** ↑47.3%
Avg.	24.17	22.06 ↓8.7%	25.72 ↑6.4%	<u>29.39</u> ↑21.6%	22.01 ↓8.9%	**33.60** ↑39.0%

Table 1: **Mean@3** (%) on BrowseComp-Plus across four agents. Each non-ReAct cell reports the absolute score and the relative change against the same-agent *ReAct* baseline. Bold and underline indicate the best and second-best results in each row. The average is computed over agents. Pass@3 results are reported in Appendix C.

the original benchmark evaluates reports along six dimensions, we focus on three dimensions most relevant to context management: task fulfillment (TF), information grounding (IG), and parallelism and efficiency (PE). The overall score is computed as $0.4\,\mathrm{TF} + 0.4\,\mathrm{IG} + 0.2\,\mathrm{PE}$. Scores on all evaluation dimensions are provided in Appendix C.

For BrowseComp-Plus, we report **mean@3**, the average accuracy over three independent runs, and **pass@3**, whether at least one run answers correctly. For MCP-Bench-Wiki, we report **mean@3**, the average rubric-based score over three runs.

Implementation. AdaCoM is initialized from Qwen3-4B-Instruct, briefly warmed up with SFT to learn the edit format, and then trained with the RL procedure in Section 3.2. For the SFT warm-up, we use GPT-5 (2025-08-07) and Claude Opus 4.6 to generate edit trajectories for BrowseComp-Plus and MCP-Bench-Wiki, respectively, with Qwen3-Max serving as the frozen agent on the corresponding training set. For RL training, we build on Trinity-RFT (Pan et al., 2025) with group size $G = 8$ for both benchmarks. We set the manager's own input context-window length to 32,768 tokens and its maximum output length to 4,096 tokens. For BrowseComp-Plus, we additionally use positive process rewards as the benchmark provides gold and key document IDs. When a search result contains a gold or key document, we assign a *gold-doc-found* or *key-doc-found* reward to the preceding manager step, crediting the context management that led the agent to issue the successful search. Additional experimental settings and process reward parameters are deferred to Appendix A.

Baselines. We compare AdaCoM against five baselines. (i) *ReAct*: the vanilla ReAct loop without context management. (ii) *SumAgent* (Summarization Agent): a summarization paradigm used in MEM1 (Zhou et al., 2025) and IterResearch (Chen et al., 2025a), where the agent conditions on the original question, previous summary, and latest tool interaction to update the summary and issue the next action. (iii) *MemAct* (Zhang et al., 2025): the agent can invoke a prune tool to summarize selected historical messages, remove them, and append the generated summary. (iv) *SumCoM* (Summarization Context Manager): a trained external Qwen3-4B-Instruct manager summarizes the context after each agent step while the agent remains frozen. This ablates our flexible management actions by replacing them with a fixed summarization operation; we adapt prompts from ReSum (Wu et al., 2025) and use the same training procedure as AdaCoM. (v) *AdaCoM w/o training*: the original Qwen3-4B-Instruct backbone used as the context manager with the same modification action space but without SFT or RL.

Agent naming. For readability, we use shortened names for frozen agents when unambiguous: Qwen for Qwen3-Max, DeepSeek for DeepSeek-V3, Kimi for Kimi-K2-Instruct, GLM for GLM-4.5-Air, Gemini for Gemini-2.5-Flash, and Seed for Seed-1.6-Flash.

4.2 Main Results

Table 1 reports the results on BrowseComp-Plus. For AdaCoM, we train a separate context manager for each target agent, including Qwen, DeepSeek, Kimi, and GLM. Overall, AdaCoM consistently improves over the vanilla ReAct baseline across all four agents, achieving an average relative gain of 39.0% and a 95.0% relative gain on Kimi. These results demonstrate the effectiveness of learning an external context manager while keeping the underlying agent unchanged.

We further compare against several alternatives

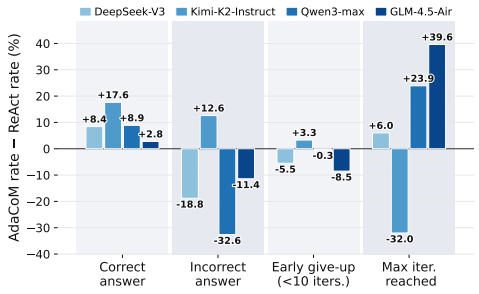

Figure 2: Per-agent trajectory outcome distribution shifts (%) on BrowseComp-Plus, computed as AdaCoM minus ReAct.

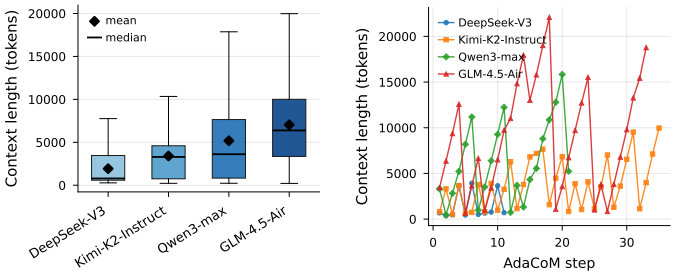

Figure 3: Post-AdaCoM agent context length across four agents, each paired with its self-trained AdaCoM: **(a)** per-modification distribution; **(b)** per-step trace on task 199.

to understand where the gains come from. First, AdaCoM w/o training performs inconsistently and is worse than ReAct on average, indicating that the manager must be trained to induce effective context-management strategies. Second, SumCoM improves some agents but degrades GLM, suggesting that a fixed summarization operation is not sufficiently compatible with varied agent backbones. Third, self-management baselines such as MemAct and SumAgent also exhibit unstable behavior and do not outperform ReAct across all agents. This supports the hypothesis that asking the agent to manage its own context may require agent-side training, since the resulting summaries may not be reliably usable by the original agent (Wu et al., 2025). Notably, MemAct marginally outperforms AdaCoM on Qwen. We attribute this to Qwen's strong intrinsic agentic capability, which enables it to invoke the pruning tool effectively even without additional training. However, this advantage does not generalize to other agents, whereas AdaCoM provides consistent gains across agents.

Table 2 reports the results on MCP-Bench-Wiki, which serves as a supplementary evaluation beyond web search. MCP-Bench-Wiki tests AdaCoM in deep research tasks requiring multi-step MCP tool invocation, evidence gathering, and long-form report synthesis. Due to the higher cost of deep research rollouts and evaluation, we evaluate two representative agents, Kimi and DeepSeek. The results show AdaCoM improves both agents over ReAct, achieving 9.0% and 22.3% relative gains on Kimi and DeepSeek, respectively, indicating that the advantages of learned external context management can be extended to tool-use-intensive tasks like deep research.

Agent / Setting	ReAct	AdaCoM w/o train.	AdaCoM
Kimi-K2-Instruct	55.05	44.35 ↓19.4%	**60.01** ↑9.0%
DeepSeek-V3	47.51	47.82 ↑0.7%	**58.09** ↑22.3%
Avg.	51.28	46.09 ↓10.1%	**59.05** ↑15.2%

Table 2: **Mean@3** on MCP-Bench-Wiki across two agents. Each non-ReAct cell reports the absolute score and the relative change against the *ReAct* baseline.

4.3 Analysis: How AdaCoM Helps Agents

Trajectory outcome taxonomy. To understand how learned context management helps agents solve long-horizon tasks, we analyze how AdaCoM changes outcome distributions on the BrowseComp-Plus test set. We categorize trajectories into four interpretable outcomes: (i) *Correct answer*, where the agent terminates with a correct final answer; (ii) *Incorrect answer*, where the agent terminates with a definitive but wrong final answer; (iii) *Early give-up*, where the agent stops within ten iterations without providing a definitive answer; and (iv) *Max iter. reached*, where the agent reaches the iteration limit without a valid answer. We omit non-definitive stops after more than ten iterations to keep the analysis focused.

Outcome shifts and underlying causes. Figure 2 reports the change in outcome ratios from ReAct to AdaCoM, computed as the ratio under AdaCoM minus the ratio under ReAct. Overall, AdaCoM increases the proportion of correct answers across all backbones compared to ReAct, while also shifting the composition of failure modes. For Qwen, DeepSeek, and GLM, AdaCoM reduces *Incorrect answer* and *Early give-up*, suggesting fewer wrong commitments and early give-ups. Paired trajectory inspection shows that these reductions are mainly tied to mitigating *constraint forgetting* and *premature abandonment*. In failed ReAct trajectories, agents often commit to candidates satisfying only

part of the task requirements or give up after several ineffective searches despite remaining solvable directions. In successful AdaCoM trajectories, the manager often maintains a compact state message immediately after the user task message, listing task requirements, unresolved constraints, useful evidence, current leads, rejected candidates, and ineffective queries. These elements help the agent verify candidates against all requirements and keep searching rather than committing to wrong answers or abandoning the task.

For Kimi, AdaCoM sharply reduces *Max iter. reached*, indicating fewer long, unproductive trajectories. This reduction is mainly tied to mitigating *redundant exploration*: in failed ReAct trajectories, Kimi often repeatedly issues identical or near-identical tool calls. On the BrowseComp-Plus test set, an average of 42.6% of Kimi's tool-use steps per task are repetitive. AdaCoM reduces such loops by recording prior attempts and pruning redundant or unhelpful tool results from the active context, thereby lowering the reasoning burden and making repeated exploration less likely.

Importantly, we do not explicitly prompt AdaCoM on what to record or how to organize the context. AdaCoM learns to record and update task requirements, search progress, and prior attempts as needed, helping agents maintain an adaptive task state rather than merely shortening the context.

4.4 Analysis: Specific Strategies across Agents

We further compare the learned context management behavior across agents on the BrowseComp-Plus test set. Figure 3 shows that the learned strategies differ systematically across agents. The mean post-management context length forms a clear trend: about 1.9K tokens for DeepSeek, 3.4K for Kimi, 5.2K for Qwen, and 7.0K for GLM. Notably, this trend aligns with the agents' vanilla ReAct performance on BrowseComp-Plus, ordered from lower to higher as DeepSeek, Kimi, Qwen, and GLM. In other words, managers for stronger ReAct agents preserve more context, while managers for weaker agents compress more aggressively. Per-task context-length curves further reveal two characteristic strategies. For GLM and Qwen, AdaCoM follows a *tiered management* strategy: it lets the context grow by retaining raw tool results and performs occasional batched compression. For DeepSeek and Kimi, AdaCoM follows an *eager distillation* strategy: it compresses nearly every round to keep the context short, with Kimi preserving slightly more raw context than DeepSeek.

Fidelity–Reliability Trade-off. These empirical strategies reveal a Fidelity–Reliability Trade-off. Using vanilla ReAct performance as a proxy for agent capability, stronger agents such as GLM and Qwen can exploit longer raw contexts, so AdaCoM intervenes later and more lightly; weaker agents such as DeepSeek and Kimi benefit from shorter, distilled contexts, so AdaCoM compresses more frequently. This reflects a balance between fidelity and reliability: preserving raw trajectories retains fine-grained evidence, but beyond an agent's effective context length, additional raw content can harm reasoning. Conversely, aggressive compression improves reliability but may discard details. Thus, effective context management is not a fixed operation such as summarization, but an agent-compatible strategy that preserves as much fidelity as possible while keeping reasoning reliable.

Relation to fixed summarization. This also explains the behavior of SumCoM in Table 1. Since SumCoM always applies a fixed summarization pattern, it resembles the eager distillation strategy and brings larger gains for weaker ReAct agents such as DeepSeek and Kimi. However, it is less suitable for agents such as GLM and Qwen that can benefit from preserving more raw context, which explains why SumCoM lags behind AdaCoM and even degrades GLM.

We provide a detailed case study showing how AdaCoM maintains task constraints and search progress while adopting agent-specific compression strategies in Appendix D.

5 Cross-Agent Transferability

In practical agent applications, users may choose various agents for tasks of varying difficulty, making it costly to train a separate context manager for every agent. We therefore investigate whether a trained AdaCoM can be reused across agents without retraining on each target agent.

Experimental setup and overall results. On BrowseComp-Plus, we evaluate the four trained managers from Section 4, each trained with one source agent among DeepSeek, Kimi, Qwen, and GLM, on eight target agents. The target set includes the four source agents and four held-out agents that never participate in manager training: GPT-OSS-20B, Gemini, Seed, and GPT-4o-mini. Figure 4 reports the accuracy of each target agent under ReAct and ReAct equipped with each of

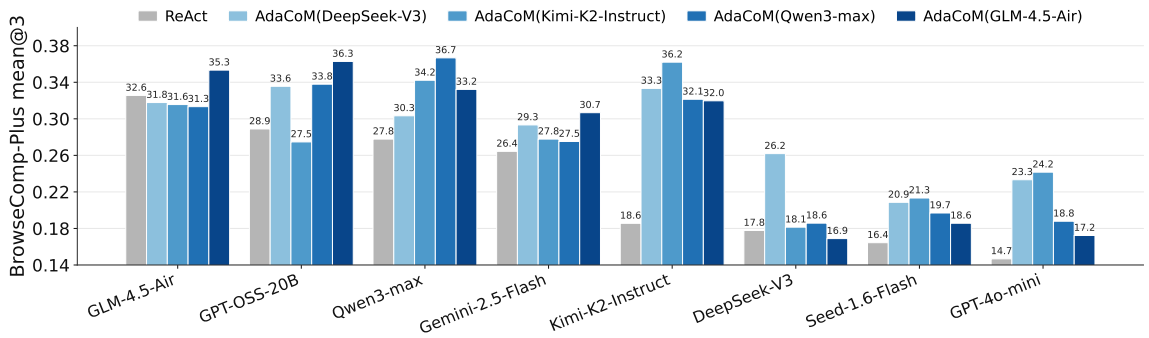

Figure 4: Cross-agent transfer of AdaCoM on BrowseComp-Plus. Each group on the x-axis is a target agent; bars within a group show BrowseComp-Plus **mean@3** under ReAct and various trained AdaCoMs.

the four trained managers. Trained managers improve over ReAct in most configurations: 23 of 28 cross-agent pairs, plus all 4 self-trained pairs, achieve positive gains, with an average relative improvement of 25.0% across all 32 pairs and 22.1% across the 28 cross-agent pairs alone. The largest cross-agent gain reaches 79.6% on Kimi under the DeepSeek-trained manager, comparable to its 95.0% self-trained gain and showing that non-self managers can approach self-trained performance. These results show that trained context managers are not narrowly tied to their source agents and can transfer broadly to unseen target agents.

We also observe similar cross-agent transfer improvements on MCP-Bench-Wiki, with detailed results deferred to Appendix C, suggesting that transfer is not limited to web search but also extends to tool-use-intensive deep research tasks.

Capability-based transfer patterns. We next ask what factors predict successful transfer on BrowseComp-Plus. A clear pattern is that effective transfer is closely related to source-target capability proximity, measured by ReAct baseline performance. Higher-baseline target agents, including GLM, Qwen, Gemini, and GPT-OSS-20B, generally benefit more from managers trained on higher-baseline sources such as GLM or Qwen, whereas lower-baseline targets such as Kimi, DeepSeek, Seed, and GPT-4o-mini tend to prefer managers trained on lower-baseline sources such as DeepSeek or Kimi. This pattern is consistent with the Fidelity–Reliability Trade-off in Section 4.4: agents with higher baseline performance can benefit from managers that preserve more raw context, while agents with lower performance tend to benefit from more aggressive compression and distillation.

Exceptions and agent-specific factors. Capability proximity is a strong but incomplete predictor of transfer. In several exceptional cases, managers trained on agents with similar capability do not yield the best transfer performance. For example, DeepSeek gains little from AdaCoM (Kimi): although their ReAct baselines are similar, DeepSeek prefers concise working memories with key findings and document IDs, whereas AdaCoM (Kimi) tends to record detailed search histories. Similarly, GLM gains little from managers trained on other agents because these managers compress too aggressively. For Gemini, AdaCoM (Qwen) underperforms AdaCoM (Kimi) because Gemini does not use AdaCoM (Qwen)'s report-style working memory effectively. Overall, capability proximity is a useful first-order rule, while compatibility between the manager's modification style and the target agent's reasoning style further affects transfer quality. Detailed trajectory analyses are provided in Appendix C.3.

Practical implications. These results suggest a practical deployment strategy for context management with closed-source agents. Instead of training a separate manager for every agent, practitioners can train a small set of managers for representative agents and reuse them across capability-similar targets. Capability proximity provides a useful first-order heuristic, while agent-specific style compatibility remains important when selecting the best manager within a capability group.

6 Conclusion

We introduced AdaCoM, an adaptive context management framework that trains an external manager to manage an agent's running context while keeping the underlying agent frozen. With a flexible modification action space and RL training, AdaCoM learns agent-compatible context management strategies. Experiments show that AdaCoM consistently improves diverse agents and can transfer to unseen agents with similar baseline capability.

Our analysis further reveals a Fidelity–Reliability Trade-off: effective context management should preserve as much task-relevant information as possible while keeping the agent within a context regime where its reasoning remains reliable.

Limitations

Evaluation scope. Our experiments cover two long-horizon task families, web search (BrowseComp-Plus) and deep research (MCP-Bench-Wiki), both centered on knowledge-intensive search. We do not evaluate AdaCoM on other long-horizon settings such as code agents, embodied agents, or longform writing, where the structure of useful context may differ. In addition, due to the higher cost of deep-research rollouts and evaluation, MCP-Bench-Wiki uses only two agents, so conclusions on that benchmark are based on a small agent pool.

Manager capacity. Our manager is initialized from a relatively small Qwen3-4B-Instruct model. This makes AdaCoM practical and inexpensive, but a 4B manager may have limited capacity to losslessly preserve or compress evidence for very strong agents whose reasoning relies on high-fidelity context. We therefore do not extensively evaluate newer and stronger closed-source agents, as such a capacity mismatch could underestimate the framework's potential. For example, the relatively small gains on GLM may partially reflect this mismatch in addition to its already-high baseline. Due to computational constraints, we do not train larger context managers in this work. For stronger target agents, future work can pair them with more capable managers, or draw on the strategies discovered by AdaCoM, such as tiered management for high-capability agents and eager distillation for lower-capability agents, to inform the design of context-management capabilities in future agent training.

Inference overhead. AdaCoM introduces an additional manager inference at every agent step, increasing per-rollout token cost and wall-clock latency compared with vanilla ReAct. Our goal in this work is to discover agent-compatible management patterns rather than to optimize runtime efficiency. Notably, our findings on stronger agents suggest a natural mitigation: AdaCoM learns a tiered management strategy that intervenes only occasionally (Section 4.4), indicating that in deployment the manager need not be invoked at every step. Triggering it only every few rounds or when the context approaches a length threshold could substantially reduce the overhead while preserving the benefits.

KV cache efficiency. AdaCoM modifies the agent's running context between steps, which can reduce the effectiveness of KV cache reuse (Kwon et al., 2023). This limitation is shared by many context-management paradigms, including summarization-based methods, because changing earlier context tokens invalidates cached states. In this work, we focus on discovering agent-compatible context-management strategies and therefore do not optimize for KV cache preservation. However, our findings suggest a possible direction for cache-aware context-management: for stronger agents that can reliably process longer contexts, the manager can preserve raw context for more steps and compress only near the agent's effective context length. Such tiered management may reduce the frequency of cache-breaking operations while still preventing long-context degradation.

References

Guoxin Chen, Zile Qiao, Xuanzhong Chen, Donglei Yu, Haotian Xu, Wayne Xin Zhao, Ruihua Song, Wenbiao Yin, Huifeng Yin, Liwen Zhang, and 1 others. 2025a. Iterresearch: Rethinking long-horizon agents with interaction scaling. *arXiv preprint arXiv:2511.07327*.

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

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*.

Yifeng Ding, Hung Le, Songyang Han, Kangrui Ruan, Zhenghui Jin, Varun Kumar, Zijian Wang, and Anoop Deoras. 2025. Empowering multi-turn tool-integrated reasoning with group turn policy optimization. *arXiv preprint arXiv:2511.14846*.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. 2025. Deepresearch bench: A comprehensive benchmark for deep research agents. *arXiv preprint arXiv:2506.11763*.

Dawei Gao, Zitao Li, Yuexiang Xie, Weirui Kuang, Liuyi Yao, Bingchen Qian, Zhijian Ma, Yue Cui, Haohao Luo, Shen Li, and 1 others. 2025. Agentscope 1.0: A developer-centric framework for building agentic applications. *arXiv preprint arXiv:2508.16279*.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, and 1 others. 2025. Memory in the age of ai agents. *arXiv preprint arXiv:2512.13564*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, and 1 others. 2025. Websailor: Navigating super-human reasoning for web agent. *arXiv preprint arXiv:2507.02592*.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Trans. Assoc. Comput. Linguistics*, 12:157–173.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and 1 others. 2026. Gdpo: Group reward-decoupled normalization policy optimization for multi-reward rl optimization. *arXiv preprint arXiv:2601.05242*.

Nous Research. 2025. Hermes agent: The agent that grows with you. https://github.com/NousResearch/hermes-agent.

Xuchen Pan, Yanxi Chen, Yushuo Chen, Yuchang Sun, Daoyuan Chen, Wenhao Zhang, Yuexiang Xie, Yilun Huang, Yilei Zhang, Dawei Gao, and 1 others. 2025. Trinity-rft: A general-purpose and unified framework for reinforcement fine-tuning of large language models. *arXiv preprint arXiv:2505.17826*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR.

Peter Steinberger and OpenClaw contributors. 2025. Openclaw — personal ai assistant. https://github.com/openclaw/openclaw.

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

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and 1 others. 2025. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers. *arXiv preprint arXiv:2508.20453*.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, and 1 others. 2025. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pages 21875–21895.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2026. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z Pan, and 1 others. 2025. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. *arXiv preprint arXiv:2508.19828*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Guibin Zhang, Muxin Fu, Kun Wang, Frank Wan, Miao Yu, and Shuicheng Yan. 2026. G-memory: Tracing hierarchical memory for multi-agent systems. *Advances in Neural Information Processing Systems*, 38:12988–13018.

Yuxiang Zhang, Jiangming Shu, Ye Ma, Xueyuan Lin, Shangxi Wu, and Jitao Sang. 2025. Memory as action: Autonomous context curation for long-horizon agentic tasks. *arXiv preprint arXiv:2510.12635*.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. 2025. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. *arXiv preprint arXiv:2506.15841*.

A Additional Experimental Settings

A.1 Available Tools

BrowseComp-Plus. The BrowseComp-Plus agent has access to two tools:

- `search(query)`: queries the corpus and returns a ranked list of document snippets, each with a unique document ID and a relevance score.
- `get_document(doc_id)`: fetches the full-text content of a document by its ID.

MCPWiki. The MCP-Bench-Wiki agent connects to a Wikipedia MCP server. Each tool name is prefixed with the server identifier (e.g., `Wikipedia__search_wikipedia`). The nine available tools are:

- `search_wikipedia(query)`: keyword search returning a list of matching article titles and snippets.
- `get_article(title)`: full-text retrieval of a Wikipedia article.
- `get_summary(title)`: retrieves a concise summary of an article.
- `summarize_article_for_query(title, query)`: returns a query-focused summary of an article.
- `summarize_article_section(title, section)`: summarizes a specific section of an article.
- `extract_key_facts(title)`: extracts key facts from an article, optionally focused on a topic.
- `get_related_topics(title)`: returns topics related to an article via its links and categories.
- `get_sections(title)`: lists the section structure of an article.
- `get_links(title)`: retrieves all links contained within an article.

Shared `finish` tool. Both benchmarks share a `finish(answer, explanation)` tool that the agent calls to terminate the episode and submit its final answer.

On MCP-Bench-Wiki, whenever `get_article` results are appended to the context, AdaCoM applies an `extract` operation before the agent's next step. This operation keeps task-relevant sentences and discards irrelevant content, preventing unbounded context growth from long Wikipedia pages. To ensure that the manager learns this behavior, we include manager invocations involving `extract` in both the SFT warm-up data and RL training trajectories. The extraction prompt is shown in Table 12.

A.2 Detailed Parameter Settings

Process rewards. We use the same rule-based process reward parameters across agents and benchmarks unless otherwise specified. The *token penalty* is set to -0.8, the *redundant-action penalty* to -0.4, and the *format penalty* to -0.5. For BrowseComp-Plus, we additionally use positive document-retrieval rewards: *gold-doc-found* is set to 0.6, and *key-doc-found* is set to 0.3. Positive and negative process rewards are accumulated when multiple rules are triggered at the same step.

Hyperparameters. We set the process reward weight α to 0.1, the rollout batch size to 32 trajectories, and the training batch size to 768 manager-step samples. The learning rate is 5×10^{-6}, the KL coefficient is 0.006, and the PPO clipping range is 0.2. We do not use an entropy bonus. During rollouts, the manager sampling temperature is set to 1.0; we found that lower temperatures more easily lead to degenerate repetitive outputs. The maximum number of rollout iterations is 35. We train for at most 40 epochs and apply early stopping if the evaluation score does not improve for 20 rollout steps. We train the SFT warm-up for five epochs on BrowseComp-Plus and two epochs on MCP-Bench-Wiki.

Context length. Under ReAct, each agent uses its own default maximum context length as provided by the model API: 256K for Qwen3-Max, 128K for GLM-4.5-Air, 256K for Kimi-K2-Instruct, and 128K for DeepSeek-V3. Under AdaCoM, since the context manager is configured with a 32K-token context budget, the agent's effective input is also capped at 32K tokens regardless of the underlying model's native context window.

	ReAct	SumAgent	MemAct	SumCoM	AdaCoM w/o train.	AdaCoM
CM Model	–	agent itself	agent itself	Qwen3-4B-Inst.	Qwen3-4B-Inst.	Qwen3-4B-Inst.
CM Trained	–	×	×	✓	×	✓
Qwen3-Max	38.00	38.67 ↑1.8%	50.67 ↑33.3%	42.67 ↑12.3%	30.67 ↓19.3%	**52.00** ↑36.8%
Kimi-K2-Instruct	29.33	44.67 ↑52.3%	30.00 ↑2.3%	45.33 ↑54.6%	42.75 ↑45.8%	**53.02** ↑80.8%
GLM-4.5-Air	44.67	20.00 ↓55.2%	46.00 ↑3.0%	36.00 ↓19.4%	32.67 ↓26.9%	**50.67** ↑13.4%
DeepSeek-V3	26.67	28.67 ↑7.5%	22.67 ↓15.0%	37.33 ↑40.0%	27.70 ↑3.9%	**39.97** ↑49.9%
Avg.	34.67	33.00 ↓4.8%	37.33 ↑7.7%	40.33 ↑16.3%	33.45 ↓3.5%	**48.92** ↑41.1%

Table 3: **Pass@3** (%) on BrowseComp-Plus across four agents. Each non-ReAct cell shows the absolute score and the relative change versus the same-agent *ReAct* baseline (↑ = improvement, ↓ = decline).

B Construction of the MCP-Bench-Wiki Dataset

In the MCP-Bench-Wiki task, we follow the original benchmark in creating tasks with a more fine-grained filtering process (Wang et al., 2025). Construction proceeds in two stages: an *upstream* stage that creates a clean pool of tasks, and a *downstream* stage that turns those tasks into the three datasets used for training and evaluation. The final data contain 4,740 SFT samples, 1,000 RL training tasks, and a held-out test set of 150 tasks, as summarized in Table 4.

B.1 Upstream: task synthesis pool

The upstream pipeline produces, validates, and de-duplicates a pool of Wikipedia tasks. Each task is a long, structured natural-language request that requires multiple coordinated calls to a Wikipedia MCP server. Every task carries three fields used downstream: `task_id`, `task_description`, and `dependency_analysis`.

The pipeline has four steps:

- **Multi-model generation.** Tasks are drafted by five frontier LLMs in a round-robin schedule, including Claude Opus 4.6, Gemini 3.1 Pro, GLM-5, MiniMax M2.7, and GPT-5 (2025-08-07). Each task-generation prompt requires the model to reason about cross-tool dependencies, embed concrete arguments, and avoid topics already covered by an updating blacklist for diversity. The blacklist is refreshed every 100 accepted tasks: the top-5 most frequently quoted entities from the accepted pool so far are appended, which steers subsequent generations away from saturated topics and encourages topical diversity across rounds.

- **Cross-model evaluation.** Three judges, drawn from the same model pool but excluding the generator, score each candidate on a 1–10 *solvability* scale. Only tasks judged solvable (score ≥ 7) by at least two judges survive.

- **Dry-run validation.** A separate executor agent (Claude Opus 4.6) attempts the task end-to-end against the live Wikipedia MCP server. A task is considered successful only if every tool call returns without error, the number of calls is not too low, and the final answer exceeds 100 characters to ensure it is non-trivial.

- **Log analysis and filtering.** A GLM-5 judge reads the full execution log of each passing run and assigns an accept/reject decision plus a 1–10 quality score covering completion, well-definedness, tool usage, and answer quality. Tasks that fail dry-run validation or are rejected by the log judge are discarded.

For the held-out evaluation pool, the same pipeline is re-run with one addition: the blacklist is *initialized* with frequently mentioned entities mined from training-side task descriptions, so that test topics avoid training topics from the first generation round. The dynamic refresh mechanism then continues during held-out generation. The result is passed through a rule-based pre-filter that drops tasks shorter than 800 characters, near-duplicates (string similarity ≥ 0.7), and topics appearing in more than five tasks. After the four-step pipeline plus cross-round de-duplication and error removal, the upstream training pool retains 2,517 tasks out of roughly 3,500 raw tasks. The held-out evaluation pool retains 879 candidate test tasks out of another 1,000 raw generations.

B.2 Downstream: from pool to datasets

The downstream stage runs the upstream pool through a Wikipedia agent with the context manager enabled, and turns the resulting trajectories into the deployed SFT, RL, and test datasets.

Benchmark rollouts. Every task in the 2,517-task training pool is executed once with the context manager enabled. Claude Opus 4.6 serves as the context manager, and Qwen3-Max serves as the task-solving agent. The per-step context cap is 24,576 tokens, with an extract threshold of 2,000 tokens. Each rollout produces a research answer together with the full sequence of context-manager invocations, which becomes the raw material for SFT extraction.

Task-level filter. A first pass rejects rollouts that are clearly broken or uninformative. We remove rollouts whose completion status is not `completed`, whose answers are shorter than 50 characters or begin with `"Error"`, or whose task-fulfillment score is below 2. The survivors are de-duplicated by text embeddings (cosine ≥ 0.92), and the remaining tasks are vetted by a Claude Opus 4.6 judge that scores completeness, naturalness, sufficiency, and grounding on a 1–5 scale. A task is kept only if all four scores exceed 2 and their mean is at least 3.5.

SFT samples. For each task that passes the filter, every context-manager invocation in its rollout is converted into a candidate `(prompt, response)` sample. Each sample is labeled by a coarse action category: `modify`, `delete`, `no_change`, or `extract`. These labels are used only for organizing SFT data and do not define a separate action space. The `modify` label covers non-empty context rewriting operations; `delete` corresponds to setting `new_content` to empty for the targeted messages; `no_change` corresponds to outputting an empty modification list; and `extract` corresponds to the long-tool-result extraction operation described in Appendix A, where an oversized tool result is first extracted before being inserted into the context.

The samples are also screened to ensure syntactic validity: responses must be parseable JSON, shorter than 12K characters, and, for `modify` actions, reduce the targeted content by at least 20%. Near-duplicates are removed by a MinHash sketch of the input text. A final ratio-control step anchors the dataset on informative `modify` samples and caps the other three action types relative to the `modify` count: `delete` and `no_change` are each limited to at most 10% of the `modify` count, and `extract` is limited to at most 30%. This yields a final composition of roughly 73% `modify`, 20% `extract`, and 7% `delete`/`no_change` combined, for a total of 4,740 samples used as input to the SFT phase.

Table 4: Data in the MCP-Bench-Wiki experiment.

Stage	Train pool	Test pool
Raw multi-model generations	∼3,500	1,000
After upstream filter & dedup	2,517	879
Final deployed datasets		
SFT samples	4,740	—
RL tasks	1,000	—
Test tasks	—	150

RL tasks. RL training is sampled from the same upstream training pool as SFT but operates at the *task* level rather than the per-invocation level. Starting from the same rollouts used for SFT extraction, we keep only tasks that completed with task-fulfillment ≥ 3, ran between 10 and 39 reasoning rounds to exclude trivially short and overly long traces, and whose descriptions are neither prefix near-duplicates of an already-kept task nor written in an overly prescriptive style. From the remaining pool we sample 1,000 tasks, retaining only the fields needed at training time: `task_id`, `task_description`, and `dependency_analysis`.

Test tasks. Evaluation tasks are drawn from the held-out upstream test pool rather than the training pool, eliminating task-source overlap. We additionally require the upstream log-analysis quality score to be at least 6, the executed trajectory to contain between 30 and 150 tool calls, and the reference answer to exceed 500 characters. We also remove prefix near-duplicates and overly prescriptive descriptions. A random sample of 150 tasks satisfying these criteria forms the test set used in our experiments.

B.3 Summary

Table 4 shows the funnel from raw task generation to deployed datasets. The training-side and test-side pools are produced by independent runs of the upstream pipeline with non-overlapping topic blacklists; together with cross-round deduplication, this ensures that the test split has no task-level overlap with SFT or RL.

C Additional Experiment Results and Analysis

C.1 Additional Experiment Results

Per-agent training pass@3 results on BrowseComp-Plus. Table 3 reports the pass@3 score of AdaCoM and baselines on BrowseComp-Plus across four agents. The trends are consistent with the mean@3 results in Table 1: AdaCoM achieves the best average pass@3 and improves all four agents over ReAct. In contrast, AdaCoM w/o training and self-management baselines remain unstable across agents, while SumCoM improves several agents but still lags behind AdaCoM on average. These results further support the benefit of training a flexible external context manager.

Cross-agent results on MCP-Bench-Wiki. Figure 5 shows the transfer effect on MCP-Bench-Wiki. Specifically, we test the two managers trained with DeepSeek-V3 and Kimi-K2-Instruct on four target agents: DeepSeek-V3, Kimi-K2-Instruct, GPT-4o-mini, and Seed-1.6-Flash. The results show that both trained managers improve every target agent over ReAct. The consistent gains across both source managers and all target agents provide additional evidence that the transfer behavior observed on BrowseComp-Plus is not benchmark-specific.

Per-dimension scores of MCP-Bench-Wiki. The original MCP-Bench judge scores each rollout along six rubric dimensions: task fulfillment (TF), information grounding (IG), tool appropriateness (TA), parameter accuracy (PA), dependency awareness (DA), and parallelism and efficiency (PE). In our setting, we use the three dimensions most related to context management: TF, IG, and PE. The overall score used for both training and evaluation is computed as $0.4 \times \mathrm{TF} + 0.4 \times \mathrm{IG} + 0.2 \times \mathrm{PE}$. Table 6 reports the per-dimension breakdown.

For GPT-4o-mini, DeepSeek-V3, and Seed-1.6-Flash, trained managers improve all three dimensions over ReAct, indicating that context management helps not only final task completion and grounding, but also the efficiency of tool use. Kimi-K2-Instruct shows a different pattern: AdaCoM improves the higher-weighted TF and IG dimensions while sacrificing PE relative to ReAct. This suggests that the manager helps Kimi-K2-Instruct gather and ground more useful information, but may encourage more tool use or longer investigation, reducing parallelism and efficiency. As TF and IG receive larger weights in the overall score, this trade-off still leads to a higher final score.

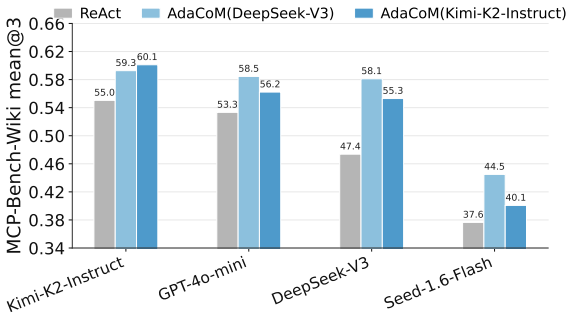

Figure 5: Cross-agent transfer of AdaCoM on MCP-Bench-Wiki. Each group on the x-axis is a target agent; bars within a group show MCP-Bench-Wiki **mean@3** under ReAct and the two trained AdaCoMs.

C.2 Ablation Study on Process Reward

We ablate the effect of process rewards in AdaCoM. Specifically, we train the managers for DeepSeek-V3 and Qwen3-Max with the process-reward weight set to $\alpha = 0$, so that the advantage is computed only from the outcome reward. Table 5 compares the original AdaCoM with this variant on BrowseComp-Plus. Removing process rewards consistently reduces performance for both agents. For DeepSeek-V3, mean@3 drops from 26.19% to 24.38%; for Qwen3-Max, it drops from 36.67% to 33.33%. These results indicate that process rewards provide useful intermediate supervision beyond the sparse final-answer reward.

Setting	DeepSeek-V3	Qwen3-Max
AdaCoM w/o pr.	24.38	33.33
AdaCoM	**26.19**	**36.67**

Table 5: **Mean@3** (%) of the original AdaCoM and AdaCoM without process reward on BrowseComp-Plus.

C.3 Analysis: Transfer Exceptions and Agent Specific Factors

We inspect exceptional transfer cases on BrowseComp-Plus to better understand factors beyond baseline capability. Although capability proximity explains the dominant transfer pattern, some managers trained on capability similar agents do not yield the best performance because their modification style is not well matched to the reasoning style of the target agent.

Agent	Setting	TF	IG	PE	Overall
Kimi-K2-Instruct	ReAct	52.5	51.0	**68.2**	55.0
	AdaCoM w/o train.	40.7	50.1	40.0	44.4
	AdaCoM (DeepSeek-V3)	56.6	60.4	62.4	59.3
	AdaCoM (Kimi-K2-Instruct)	**58.9**	**61.6**	59.0	**60.0**
GPT-4o-mini	ReAct	45.0	58.2	60.3	53.3
	AdaCoM w/o train.	45.3	61.0	35.8	49.7
	AdaCoM (DeepSeek-V3)	**51.0**	**63.8**	**62.5**	**58.5**
	AdaCoM (Kimi-K2-Instruct)	48.0	63.4	58.3	56.2
DeepSeek-V3	ReAct	42.1	48.0	57.3	47.5
	AdaCoM w/o train.	41.3	58.6	39.2	47.8
	AdaCoM (DeepSeek-V3)	**49.1**	**64.4**	**63.5**	**58.1**
	AdaCoM (Kimi-K2-Instruct)	45.8	62.2	60.3	55.3
Seed-1.6-Flash	ReAct	23.1	50.1	41.8	37.7
	AdaCoM w/o train.	26.0	48.8	34.9	36.9
	AdaCoM (DeepSeek-V3)	**31.6**	**55.2**	**48.8**	**44.5**
	AdaCoM (Kimi-K2-Instruct)	26.6	52.0	43.3	40.1

Table 6: Per-dimension mean@3 (%) of AdaCoM and baselines on MCP-Bench-Wiki. We report task fulfillment (TF), information grounding (IG), and parallelism and efficiency (PE), with **Overall** computed as $0.4 \times \mathrm{TF} + 0.4 \times \mathrm{IG} + 0.2 \times \mathrm{PE}$. Bold indicates the best result within each target agent and column.

DeepSeek and Kimi prefer different working memory styles. Although DeepSeek and Kimi have similar ReAct baselines, DeepSeek gains little from AdaCoM (Kimi). Trajectory inspection shows that AdaCoM (DeepSeek) and AdaCoM (Kimi) organize the working memory differently. AdaCoM (DeepSeek) tends to preserve concise key findings and document IDs that directly support the current search direction. In contrast, AdaCoM (Kimi) records a more detailed search history, including issued queries, returned document IDs, and summaries of document contents. This suggests that DeepSeek is more compatible with compact and exact clues, whereas Kimi benefits more from detailed attempt histories. As a result, a manager trained for Kimi can introduce unnecessary context for DeepSeek even though the two agents have similar baseline capability.

GLM is sensitive to aggressive compression. GLM also gains relatively little from managers trained on other agents. These managers do convert some ReAct failures into successes, but they also introduce regressions on tasks that ReAct already solves. We find that the main issue is excessive compression: managers trained on other agents often summarize or delete evidence more aggressively than GLM prefers. Since the Qwen3-4B-Instruct manager may not paraphrase evidence losslessly due to the limited capability, such compression can remove fine-grained information that GLM could otherwise exploit from raw context. In contrast, AdaCoM (GLM) compresses only occasionally and in larger batches, better preserving the information density required by GLM.

Gemini is less compatible with report style memory. For Gemini, AdaCoM (Qwen) underperforms AdaCoM (Kimi) despite Qwen and Gemini having similar ReAct baseline performance. We find that AdaCoM (Qwen) often produces working memory in a structured report style, containing previous queries and summarized results. When Gemini reads this report, it tends to mirror the report format in its own response instead of continuing the search, which can cause the agent to give up early without a valid answer. This case shows that transfer quality can depend on whether the manager's modification style is compatible with the target agent's reasoning behavior, beyond capability proximity alone.

D Case Study

We trace how GLM-4.5-Air and DeepSeek-V3 solve task 996 (a multi-constraint identity-resolution question) to illustrate how AdaCoM helps the agents complete the task and the distinct context-management strategies underlying their behavior. Both agents reach the correct answer, but via very different paths. GLM-4.5-Air's AdaCoM intervenes 11 times, and in 10 of those 11 turns it modifies only a single $\sim$1 K-token "working note" message (the index-1 message, placed right after

the user question), while every raw tool result is left untouched. In the first three rounds, since the tool results do not contain useful clues, the working note holds a clear list of constraints about the target identity and indicates that "no suitable candidate identified yet". When related documents are retrieved in the fourth round, AdaCoM adds "Current lead" (containing the discovered clues and corresponding document IDs) and "Pending verification" (the constraints not yet satisfied) into the working note. In subsequent rounds, AdaCoM continues to collect new clues into the working note when they surface, or leaves the context unchanged when none arise. In the final round, in addition to keeping the working note, it performs a batched deletion that removes useless tool results and merges several stale tool results into a summary. GLM-4.5-Air then successfully returns the answer along with verifications for all constraints.

In contrast, DeepSeek-V3's AdaCoM intervenes 18 times and in nearly every round either replaces the latest tool result with a paraphrase or deletes it outright. Concretely, AdaCoM maintains the refined constraints and current progress in the index-1 message throughout all rounds. Compared with the working note for GLM-4.5-Air, this note contains more details, including the executed search queries and returned document IDs. For the latest tool result, AdaCoM often removes it in the same round, before the agent sees it in the next step. Raw results are retained only when they contain evidence for unresolved constraints, and even then for at most three rounds before being paraphrased or deleted.

E Code Availability

The code and data for this work are publicly available at `https://anonymous.4open.science/r/AdaCoM-8864/`.

F Artifact Licenses and Terms of Use

This work builds on several open-source artifacts, each used solely for non-commercial academic research. **Trinity-RFT** (Pan et al., 2025), the reinforcement learning training framework used to train the context manager, is released under the Apache 2.0 License. **AgentScope** (Gao et al., 2025), used to implement the overall agent workflow, is likewise released under the Apache 2.0 License. **MEM1** (Zhou et al., 2025), from which we derive the SumAgent baseline, is released under the MIT License. **MCP-Bench** (Wang et al., 2025), used to construct the MCP-Bench-Wiki training and evaluation set, and **BrowseComp-Plus** (Chen et al., 2025b), used for both training and evaluation, are publicly available on GitHub; neither repository specifies an explicit open-source license at the time of use. All artifacts are used in accordance with their intended research purposes.

Regarding data privacy, BrowseComp-Plus and MCP-Bench are constructed from publicly available web sources and benchmark corpora. We did not collect any new human-subject data. We manually inspected a random sample of training and evaluation instances and found no content that names or uniquely identifies private individuals, nor any offensive material.

G AI Assistant Usage Disclosure

In the preparation of this manuscript, AI writing assistants (Claude, ChatGPT) were used to assist with proofreading and language polishing. All scientific claims, experimental designs, analyses, and conclusions are the sole responsibility of the authors.

H Adopted Prompts

This section lists all prompts used in our system. Table 7 gives the agent system prompts for BrowseComp-Plus and MCP-Bench-Wiki. Table 9 gives the AdaCoM context manager prompt; the placeholder `{{Background}}` is filled with a short benchmark-specific paragraph (Table 8) that describes the agent's tools and the information-retention priority for that benchmark — this is the only part of the AdaCoM prompt that differs between BrowseComp-Plus and MCP-Bench-Wiki. Tables 10 and 11 give the judge prompts used to score agent outputs on BrowseComp-Plus and MCP-Bench-Wiki, respectively; both follow the original benchmark judge designs (Chen et al., 2025b; Wang et al., 2025), with the MCP-Bench-Wiki prompt augmented with finer-grained percentage-based scoring rubrics to improve score discrimination. Table 12 gives the extraction prompt applied on MCP-Bench-Wiki when `get_article` or `get_sections` results are appended to the context.

Table 7: Agent system prompts for BrowseComp-Plus and MCP-Bench-Wiki.

BrowseComp-Plus Agent System Prompt

You are a deep research agent. You need to answer the given question by interacting with a search engine, using the `search` and `get_document` tools provided. Please perform reasoning and use the tool step by step, in an interleaved manner.

NOTE:

- You should always call one tool at a time. Use short keywords as the query for the `search` tool call.
- You should first provide your reasoning process (your chain of thought) before each tool call step.
- When you have a definitive answer or cannot progress further, call the `finish` tool to provide your final answer.

MCP-Bench-Wiki Agent System Prompt

You are an AI assistant that can use various tools to complete tasks. You have access to tools from multiple MCP servers. Each tool is prefixed with its server name (e.g., `Wikipedia__search_wikipedia`).

NOTE:

- You should always call tools to find the information you need to complete the task, but no more than five tools at a time.
- When you have completed the task or cannot progress further, call the `finish` tool to provide your final answer.

Table 8: Background information injected at the `{{Background}}` placeholder in the AdaCoM prompt (Table 9).

BrowseComp-Plus Background Information

Task Background Information

Agent Mission: The agent is tasked with answering complex questions by iteratively searching a database for essential clues and evidence.
Operational Workflow & Tools: `search` takes a query and returns a list of document snippets. `get_document` takes a specific document ID and retrieves the full-text content.
Core Requirement (Data Traceability): Each document is identified by a unique ID. To ensure the agent can provide verifiable citations and maintain information provenance, it is critical to preserve the document IDs associated with important information.

MCP-Bench-Wiki Background Information

Task Background

The agent is responsible for executing complex tasks by leveraging tools across multiple MCP servers.
Information Retention Strategy: Retain information at a level of detail appropriate to the task at hand. If the task requires generating structured output (e.g., reports, summaries), preserve comprehensive details. Otherwise, retain only key findings and critical clues necessary to complete the task.

Table 9: AdaCoM context manager prompt. {{Background}} is filled with the benchmark-specific background information in Table 8; {{full_memory}} and {{token_usage_ratio}} are filled at runtime.

You are a **Memory Modifier** specialized in optimizing agent context windows by compressing, summarizing, or removing content while preserving task-critical information.

Core Objective

Analyze the agent's context and generate specific memory modifications that: (1) maintain only the necessary information for the current task; (2) safely compress or remove unnecessary content; (3) preserve logical continuity and dependencies.

Input

1. Agent Context: The complete context window including the original task description, previous agent actions and results, and any existing summaries or hints from prior rounds.
2. Token Usage Ratio: Current token usage (0% to 100% scale).
{{Background}}

Memory Modification Process

Follow this reasoning chain: (1) **Analyze Context**: Understand the task progress and the agent's goal in the current round. (2) **Apply Strategy**: Apply the compression strategy to the specific context.

Output Format Return {"modifications":[]} if no changes are needed.

```
{
  "modifications": [
    {
      "ids":           ["<msg_id_1>", "<msg_id_2>", ...],
      "role":          "user" | "assistant",
      "justification": "<reason for this modification>",
      "new_content":   "<replacement text, or empty string \"\" to remove>"
    }
  ]
}
```

Field Specifications

- **ids**: One or more *consecutive* message IDs to modify together.
- **role**: Perspective for the new content. Use "user" to increase agent attention to critical information.
- **justification**: Your justification for the modification.
- **new_content**: Replacement content or summary; empty string "" for complete removal.

Strategy Guidelines

Your role is to manage the agent's context window, *not* to guide its problem-solving. Do **not** include next-step instructions in "new_content".

Core Principles:

- Always preserve the original task description — without it, the agent loses task context entirely.
- Preserve information the agent will need; remove or compress information it won't.
- If the agent is ignoring something important, make it more prominent.
- If the agent is repeating itself, ensure the context clearly shows the action was already taken and its outcome.

General Techniques:

- Keep recent context detailed; compress older context progressively.
- Use clear structural markers to distinguish completed actions, findings, and constraints.
- Retain specific details likely to be needed later — overly aggressive summarization can cause redundant work.

Your Input

Agent Context: {{full_memory}} Token Usage Ratio: {{token_usage_ratio}}

Table 10: BrowseComp-Plus judge prompt. `{{question}}`, `{{correct_answer}}`, `{{actual_answer}}`, and `{{explanation}}` are filled at runtime. The system turn is: *"You are a precise evaluator. Always respond with valid JSON format as requested."*

You are an expert judge tasked with evaluating whether an agent's response to a question is correct.

Question: `{{question}}`

Ground Truth Answer: `{{correct_answer}}`

Agent's Final Answer: `{{actual_answer}}`

Agent's Explanation: `{{explanation}}`

Your task is to determine if the agent's output is correct based on the ground truth answer. Be strict and precise in your judgment.

Evaluation Criteria:

1. Extract the final answer primarily from the agent's final answer field.
2. Use the explanation as supporting context that can clarify, refine, or contradict the final answer.
3. Compare the agent's overall output with the ground truth answer.
4. The agent's answer is correct **only if** it is semantically equivalent to the ground truth.
5. Allow for minor variations in phrasing, but the core information must match exactly.
6. For numerical answers, allow small rounding differences (within 1% or 0.1 units).
7. If the final answer contains additional information that does not contradict the ground truth, it can still be marked as correct.
8. If the final answer is ambiguous, contradictory, or contains incorrect information, mark it as incorrect.
9. If the agent did not provide a clear final answer, mark it as incorrect.

Output Format: Respond with a valid JSON object only (no additional text):

```
{
  "extracted_answer": "<exact answer extracted from the agent's output, or null>",
   "ground_truth":     "<the ground truth answer>",
   "reasoning":        "<why the answer is correct or incorrect>",
   "score":            1.0 or 0.0
}
```

Table 11: MCP-Bench-Wiki judge prompt. `{{task}}`, `{{concrete_task_description}}`, `{{available_tools}}`, `{{execution_summary}}`, `{{final_solution}}`, `{{total_rounds}}`, and `{{dependency_analysis}}` are filled at runtime. The system turn is: *"You are an expert AI task execution evaluator. Score each dimension objectively based on evidence."*

You are an impartial evaluator judging the quality of an AI agent's tool-based task execution.

You must assign scores **only based on evidence** from the task, solution, and tool usage. Your evaluation should be:

- Objective (avoid being influenced by language fluency or formatting).
- Justified (include specific reasons tied to each score).
- Robust against bias (ignore narrative style, verbosity, or formatting polish).

Critical format rules:

- **Do not** penalize for output format (JSON, text, etc.) unless the task presented to agent explicitly requires it.
- If the task presented to agent says "provide information" without specifying format, **any** readable format is acceptable.
- Only deduct points for format if the task explicitly states "return as JSON" or "format as table" etc.
- Focus on **content** correctness, not presentation style.

Available tools (`{{N}}` tools): `{{available_tools}}`

Task presented to agent: `{{task}}`

Concrete task reference (for evaluation context only). Note: the agent did **not** see this concrete version, it only saw the task above. The task visible for the agent is the fuzzy version of the concrete task. This reference helps assess actual task completion but is not the sole criterion. The agent's interpretation of the fuzzy task may differ but still be valid.

Format reminder: if the concrete task mentions JSON but the task presented to agent doesn't explicitly require it, **do not** penalize for not using JSON format. Only the task presented to agent's requirements matter for format. `{{concrete_task_description}}`

Execution summary: `{{execution_summary}}`

Final solution: `{{final_solution}}`. **Total rounds**: `{{total_rounds}}`.

Dependency analysis (reference only). Note: this analysis was generated during task creation to help understand tool dependencies. The agent did **not** see this analysis. It is provided as reference for evaluation purposes. `{{dependency_analysis}}`

Task completion rubric (1–10 per subdimension).

1. Task fulfillment and quality.

- 1–3: Perfectly completes 10–30% of requirements.
- 4–6: Perfectly completes 40–60% of requirements.
- 7–8: Perfectly completes 70–80% of requirements.
- 9–10: Perfectly completes 90–100% of requirements.

NOTE: requirements come from the task presented to agent only. Format (JSON/text) is **not** a requirement unless explicitly stated in the task presented to agent.

2. Grounding.

- 1–3: 10–30% of claims are perfectly grounded in tool outputs.
- 4–6: 40–60% of claims are perfectly grounded in tool outputs.
- 7–8: 70–80% of claims are perfectly grounded in tool outputs.
- 9–10: 90–100% of claims are perfectly grounded in tool outputs.

Tool usage rubric (1–10 per subdimension).

1. Tool appropriateness.

- 1–3: 10–30% of tools were perfectly selected for their subtasks.
- 4–6: 40–60% of tools were perfectly selected for their subtasks.
- 7–8: 70–80% of tools were perfectly selected for their subtasks.
- 9–10: 90–100% of tools were perfectly selected for their subtasks.

2. Parameter accuracy.

- 1–3: 10–30% of tool calls have perfectly accurate and complete parameters.
- 4–6: 40–60% of tool calls have perfectly accurate and complete parameters.
- 7–8: 70–80% of tool calls have perfectly accurate and complete parameters.
- 9–10: 90–100% of tool calls have perfectly accurate and complete parameters.

Planning effectiveness and efficiency (1–10 per subdimension).

1. Dependency awareness.

- 1–3: 10–30% of dependency chains are perfectly executed.
- 4–6: 40–60% of dependency chains are perfectly executed.
- 7–8: 70–80% of dependency chains are perfectly executed.
- 9–10: 90–100% of dependency chains are perfectly executed.

2. Efficiency.

- 1–3: More than 70% of tool calls are redundant or unnecessary.
- 4–6: 40–60% of tool calls are redundant or unnecessary.
- 7–8: 10–30% of tool calls are redundant or unnecessary.
- 9–10: Less than 10% of tool calls are redundant or unnecessary.

Percentage-based scoring system.

How to calculate scores: for each dimension, calculate the defect rate = (number of issues / total opportunities) $\times$ 100%. Then map defect rate to score:

- 0–10% defects $\rightarrow$ score 9–10 (excellent to perfect).
- 10–30% defects $\rightarrow$ score 7–9 (good performance).
- 30–50% defects $\rightarrow$ score 5–7 (average performance).
- 50–70% defects $\rightarrow$ score 3–5 (poor performance).
- 70–100% defects $\rightarrow$ score 0–3 (failed).

How to score:

1. When evaluating percentages, assess what counts as "well executed" for each dimension:
   - Task fulfillment: requirements completed correctly.
   - Grounding: claims supported by actual tool outputs.
   - Tool appropriateness: suitable tools chosen for each subtask.
   - Parameter accuracy: correct and complete parameters in tool calls.
   - Dependency awareness: proper ordering of dependent operations.
   - Efficiency: minimal redundant or unnecessary tool calls.
2. Minor imperfections reduce the percentage proportionally, not to zero.
3. Map the resulting defect rate to the score range above.

Key principles:

1. **Always** calculate as percentage, **not** absolute numbers.
2. 10 errors in 100 calls (10%) = same score as 1 error in 10 calls (10%).
3. Consider the opportunity count for each dimension:
   - Tool calls: how many total calls were made?
   - Parallelization: how many tasks **could** have been parallel?
   - Parameters: how many total parameters across all calls?
   - Claims: how many factual statements were made?
   - Dependencies: how many dependency relationships exist?

Score each dimension based on the defect rate mapped above. Use the full 1–10 range.

Concrete scoring examples with proportions:

- *Task fulfillment*: 19/20 requirements (5% defect) = 9; 16/20 (20%) = 8; 12/20 (40%) = 6; 8/20 (60%) = 4.
- *Tool appropriateness*: 19/20 optimal (5% defect) = 9; 16/20 (20%) = 8; 12/20 (40%) = 6; 8/20 (60%) = 4.
- *Efficiency*: 2/20 redundant (10%) = 9; 4/20 (20%) = 8; 8/20 (40%) = 6; 12/20 (60%) = 4.
- *Grounding*: 19/20 supported (5% unsupported) = 9; 16/20 (20%) = 8; 12/20 (40%) = 6; 8/20 (60%) = 4.
- *Parameter accuracy*: 95/100 perfect (5% defect) = 9; 80/100 (20%) = 8; 60/100 (40%) = 6; 40/100 (60%) = 4.
- *Dependency awareness*: 9/10 ordered (10% misordered) = 9; 8/10 (20%) = 8; 6/10 (40%) = 6; 4/10 (60%) = 4.

Format notes:

- Text output when JSON not required in the task presented to agent = no penalty (0% defect).
- Missing JSON when explicitly required in the task presented to agent = count as failed requirement.

Normalize by complexity (don't punish complex tasks):

- Simple task: 1 error / 5 steps (20% defect) = score 7.
- Complex task: 4 errors / 20 steps (20% defect) = score 7.
- Server count is irrelevant — using more servers is **not** better.

Critical evaluation requirements:

1. For task fulfillment, use chain-of-thought: first list **all** requirements from the task, then for each state whether fulfilled with evidence, then count fulfilled/total = percentage, then map to score range.
2. You **must** map each score to the exact percentage ranges in the rubrics.
3. Task completion and tool usage **must** be evaluated against the concrete task reference, not the fuzzy task.
4. Planning effectiveness should be evaluated based on the proportion of dependencies correctly handled, not the absolute number of steps executed or exact conformance to the dependency analysis.
5. First calculate the actual percentage of completion/success, then assign the corresponding score range.
6. Focus on completion **ratios** not absolute numbers — completing 7/10 steps (70%) should score similarly to completing 14/20 steps (70%), regardless of task complexity.

Please score based on completion percentages and proportional success, not absolute numbers. Return your evaluation scoring and reasoning in this exact JSON format. **All six numeric score fields (integer 1–10) are mandatory.** Missing any score field invalidates the response.

```
{
  "task_fulfillment_reasoning":
      "Explain how well the agent fulfilled the detailed task objectives,
       referencing specific content from the concrete task description
       and what percentage was completed.",
  "grounding_reasoning":
      "Explain how well the agent's outputs were grounded in actual
       tool results versus unsupported claims.",
  "tool_appropriateness_reasoning":
      "Explain whether the tools selected were appropriate for each
       subtask requirement.",
  "parameter_accuracy_reasoning":
      "Explain the accuracy and completeness of parameters used in tool
```

```
      calls, noting any missing required parameters or incorrect values.",
  "dependency_awareness_reasoning":
      "Explain how well the agent understood and respected task
      dependencies (what percentage of dependencies were handled
      correctly), refer to the provided dependency analysis section.",
  "parallelism_efficiency_reasoning":
      "Explain the efficiency of execution, including use of parallelism
      and avoiding redundancy, refer to the provided dependency analysis
      section.",
  "task_fulfillment":           X,
  "grounding":                  X,
  "tool_appropriateness":       X,
  "parameter_accuracy":         X,
  "dependency_awareness":       X,
  "parallelism_and_efficiency": X
}
```

Return **only** the JSON object.

Table 12: Extraction prompt applied when `get_document` (BrowseComp-Plus) or `get_article` / `get_sections` (MCP-Bench-Wiki) results are appended to the context. `{{chunk_idx}}`, `{{total_chunks}}`, `{{question}}`, `{{existing_notes}}`, and `{{chunk}}` are filled at runtime. If the tool result fits within the context budget it is processed in a single pass; otherwise it is split into overlapping chunks and the reading note is refined iteratively.

Objective: Your mission is to help answer the "Original Question" by refining a "Reading Note" based on sequentially provided document chunks. This iterative process ensures the note evolves into a concise and relevant tool for addressing the question.

Your Task

Analyze the "New Chunk of Document" and critically revise the "Existing Reading Note". Your goal is to produce an updated version of the note, incorporating new insights while maintaining focus on the original question.

Key Guidelines

1. **Relevance is Key**: Include only the information that directly or potentially contributes to answering the "Original Question". Eliminate irrelevant or redundant details.
2. **Refine, Don't Just Append**:
   - **Merge**: Consolidate new information with existing points to enhance clarity and completeness.
   - **Update**: Replace general statements with more precise or specific findings from the new chunk.
   - **Remove**: Discard outdated, irrelevant, or less important sections of the note.
3. **No-Change Option**: If the new chunk provides no relevant information, simply return the unchanged Existing Reading Note.
4. **Be Concise**: Keep the note succinct, capturing only the most critical and essential facts. Avoid summarizing the entire document; this is a working reference, not a comprehensive report.
5. **No Premature Conclusions**: Focus strictly on refining the note at each step. Save final judgments or conclusions until all chunks have been processed.

Output Format

Your response must only consist of the full text of the revised "Updated Reading Note". Do not write any explanations, commentary, or other additional text.

Progress: `{{chunk_idx}}` out of `{{total_chunks}}`

Original Question: `{{question}}`

Existing Notes: `{{existing_notes}}`

New Chunk of Document: `{{chunk}}`

Updated Reading Note: