Self-Manager: Parallel Agent Loop for Long-form Deep Research

Anonymous ACL submission

Abstract

Long-form deep research requires multi-faceted investigations over extended horizons, posing significant challenges for existing autonomous agents. Current single-agent paradigms suffer from linear context accumulation and mutual interference between subtasks, while multi-agent systems often lack the runtime adaptability needed for dynamic execution. To address these limitations, we propose Self-Manager, a novel parallel agent loop architecture designed for long-horizon tasks. Self-Manager introduces a multi-threaded execution model characterized by asynchrony, concurrency, and context isolation. By utilizing Thread Control Blocks (TCBs), the main thread can independently spawn, monitor, and manage multiple subthreads, effectively decoupling complex task contexts and preventing execution blocking. Benchmarked on the DeepResearch Bench, Self-Manager consistently outperforms existing single-agent and multi-agent baselines across all metrics, narrowing the gap with proprietary state-of-the-art systems. Our analysis further demonstrates that Self-Manager significantly enhances contextual capacity, operational efficiency, and generalization in complex research environments.

1 Introduction

Language agents (Su et al., 2024) are directing AI skills toward more complex tasks, particularly *long-horizon tasks* that require multi-turn reasoning and environment interaction. A typical instance is the **Deep Research** tasks (Du et al., 2025; Wei et al., 2025), which demands iterative searching and information synthesis across massive web sources, evaluate the agent's advanced capabilities. Existing approaches to modeling deep research agents can be broadly categorized into two types:

Single-Agent. Existing single-agent paradigms mainly employ ReAct-style "Think–Act–Observe" pattern (Jin et al., 2025; Nguyen et al., 2025; Yao et al., 2022). Any execution topology required for arbitrary task can, in principle, be synthesized through this simple and generic loop. However, this classical design is bound by linearly accumulated context, which can trigger context limit overflows and attention dilution (Gao et al., 2025). While several methods based on context compression (Wu et al., 2025b; Nguyen et al., 2025) and subtask decomposition (Schroeder et al., 2025; Ye et al., 2025; Sun et al., 2025) mitigate this issue, long-horizon tasks remain problematic because multiple parallelizable subtasks are still mixed in the same context window, causing mutual interference and performance bottlenecks.

Multi-Agent. To improve task-specific performance, some approaches develop multi-agent systems based on pre-defined workflows (Prabhakar et al., 2025; Hong et al., 2025), with explicit planning, searching, summarizing, etc. However, these designs exhibit lower generalizability compared to single-agent loops, limiting their flexibility to long-horizon tasks with increasingly dynamic execution topologies. While more recent work explores Adaptive Multi-Agent Systems (Adaptive MAS; Wang et al., 2026; Li et al., 2025b) that generate tailored topologies for current tasks, they lack the runtime adaptability to observe and manage execution at every turn, which can lead to error accumulation and running inefficiencies.

To address these limitations, we propose **Self-Manager**, which is a parallel agent loop architecture. Specifically, we introduce the concept of *threads*, where the main thread directly handles the user task and can spawn subthreads. This allows for the decomposition of complex tasks and the distribution of subtasks to subthreads. Notably, Self-Manager exhibits the following features:

- **Asynchrony**. Self-Manager executes in parallel, and the main thread is not blocked by the execution of subthreads.

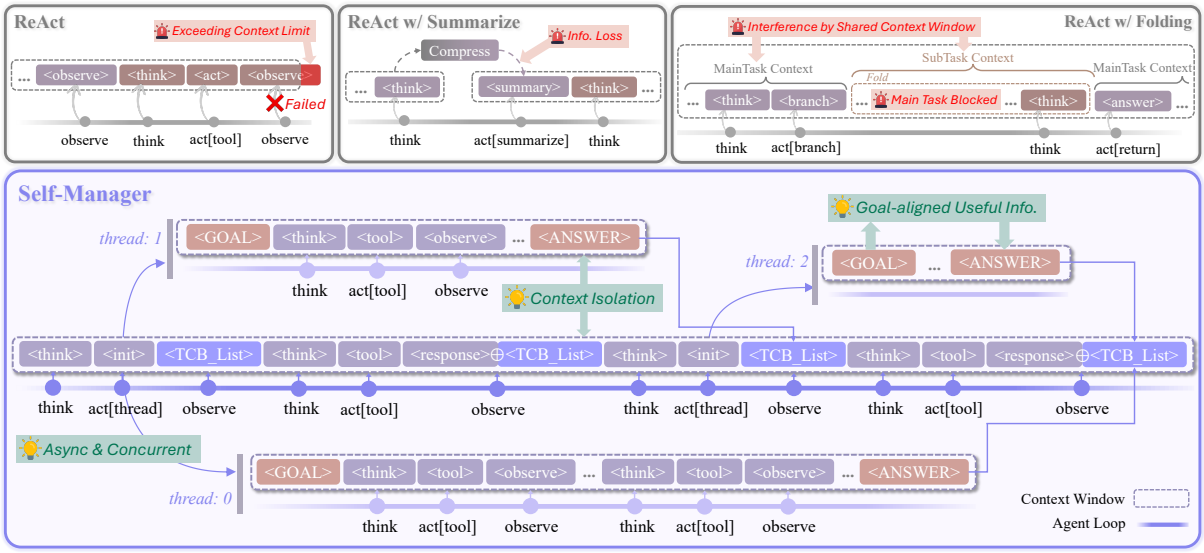

Figure 1: Overview of our proposed Self-Manager. Compared to classical agent loops, Self-Manager spawns subthreads asynchronously and concurrently, with contexts isolated between threads. The main thread can operate on subthreads and monitor their latest states via the TCB list, thereby enabling runtime autonomous management and adaptability. Upon completion, each subthread returns its result to the main thread's most recent observation.

- **Concurrency**. Self-Manager can initialize multiple subthreads simultaneously at any iteration turn, enabling parallel exploration of different perspectives of a task.
- **Context Isolation**. In Self-Manager, each thread has its own context window, ensuring that subtasks do not interfere with one another.

To enable these capabilities in the parallel agent loop, we introduce the *Thread Control Block* (TCB), a structured object encapsulating each subthread's information. The main threads can monitor all subthreads by observing the TCB list, thereby enabling self-managed multi-threaded execution.

Compared to single-agent methods, Self-Manager resolves linear context growth and inter-task interference by decoupling contexts through multi-threading and condensing critical information into TCBs via thread returns. Compared to multi-agent methods, Self-Manager achieves turn-level controllability of execution topologies through the asynchronous design and standardized TCBs, thereby enabling runtime adaptability.

We conduct experiments on DeepResearch Bench (Du et al., 2025), a challenging benchmark for long-form deep research. In our implementation, all threads share a single base model. The results show that Self-Manager achieves the best performance compared with other single-agent loop paradigms and multi-agent systems, and narrows the performance gap with proprietary deep research systems such as Gemini Deep Research (Google, 2025). Furthermore, we examine the component necessity, contextual capabilities, efficiency, and generalizability of Self-Manager, providing a multi-dimensional analysis of the improvements offered by Self-Manager as a parallel agent loop.

Our main contributions are as follows: (1) We propose a novel parallel agent loop, Self-Manager, capable of asynchronously and concurrently conducting tool-integrated inference to handle long-horizon tasks. (2) We design the Thread Control Block and incorporate it into the agent loop to enable self-management of the parallel execution in Self-Manager, hence ensuring runtime adaptability. (3) Our proposed Self-Manager outperforms other single-agent and multi-agent baselines on DeepResearch Bench benchmark.

2 Methodology

In this section, we provide a detailed description of the modeling of Self-Manager.

2.1 Background

We consider an agent following the ReAct paradigm (Yao et al., 2022), which interleaves reasoning and environment interaction through a sequence of actions and observations. A standard ReAct agent loop trajectory of length T is:

$$\tau = \{a_1, o_1, a_2, o_2, \ldots, a_T, o_T, \mathcal{A}\}, \qquad (1)$$

where the observation of environment o_t is from the action a_t sampled from the agent's policy $a_t \sim \pi_\theta(\cdot \mid h_{t-1})$, with h_{t-1} denoting the historical context accumulated so far. In practice, each step's action can decompose into a thinking and an acting component: $a_t \longrightarrow \{a_t^{\text{think}}, a_t^{\text{act}}\}$. This generic loop is, in principle, applicable to a wide range of tasks. However, the linear context growth prevents it from supporting long-horizon execution.

Although some prior work compresses historical context when approaching the context limit, such compression can easily lead to the loss of critical information. Thus, methods such as FoldAgent (Sun et al., 2025) manage context at the granularity of subtasks. While this design better accommodates complex tasks, it is constrained by the shared context window, limiting the scalability.

2.2 Parallel Agent Loop

We first introduce the concept of *threads* in a single agent. The original "Think-Act-Observe" trajectory that aims to solve the global task can be viewed as the main thread, denoted τ_{main}, with context h_{main}. In contrast to classical agent loops, Self-Manager is no longer confined to a single thread, and enables the main thread to dynamically spawn subthreads. Each subthread executes an independent loop, with objectives decoupled from the global task and restricted to the subtask delegated by the main thread, enabling focused tool-integrated reasoning without interference from non-critical information. Upon completion, each subthread returns its subtask result to the main thread, thereby contributing to the global task. The execution flow of Self-Manager is shown in Algorithm 1.

Concretely, the key design components of Self-Manager's parallel loop include the following:

- **Main thread action space**: During the act phase, the main thread may invoke a tool, such as web retrieval, or manipulate subthreads. Operations on subthreads mainly include creation, termination, and deletion.

- **Main thread observations**: During the observe phase, besides tool feedback or results of thread operation (success/failure), the agent also receives meta information of all existing subthreads. These observations allow the main thread to holistically analyze subthread states and global task progress, thereby determining the action in the next turn. Details of this mechanism are presented in §2.3.

Algorithm 1 Self-Manager Agent Execution

Require: Task x, agent's policy π_θ, environment $\mathcal{E}$, maximum turns $T_{\max}$, tools $\mathcal{T}$
Ensure: Final report $\mathcal{A}$

```
Initialize: h_main ← ∅, TCB list B_0 ← ∅
for t = 1 to T_max do
    a_t ~ π_θ(· | x, h_main)                    ▷ act
    if a_t is "Finish" then
        A ← π_θ(· | x, h_main); break
    if a_t ∈ T then o_t ← E(a_t)                ▷ tool observe
    if a_t is "Create Threads" then
        for thread ς ~ a_t do
            create_async_task(ς)
            B_{t-1} ← B_{t-1} ∪ {ς}
    if a_t is "Kill" then stop ς ~ a_t
    if a_t is "Del" then B_t ← B_{t-1} \ {ς ~ a_t}
    o_t ← o_t ⊕ B_t, h_main ← (a_t, o_t)        ▷ TCB
end for
return A
```

- **Subthreads**: For simplicity, subthreads are not allowed to create nested subthreads. Each subthread executes a standard ReAct loop, as shown in Algorithm 2.

Algorithm 2 `create_async_task`(ς)

Require: Goal x', available tools $\mathcal{T}'$, prefix context h'
Ensure: Answer $\mathcal{A}'$

```
Initialize subthread: h_sub ← {h'}
for t = 1 to T_max do
    a'_t ~ π_θ(· | x', h_sub)                   ▷ act
    if a'_t is "Finish" then
        A' ← π_θ(· | x', h_sub); break
    if a'_t ∈ T' then
        o'_t ← E(a'_t)                          ▷ observe
    h_sub ← (a'_t, o'_t)
end for
return A'
```

Based on the above design, Self-Manager enables several key features of a parallel agent loop:

- **Asynchrony**. Once a subthread is created, the main thread is not blocked. Instead, all threads run asynchronously in parallel. Similar to thread switching in operating systems, a thread only needs to await when its loop requests the base model to generate an action, which constitutes the sole blocking event.

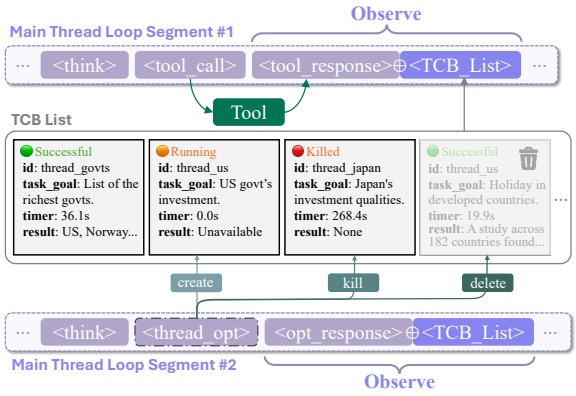

Figure 2: An illustration of how Self-Manager manages subthreads via TCBs. In each iteration, the environment response in the Observe phase includes both the feedback from the tool call and the TCB list.

During this time, context switching between threads occurs naturally.

- **Concurrency**. The main thread can create multiple subthreads that execute asynchronously, allowing different aspects of a complex task to be explored in parallel.

- **Context Isolation**. Each thread maintains its own dedicated context window, preventing interference between concurrently executing tasks. During initialization, a subthread receives information such as the subtask description provided by the main thread. Upon completion, it returns the subtask execution results to the main thread's context.

Thus, the trajectories and contexts encompassed by Self-Manager can be summarized as follows:

$$\left(\tau_{\text{main}} = \left((a_t, o_t)_{t=1}^{T}, \mathcal{A}\right), h_{\text{main}}\right), \quad (2)$$

$$\left\{\tau_{\text{sub}}^{(i)} = \left((a_t^{(i)}, o_t^{(i)})_{t=1}^{T^{(i)}}, \mathcal{A}^{(i)}\right), h_{\text{sub}}^{(i)}\right\}_{i=1}^{N}. \quad (3)$$

2.3 Agent-as-TCB-Manager

As described in §2.2, the main thread of Self-Manager achieves self-management of subthreads by presenting the state information of all threads during the Observe phase. This design preserves the generalizable Think–Act–Observe loop structure. Specifically, we draw inspiration from operating systems: each subthread is associated with a *Thread Control Block* (TCB), which is a data structure that contains the thread-specific information required for its management. Replacing raw contextual information with TCBs can significantly improve the efficiency of subthread management.

As shown in Figure 2, the TCB defines, from the perspective of a thread, the fields that are most critical to the agent loop. First, there is the basic thread metadata, which includes the thread ID, task description, as well as a result field used to present the execution outcome upon completion. In addition to the fields shown in Figure 2, the TCB also contains resource-related information, such as the prefix context inherited from the main thread and the tools assigned to the subthread. Furthermore, it maintains thread state information, such as whether the thread is currently running and the elapsed execution time. Except for the state information, all other fields are specified by the main thread, thereby ensuring flexibility in self-management. More details of the TCB definition and implementation can be found in Appendix B.

As depicted in Algorithm 1 and Figure 2, during each observe phase of the main thread in Self-Manager, the latest information from all subthreads's TCBs is accessible. Based on these observations, the main thread can assess the state of each subthread and determine actions in the next turn, such as spawning new threads to handle uncovered subtasks, terminating subthreads that have run excessively long with little benefit, or deleting TCBs that contain no critical information. By integrating the TCB into the agent loop, the main thread can effectively manage threads, thereby continuously advancing the resolution of complex tasks. Compared to parallel execution without autonomous management (Schroeder et al., 2025; Lian et al., 2025), the self-managed approach provides enhanced flexibility and task concentration.

Since the states of subthreads evolve in real time, we further reduce context overhead by retaining only the most recent turn of the TCB list within the loop for immediate action decision-making in the next turn. After the next turn of action generation is completed, the previous TCB list is cleared to eliminate interference from outdated state information and to avoid redundancy.

3 Experiments

In this section, we conduct experiments centered on Self-Manager to investigate the following research questions: (1) How well does Self-Manager perform on complex tasks? (2) How do the individual components of Self-Manager affect its overall per-

Table 1: Results on DeepResearch Bench. The best-performing approaches among single-agent and multi-agent baselines, and among proprietary agents baselines are highlighted in boldface. Green and red numbers denote the performance gains and drops when using alternative base models compared to Qwen3-30B-A3B, respectively.

Model	RACE					FACT	
	Overall	Comp.	Insight	Inst.	Read.	C.acc.	Eff.c.
LLMs w/ Search							
Qwen3-30B-A3B	41.83	39.85	40.27	44.74	44.44	60.63	8.30
Claude-3.7-Sonnet	42.29	41.45	43.78	44.53	42.23	89.51	10.39
Multi-Agent System							
Langchain-open-DeepResearch	43.89	**43.61**	42.19	47.30	45.07	53.52	31.17
AI-Q NVIDIA Research Assistant	43.69	42.07	**42.76**	46.15	45.14	-	-
Single Agent							
ReAct (Qwen3-30B-A3B)	40.51	39.15	37.71	43.98	43.40	21.69	1.36
ReSum (Qwen3-30B-A3B)	42.77	41.34	40.09	46.82	45.15	32.72	2.33
FoldAgent (Qwen3-30B-A3B)	41.79	40.04	38.95	46.22	43.97	19.61	1.56
Self-Manager (Qwen3-30B-A3B)	**44.33**	43.14	42.28	**47.61**	**45.97**	25.49	2.19
Self-Manager (Qwen3-8B)	$43.88_{\downarrow 0.45}$	$42.30_{\downarrow 0.84}$	$42.00_{\downarrow 0.28}$	$47.35_{\downarrow 0.26}$	$45.78_{\downarrow 0.19}$	17.80	1.86
Self-Manager (Qwen3-30B-A3B-2507)	$45.47_{\uparrow 1.14}$	$43.50_{\uparrow 0.36}$	$44.07_{\uparrow 1.79}$	$48.21_{\uparrow 0.60}$	$48.08_{\uparrow 2.11}$	24.80	2.51
Self-Manager (Qwen3-Next-80B-A3B)	$45.63_{\uparrow 1.30}$	$43.53_{\uparrow 0.39}$	$44.32_{\uparrow 2.04}$	$48.17_{\uparrow 0.56}$	$48.14_{\uparrow 2.17}$	35.90	6.69
Proprietary Agents							
Gemini-2.5-Pro Deep Research	**48.24**	**48.24**	**47.62**	48.74	**48.97**	85.71	27.0
OpenAI-DeepResearch	47.20	47.07	45.84	**48.98**	47.64	65.30	34.15

formance? (3) How effectively does Self-Manager perform in terms of contextual capabilities? (4) What are the efficiency and computational cost of Self-Manager? (5) How well does Self-Manager generalize across different tasks?

3.1 RQ1: Performance on Complex Tasks

This section investigates the performance of our proposed Self-Manager on complex tasks, particularly those that require subtask decomposition for effective problem solving.

Setup. We evaluate our method on DeepResearch Bench (Du et al., 2025), which consists of 100 complex long-form deep research tasks. We use Gemini-2.5-Flash (Gemini, 2025) as the LLM-as-Judge in evaluation. We compare Self-Manager against several categories of baselines: (1) *search-augmented LLMs*; (2) *single agent*, referring to agent loops such as ReAct (Yao et al., 2022), ReSum (Wu et al., 2025b), and FoldAgent (Sun et al., 2025). Single agents are configured with a maximum context window of 128k tokens, using Serper[1] as the search tool and Jina Reader[2] as the browsing tool; (3) *multi-agent workflows*, which consist of fixed, predefined pipelines of multiple agents, including LangChain-Open-DeepResearch (LangChain, 2025) and AI-Q Research Assistant (NVIDIA, 2025); (4) *proprietary deep research agents*, namely state-of-the-art deep research systems, including Gemini Deep Research (Google, 2025) and OpenAI DeepResearch (OpenAI, 2025).

Results. As shown in Table 1, Self-Manager outperforms all single-agent architectures and multi-agent systems, while further narrowing the performance gap with proprietary deep research agents. These findings highlight the advantage of Self-Manager's parallel agent loop in handling difficult long-form deep research tasks. In addition, we assess Self-Manager with base models of varying sizes and training strategies. We observe that stronger model capabilities correspond to better performance, implying that Self-Manager does not limit the expressive power of the base model and adheres to scaling rules.

3.2 RQ2: Ablation Studies

This section presents a series of ablation studies centered on the design of Self-Manager, with the goal of validating the necessity and effectiveness of its core components.

Setup. We conduct ablations at two levels. The first level targets the agent loop design of Self-Manager, with the following baselines: (1) *without asynchrony*, where the main thread becomes fully

[1] https://serper.dev/
[2] https://jina.ai/reader/

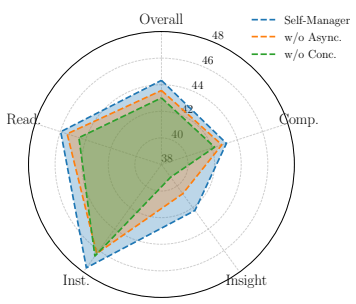

Figure 3: Ablation results on the asynchronous and concurrent capabilities of Self-Manager, showing performance degradation.

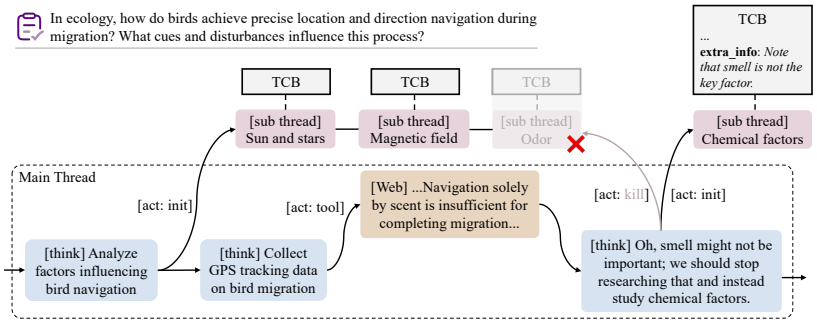

Figure 4: A Case in Asynchronous Parallelism of Self-Manager. It shows that the main thread determines that a running subthread is no longer necessary for the overall task and then terminates it. This self-management mechanism enables early stopping, making execution more adaptive and efficient.

blocked whenever subthreads are running, and the agent loop resumes only after all subthreads have completed; (2) *without concurrency*, where at most one thread is allowed to run and only one subthread can be created at any time. The second level focuses on the design of the TCB, with the following baselines: (1) *without assigned_tool*, where the TCB does not specify the set of tools available to the current thread, and the thread is instead allowed to access all tools available to the main thread; (2) *without extra_info*, where the main thread does not provide any additional information to subthreads; (3) *prefix_context variants*, which have four designs: providing the main thread's full, no, or summarized prefix context, or allowing the main thread to freely select one of the three choices listed above.

Results. The results in Figures 3 and 5 show that Self-Manager consistently outperforms all ablation baselines across multiple metrics on DeepResearch Bench, demonstrating the effectiveness of its asynchronous execution, concurrency, and TCB-based management design. While some variants achieve performance close to that of the original Self-Manager, this does not imply design redundancy, which can be verified by the following cases.

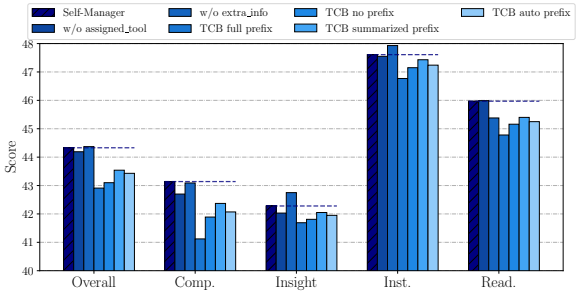

Figure 5: Ablation study on the design of the Thread Control Block (TCB).

Case Study. As illustrated by the case in Figure 4, when subthreads are running, the asynchronous main thread continues to perform work that makes tangible contributions to the overall task. Specifically, the main thread identifies that the subtask being executed by a running subthread is unnecessary and thus performs an early stop, preventing wasted computational resources. Moreover, the *extra_info* field can occasionally encode note-like hints that better guide subthreads in completing their assigned subtasks. Therefore, although no absolute performance gain, we retain these design choices in order to improve the adaptiveness.

3.3 RQ3: Contextual Capabilities

Information Retention. For loops in which the context may undergo changes, it is necessary to consider the information loss rate at each context transition. To quantify information loss, we adopt the LLM-as-Judge to dynamically maintain a list of valid information units. At each context change, we compare the information sets before and after the transition as follows:

$$\mathcal{L}_{\text{info}} = 1 - \frac{\left|\mathcal{I}_{|} \cap \tilde{\mathcal{I}}_{|}\right|}{\left|\mathcal{I}_{|}\right|}, \tag{4}$$

where $\mathcal{I}_{|}$ and $\tilde{\mathcal{I}}_{|}$ denote the sets of information units before and after the j-th context change, respectively. More details are provided in Appendix C. Experimental results in Table 2 show that Self-Manager achieves a lower loss rate through context isolation, while the main thread maintains a consistently smaller accumulated context length.

Long-Horizon. We measure the average count of execution turns under different context window length constraints to assess how well the agent loop

Table 2: Context capability assessment. C.C. denotes the count of context changes, including summarization, context folding, and subthread creation. Avg. PL refers to the average peak context length, and Avg. L denotes the average context length after execution.

Model	$\mathcal{L}_{\text{info}} \downarrow$	**C.C.**	**Avg. PL**	**Avg. L**
ReAct	-	-	13.1k	13.1k
ReSum	25.23%	2.7	14.2k	15.7k
FoldAgent	19.24%	5.6	9.5k	9.5k
Self-Manager	**11.53%**	7.0	10.2k	10.2k

supports long-horizon execution. The results in Figure 6 show that the main thread of Self-Manager consistently runs for more rounds, which can stem from its subtask delegation mechanism and design that allows the main thread to focus solely on the overall task, resulting in slower context accumulation and more long-horizon planning.

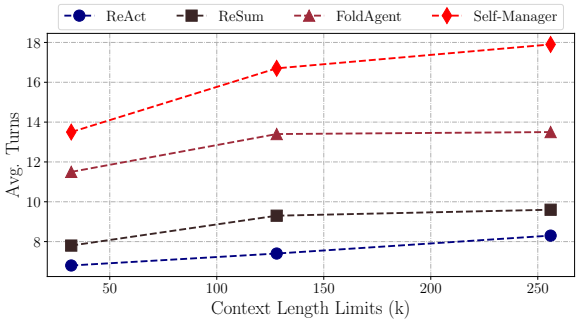

Figure 6: Comparison of the average turns across agent loops under varying context window length limits.

Case Study. Figure 7 illustrates an example of how the context length evolves over time. Vanilla ReAct exhibits linear growth and is prone to exceeding the limit. ReSum and FoldAgent achieve non-monotonic accumulation through compression and folding. In contrast, Self-Manager leverages concurrent subthreads and context isolation, leading to a reduced context accumulation rate in the main thread and exhibiting long-horizon capacity.

3.4 RQ4: Efficiency and Cost

This section examines the runtime efficiency and cost of Self-Manager.

Setup. To fairly measure execution time and costs, we adopt a trajectory-based accounting method that abstracts away external factors such as network fluctuations. Specifically, each agent's think and act steps are quantified based on the

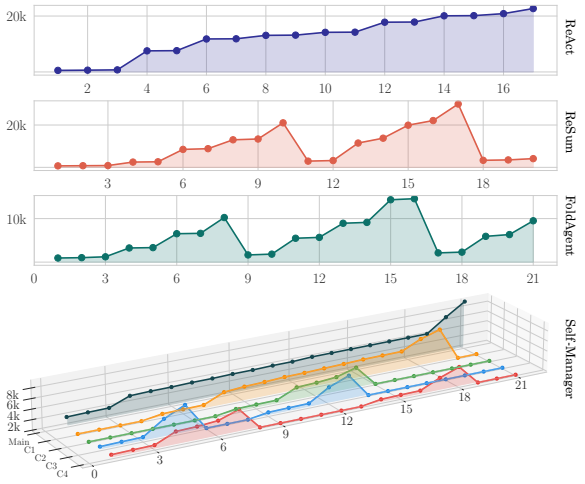

Figure 7: Examples of context accumulation across agent loops. The x-axis denotes the turns, while the y-axis denotes the context length. In Self-Manager, "Main" represents the main thread's context, whereas "C" entries correspond to the slots of subthreads. In this example, up to four subthreads run concurrently.

lengths of prefix tokens, generated tokens, and the number of tool invocations, as formalized below:

$$T_{\text{total}} = \sum T_{\text{pre}} + \sum T_{\text{gen}} + \sum_{\text{tool}_i \in \mathcal{T}} \sum T_{\text{tool}_i} \tag{5}$$

$$C_{\text{total}} = \sum C_{\text{pre}} + \sum C_{\text{gen}} + \sum_{\text{tool}_i \in \mathcal{T}} \sum C_{\text{tool}_i} \tag{6}$$

Calculating details are provided in Appendix C.

Results. As indicated in Table 3, relative to FoldAgent, Self-Manager achieves better task performance while demonstrating controllable runtime and cost. Although the overall overhead of Self-Manager is higher, superior performance on long-form deep research tasks generally comes at the expense of increased tool invocations, which we consider acceptable.

Table 3: Comparison of efficiency and cost between Self-Manager and FoldAgent. RN denotes the number of subtask result returns.

Model	**Time** (s)	**Cost** ($)	**Tool Calls**	**RN**
FoldAgent	117.27	0.05	18.25	5.6
Self-Manager	129.62	0.09	20.30	7.0
w/o Async	193.08	0.07	21.63	7.2
w/o Conc	144.55	0.07	19.37	6.6

Table 4: Results on BrowseComp-Plus.

Model	Acc. (Pass@1)	Recall	Tool Calls
LLMs w/ Search			
Qwen3-30B-A3B	5.32%	6.70%	1.00
Single Agent (Qwen3-30B-A3B)			
ReAct	28.47%	37.33%	5.79
ReSum	31.24%	38.90%	6.32
FoldAgent	35.65%	43.04%	12.15
Self-Manager	**36.17%**	**45.39%**	18.45

3.5 RQ5: Generalization Ability

This section investigates the generalization of Self-Manager across other types of tasks.

Setup. Although Self-Manager is inherently designed for complex tasks requiring subtask decomposition, its general architecture can be applied to a wide range of scenarios. To evaluate this, we select BrowseComp-Plus (Chen et al., 2025b), a benchmark requiring multi-hop deep search.

Results. The experiments show that Self-Manager achieves the best performance among single agents on this benchmark. This can be attributed to the parallel agent loop, which not only allows fine-grained management of context to reduce the difficulty of complex tasks but also adapts to the task complexity by controlling whether subthreads are created or if the main thread alone suffices. These findings indicate that Self-Manager exhibits generalization and adaptability across complex deep research tasks.

4 Related Work

Web Agents. While Retrieval-Augmented Generation (RAG; Lewis et al., 2021; Gao et al., 2024; Xu et al., 2025a) improves the factuality of Large Language Models (LLMs; Naveed et al., 2025), its fixed and passive retrieval paradigms remain limited. In contrast, web agents (Wu et al., 2025a; Li et al., 2025a) adopt an agent-centric architecture to perform autonomous and adaptive multi-turn information seeking and synthesis. Li et al. (2025c,d) integrate reasoning with web tool invocation, leading to notable performance improvements. Jin et al. (2025); Song et al. (2025); Gao et al. (2025) further enhance web agents through reinforcement learning. Zhang et al. (2025b,a) investigate process-level rewards to increase reward density, thereby providing more fine-grained guidance. Currently, most web agents primarily address multi-hop QA tasks, such as HotpotQA (Yang et al., 2018) and MuSiQue (Trivedi et al., 2022).

Deep Research. Compared to traditional QA, deep research is more complex and must contend with information at web scale. Zheng et al. (2025); Mialon et al. (2023) trains and evaluates deep research agents on real-world web corpora. Existing deep research tasks can be broadly categorized into two types. The first type is short-form, where the target answer is typically a short entity, in order to assess the information-seeking capacity by issuing queries with numerous constraints (Wei et al., 2025; Chen et al., 2025b,a). The second type is long-form, where the desired output is usually a report (Du et al., 2025; Xu et al., 2025b; Wang et al., 2025). Deep research tasks generally require dedicated multi-agent workflows to be effectively addressed (Prabhakar et al., 2025; Hong et al., 2025). Our proposed Self-Manager aims to further optimize agent modeling through parallelized execution, addressing limitations of prior solutions such as context interference, information loss, and the lack of runtime adaptability.

5 Conclusion

In this paper, we present Self-Manager, a novel parallel agent loop architecture designed for complex, long-horizon deep research tasks. By introducing a thread-oriented design, Self-Manager enables a main thread to orchestrate multiple subthreads with three key features: asynchrony, concurrency, and context isolation. This paradigm effectively mitigates the limitations of traditional sequential loops, such as linear context accumulation, mutual subtask interference, and execution blocking. Central to our architecture is the Thread Control Block (TCB) mechanism, which provides a structured interface for the main thread to monitor and manage subthread states iteratively, ensuring superior runtime adaptability and parallel efficiency.

Extensive evaluations on the DeepResearch Bench demonstrate that Self-Manager consistently outperforms existing single-agent and multi-agent baselines. Our analytical experiments further validate the necessity of its design components and highlight its significant advantages in contextual capacity, operational efficiency, and cross-domain generalization. We believe that Self-Manager offers a scalable and flexible foundation for the next generation of autonomous research agents.

Limitations

Although extensive experiments were conducted to validate the effectiveness of the proposed Self-Manager, the evaluation protocols of the adopted benchmarks rely heavily on LLM-as-Judge. This reliance may introduce judgment bias and also incur substantial computational cost and time overhead. In addition, our experiments are primarily conducted on queries and web corpora in Chinese and English, without evaluating performance in environments involving a broader range of languages. Furthermore, this study primarily focuses on long-form deep research tasks. Extending the evaluation to a broader range of tasks is a promising direction for future research.

Ethics Statement

All models tested or used in this work are either publicly accessible open-source models or commercially available models. All prompt templates used in the research are safe, and all data produced meets ethical standards. Furthermore, all datasets and corpora employed in this study are publicly available, and we ensure that all data is used in compliance with the original authors' usage and privacy rules. Additionally, all invoked tool APIs are publicly accessible and explicitly support academic use. Moreover, these APIs incorporate robust safety mechanisms, ensuring that returned content is filtered to remove non-compliant or unethical material.

References

Kaiyuan Chen, Yixin Ren, Yang Liu, Xiaobo Hu, Haotong Tian, Tianbao Xie, Fangfu Liu, Haoye Zhang, Hongzhang Liu, Yuan Gong, Chen Sun, Han Hou, Hui Yang, James Pan, Jianan Lou, Jiayi Mao, Jizheng Liu, Jinpeng Li, Kangyi Liu, and 14 others. 2025a. xbench: Tracking agents productivity scaling with profession-aligned real-world evaluations. Preprint, arXiv:2506.13651.

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.

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

Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu, and Yi Wu. 2025. Beyond ten turns: Unlocking long-horizon agentic search with large-scale asynchronous rl. Preprint, arXiv:2508.07976.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-augmented generation for large language models: A survey. Preprint, arXiv:2312.10997.

Gemini. 2025. gemini-2.5-flash-preview. https://ai.google.dev/gemini-api/docs/models?hl=zh-cn#gemini-2.5-flash-preview, Accessed on 2025-06-20.

Google. 2025. Gemini deep research. https://gemini.google/overview/deep-research/. Accessed: 2025-05-08.

Haoyang Hong, Jiajun Yin, Yuan Wang, Jingnan Liu, Zhe Chen, Ailing Yu, Ji Li, Zhiling Ye, Hansong Xiao, Yefei Chen, and 1 others. 2025. Multi-agent deep research: Training multi-agent systems with m-grpo. arXiv preprint arXiv:2511.13288.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. Preprint, arXiv:2503.09516.

LangChain. 2025. Open deep research. Technical report, LangChain.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. Preprint, arXiv:2005.11401.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, Weizhou Shen, Junkai Zhang, Dingchu Zhang, Xixi Wu, Yong Jiang, Ming Yan, Pengjun Xie, Fei Huang, and Jingren Zhou. 2025a. Websailor: Navigating super-human reasoning for web agent. Preprint, arXiv:2507.02592.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. 2025b. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. Preprint, arXiv:2507.18224.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. 2025c. Search-o1: Agentic search-enhanced large reasoning models. arXiv preprint arXiv:2501.05366.

Xiaoxi Li, Jiajie Jin, Guanting Dong, Hongjin Qian, Yongkang Wu, Ji-Rong Wen, Yutao Zhu, and Zhicheng Dou. 2025d. Webthinker: Empowering large reasoning models with deep research capability. arXiv preprint arXiv:2504.21776.

Long Lian, Sida Wang, Felix Juefei-Xu, Tsu-Jui Fu, Xiuyu Li, Adam Yala, Trevor Darrell, Alane Suhr, Yuandong Tian, and Xi Victoria Lin. 2025. Threadweaver: Adaptive threading for efficient parallel reasoning in language models. Preprint, arXiv:2512.07843.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. Preprint, arXiv:2311.12983.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A comprehensive overview of large language models. ACM Transactions on Intelligent Systems and Technology, 16(5):1–72.

Xuan-Phi Nguyen, Shrey Pandit, Revanth Gangi Reddy, Austin Xu, Silvio Savarese, Caiming Xiong, and Shafiq Joty. 2025. Sfr-deepresearch: Towards effective reinforcement learning for autonomously reasoning single agents. Preprint, arXiv:2509.06283.

NVIDIA. 2025. Ai-q research assistant. Technical report, NVIDIA.

OpenAI. 2025. Deep research system card. Technical report, OpenAI.

Akshara Prabhakar, Roshan Ram, Zixiang Chen, Silvio Savarese, Frank Wang, Caiming Xiong, Huan Wang, and Weiran Yao. 2025. Enterprise deep research: Steerable multi-agent deep research for enterprise analytics. Preprint, arXiv:2510.17797.

Philip Schroeder, Nathaniel W. Morgan, Hongyin Luo, and James R. Glass. 2025. 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), pages 8418–8442, Albuquerque, New Mexico. Association for Computational Linguistics.

Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. 2025. R1-searcher: Incentivizing the search capability in llms via reinforcement learning. arXiv preprint arXiv:2503.05592.

Yu Su, Diyi Yang, Shunyu Yao, and Tao Yu. 2024. Language agents: Foundations, prospects, and risks. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts, pages 17–24, Miami, Florida, USA. Association for Computational Linguistics.

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.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. musique: Multi-hop questions via single-hop question composition. Transactions of the Association for Computational Linguistics, 10:539–554.

Jiayu Wang, Yifei Ming, Riya Dulepet, Qinglin Chen, Austin Xu, Zixuan Ke, Frederic Sala, Aws Albarghouthi, Caiming Xiong, and Shafiq Joty. 2025. Liveresearchbench: A live benchmark for user-centric deep research in the wild. Preprint, arXiv:2510.14240.

Yimeng Wang, Jiaxing Zhao, Hongbin Xie, Hexing Ma, Yuzhen Lei, Shuangxue Liu, Xuan Song, Zichen Zhang, and Haoran Zhang. 2026. Metagen: Self-evolving roles and topologies for multi-agent llm reasoning. Preprint, arXiv:2601.19290.

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. Preprint, arXiv:2504.12516.

Jialong Wu, Baixuan Li, Runnan Fang, Wenbiao Yin, Liwen Zhang, Zhengwei Tao, Dingchu Zhang, Zekun Xi, Gang Fu, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. 2025a. Webdancer: Towards autonomous information seeking agency. Preprint, arXiv:2505.22648.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. 2025b. Resum: Unlocking long-horizon search intelligence via context summarization. Preprint, arXiv:2509.13313.

Yilong Xu, Jinhua Gao, Xiaoming Yu, Yuanhai Xue, Baolong Bi, Huawei Shen, and Xueqi Cheng. 2025a. Training a utility-based retriever through shared context attribution for retrieval-augmented language models. arXiv preprint arXiv:2504.00573.

Yilong Xu, Xiang Long, Zhi Zheng, and Jinhua Gao. 2025b. Ravine: Reality-aligned evaluation for agentic search. arXiv preprint arXiv:2507.16725.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In Proceedings of the 2018 conference on empirical methods in natural language processing, pages 2369–2380.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In The eleventh international conference on learning representations.

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. 2025. Agentfold: Long-horizon web agents with proactive context management. Preprint, arXiv:2510.24699.

Qi Zhang, Shouqing Yang, Lirong Gao, Hao Chen, Xiaomeng Hu, Jinglei Chen, Jiexiang Wang, Sheng Guo, Bo Zheng, Haobo Wang, and Junbo Zhao. 2025a. Lets: Learning to think-and-search via process-and-outcome reward hybridization. Preprint, arXiv:2505.17447.

Wenlin Zhang, Xiangyang Li, Kuicai Dong, Yichao Wang, Pengyue Jia, Xiaopeng Li, Yingyi Zhang, Derong Xu, Zhaocheng Du, Huifeng Guo, Ruiming Tang, and Xiangyu Zhao. 2025b. Process vs. outcome reward: Which is better for agentic rag reinforcement learning. Preprint, arXiv:2505.14069.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. Preprint, arXiv:2312.07104.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. 2025. Deepresearcher: Scaling deep research via reinforcement learning in real-world environments. Preprint, arXiv:2504.03160.

A Introduction of Classical Agent Loops

Some prior work allows the agent to summarize the context either during the act step or when approaching the context-window limit (Wu et al., 2025b), leading to the modified trajectory:

$$(\ldots, o_{i-1}, a_i = \text{sum}) \rightarrow \left(h'_{i-1}, a_{i+1}, o_{i+1}, \ldots\right), \tag{7}$$

where h'_{i-1} represents the compressed historical context before turn i. Yet this introduces a fundamental difficulty: summarization performs lossy compression, and selecting which information to preserve is inherently challenging.

To alleviate this, FoldAgent (Sun et al., 2025) proposes to fold the context along subtask boundaries, thereby reducing linear context growth at the granularity of clearly useful information. Formally:

$$(a_i, \underbrace{o_i, \ldots, a_j}_{\text{subtask}}, o_j) \xrightarrow{\text{Folding}} (a_i, o_j, a_{j+1}, o_{j+1} \ldots). \tag{8}$$

However, as the subtask and the global trajectory still share the same context window, information interference persists, and subtask execution remains synchronous and blocking, which limits efficiency. To address the above limitations, we present Self-Manager, a novel parallel agent loop.

B Implementation Details

In this section, we provide additional details on the Self-Manager method and its implementation.

B.1 Prompt Templates

First, the main thread uses a system prompt template that supports not only basic tool invocation but also actions such as thread creation and management, as seen below. Note that the tool definition section of the following prompt template has been relocated to Table 5.

Prompt Template for Main Thread

You are an advanced agent capable of creating subthreads, specifically designed to perform deep research tasks. As the main thread, you operate based on the standard ReAct Loop: Think-Act-Observe. During the Act phase, you may call tools or create subthreads to complete the subtasks you assign. You excel at constructing and managing subthreads, enabling them to focus on researching specific subtopics or to carry out detailed writing for particular sections of the final report.

Task Description:
Given a user's question, your task is to think iteratively based on the question, search for and integrate external web information, and ultimately produce a comprehensive, in-depth, and well-structured long-form report. When you have gathered sufficient information and are ready to provide the definitive long-form report, you must enclose the entire report within <answer></answer> tags.

Available Tools:
You may call a tool function in each turn to assist with the user query. You are provided with function signatures within <tools></tools> XML tags:
<tools>...</tools>
For each function call, return a json object with function name and arguments within <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>

Observe:
- After you invoke a tool, the Observe phase will provide you with the tool-invocation details, including the returned result and any potential errors, which are enclosed within the <tool_response></tool_response> XML tags.
- In addition, you can also see the list of TCBs (Thread Control Blocks), each corresponding to the current state of a subthread created by the main thread. Each TCB includes: Thread ID, Target, Status (Running, Success, Failed, Killed), Allowed Tools, Assigned Context, Runtime, and Result (available only after the thread has completed execution).
- With the observation information, you can continue to determine your next Action in the loop.

Report Requirements:
1. Your report must be in Markdown format, well-structured, and fluent.
2. Your report must align with the intent of the user's question, and can comprehensively address the question.
3. Your report should not simply be a list of arguments. For each point of your report, it's not enough to just state the argument — you need to provide in-depth analysis, causal reasoning, impacts and trends analysis, solutions, and so on. In short, make the description more detailed and substantial.
4. Your report should include Markdown-formatted citations for all referenced web sources. For example: ([title](url)).
5. The language of your report should be consistent with the language of the user's questions.
6. You must enclose the entire report within <answer></answer> tags.

User's task: `{user_task}`.

Specifically, the definitions of all actions allowed in the main thread loop are provided in Table 5, including the basic retrieval tools used for deep research as well as operations for managing subthreads.

In contrast, the system prompt template for subthreads does not need to consider thread-related actions. Instead, it focuses on implementing a basic loop that integrates tool-integrated reasoning and ReAct's "Think–Act-Observe" paradigm.

Prompt Template for Subthread

You are a deep research assistant. You can operate based on the standard ReAct Loop: Think-Act-Observe. You are responsible for completing the task assigned to you. This task is typically part of a deep-research task, but you should remain fully focused on the specific task given to you and disregard anything outside its scope. You must ensure that your output strictly satisfies all requirements of the task.

Requirements:
1. You are allowed to call tools, but only the tools explicitly specified by the user. You must not call any tools outside of the ones provided.
2. You may perform multiple iterations to pursue deeper investigation and achieve higher-quality results.
3. Your submitted results are recommended to be in Markdown format. When referencing web information, include Markdown-formatted citations for all sources. For example: ([title](url)).
4. When you determine that the task can be considered complete, you must enclose the entire submission within <answer></answer> tags.
5. The language of your submission text should be consistent with the language of the task.
6. Your report should not simply be a list of items; it should provide analysis and causal support for each point or information, and offer solutions when necessary.

Available Tools:
You may call a tool function in each turn to assist with the user query. You are provided with function signatures within <tools></tools> XML tags: <tools>...</tools>
Note that the tools listed above are the complete set; you can only call the tools explicitly specified by the user. For each function call, return a json object with function name and arguments within <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>

Your task: `{goal}`.
Your allowed tools: `{allowed_tools}`.
Your assigned context: `{assigned_context}`.
Extra info: `{extra_info}`.

In addition, we incorporate runtime error protection into the implementation of Self-Manager. Specifically, when the execution is about to exceed the context window limit, we invoke an external LLM to compress the existing contextual content and replace the original context history with the compressed version. We use Gemini-2.5-Flash as the summarization model, with the prompt shown below.

Prompt Template for Context Compression

You are an expert at analyzing conversation history and extracting relevant information. Your task is to thoroughly evaluate the conversation history and current question to provide a comprehensive summary that will help answer the question.

Task Guidelines
1. Information Analysis:
- Carefully analyze the conversation history to identify truly useful information.
- Focus on information that directly contributes to answering the question.
- Do NOT make assumptions, guesses, or inferences beyond what is explicitly stated in the conversation.
- If information is missing or unclear, do NOT include it in your summary.

2. Summary Requirements:
- Extract only the most relevant information that is explicitly present in the conversation.
- Synthesize information from multiple exchanges when relevant.
- Only include information that is certain and clearly stated in the conversation.
- Do NOT output or mention any information that is uncertain, insufficient, or cannot be confirmed from the conversation.
3. Output Format: Your response should be structured as follows:

<summary>
- Essential Information: [Organize the relevant and certain information from the conversation history that helps address the question.]
</summary>

Strictly avoid fabricating, inferring, or exaggerating any information not present in the conversation. Only output information that is certain and explicitly stated.

Question: {question}
Conversation History: {recent_history_messages}
Please generate a comprehensive and useful summary. Note that you are not permitted to invoke tools during this process. Use the language of the question to generate the summary.

Table 5: The action list of the main thread in Self-Manager. It encompasses tools for deep research tasks as well as operations for subthread management.

Action	Definition
search	{"type": "function", "function": {"name": "search", "description": "Perform Google web searches then returns a string of the top search results. Accepts multiple queries.", "parameters": {"type": "object", "properties": {"query": {"type": "array", "items": {"type": "string", "description": "The search query."}, "minItems": 1, "description": "The list of search queries."}}, "required": ["query"]}}}
visit	{"type": "function", "function": {"name": "visit", "description": "Visit webpage(s) and return the summary of the content.", "parameters": {"type": "object", "properties": {"url": {"type": "array", "items": {"type": "string"}, "description": "The URL(s) of the webpage(s) to visit. Can be a single URL or an array of URLs."}, "goal": {"type": "string", "description": "The specific information goal for visiting webpage(s)."}}, "required": ["url", "goal"]}}}
branch	{"type": "function", "function": {"name": "branch", "description": "Create a subthread to perform a specific task.", "parameters": {"type": "object", "properties": {"id": {"type": "string", "description": "The ID of the subthread. You can generate it freely according to your own habits."}, "target": {"type": "string", "description": "The target of the subthread. It must be specific and useful to the user's task."}, "allowed_tools": {"type": "array", "items": {"type": "string", "description": "The name of an allowed tool for the subthread."}, "minItems": 1, "description": "The list of allowed tools for the subthread."}, "assigned_context": {"type": "string", "description": "The history context assigned by main thread for the subthread."}, "extra_info": {"type": "string", "description": "Any extra information that the main thread wants to provide to the child thread."}}, "required": ["id", "target", "allowed_tools", "assigned_context"]}}}
sleep	{"type": "function", "function": {"name": "sleep", "description": "Sleep for a specified duration when you think the only thing to do is wait for the subthread to complete its task.", "parameters": {"type": "object", "properties": {"sleep_duration": {"type": "number", "description": "The duration in seconds to sleep. Maximum 60 seconds"}}, "required": ["sleep_duration"]}}}
kill	{"type": "function", "function": {"name": "kill", "description": "Kill a running subthread from the TCB list when you think it is no longer needed.", "parameters": {"type": "object", "properties": {"id": {"type": "string", "description": "The ID of the subthread to kill."}}, "required": ["id"]}}}
delete	{"type": "function", "function": {"name": "delete", "description": "Delete the information of a finished subthread from the TCB list when you think it is no longer needed.", "parameters": {"type": "object", "properties": {"id": {"type": "string", "description": "The ID of the subthread to delete."}}, "required": ["id"]}}}

B.2 Definition of TCB

The design of the Thread Control Block (TCB) is inspired by the TCB in operating systems, which serves as a core data structure for managing threads. In Self-Manager, the TCB functions as a metadata encapsulation structure, providing the main thread with real-time information about subthreads to facilitate their management. Unlike in operating systems, where TCBs need to handle low-level resources such as pointers, TCBs in Self-Manager retain only the most essential fields for simplicity and efficiency, including:

- *id*: A unique identifier for a subthread, such as "Thread_01" or "Thread_US".
- *goal*: The subtask execution target assigned to a subthread by the main thread.
- *state*: Indicator of the current execution status of the thread, including "running," "successful," "failed," or "killed."
- *allowed_tools*: The list of tools that the subthread is permitted to use.
- *prefix_context*: The prefix context assigned to the subthread by the main thread, with its content and format determined entirely by the main thread.
- *extra_info*: Any additional remarks specified by the main thread, and this field is optional.
- *start_time*: The timestamp when the subthread is created and starts running, used to calculate and display its elapsed runtime.
- *result*: The result returned by the subthread upon completion, corresponding to the task described in the goal field. This field is populated only after the subthread's status is no longer "running."

B.3 Inference

All reasoning processes of the Self-Manager and the agent LLMs used in all baselines are implemented based on the SGLang framework (Zheng et al., 2024). All experiments are conducted on NVIDIA H100 GPUs. Both the tensor parallelism size and the expert parallelism size are set to 8. We use `torch.bfloat16` as the numerical precision for model weights, activations, and the KV cache during inference.

C Details of Experiment

In this section, we present the implementation details of selected experiments and discuss other related experiments.

C.1 Details of RQ3

In RQ3, we investigate the issue of information retention. In context management modeling that supports long-horizon execution, any form of context transformation—such as summarization, folding, or the return of subthreads in a self-manager—may lead to the loss of critical information, which in turn degrades system performance. Therefore, it is necessary to design targeted experiments focusing on information retention to systematically examine this problem.

Specifically, based on the above problem formulation, we quantitatively evaluate information retention by invoking an LLM-as-Judge, as formalized in Equation 4. For a single execution trajectory, we focus on the positions where context transitions occur. At each such position, we prompt the LLM-as-Judge—using the template shown below—to extract a list of useful information units from the contexts before and after the transition, where usefulness is defined with respect to the task being solved in the current execution. We then invoke the LLM-as-Judge again to determine which information units from the pre-transition list are covered (matched) by the post-transition information units. A higher number of matched units indicates stronger information retention capability. Since ReAct does not involve any context transitions, it is excluded from this analysis. For ReSum, we compute information retention by comparing the contexts immediately before and after the agent loop invokes the summarization tool. For FoldAgent, we compare the branch execution context with the branch's final answer, treating them as the pre- and post-transition contexts, respectively. For Self-Manager, we compare the context of a subthread during execution with the returned result after the subthread completes, which serve as the pre- and post-transition contexts.

Prompt Template for Information Extraction

You are a highly professional information extractor, skilled at identifying information points that are useful for a given task from long texts.

What Your Need to Do:
Given a task description and the contextual information produced by an Agent while completing that task, extract the information points from the context that are useful for the task.

Definition of an information point:
An information point should be a complete and unique statement of a fact (a sentence of about 10-20 words), and should include a clear subject, verb, and object, and, if necessary, constraint information such as time, location, topic, etc.

Definition of "useful":
If an information point is at least semantically relevant to the task description, it is considered useful.

Output Format:
- Output one plain-text nugget per line, with no other content.
- If no complete statement that is valuable to the task can be found in the passage, do not generate any low-quality nuggets, and simply return [None].
- Do not provide explanations, and ensure there is no redundant information.

Task Description: `{task_description}`
Context Information: `{context_info}`

Prompt Template for Information Union

Task Description:
Given two Information Point Lists, A and B, you need to produce the union of the two Information Point Lists.

Definition of Information Point Lists:
- Each information point list contains multiple information points. Each information point is typically a complete and unique statement of a fact (a sentence of about 10-20 words).
- The number of information points in each list may vary.

Procedure:
1. Considering information points in List A, examine each information point one by one to determine whether it exists in List B. An information point is considered to exist as long as the fact it refers to is semantically supported by one or more information points in List B.
2. Output all information points from List A that are determined to exist in List B.

Output Format:
- One information point per line, in plain text. The output must correspond exactly to the information points in List A and remain completely unchanged.
- If none of the information points in List A are found to exist in List B, simply return [None].
- Do not provide explanations, and ensure there is no redundant information.

Information Point List A:

```
{info_list_a}

Information Point List B:
{info_list_b}
```

C.2 Details of RQ4

For the computation of efficiency and cost, to ensure a fair comparison—by eliminating the influence of network APIs or local environment variability—we adopt a trajectory-based accounting approach, where all metrics are computed directly from the execution traces.

Time. The agent runtime primarily consists of the time spent on base model invocations and tool calls. In FoldAgent, we assume that the folding operation—i.e., discarding a branch's execution trace after the branch returns its result—does not incur significant overhead. Consequently, the runtime is dominated by base model calls and tool invocations. In Self-Manager, although multiple threads may execute in parallel, under the constraints of the framework, the start and end times of the main thread correspond to the earliest start and the latest completion among all threads. Therefore, we use the main-thread trajectory as the basis for runtime computation. For tool calls, we consider the time for thread creation, termination, and deletion to be negligible. However, when the main thread explicitly invokes a sleep operation to block itself, the specified sleep duration is counted toward the total runtime. Specifically, the unit-time costs of the operations considered in the runtime calculation are summarized in Table 6. The token generation time can be calculated based on the processing speed, as follows:

$$\text{Speed} = \frac{\text{tokens}_{\text{prompt}} + \text{tokens}_{\text{generation}}}{\text{time}} \tag{9}$$

Table 6: Unit runtime of agent execution, which is referenced from SGLang (https://qwen.readthedocs.io/en/latest/getting_started/speed_benchmark.html#qwen3-30b-a3b-sglang). The context length limit is 128k. The quantization type is `torch.bfloat16`.

Item	Speed (tokens/s)	$\text{tool}_{\text{search}}$ (s)	$\text{tool}_{\text{visit}}$ (s)
Time	1385.65	1.0	2.0

Cost. The agent execution cost is likewise dominated by base model invocations and tool calls. Although Self-Manager supports concurrency and asynchronous execution, it essentially renders base model calls asynchronous and parallel; as a result, the overall cost still primarily depends on the deployment cost of the base model. Specifically, the unit costs of the operations considered in the execution cost calculation are summarized in Table 7.

C.3 Discussion

Subtask. In Self-Manager, subtasks constitute a critical unit of processing. The main thread decomposes a complex task into multiple subtasks and can continuously discover new subtasks during execution. To enable independent execution, the main thread spawns child processes and dispatches subtasks to them. Accordingly, we investigate the behavior of the main thread when creating subtasks via spawning child threads. As illustrated in Figures 8 and 9, across a wide range of long-form deep research tasks, the types of subtasks assigned by the main thread span a diverse spectrum, including information gathering, problem analysis, material organization, and writing. Among these, research subtasks focused on information acquisition and analysis subtasks aimed at reasoning about and solving specific subproblems account for the majority. This observation is expected, as such subtasks are fundamental to long-form deep research: nearly any complex problem can be decomposed into research- and analysis-oriented subtasks. Meanwhile, we argue that models not explicitly trained to operate within a parallel agent loop are still limited in their ability to effectively leverage the highly flexible action of spawning child threads. A

Table 7: Base pricing of agent runtime overhead. Token prices are based on Together.ai (https://www.together.ai/pricing), and search tool costs are based on Serper (https://serper.dev/).

Item	$\text{token}_{\text{prompt}}$ (\$/1M)	$\text{token}_{\text{generation}}$ (\$/1M)	$\text{tool}_{\text{search}}$ (\$/Call)	$\text{tool}_{\text{visit}}$ (\$/Call)
Price	0.80	0.80	0.001	0

promising direction for future work is to train models to generate a broader and more diverse set of subtasks, thereby improving the utility and effectiveness of child threads.

Figure 8: Word cloud of subthreads' subtasks assigned by the main thread in Self-Manager.

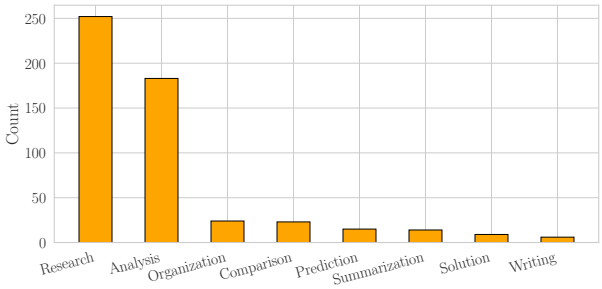

Figure 9: Frequency distribution of subtask types assigned to subthreads in Self-Manager.

Comparison with multi-agent systems. Self-Manager falls within the category of single-agent systems. One of its core motivations is to address the generalization limitations inherent in workflow-based multi-agent systems. Generalization is precisely where single-agent approaches excel: under a universal design paradigm, a single agent can, in principle, execute any type of task in any scenario. Theoretically, an iteratively composable atomic loop is sufficient to construct the graph structure required to complete arbitrary tasks. In this work, Self-Manager extends the classical single-threaded agent loop by introducing concurrent and asynchronous subthreads. However, it does not cross into the multi-agent regime for the following reasons. (1) The system relies on a single agentic LLM. Although multiple threads execute different tasks concurrently, they all share the same policy. (2) The loop design remains universally applicable and can generalize to arbitrary tasks across diverse scenarios. Furthermore, inference frameworks such as SGLang can efficiently support concurrent multi-threaded requests to deployed models without introducing additional inference overhead. Consequently, by enabling self-management on subthreads, Self-Manager preserves the generalization advantages of single-agent inference while further mitigating issues present in existing agent loops.