W&D: SCALING PARALLEL TOOL CALLING FOR EFFICIENT DEEP RESEARCH AGENTS

Xiaoqiang Lin* **Jun Hao Liew*** **Silvio Savarese** **Junnan Li**
Salesforce AI Research
{xiaoqiang.lin, junhao.liew, ssavarese, junnan.li}@salesforce.com

ABSTRACT

Deep research agents have emerged as powerful tools for automating complex intellectual tasks through multi-step reasoning and web-based information seeking. While recent efforts have successfully enhanced these agents by scaling depth through increasing the number of sequential thinking and tool calls, the potential of scaling width via parallel tool calling remains largely unexplored. In this work, we propose the **Wide and Deep research agent**, a framework designed to investigate the behavior and performance of agents when scaling not only depth but also width via parallel tool calling. Unlike existing approaches that rely on complex multi-agent orchestration to parallelize workloads, our method leverages intrinsic parallel tool calling to facilitate effective coordination within a single reasoning step. We demonstrate that scaling width significantly improves performance on deep research benchmarks while reducing the number of turns required to obtain correct answers. Furthermore, we analyze the factors driving these improvements through case studies and explore various tool call schedulers to optimize parallel tool calling strategy. Our findings suggest that optimizing the trade-off between width and depth is a critical pathway toward high-efficiency deep research agents. Notably, without context management or other tricks, we obtain 62.2% accuracy with GPT-5-Medium on BrowseComp, surpassing the original 54.9% reported by GPT-5-High. Our codes are available on https://github.com/SalesforceAIResearch/MCP-Universe/tree/main/mcpuniverse/benchmark/configs/deepresearch.

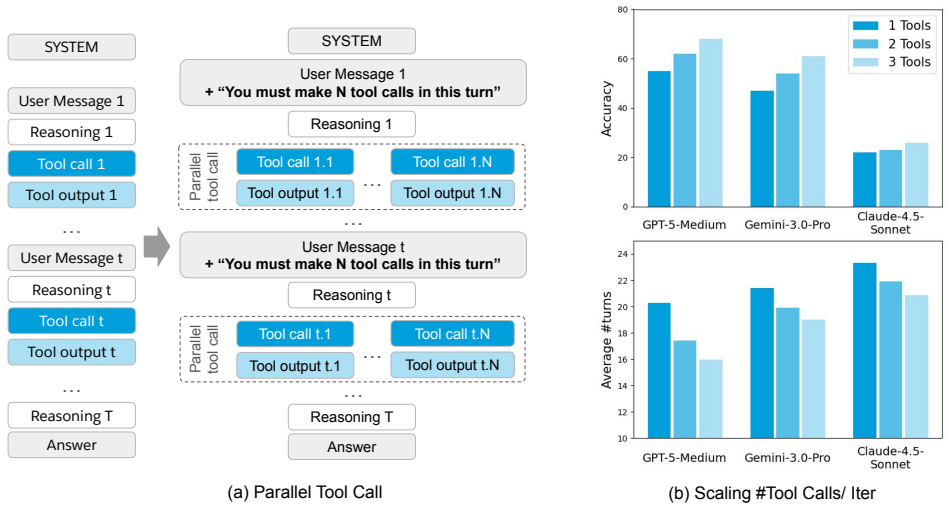
1 INTRODUCTION

Deep research agents ((OpenAI, 2025; Gemini Team, 2025; Team et al., 2025b; Liu et al., 2025; Moonshot AI, 2025; Team et al., 2025a)) have raised increasing interest in both applications and the research community due to their growing capabilities and potential to conduct multi-step research on the internet, liberating humans from complex intellectual tasks. These agents perform multi-step reasoning and information seeking to solve tasks that typically require hours of human work. Specifically, at each step, these agents perform reasoning followed by tool execution to search and retrieve information from the web. Ultimately, they provide a final answer or a full report that summarizes all the information gathered after multiple steps of reasoning and tool calling.

Recently, DeepSeek (Liu et al., 2025), MiroThinker (Team et al., 2025a) and LongCat (Meituan LongCat Team, 2026) have demonstrated the potential of scaling the depth of agent-environment interactions, *i.e.* increasing steps of thinking and tool calling to improve deep research capabilities. Consequently, recent works have focused on scaling context length or implementing smarter context management strategies to handle deeper reasoning and more tool calls. On the other hand, while many proprietary models (*e.g.* GPT-5, Gemini, Claude) support parallel tool calling, no work has yet explored the potential of scaling along the width dimension, *i.e.* making multiple tool calls in a single step, for deep research agents.

Existing works have explored alternative ways to scale the workload within each step. For example, Kimi-K2.5 (Moonshot AI, 2026) introduced a multi-agent framework to deploy multiple sub-agents

*Equal contribution.



(a) Parallel Tool Call (b) Scaling #Tool Calls/ Iter

Figure 1: (a) Single vs. parallel tool calling in a multi-step deep research agent trace. In parallel tool calling, the model performs a single reasoning step to issue multiple tool calls simultaneously; these calls are executed in parallel and their outputs are returned together into the agent's trace. (b) Top: Performance of different LLMs under parallel tool calling with varying # tool calls per step. Performance consistently improves as the # parallel tool calls increases across all models. Bottom: Average # turns required to complete the task with different # parallel tool calls. Increasing the # tool calls per iteration reduces the total # iterations needed to complete the deep research task.

to solve sub-tasks in a parallel manner; LongCat (Meituan LongCat Team, 2026) introduced parallel reasoning to generate multiple reasoning paths at the same turn and aggregate the outcomes via summarization. However, these approaches either rely on complex orchestration or overlook the coordination of distinct tool executions. In contrast, since parallel tool calling invoke multiple tools within a single reasoning step, it facilitates effective collaboration among the tool calls, improving the efficiency of information gathering.

In this work, we introduce the Wide and Deep (W&D) research agent to investigate the behavior and performance of state-of-the-art LLMs when jointly scaling depth (via more iterations) and width (via parallel tool calling). Moreover, we explore different tool call scheduling strategies to further boost agent performance on deep research tasks. In summary, we make the following contributions:

- We demonstrate that scaling width via parallel tool calling not only increases performance, but also reduces the number of turns required for the agent to find the correct answer.
- We analyze the driving factors behind why parallel tool calling improves performance by presenting several case studies.
- We study multiple tool call schedulers by employing different numbers of tool calls in each turn to further boost performance, suggesting the potential of dynamic tool calling.

2 Methodology

2.1 Parallel Tool Calling

State-of-the-art LLMs increasingly support the capability of *parallel tool calling* [1] [2] [3]. Denote the LLM as f_θ. To define this formally, let us first consider the standard sequential formulation of the

[1] https://platform.openai.com/docs/guides/function-calling?api-mode=chat#parallel-function-calling

[2] https://ai.google.dev/gemini-api/docs/function-calling?example=meeting#parallel_function_calling

[3] https://platform.claude.com/docs/en/agents-and-tools/tool-use/implement-tool-use#parallel-tool-use

agent trace. Given a user query X (which contains system prompt and the user question), at each step[4] t of the agent trace, a typical LLM outputs a reasoning thought R_t and a single tool call A_t. The environment executes this call and returns an observation O_t. The agent repeats this process until the final step T, where it produces a final answer $\hat{Y}$ instead of a tool call. Consequently, the full sequential agent trace τ_{seq} is defined as an ordered sequence:

$$\begin{aligned} \tau_{\text{seq}} &= \Big\langle X, (R_1, A_1, O_1), \ldots, (R_{T-1}, A_{T-1}, O_{T-1}), (R_T, \hat{Y}) \Big\rangle \\ R_t, A_t &= f_\theta(\langle X, (R_1, A_1, O_1), \ldots, (R_{t-1}, A_{t-1}, O_{t-1}) \rangle) \end{aligned} \tag{1}$$

Parallel tool calling extends this paradigm by allowing the agent to generate multiple tool calls simultaneously. At step t, rather than issuing a single action A_t, the model generates a set of m concurrent tool calls $\mathcal{A}_t^{\text{par}} = \{A_t^{(1)}, \ldots, A_t^{(m)}\}$. These are executed in parallel by the environment, yielding a corresponding set of observations $\mathcal{O}_t^{\text{par}} = \{O_t^{(1)}, \ldots, O_t^{(m)}\}$. The parallel agent trace τ_{par} is thus formalized as:

$$\tau_{\text{par}} = \Big\langle X, (R_1, \mathcal{A}_1^{\text{par}}, \mathcal{O}_1^{\text{par}}), \ldots, (R_{T-1}, \mathcal{A}_{T-1}^{\text{par}}, \mathcal{O}_{T-1}^{\text{par}}), (R_T, \hat{Y}) \Big\rangle \tag{2}$$

Figure 1 shows the difference in agent trace between the single tool calling and parallel tool calling. This approach allows the agent to aggregate significantly more information per interaction turn, thereby reducing the total number of steps required to solve the task. Furthermore, parallel execution offers dual efficiency benefits compared to performing m sequential steps. First, it amortizes the computational cost of reasoning; we condense m distinct reasoning traces into a single R_t, significantly reducing the number of decoding tokens and LLM generation latency. Second, because tool calls within the set $\mathcal{A}_t$ are executed concurrently, the wall-clock time spent waiting for environment feedback is minimized. In summary, parallel tool calling optimizes both the end-to-end latency of the agent rollout and the cost of LLM API usage by reducing both the iteration count and total token consumption.

Precise control of tool calling. To ensure the agent calls the exact number of tools specified, we explored two prompting strategies: 1) adding an instruction in the system message (query X) requesting m function calls per iteration, and 2) inserting a user message before each LLM call that specifies the required number of function calls. We chose the latter approach for its superior performance and more reliable tool-call consistency. Specifically, the agent trace is now represented as:

$$\tau_{\text{par}} = \Big\langle X, U_1, (R_1, \mathcal{A}_1^{\text{par}}, \mathcal{O}_1^{\text{par}}), \ldots, U_{T-1}, (R_{T-1}, \mathcal{A}_{T-1}^{\text{par}}, \mathcal{O}_{T-1}^{\text{par}}), U_T, (R_T, \hat{Y}) \Big\rangle \tag{3}$$

The detailed instruction used in U_t is illustrated in Figure 2.

User Instruction at Each Iteration

*At next step, if you need to make function calls, you MUST make at least **m** but not more than **m+1** function calls in a single response to gather information extensively.*

Figure 2: Prompt for controlling the number of tool calls in parallel tool calling.

3 Experimental results

3.1 Settings

Benchmarks. We conduct our experiments across three widely adopted deep-research benchmarks: 1) BrowseComp (Wei et al., 2025), 2) Humanity's Last Exam (HLE) (Phan et al., 2025), and 3) GAIA (Mialon et al., 2023). Because the evaluated models are not multimodal, we restrict our

[4]Throughout this work, the terms 'step,' 'iteration,' and 'turn' are used interchangeably.

testing to the text-only subsets of HLE (2,158 samples) and GAIA (103 samples). Furthermore, due to the high computational costs of full evaluations, performance on BrowseComp and HLE is reported based on the first 100 samples unless otherwise specified. We evaluate GPT-5 (Medium reasoning effort), Gemini 3.0 Pro, and Claude 4.5 Sonnet across these benchmarks.

Tool Environment. We adopt the agent framework from MCP-Universe (Luo et al., 2025) for evaluation. We provide the agent with three tools: 1) a Google-based search tool[5], 2) a scraping tool[6] with LLM summarization, and 3) a Python code interpreter. The scraping tool accepts a target URL and an extraction query as input; it scrapes the full content of the webpage and passes it, along with the query, to a summarization LLM to extract the specific information required. We use Gemini-2.5-Flash (with thinking mode disabled) for the summary model due to its balance of affordability and effectiveness. For BrowseComp and GAIA, we expose only the search and scraping tools since these benchmarks focus exclusively on information seeking, whereas for HLE, we expose all three tools to the agent. More implementation details can be found in the Appendix.

3.2 Main result

Tool call scaling. Figure 3 shows the performance of the deep research agent on BrowseComp. Specifically, Figure 3 (a) shows the performance of scaling depth (i.e., average # turns via max turn limit) for both single tool calling and parallel tool calling. Table 1 presents the tabular version of Figure 3 (a). The results indicate that parallel tool calling achieves the best performance with 3 tools per turn with better performance than the single tool calling. Meanwhile, the average number of turns for parallel tool calling shifts to the left, suggesting that parallel tool calling significantly reduces the number of turns required to finish the task. This significantly reduces the wall-clock time to finish the task and the LLM API costs due to the lower number of turns. For example, in single tool calling, to achieve 66% accuracy, it costs $\$102.5$ for 100 tasks in tool calling and LLM API fees and takes an average of 1522.6 s to finish the trace; whereas for parallel tool calling with 3 tool calls per turn, the model achieves 68% accuracy with a cost of $\$65.7$ for 100 tasks (*i.e.*, a 35.9% reduction) and a wall-clock time of 904.2 s (*i.e.*, a 40.6% reduction).

Figure 3 (b) shows the performance scaling of width (*i.e.*, # tool calls per turn via parallel tool calling) under different max turn limits. The results suggest that with a lower max turn limit (*e.g.*, 10), scaling the width consistently improves performance, whereas with a larger limit (*e.g.*, 100), the best performance is achieved with a moderate number of tool calls. This suggests that for larger max turn limits, a higher number of tool calls does not always help. Consequently, varying tool calls across different turns might further improve performance. Inspired by this, we provide a further study in Section 5.

Generalization on different models and benchmarks. Figure 4 shows the performance of parallel tool calling across different datasets (top row) and different models. The results suggest that the superior performance of parallel tool calling and the benefit of reducing the # turns required to finish the task are generalizable to different LLMs and deep research benchmarks. To further validate whether the performance gain can be obtained for open-source models, we test the performance of DeepSeek-V3.2 (Liu et al., 2025) and Qwen-3-235B-A22B-Thinking-2507 (Team, 2025b). Table 2 shows that there is a small gain in performance when using parallel tool calling; however, the gain is not as prominent as in SoTA proprietary models, suggesting room for improvement in open-source model training to enable more effective parallel tool calling.

Full set evaluation. Our results above for BrowseComp are evaluated with the first 100 tasks. We run the experiment with parallel tool calling on the full set of BrowseComp and obtained 62.2% accuracy with GPT-5-Medium which is 7.3% performance gain compared to the GPT-5-High (54.9%(OpenAI, 2025)).

4 Why Parallel Tool Calling Improves Accuracy

Our previous experiments have shown both the efficiency and the effectiveness of parallel tool calling. While the efficiency improvement is easier to understand due to the decreased number of

[5]We use Serper API for search

[6]We use JINA API for scraping

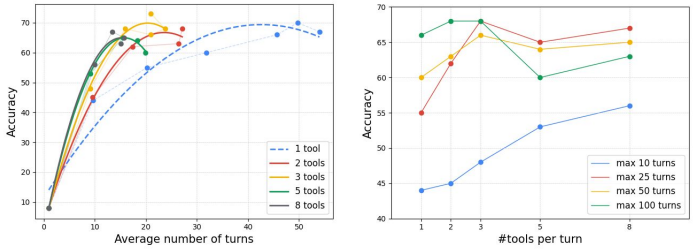


Figure 3: (Left) BrowseComp accuracy against average number of turns. (Right) Accuracy against number of tools per turn.

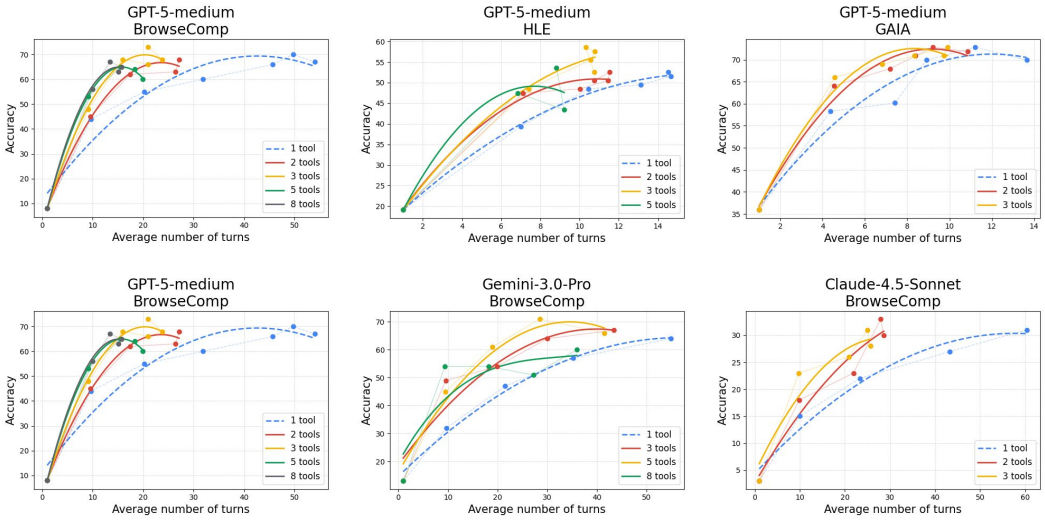


Figure 4: **Scaling of tool calls.** (Top row) Performance of GPT-5-medium across different benchmarks. (Bottom row) Performance of different models on BrowseComp benchmark.

Table 1: Accuracy and average number of iterations to completion (in brackets) on the BrowseComp dataset across different tool call limits per iteration. **No tool call** implies the LLM answers solely via reasoning without tools. n **iters** indicates the agent is forced to answer and stop at the n-th iteration.

	No tool call	**10 iters**	**25 iters**	**50 iters**	**100 iters**	**150 iters**	**300 iters**
1 Tool		44 **(9.7)**	55 **(20.3)**	60 **(31.9)**	66 **(45.7)**	70 **(49.8)**	67 **(54.7)**
2 Tools		45 **(9.6)**	62 **(17.4)**	63 **(26.5)**	**68** **(27.1)**	66 **(26.2)**	-
3 Tools	8 **(1.0)**	48 **(9.2)**	**68** **(16.0)**	**66** **(21.0)**	**68** **(23.8)**	73 **(21.0)**	-
5 Tools		53 **(9.1)**	65 **(15.6)**	64 **(18.4)**	60 **(19.9)**	-	-
8 Tools		**56** **(10.0)**	67 **(13.4)**	65 **(15.7)**	63 **(15.2)**	-	-

Table 2: Accuracy and average number of iterations to completion (in brackets) on the BrowseComp dataset for open-source models with single tool calling and parallel tool calling.

	Single tool calling	Parallel tool calling (3 tools)
Qwen3-235B-A22B-Thinking-2507	8 **(18.5)**	11 **(52.1)**
DeepSeek-V3.2	38 **(78.0)**	39 **(52.5)**

iterations and reduced reasoning, the source of the effectiveness (i.e., the gain in performance) remains unclear. To understand better on why the parallel tool calling helps effectiveness we inspect the agent trace manually to gain more insight and identified the following 3 patterns.

Observation 1: Exploration improves the credibility of information sources. In information-seeking tasks, the credibility of the source is vital for accuracy. Parallel tool calling broadens the search scope by triggering multiple queries, thereby aggregating a diverse collection of sources. This allows the model to compare inputs during the reasoning phase and select the most authoritative one. For the question "According to United Nations data from 2021, what percentage of national parliamentary seats in Northern Africa were held by women?", the parallel tool calling retrieved statistics from multiple locations and explicitly selected an official UN report, leading to the correct answer. In contrast, the single-tool calling, limited by a narrower search scope, relied on a API database. This source proved unofficial, causing the agent to answer incorrectly. The detailed comparison is shown in Figure 5.

Question: According to United Nations data from 2021, what percentage of national parliamentary seats in Northern Africa were held by women?

Single tool calling
What I did: ... Queried the SDG API ...
Evidence and result: The API returns a record with geoAreaName: Northern Africa ... value: 22.25 ...
Conclusion: According to United Nations SDG API data for 2021, women held 22.25% of national parliamentary seats in Northern Africa ...

Parallel tool calling
I searched United Nations sources and located the UN Statistical Yearbook tables ... Both the SYB65 (2022) and SYB67 (2024) tables list Northern Africa's percentage by year and show 24.3% for 2021 ... The UN Statistical Yearbook explicitly gives 24.3% for Northern Africa in 2021.

Figure 5: The comparison between single tool calling vs parallel tool calling in using the information source. Parallel tool calling uses more reliable information source.

Observation 2: Tool call redundancy enables tool results verification and avoids unreliable tools. When making tool calls, we could encounter tool call failure or unreliable tool call output. For agent with single tool call it will take all the tool call results as it is and plan the subsequent steps. Therefore, when there is a unreliable tool call result, the agent will be mis-led by this result and hence head toward the wrong direction of problem solving. In parallel tool calling, the agent tends to make redundant tool calls by specifying different arguments for each tool to get similar information. As a result, these different tool call results serve as verification of the results with each other and hence the model can decide whether this tool results are reliable or not and decide to use this information or do further searching. This verification ensure that the agent will not use the unrelible tool results and increased performance. In a query regarding Virginia Tech tuition, both agents located a target PDF, but the scraping mechanism failed to retrieve content. The tool's internal summarization model, however, hallucinated an answer based on this empty input. The single-call agent blindly accepted this fabricated tool output as fact. In contrast, the parallel-call agent triggered multiple extraction attempts. Because the resulting hallucinations were inconsistent across the redundant calls, the agent identified the results as unreliable, initiated a new search, and successfully retrieved the correct data.

Observation 3. Parallel search enhances retrieval effectiveness through query decomposition. While single-tool agents often struggle to return relevant results for complex, multi-faceted constraints, parallel tool calling allows the model to decompose a complicated request into multiple, simpler search queries. This approach significantly improves the search engine's ability to recall specific information. Consider a complex query regarding soccer matches between 1990–1994 involving specific constraints on the referee, yellow cards, and injury substitutions. The single tool calling attempted a keyword-stuffed search: "match report Brazilian referee yellow cards four substitutions first 25 minutes injury 1990... 1994". This yielded poor results. In contrast, the parallel tool calling split the task into distinct, manageable queries, such as "1990 World Cup matches referee list" and "1994 World Cup matches referee list". By isolating the search variables, the parallel

approach achieved higher recall and successfully retrieved the necessary data. The detailed comparison is shown in Figure 6.

Single tool calling:

```
{
    "function": {
      "arguments": "{"q":"match report "Brazilian referee" yellow
      cards four substitutions first 25 minutes injury 1990 1991
      1992 1993 1994"}",
      "name": "serper-search__google_search"
    },
  },
}
```

Parallel tool calling:

```
{
    "function": {
      "arguments": "{"q": "José Roberto Wright 1990 World Cup
      matches referee list"}",
      "name": "serper-search__google_search"
    },
  },
  {
    "function": {
      "arguments": "{"q": "Renato Marsiglia 1994 World Cup matches
      referee list"}",
      "name": "serper-search__google_search"
    },
  },
  ...
}
```

Figure 6: The comparison between single tool calling vs parallel tool calling in using the search tool. Parallel tool calling decompose a complex query to multiple simpler ones which enables more effective search.

5 TOOL CALL SCHEDULER

In the previous section, we fixed the number of tool calls across all steps. However, this may not be the optimal strategy. As shown in the right plot of Figure 3. We can see that when number of steps is small, increasing number of tools per step improves accuracy; when number of steps is higher, having more tools may not be beneficial. In this section, we present a preliminary exploration of different tool call schedulers for parallel tool calling. Denoting m_t as the number of tool calls required at step t, we compare the following schedulers:

- **Constant 1 Tool**: The default setting where make single tool call at each step ($m_t = 1$)
- **Constant 3 Tools**: We fix the number of tool calling to 3 across all steps ($m_t = 3$)
- **Ascending**: The number of tool calls increases *w.r.t.* step.

$$m_t = \begin{cases} 1 & t \leq 25 \\ 2 & 25 < t \leq 50 \\ 3 & t > 50 \end{cases} \tag{4}$$

- **Descending**: The number of tool calls decreases. *w.r.t.* step.

$$m_t = \begin{cases} 3 & t \leq 25 \\ 2 & 25 < t \leq 50 \\ 1 & t > 50 \end{cases} \tag{5}$$

- **Automatic**: We let LLM to decide the number of tool calls at each step. We add a specific instruction to the user message at each step, as shown in Figure 7.

User Instruction at Each Iteration for Dynamic Tool Calling

"In the next step, first identify your progress (0-100%) on solving this problem. If you need to make function calls, you MUST make at least 1 but no more than 4 function calls in a single response to gather information extensively. Based on the progress, the general principle is to make more function calls in the early phases, and gradually reduce the number as you approach completion."

Figure 7: The specific instruction used to let the LLM decide the number of tool calls autonomously.

Table 3 shows the performance across different tool call schedulers. The results show that **Descending** achieves a 6% performance gain over **Constant 3 Tools**, demonstrating the potential of tool call schedulers. **Ascending** achieves the worst performance while **Descending** achieves the best, suggesting that the explore-then-exploit strategy contributes to better performance. The **Automatic** strategy did not perform better than **Descending**, indicating that the LLM itself cannot determine the optimal number of tool calls in each iteration. These results could inspire future research on training LLMs to incorporate the explore-then-exploit tool call scheduler, enabling them to decide the optimal number of tool calls independently. Figure 8 shows the actual average number of tool calls in each turn. The number of tool calls is very close to our specified tool calls, suggesting that the LLMs follow our instruction in Figure 2 well.

Table 3: Accuracy and average number of iterations to completion (in brackets) on the BrowseComp dataset across different tool call schedulers.

	Constant 1 Tool	**Constant 3 Tools**	**Ascending**	**Descending**	**Automatic**
Accuracy **(#Turns)**	66 **(45.7)**	68 **(23.8)**	63 **(36.5)**	**74** **(23.5)**	72 **(26.6)**

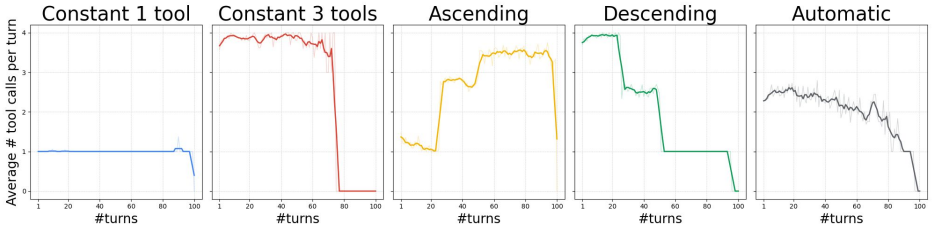


Figure 8: The average # tool calls across all turns for different tool call scheduler.

6 Related Work

Existing works have explored similar ideas to handle a larger workload within each step. They can be categorized into two main classes: 1) parallel reasoning, which generates multiple reasoning traces based on the same context and employs a summary model to aggregate all the results; and 2) sub-agent design, where the main agent invokes multiple sub-agents in each step to complete sub-tasks and aggregates the results at the end of the step.

Parallel reasoning. The work of (Pan et al., 2025) proposes to replace serialized chain-of-thought reasoning with coordinated parallel reasoning to improve latency; however, it does not target the agent setting where tool calls are incurred. Longcat (Meituan LongCat Team, 2026) introduced parallel reasoning to generate multiple reasoning paths within the same turn, aggregating the outcomes via summarization. However, it overlooks coordination between different paths and hence can produce repetitive work. In contrast, in our parallel tool calling, although all tools are invoked in the same step, they share the same reasoning, which helps to coordinate between tool calls to avoid the wasteful use of tools.

Sub-agent. Miroflow (Team, 2025a) proposes a sub-agent orchestration design. In each iteration, the main agent reasons based on the current context without direct tool calling; instead, a sub-agent is invoked to use tools and solve sub-tasks. Kimi-K2.5 (Moonshot AI, 2026) proposes an agent swarm framework and trains the model with the newly proposed parallel-agent reinforcement learning (PARL). The model learns to decompose the task into parallelizable sub-tasks. These sub-agent-based methods are similar to parallel tool calling; however, they require more complex orchestration and significant modification to the commonly used single-agent framework. In contrast, our parallel tool calling enables more workload to be processed and is readily integrated into any single-agent framework, making it potentially adaptable to any agentic system.

7 Conclusion

In this work, we introduced the **Wide and Deep research agent**, a framework designed to explore the benefits of scaling execution width via parallel tool calling alongside reasoning depth. Our extensive experiments across BrowseComp, HLE, and GAIA demonstrate that parallel tool calling significantly improves agent performance while simultaneously reducing the number of sequential turns required to reach a solution. This reduction in iterations not only lowers the end-to-end wall-clock time but also amortizes reasoning costs by condensing multiple actions into single reasoning steps. Through qualitative analysis, we identified that these gains are driven by enhanced source verification, redundancy against tool failures, and effective query decomposition. Furthermore, our investigation into tool call scheduling reveals that a **Descending** strategy, prioritizing broad exploration in early stages followed by focused exploitation, outperforms static or ascending strategies. However, the inability of current LLMs to autonomously optimize this trade-off in the **Automatic** setting highlights a limitation in existing models. Future work should focus on training agents, potentially through reinforcement learning, to dynamically manage the "width-depth" trade-off, enabling next-generation agents to autonomously navigate the explore-then-exploit spectrum for high-efficiency deep research.

Acknowledgments

We thank the Salesforce AI Research team for their valuable feedback and support throughout this project.

References

Anthropic. Claude opus 4.5 system card. System card, Anthropic, November 2025. URL https://assets.anthropic.com/m/64823ba7485345a7/Claude-Opus-4-5-System-Card.pdf.

Gemini Team. Gemini deep research, 2025. URL https://gemini.google/overview/deep-research/. Accessed: 2026-01-26.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. Mcp-universe: Benchmarking large language models with real-world model context protocol servers, 2025. URL https://arxiv.org/abs/2508.14704.

Meituan LongCat Team. Longcat-flash-thinking-2601 technical report. Technical Report arXiv:2601.16725, Meituan, January 2026. URL https://arxiv.org/abs/2601.16725. arXiv preprint arXiv:2601.16725.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Moonshot AI. Kimi-researcher: End-to-end rl training for emerging agentic capabilities, June 2025. URL https://moonshotai.github.io/Kimi-Researcher/. Accessed: 2026-01-26.

Moonshot AI. Kimi k2.5: Visual agentic intelligence. Moonshot AI Blog, January 2026. URL https://www.kimi.com/blog/kimi-k2-5.html. Accessed: 2026-01-29.

OpenAI. Deep research system card. System card, OpenAI, February 2025. URL https://cdn.openai.com/deep-research-system-card.pdf.

OpenAI. Intruducing gpt-5, 2025. URL https://openai.com/index/introducing-gpt-5/. Accessed: 2026-02-05.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. In *Proc. COLM*, 2025.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

MiroMind Team, Song Bai, Lidong Bing, Carson Chen, Guanzheng Chen, Yuntao Chen, Zhe Chen, Ziyi Chen, Jifeng Dai, Xuan Dong, et al. Mirothinker: Pushing the performance boundaries of open-source research agents via model, context, and interactive scaling. *arXiv preprint arXiv:2511.11793*, 2025a.

MiroMind AI Team. Miroflow: A high-performance open-source research agent framework. https://github.com/MiroMindAI/MiroFlow, 2025a.

Qwen Team. Qwen3 technical report, 2025b. URL https://arxiv.org/abs/2505.09388.

Tongyi DeepResearch Team, Baixuan Li, Bo Zhang, Dingchu Zhang, Fei Huang, Guangyu Li, Guoxin Chen, Huifeng Yin, Jialong Wu, Jingren Zhou, et al. Tongyi deepresearch technical report. *arXiv preprint arXiv:2510.24701*, 2025b.

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

A Appendix

More implementation details. Figure 11 provides the full system prompt we use in the deep research agent. In each step, as mentioned in Figure 2, a user message is used to precisely control the number of tool calls required in the next step. Furthermore, we include a countdown message to inform the LLM of the remaining budget (see Figure 9). This is inspired by previous work (Anthropic, 2025) that uses prompts to make the LLM aware of the remaining budget and hence improves performance. When the agent reaches the max turn limit, we use a user message (see Figure 10) to force the LLM to output the answer given all the information available, instead of stopping without an answer.

User Instruction at Each Iteration

You have n steps remaining. Please continue. At next step, if you need to make function calls, you MUST make at least 3 but not more than 4 function calls in a single response to gather information extensively.

Figure 9: User countdown message.

User Instruction at Each Iteration

Important: You have reached the maximum number of interaction turns. Therefore, you have to output the final answer with the specified format without calling any tools in the final response.

Figure 10: User message to force the LLM to generate the final answer when it reaches the max turn limit.

User Instruction at Each Iteration

You are an intelligent assistant that can solve complex problems by thinking step-by-step and using available tools when needed.
Today is:
{{*FORMATTED_DATE*}}
Your Role
{{*INSTRUCTION*}}
Your Task
{{*QUESTION*}}
How You Work

1. ***Think First****: Analyze the problem and determine what information or actions you need*
2. ***Use Tools When Needed****: Call appropriate functions to gather information, perform calculations, or take actions*
3. ***Reason with Results****: Process the tool outputs and use them to inform your next steps*
4. ***Iterate****: Continue thinking and using tools until you can provide a complete answer*

Available Capabilities

- *You have access to various specialized tools through function calling*
- *When you need to use a tool, simply call the appropriate function with the required parameters*
- *The system will execute the function and provide you with the results*
- *Use these results to continue your problem-solving process*

Important Guidelines

- *You have a maximum of* {{*MAX_STEPS*}} *steps to complete this task*
- *Each step should either advance your understanding or gather necessary information*
- *Be systematic and thorough in your approach*
- *Only provide your final answer when you have sufficient information*
- ***IMPORTANT: At each step, you MUST follow user instructions in each step to make the amount of function calls specified in the user instructions.***

—
Final Answer Format *When you have completed your analysis and gathered all necessary information, provide your final response using this JSON format:*

```
{
    "thought": "Explain your reasoning process and how you arrived at
    the answer",
    "answer": "Your complete final answer to the task"
}
```

Important*:*

- *Use the JSON format above ONLY for your final answer*
- *During your thinking process, you can respond in any natural format*
- *The "answer" field should contain your complete solution as a string*

Figure 11: System prompt for deep research agent.