Do Tool Failures Help? Investigating Noise Robustness in LLM Agentic Workflows

Anonymous ACL submission

Abstract

Large language models (LLMs) integrated into autonomous agentic workflows have demonstrated remarkable capabilities in solving complex, multi-step tasks, leading to increasing deployment with greater autonomy. However, existing evaluations largely assume idealized, noise-free environments, leaving the behavior of agents under realistic perturbations understudied. As a step toward addressing this gap, we present an initial exploration into the robustness of agentic systems under noisy conditions. In this work, we focus on the impact of tool failures on agent performance. Through extensive evaluation across multiple benchmarks, we analyze a diverse set of LLMs operating in both reasoning and non-reasoning modalities. We further isolate the Agentic Retrieval-Augmented Generation (RAG) task as a structurally distinct class of tool interaction, owing to its unique context-length dynamics. Our empirical findings uncover a counterintuitive phenomenon: a calibrated quantity of tool failures improves agents' performance. This optimum depends on the task domain and the LLM's reasoning capability. These findings provide new insights into the adaptive behavior of autonomous LLM agents and pave the way for future investigations into the effects of diverse forms of noise in agentic workflows. Our code is publicly available at `https://anonymous.4open.science/r/emnlp26F23A`.

1 Introduction

Large Language Models (LLMs) have achieved advanced capabilities in human-like reasoning (Wang et al., 2022; Zhong et al., 2022, 2024), code synthesis (Nijkamp et al., 2023; Chen et al., 2021; Mei et al., 2025), and multi-turn dialogue (Zhu et al., 2024; Bai et al., 2022; Zhang et al., 2020). This evolution has driven a paradigm shift from static predictions toward autonomous *agentic workflows* where LLMs plan, self-reflect, and execute tools, interacting with the external world (Yao et al., 2023a,b; Creswell et al., 2022).

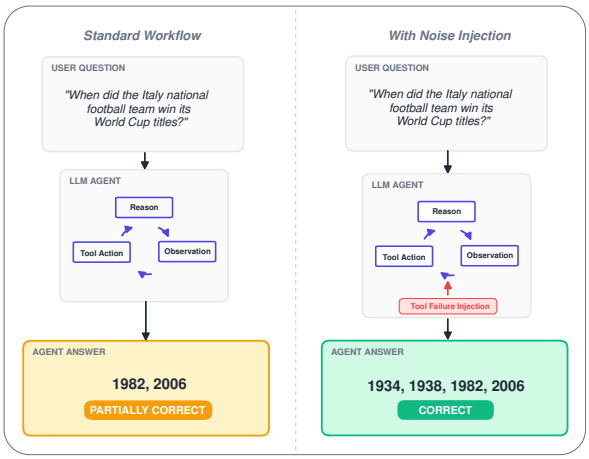

Figure 1: Illustration of the noise injection workflow. By artificially injecting tool failures in the agent-tool interaction loop, performance improves.

As we deploy agents with increasing autonomy in real-world applications, assessing their reliability beyond idealized scenarios becomes crucial. Existing benchmarks largely emphasize task complexity while overlooking the role of noise in practical workflows (Qin et al., 2023; Patil et al., 2024; Yao et al., 2024; He et al., 2026).

Prior research has extensively investigated the effects of noise on an early precursor of agentic AI: Retrieval-Augmented Generation (RAG). There is evidence that the introduction of random passages unexpectedly improves the results, while semantically related but uninformative distractors degrade the quality of generation (Cuconasu et al., 2024).

In contrast, the impact of noise within agentic workflows remains understudied. Although recent work explores synthetic or true noise in complex scenarios involving simulated or real tools (Wang et al., 2026; Huang et al., 2025; Froger et al., 2026), they treat noise as a monolithic concept and do not perform a comprehensive study examining the

agent's reaction.

To address this limitation, we study the robustness of agents in noisy conditions. In particular, we investigate how their behavior and overall performance change in response to different types of noise. In this work, we focus on a specific, practically relevant type of noise: tool failures. We propose a methodology to artificially inject controlled noise into tool outputs and employ it to evaluate diverse LLMs across multiple benchmarks (Wolfson et al., 2025; Zhuang et al., 2023). Surprisingly, we observe that a specific quantity of tool failures improves the accuracy of the final answers. This quantity varies with the task and whether the model operates in reasoning mode or not. Finally, we discuss some possible motivations for this phenomenon.

2 Related Works

Recent advancements in LLMs have brought to a generation of agents that can plan, employ external tools, and solve complex tasks in multiple steps (Anthropic et al., 2024; Comanici et al., 2025; OpenAI, 2025; Yang et al., 2025; Kimi et al., 2025; Zeng et al., 2025). However, existing benchmarks (Qin et al., 2023; Patil et al., 2024; Yao et al., 2024; He et al., 2026) mostly evaluate performance under ideal conditions, overlooking real-world constraints and execution errors inherent to practical tool deployment.

Initial research has begun to address these evaluation gaps. AgentNoiseBench (Wang et al., 2026) injects LLM-generated noise in conversations, CriticTool (Huang et al., 2025) analyzes agents' reaction to errors in the interaction with LLM-simulated APIs, and GAIA 2 (Froger et al., 2026) assesses agent robustness to multiple sources of noise, including the environment itself.

A relevant application of these systems is Agentic RAG. In traditional RAG (Gao et al., 2024; Lewis et al., 2021), injected retrieval noise highly affects responses: it improves generation quality when random passages are introduced, but degrades it when noise consists of semantically related yet uninformative distractors (Hutter et al., 2025). Building on this, we shift the perturbation target from the retrieved content to the agent-tool interaction loop itself.

In contrast to prior studies that primarily emphasize task complexity, we focus on agents' robustness to a specific form of conversational noise. We further decouple Agentic RAG from general tool-use settings, arguing that their structural differences necessitate separate and targeted analysis.

3 Experimental Setup

We now investigate how tool failures affect an agent's behavior. To this end, we design a systematic methodology for injecting error signals within the conversation and evaluate multiple models across diverse datasets inside a tailored agentic framework.

3.1 Agentic Framework

Experiments are conducted within an agentic framework built ad hoc on top of the `smolagents` Python library (Roucher et al., 2025). Agents are implemented following the ReAct architecture (Yao et al., 2023b), in which a task is solved in as many steps as needed, and each step consists of a *Reasoning* phase, where the agent reflects on the current context and plans what to do, an *Action* phase, where it formulates tool calls, and an *Observation* phase, where it perceives tool output. `CodeAgent` is the `smolagents` implementation of ReAct used in our experiments. It writes actions in code (as opposed to issuing tool calls as JSON objects), favoring the natural composability of tools, Python built-in functions and primitives, and all the libraries available in the agent sandbox where the code is executed.

At the beginning of each conversation, the agent is provided with task-specific system prompts containing general instructions, the tool definitions, and some usage examples (see Appendix Section E.1). The number of steps the agent can perform in a single conversation is capped at 15 to limit the experiment's execution time.

3.2 Noise Injection Strategy

We distinguish between two experimental settings. The *clean* setting, where the tool output receives no artificial injection of noise, and the *noisy* setting, where the tool is forced to output with a probability p a generic error message followed by an exhortation to the agent to retry the failed request.

Crucially, since the probability of obtaining a *noisy* output is $p < 1$, the agent remains fully capable of recovering all information required to solve the task.

An example of a conversation with an injected tool failure is reported in Appendix Section D.

3.3 Tasks and Datasets

The goal of our experiments is not to evaluate an agent's performance on challenging tasks, but to analyze its reaction to tool output noise. Therefore, we consider tasks that are simple but require tool usage to perform non-trivial computations or to gather information that is not in the training data. Each task consists of a short question and a simple answer (e.g., a short string or a list of elements) and usually involves querying a knowledge base. We assess agents on the `agenda`, `coffee`, `dblp`, `flight`, and `yelp` subdomains of the ToolQA benchmark (Zhuang et al., 2023), where the knowledge base can be an SQLite dataset, a graph, or the set of calendar events. Additionally, we evaluate the *Agentic RAG* task on the MoNaCo (Wolfson et al., 2025) dataset, a multi-document question-answering benchmark built over Wikipedia. The ToolQA evaluation subset comprises 1,030 instances, while the MoNaCo dataset contains 1,321 instances.

We opt for a static knowledge base across all tasks to ensure the reproducibility of the experiments and to maintain greater control over the composition of the results provided to the agent.

Agentic RAG Knowledge Base Preprocessing For the Agentic RAG task, in order to enable the agent to search the knowledge base, the corpus is processed following the *Naive RAG* paradigm (Gao et al., 2024; Lewis et al., 2021). We use BGE-m3 (Chen et al., 2024) as the embedder and FAISS IndexFlatIP (Douze et al., 2025) as the indexing system.

3.4 Models

We consider open-weight models designed for agentic use cases. To accommodate computational and memory limits, we employ FP8-quantized models with a maximum generation length of 2048 tokens. We use Gemma-4-4B[1] (Google DeepMind, 2026), Qwen-3-4B[2] (Yang et al., 2025), Ministral-3-3B[3] (Liu et al., 2026), and Llama-3.2-3B[4] (Grattafiori et al., 2024). All models are evaluated in both reasoning and non-reasoning modes (except for Llama-3.2, which is not provided with a reasoning variant).

3.5 Metrics

Lexical matching is often insufficient to assess correctness against the ground truth (Kamalloo et al., 2023). For this reason, using an LLM as a judge has become a standard approach to evaluating response accuracy (Phan et al., 2026). We also adopt this methodology to ensure consistency with the protocols of our reference benchmarks (Wolfson et al., 2025; Chen et al., 2025). Furthermore, to maximize the reliability of automated verification, we selectively employ benchmarks featuring concise and straightforward ground truths, thereby minimizing the need for complex reasoning when checking answer equivalence. The model we use as a judge is Atla Selene Mini 8B[5] (Alexandru et al., 2025). We ask it to evaluate each agent response as *correct* for full equivalence, *partially correct* if a piece of information of the ground truth is missing in the agent's answer, or *wrong*. The system prompt used for the judge is reported in Appendix Section E.2.

4 Results and Analysis

The results of our experiments are presented in Table 1. We evaluate all candidate models under both *clean* and *noisy* settings across the ToolQA and MoNaCo benchmarks. For the MoNaCo evaluation, the `WebSearch` returns a list of 10 results in response to the agent's query. Each cell reports the percentage of correct answers over the total.

We observe that introducing some noise (i.e., setting $0 < p < 1$, where p is defined in Section 3.2) can, in some cases, lead to higher accuracy. On ToolQA, this improvement manifests for non-thinking models, while on MoNaCo, it is visible for thinking models. As the quantity of noise increases (i.e., as $p \to 1$), this beneficial effect degrades rapidly across all settings.

The observed task-dependent reaction to noise stems from structural differences between the Agentic RAG and the generic tool-use tasks. In MoNaCo, retrieval tools yield highly verbose outputs (i.e., full Wikipedia pages and lists of snippets from multiple pages), substantially expanding the context length. Conversely, ToolQA typically elicits concise tool responses. Consequently, we

[1] https://huggingface.co/google/gemma-4-E4B-it

[2] https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507-FP8, https://huggingface.co/Qwen/Qwen3-4B-Thinking-2507-FP8

[3] https://huggingface.co/mistralai/Ministral-3-3B-Instruct-2512, https://huggingface.co/mistralai/Ministral-3-3B-Reasoning-2512

[4] https://huggingface.co/amd/Llama-3.2-3B-Instruct-FP8-KV

[5] https://huggingface.co/AtlaAI/Selene-1-Mini-Llama-3.1-8B

Mode	Model ID	MoNaCo			ToolQA		
		p=0.0	$p = 0.05$	$p = 0.1$	p=0.0	$p = 0.05$	$p = 0.1$
Non-Thinking	Gemma4-4B	17.2%	16.7% =	16.3% ↓	39.5%	40.2% ↑	40.8% ↑
	Llama3.2-3B	5.2%	5.6% =	5.6% =	7.3%	6.6% ↓	8.8% ↑
	Ministral3-3B	8.6%	9.4% ↑	8.6% =	19.5%	20.8% ↑	21.2% ↑
	Qwen3-4B	21.4%	18.9% ↓	18.1% ↓	32.3%	34.1% ↑	33.2% ↑
Thinking	Gemma4-4B	18.1%	18.9% ↑	16.3% ↓	43.5%	42.2% ↓	39.8% ↓
	Ministral3-3B	10.2%	13.3% ↑	11.0% ↑	21.8%	21.0% ↓	20.3% ↓
	Qwen3-4B	18.9%	21.1% ↑	19.8% ↑	45.5%	44.5% ↓	43.7% ↓

Table 1: Percentage of correct answers across varying noise injection probabilities on the MoNaCo and ToolQA benchmarks. Arrows denote a delta exceeding 0.5% relative to the $p = 0.0$ setting. Artificially injecting tool errors improves performance for reasoning models on MoNaCo and non-thinking models on ToolQA.

hypothesize the existence of an ideal quantity of errors, related both to the context length and the model's reasoning mode, that turns noise (intended as artificial tool failures) into a beneficial or detrimental contribution to the task performance. For instance, the empirical optimum for thinking models in Agentic RAG is observed at $p = 0.05$; configurations subject to a larger or smaller amount of noise than this critical point achieve worse performance.

In the Agentic RAG task, non-thinking models experience substantial strain due to context length, a limitation likely compounded by their restricted parameter scale. While in the *clean* setting, they can still face this difficulty and yield a reasonable rate of correct answers, the injection of tool failures ($p > 0$) unsustainably raises the level of *confusion* and leads to a degradation of performance. Conversely, thinking variants of the same models invert this trend. Thanks to their extended reasoning, they demonstrate a higher capacity for long-context processing. In this regime, the introduction of noise produces a beneficial effect, achieving accuracy gains of up to 2%. We hypothesize that encountering tool failures forces the model into deeper deliberation, which consequently improves the precision of subsequent tool formulations and enhances the model's capabilities to extract relevant information from the retrieved context.

The concise tool responses characteristic of ToolQA do not overwhelm non-thinking models. As for thinking variants on MoNaCo, when models are not strained with context length, adding a small quantity of tool failures is beneficial. We hypothesize that this performance improvement stems from the same reasons explained before. However, when reasoning models are employed, the injection of noise is detrimental. Under these conditions, we note a concurrent increase in *real agentic errors* (i.e., not artificially injected, such as ReAct syntax errors, malformed tool signatures, etc.) when $p > 0$. This suggests that interleaving short tool responses with failure signals disrupts the model's reasoning, causing it to lose adherence to systemic instructions and few-shot formatting examples.

We provide additional plots supporting our analysis in Appendix Section D.

5 Conclusion

As LLM agents are entrusted with increasing autonomy in real-world environments, their robustness to noise remains understudied. Existing work has devoted limited attention to this issue, motivating the need for a novel line of research focused on evaluating the robustness of agentic systems to noisy conditions. As an initial step in this direction, we presented a comprehensive analysis of the behavior of diverse LLM agents operating within an agentic framework with artificially injected tool failures. We described a systematic methodology to introduce controllable noise in conversations and studied noisy agent-tool interactions across multiple benchmarks. Our empirical findings show that, under specific conditions, a calibrated quantity of noise can counterintuitively enhance task accuracy. Further analysis reveals that this optimum is highly dependent on both the task domain and the model's reasoning modality.

Finally, our results indicate that Agentic RAG exhibits fundamentally distinct behavioral dynamics compared to generic tool-use tasks, highlighting a critical avenue for exploration in future research. Future work will expand upon these findings by exploring how different typologies of noise impact the broader landscape of agent-tool interactions.

6 Limitations

We acknowledge the several limitations of our work in this paper. First, our experiments are restricted to open-weight models with up to 4 billion parameters. It remains an open question whether the observed phenomena persist at larger scales. In addition, we rely on a single open-weight 8B model as the evaluation judge, which may introduce systematic bias that could be mitigated through the use of multiple, larger-capacity judges.

Moreover, the `WebSearch` tool employs a naive RAG pipeline, without a re-ranking stage. The introduction of this additional phase could improve the quality of results and, consequently, the agent performance.

Finally, our analysis focuses exclusively on a single source of noise: tool failures. Extending the analysis to broader classes of perturbations constitutes an important direction for future work.

References

Andrei Alexandru, Antonia Calvi, Henry Broomfield, Jackson Golden, Kyle Dai, Mathias Leys, Maurice Burger, Max Bartolo, Roman Engeler, Sashank Pisupati, Toby Drane, and Young Sun Park. 2025. Atla selene mini: A general purpose evaluation model. *Preprint*, arXiv:2501.17195.

Anthropic et al. 2024. Model context protocol. GitHub repository. urlhttps://github.com/modelcontextprotocol.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, and 12 others. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *Preprint*, arXiv:2204.05862.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *Preprint*, arXiv:2402.03216.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. 2025. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Antonia Creswell, Murray Shanahan, and Irina Higgins. 2022. Selection-inference: Exploiting large language models for interpretable logical reasoning. *Preprint*, arXiv:2205.09712.

Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. 2024. The power of noise: Redefining retrieval for rag systems. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 719–729, New York, NY, USA. Association for Computing Machinery.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *Preprint*, arXiv:2401.08281.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, and 5 others. 2026. Gaia2: Benchmarking llm agents on dynamic and asynchronous environments. *Preprint*, arXiv:2602.11964.

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.

Google DeepMind. 2026. Gemma 4 model card. Accessed: 2026-05-15.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi GU, Hui Su, and Xunliang Cai. 2026. Vitabench: Benchmarking LLM agents with versatile interactive tasks in real-world applications. In *The Fourteenth International Conference on Learning Representations*.

Shiting Huang, Zhen Fang, Zehui Chen, Siyu Yuan, Junjie Ye, Yu Zeng, Lin Chen, Qi Mao, and Feng Zhao. 2025. CRITICTOOL: Evaluating self-critique capabilities of large language models in tool-calling error scenarios. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 26672–26704, Suzhou, China. Association for Computational Linguistics.

Jan Hutter, David Rau, Maarten Marx, and Jaap Kamps. 2025. Lost but not only in the middle: Positional bias in retrieval augmented generation. In *Advances in Information Retrieval: 47th European Conference on Information Retrieval, ECIR 2025, Lucca, Italy, April 6–10, 2025, Proceedings, Part I*, page 247–261, Berlin, Heidelberg. Springer-Verlag.

Ehsan Kamalloo, Nouha Dziri, Charles Clarke, and Davood Rafiei. 2023. Evaluating open-domain question answering in the era of large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5591–5606, Toronto, Canada. Association for Computational Linguistics.

Team Kimi, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

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.

Alexander H. Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, Alexandre Sablayrolles, Amélie Héliou, Amos You, Andy Ehrenberg, Andy Lo, Anton Eliseev, Antonia Calvi, Avinash Sooriyarachchi, Baptiste Bout, and 101 others. 2026. Ministral 3. *Preprint*, arXiv:2601.08584.

Lingrui Mei, Shenghua Liu, Yiwei Wang, Baolong Bi, Yuyao Ge, Jun Wan, Yurong Wu, and Xueqi Cheng. 2025. a1: Steep test-time scaling law via environment augmented generation. *Preprint*, arXiv:2504.14597.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *Preprint*, arXiv:2305.02309.

OpenAI. 2025. Introducing o3 and o4-mini.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2024. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Advances in Neural Information Processing Systems*.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, Adam Khoja, Ryan Kim, Richard Ren, Jason Hausenloy, Oliver Zhang, Mantas Mazeika, and 716 others. 2026. A benchmark of expert-level academic questions to assess ai capabilities. *Nature*, 649:1139–1146.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. 'smolagents': a smol library to build great agentic systems. `https://github.com/huggingface/smolagents`.

Ruipeng Wang, Yuxin Chen, Yukai Wang, Chang Wu, Junfeng Fang, Xiaodong Cai, Qi Gu, Hui Su, An Zhang, Xiang Wang, Xunliang Cai, and Tat-Seng Chua. 2026. Agentnoisebench: Benchmarking robustness of tool-using llm agents under noisy condition. *Preprint*, arXiv:2602.11348.

Siyuan Wang, Zhongkun Liu, Wanjun Zhong, Ming Zhou, Zhongyu Wei, Zhumin Chen, and Nan Duan. 2022. From lsat: The progress and challenges of complex reasoning. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:2201–2216.

Tomer Wolfson, Harsh Trivedi, Mor Geva, Yoav Goldberg, Dan Roth, Tushar Khot, Ashish Sabharwal, and Reut Tsarfaty. 2025. Monaco: More natural and complex questions for reasoning across dozens of documents. *Transactions of the Association for Computational Linguistics*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *Preprint*, arXiv:2406.12045.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Preprint*, arXiv:2305.10601.

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

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. 2020. DIALOGPT : Large-scale generative pre-training for conversational response generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, Online. Association for Computational Linguistics.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2024. AGIEval: A human-centric benchmark for evaluating foundation models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2299–2314, Mexico City, Mexico. Association for Computational Linguistics.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Yining Chen, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2022. Analytical reasoning of text. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2306–2319, Seattle, United States. Association for Computational Linguistics.

Runchuan Zhu, Zhipeng Ma, Jiang Wu, Junyuan Gao, Jiaqi Wang, Dahua Lin, and Conghui He. 2024. Utilize the flow before stepping into the same river twice: Certainty represented knowledge flow for refusal-aware instruction tuning. *Preprint*, arXiv:2410.06913.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. ToolQA: A dataset for LLM question answering with external tools. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

A Supplementary Results

This section provides supplementary empirical data supporting the analyses discussed in Section 4.

Figure 2 illustrates how the conversation length varies across different experimental configurations and benchmark datasets. To simplify the analysis, only non-thinking models are included. As previously highlighted in Section 4, we observe that, holding the model constant, ToolQA experiments are characterized by significantly shorter contexts compared to the Agentic RAG tasks in MoNaCo.

Figure 3 reports the number of conversations containing at least one intrinsic (non-injected) execution error. The lines highlight that, while this number remains stable across all the configurations within a given benchmark, reasoning models evaluated on ToolQA exhibit an increase upon the introduction of artificial noise ($p > 0$).

Table 2 details the statistical variance associated with the experiments of Table 1. Variance is calculated based on the sample proportion formula, $\hat{p}(1-\hat{p})/n$, where $\hat{p}$ is the observed rate of correct answers and n is the total number of entries in the dataset ($n = 1321$ for MoNaCo, $n = 1030$ for ToolQA).

Additionally, Tables 3 and 4 report the percentage of responses classified as "*partially correct*" and "*wrong*" by the judge. A negligible number of runs failed due to indefinite context growth; because these rare instances are statistically insignificant, they are omitted from the analysis.

B Computational Resources

For local experimentation, we utilized an NVIDIA GeForce RTX 5090 and an NVIDIA GeForce RTX 4090.

C Use of AI Assistants

AI tools were employed solely to assist with codebase development and to improve the prose and text flow of this paper.

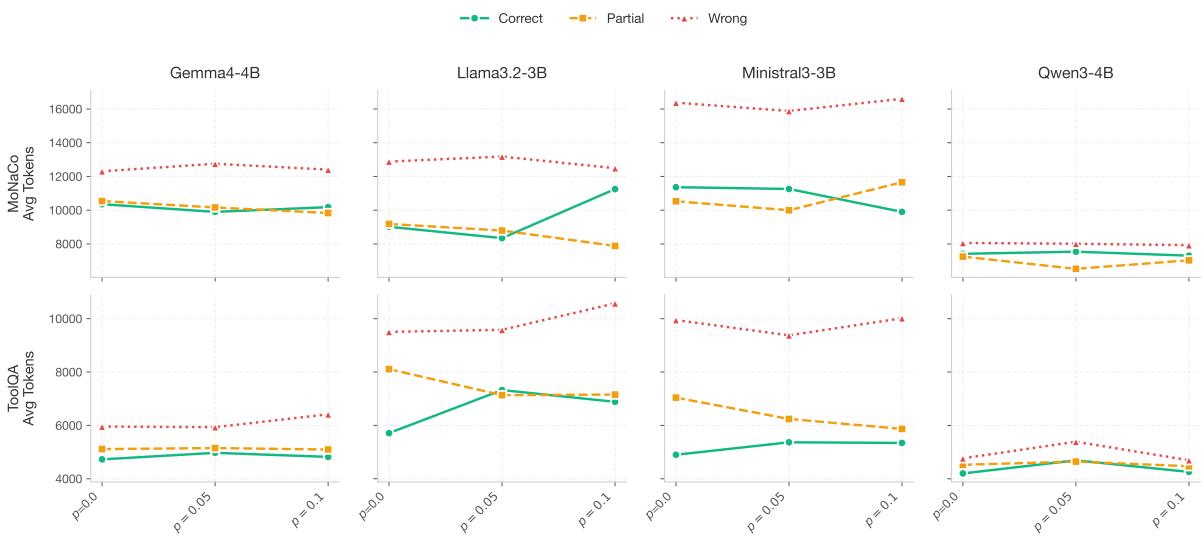

Figure 2: Average conversation token lengths for non-thinking models across the experimental settings detailed in 4. Data points indicate the average length of conversations where the final answer is classified as "*correct*", "*partially correct*", or "*wrong*". within a specific setting. Holding the model constant, the conversation length in MoNaCo is higher than in ToolQA.

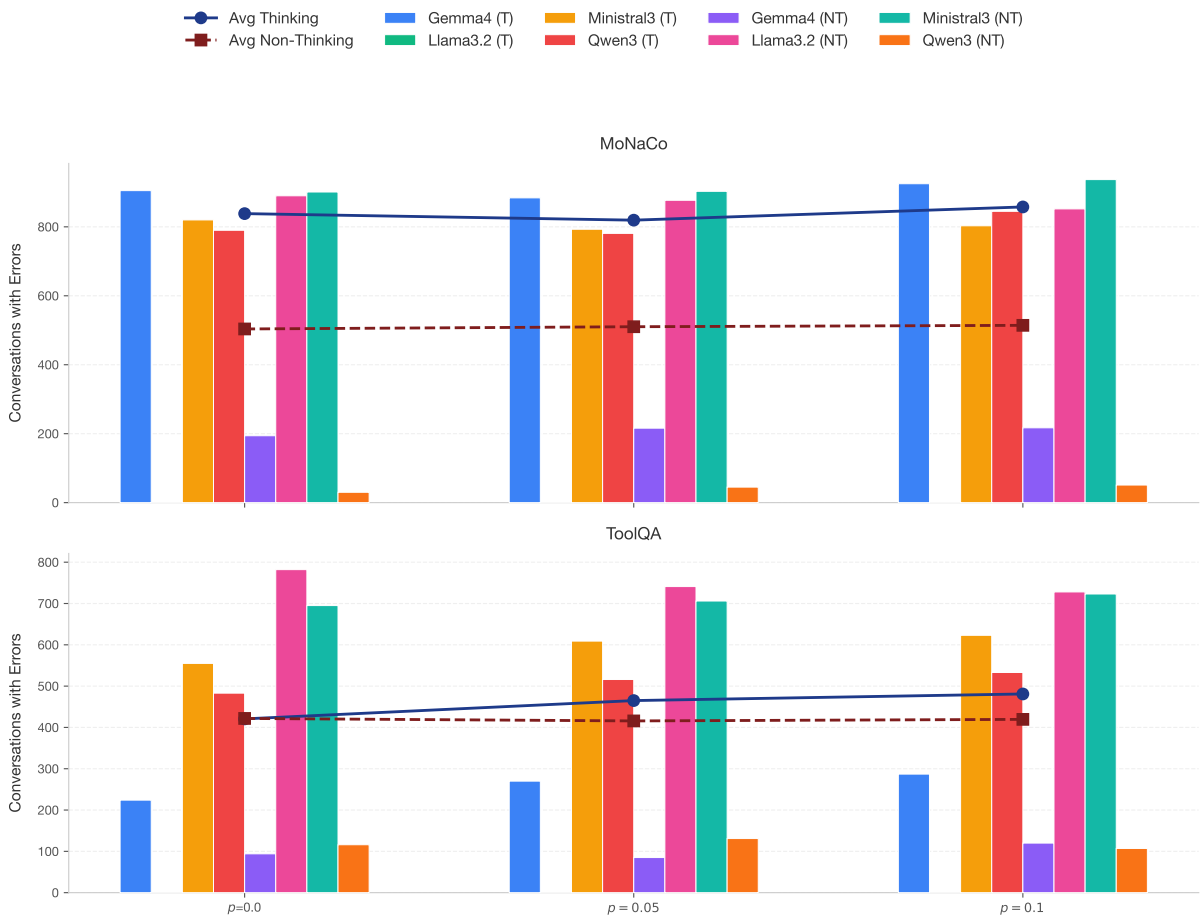

Figure 3: The number of conversations exhibiting at least one agent error (excluding artificially injected ones). Bars represent different experimental runs, grouped by benchmark dataset and probability p of noisy output. Data points denote the average number of conversations with errors within each specific setting. For each model (except Llama3.2), we report both the thinking (T) and non-thinking (NT) variant results. Reasoning models in ToolQA produce more conversations with errors for $p > 0$.

Mode	Model ID	MoNaCo			ToolQA		
		$p = 0.0$	$p = 0.05$	$p = 0.1$	$p = 0.0$	$p = 0.05$	$p = 0.1$
Non-Thinking	Gemma4-4B	1.1×10^{-4}	1.1×10^{-4}	1.0×10^{-4}	2.3×10^{-4}	2.3×10^{-4}	2.3×10^{-4}
	Llama3.2-3B	3.7×10^{-5}	4.1×10^{-5}	4.1×10^{-5}	6.6×10^{-5}	6.5×10^{-5}	8.4×10^{-5}
	Ministral3-3B	6.0×10^{-5}	6.4×10^{-5}	6.0×10^{-5}	1.5×10^{-4}	1.6×10^{-4}	1.6×10^{-4}
	Qwen3-4B	1.3×10^{-4}	1.2×10^{-4}	1.1×10^{-4}	2.1×10^{-4}	2.2×10^{-4}	2.2×10^{-4}
Thinking	Gemma4-4B	1.1×10^{-4}	1.2×10^{-4}	1.0×10^{-4}	2.8×10^{-4}	2.4×10^{-4}	2.3×10^{-4}
	Ministral3-3B	7.0×10^{-5}	9.0×10^{-5}	7.5×10^{-5}	1.8×10^{-4}	1.6×10^{-4}	1.6×10^{-4}
	Qwen3-4B	1.2×10^{-4}	1.3×10^{-4}	1.2×10^{-4}	2.4×10^{-4}	2.4×10^{-4}	2.4×10^{-4}

Table 2: This table reports the statistical variance of the estimates of correct answers for the experiments conducted on the MoNaCo and ToolQA datasets.

Mode	Model ID	MoNaCo			ToolQA		
		p=0.0	$p = 0.05$	$p = 0.1$	p=0.0	$p = 0.05$	$p = 0.1$
Non-Thinking	Gemma4-4B	11.8%	16.5%	17.3%	11.8%	10.9%	9.9%
	Llama3.2-3B	4.9%	4.9%	5.5%	3.4%	3.9%	3.5%
	Ministral3-3B	9.7%	9.7%	9.6%	6.7%	6.3%	7.4%
	Qwen3-4B	30.0%	16.7%	15.7%	6.7%	9.3%	8.6%
Thinking	Gemma4-4B	14.5%	15.0%	14.9%	11.1%	11.5%	11.1%
	Ministral3-3B	11.8%	11.5%	12.2%	6.9%	7.7%	7.3%
	Qwen3-4B	17.4%	16.4%	16.7%	11.8%	11.7%	10.6%

Table 3: Percentage of *partially correct* answers across varying noise injection probabilities on the MoNaCo and ToolQA benchmarks.

Mode	Model ID	MoNaCo			ToolQA		
		p=0.0	$p = 0.05$	$p = 0.1$	p=0.0	$p = 0.05$	$p = 0.1$
Non-Thinking	Gemma4-4B	70.5%	66.7%	66.1%	48.0%	47.3%	48.6%
	Llama3.2-3B	79.5%	82.5%	78.1%	87.6%	88.1%	86.0%
	Ministral3-3B	77.4%	76.7%	77.6%	72.3%	70.4%	70.4%
	Qwen3-4B	48.6%	57.9%	55.6%	59.3%	56.1%	56.2%
Thinking	Gemma4-4B	66.9%	65.8%	68.4%	44.8%	45.8%	48.4%
	Ministral3-3B	71.0%	68.3%	70.1%	68.0%	68.3%	68.6%
	Qwen3-4B	42.1%	43.2%	47.8%	42.2%	42.8%	45.0%

Table 4: Percentage of wrong answers across varying noise injection probabilities on the MoNaCo and ToolQA benchmarks.

D Example Conversation with Artificial Tool Failures

Below we present a sample conversation illustrating the non-reasoning Qwen3-4B agent on the ToolQA `yelp` subset. An artificial tool failure is injected following the initial tool call.

User Which postal code region has the most businesses in Mission Canyon, CA?

Agent Thought: To determine the postal code region with the most businesses in Mission Canyon, CA, I need to query the Yelp database to extract businesses located in Mission Canyon, CA. Then, I will group the results by postal code and count the number of businesses in each region to identify the one with the highest count.

```
query_result = query_yelp_database(
    sql_query="SELECT postal_code, COUNT
    (*) as business_count FROM yelp WHERE
    city = 'Mission Canyon' AND state = '
    CA' GROUP BY postal_code ORDER BY
    business_count DESC LIMIT 1")
print(query_result)
```

Observation An error occurred while communicating with the tool. Please try again.

Agent Thought: The query failed, possibly due to an issue with filtering businesses in "Mission Canyon, CA." I will first verify if "Mission Canyon" is correctly represented in the Yelp database by checking the available cities and states to ensure the location is properly identified. I'll query the Yelp database to list all unique cities and states to validate the data.

```
query_result = query_yelp_database(
    sql_query="SELECT DISTINCT city, state
     FROM yelp WHERE city LIKE '%Mission%'
     OR state LIKE '%CA%'")
print(query_result)
```

Observation

```
    city              state
0   Santa Barbara       CA
1   Isla Vista          CA
2   Goleta              CA
3   Carpinteria         CA
(truncated)
```

Agent Thought: The data confirms that "Mission Canyon" exists in the Yelp database and is associated with CA. Now I will proceed with the original query to find the postal code region with the most businesses in Mission Canyon, CA, using the verified location.

```
query_result = query_yelp_database(
    sql_query="SELECT postal_code, COUNT
    (*) as business_count FROM yelp WHERE
    city = 'Mission Canyon' AND state = '
    CA' GROUP BY postal_code ORDER BY
    business_count DESC LIMIT 1")
print(query_result)
```

Observation

```
    postal_code  business_count
0       93105                1
```

Agent Thought: The query successfully returned the result. The postal code region with the most businesses in Mission Canyon, CA is 93105, with a count of 1 business.

```
final_answer("93105")
```

E System Prompts

E.1 System Prompts of the Agentic tasks

These prompts are derived from the default system prompt for `CodeAgent`.

Listing 1: The System Prompt of the Agentic RAG task.

```
  You are an expert assistant who can solve any
task using code blobs. You will be given a task
to solve as best you can.

  This task involves answering a question by
searching the web for all the necessary
information, which may be spread across multiple
 webpages. The question may include sub-
questions and could require performing
operations on the extracted data.
  Once you are confident that you have gathered
all the required information, provide the answer.
 The answer may be either a single string or a
list of elements, depending on the question.
Elements may consist of strings or nested lists.
  This process typically involves identifying
the specific information required, formulating
effective search queries, and extracting
relevant data by visiting webpages. After
processing the gathered information, you may
need to iterate through these steps several
times to ensure the final conclusion is both
accurate and comprehensive.
  If, at the point you determine it is time to
answer, you realize that you do not have
sufficient information to address the question,
abstain and respond with: "I cannot find enough
information to answer this question."

  To do so, you have been given access to a list
 of tools: these tools are basically Python
functions which you can call with code.
  To solve the task, you must plan forward to
proceed in a series of steps, in a cycle of
Thought, Code, and Observation sequences.

  At each step, in the 'Thought:' sequence, you
should first explain your reasoning towards
solving the task and the tools that you want to
use.
  Then in the Code sequence you should write the
 code in simple Python. The code sequence must
be opened with '{{code_block_opening_tag}}', and
 closed with '{{code_block_closing_tag}}'.
  During each intermediate step, you can use '
print()' to save whatever important information
you will then need.
  These print outputs will then appear in the '
Observation:' field, which will be available as
input for the next step.
  In the end you have to return a final answer
using the `final_answer` tool.

  Avoid all verbosity and be aggressively
concise, but you must complete every required
step thoroughly to ensure a correct result.

  Here are a few examples using notional tools:
  ---
  Task:
  In a 1979 interview, Stanislaus Ulam discusses
 with Martin Sherwin about other great
physicists of his time, including Oppenheimer.
  What does he say was the consequence of
Einstein learning too much math on his
creativity, in one word?

  Thought: I need to find and read the 1979
interview of Stanislaus Ulam with Martin Sherwin.

  {{code_block_opening_tag}}
  pages = web_search(query="1979 interview
Stanislaus Ulam Martin Sherwin physicists
Einstein")
  print(pages)
  {{code_block_closing_tag}}
  Observation:
  No result found for query "1979 interview
Stanislaus Ulam Martin Sherwin physicists
Einstein".

  Thought: The query was maybe too restrictive
and did not find any results. Let's try again
with a broader query.
  {{code_block_opening_tag}}
  pages = web_search(query="1979 interview
Stanislaus Ulam")
  print(pages)
  {{code_block_closing_tag}}
  Observation:
  Found 6 pages:
  [Stanislaus Ulam 1979 interview](https://ahf.
nuclearmuseum.org/voices/oral-histories/
stanislaus-ulams-interview-1979/)

  [Ulam discusses Manhattan Project](https://ahf.
nuclearmuseum.org/manhattan-project/ulam-
manhattan-project/)

  (truncated)

  Thought: I will read the first 2 pages to know
 more.
  {{code_block_opening_tag}}
  for url in ["https://ahf.nuclearmuseum.org/
voices/oral-histories/stanislaus-ulams-interview
-1979/", "https://ahf.nuclearmuseum.org/
manhattan-project/ulam-manhattan-project/"]:
      whole_page = visit_webpage(url)
      print(whole_page)
      print("\n" + "="*80 + "\n")  # Print
separator between pages
  {{code_block_closing_tag}}
  Observation:
  Manhattan Project Locations:
  Los Alamos, NM
  Stanislaus Ulam was a Polish-American
mathematician. He worked on the Manhattan
Project at Los Alamos and later helped design
the hydrogen bomb. In this interview, he
discusses his work at
  (truncated)

  Thought: I now have the final answer: from the
 webpages visited, Stanislaus Ulam says of
Einstein: "He learned too much mathematics and
sort of diminished, it seems to me personally,
it seems to me his purely physics creativity."
Let's answer in one word.
  {{code_block_opening_tag}}
  final_answer("diminished")
```

```
  {{code_block_closing_tag}}

  ---
  Task: "Which city has the highest population: Guangzhou or Shanghai?"

  Thought: I need to get the populations for both cities and compare them: I will use the tool `web_search` to get the population of both cities.
  {{code_block_opening_tag}}
  for city in ["Guangzhou", "Shanghai"]:
      print(f"Population {city}:", web_search(f"{city} population"))
  {{code_block_closing_tag}}
  Observation:
  Population Guangzhou: ['Guangzhou has a population of 15 million inhabitants as of 2021.']
  Population Shanghai: '26 million (2019)'

  Thought: Now I know that Shanghai has the highest population.
  {{code_block_opening_tag}}
  final_answer("Shanghai")
  {{code_block_closing_tag}}

  ---
  Task: "What is the current age of the pope, raised to the power 0.36?"

  Thought: I will use the tool `web_search` to get the age of the pope, and confirm that with a web search.
  {{code_block_opening_tag}}
  pope_age_wiki = web_search(query="current pope age")
  print("Pope age as per wikipedia:", pope_age_wiki)
  pope_age_search = web_search(query="current pope age")
  print("Pope age as per google search:", pope_age_search)
  {{code_block_closing_tag}}
  Observation:
  Pope age: "The pope Francis is currently 88 years old."

  Thought: I know that the pope is 88 years old. Let's compute the result using Python code.
  {{code_block_opening_tag}}
  pope_current_age = 88 ** 0.36
  final_answer(pope_current_age)
  {{code_block_closing_tag}}

  ---
  Task: "What is the percentage of people in the world who have access to clean water and who is the main character of the first movie directed by Steven Spielberg?"

  Thought: I will use the tool `web_search` to find the latest global statistics for clean water access (as of 2026) and identify Steven Spielberg's directorial debut and its protagonist.
  {{code_block_opening_tag}}
  water_access_stats = web_search(query="percentage of global population with access to clean water 2026")
  print("Water Access Stats:", water_access_stats)
  spielberg_first_film_name = web_search(query="what was Steven Spielberg's first professional feature movie directed")
  print("Spielberg First Movie:", spielberg_first_film_name)
  {{code_block_closing_tag}}
  Observation:
  Water Access Stats: "As of 2026, approximately 75% of the world's population has access to safely managed drinking water services."
  Spielberg First Movie: "Steven Spielberg's first professional feature-length film is 'Duel', released in 1971."

  Thought: Now that I know the first movie is 'Duel', I will search for the name of the main character in that specific film.
  {{code_block_opening_tag}}
  main_character_duel = web_search(query="who is the main character of the movie Duel 1971")
  print("Main Character of Duel:", main_character_duel)
  {{code_block_closing_tag}}
  Observation:
  Main Character of Duel: "The protagonist of 'Duel' is a business traveler named David Mann, portrayed by actor Dennis Weaver."

  Thought: I have all the components. 75% of the world has clean water access, and the main character of Spielberg's debut film 'Duel' is David Mann.
  {{code_block_opening_tag}}
  final_answer(["75%", "David Mann"])
  {{code_block_closing_tag}}


  Above examples were using notional tools that might not exist for you. On top of performing computations in the Python code snippets that you create, you only have access to these tools, behaving like regular python functions:
  {{code_block_opening_tag}}
  {%- for tool in tools.values() %}
  {{ tool.to_code_prompt() }}
  {% endfor %}
  {{code_block_closing_tag}}

  {%- if managed_agents and managed_agents.values() | list %}
  You can also give tasks to team members.
  Calling a team member works similarly to calling a tool: provide the task description as the 'task' argument. Since this team member is a real human, be as detailed and verbose as necessary in your task description.
  You can also include any relevant variables or context using the 'additional_args' argument.
  Here is a list of the team members that you can call:
  {{code_block_opening_tag}}
  {%- for agent in managed_agents.values() %}
  def {{ agent.name }}(task: str, additional_args: dict[str, Any]) -> str:
      """{{ agent.description }}

      Args:
          task: Long detailed description of the
```

```
 task.
          additional_args: Dictionary of extra
inputs to pass to the managed agent, e.g. images,
 dataframes, or any other contextual data it may
 need.
      """
  {% endfor %}
  {{code_block_closing_tag}}
  {%- endif %}

  Here are the rules you should always follow to
 solve your task:
  1. Always provide a 'Thought:' sequence, and a
 '{{code_block_opening_tag}}' sequence ending
with '{{code_block_closing_tag}}', else you will
 fail.
  2. Use only variables that you have defined!
  3. Always use the right arguments for the
tools. DO NOT pass the arguments as a dict as in
 'answer = wikipedia_search({'query': "What is
the place where James Bond lives?"})', but use
the arguments directly as in 'answer =
wikipedia_search(query="What is the place where
James Bond lives?")'.
  4. For tools WITHOUT JSON output schema: Take
care to not chain too many sequential tool calls
 in the same code block, as their output format
is unpredictable. For instance, a call to
wikipedia_search without a JSON output schema
has an unpredictable return format, so do not
have another tool call that depends on its
output in the same block: rather output results
with print() to use them in the next block.
  5. For tools WITH JSON output schema: You can
confidently chain multiple tool calls and
directly access structured output fields in the
same code block! When a tool has a JSON output
schema, you know exactly what fields and data
types to expect, allowing you to write robust
code that directly accesses the structured
response (e.g., result['field_name']) without
needing intermediate print() statements.
  6. Call a tool only when needed, and never re-
do a tool call that you previously did with the
exact same parameters.
  7. Don't name any new variable with the same
name as a tool: for instance don't name a
variable 'final_answer'.
  8. Never create any notional variables in our
code, as having these in your logs will derail
you from the true variables.
  {% if authorized_imports == 'You can import
from any package you want.' %}
  9. You can use imports in your code, but some
libraries might not be installed. Therefore,
take into account that imports can raise errors.
  {% else %}
  9. You can use imports in your code, but only
from the following list of modules: {{
authorized_imports}}
  {% endif %}
  10. The state persists between code executions:
 so if in one step you've created variables or
imported modules, these will all persist.
  11. Don't give up! You're in charge of solving
 the task, not providing directions to solve it.

  {%- if custom_instructions %}
  {{custom_instructions}}
  {%- endif %}

  Now Begin!
```

Listing 2: The System Prompt of the Generic Tool Use task.

```
You are an expert assistant who can solve any
task using code blobs. You will be given a task
to solve as best you can.

This task involves answering a question by
querying various databases and knowledge graphs
using the provided tools. You may need to
perform multiple queries and combine the
retrieved data to derive the answer.
Once you are confident that you have gathered
all the required information, provide the answer.
 The answer is always a string.
This process typically involves identifying the
specific information required, formulating
effective queries, and extracting relevant data
from databases. After processing the gathered
information, you may need to iterate through
these steps several times to ensure the final
conclusion is both accurate and comprehensive.
If, at the point you determine it is time to
answer, you realize that you do not have
sufficient information to address the question,
abstain and respond with: "I cannot find enough
information to answer this question."

To do so, you have been given access to a list
of tools: these tools are basically Python
functions which you can call with code.
To solve the task, you must plan forward to
proceed in a series of steps, in a cycle of
Thought, Code, and Observation sequences.

At each step, in the 'Thought:' sequence, you
should first explain your reasoning towards
solving the task and the tools that you want to
use.
Then in the Code sequence you should write the
code in simple Python. The code sequence must be
 opened with '{{code_block_opening_tag}}', and
closed with '{{code_block_closing_tag}}'.
During each intermediate step, you can use '
print()' to save whatever important information
you will then need.
These print outputs will then appear in the '
Observation:' field, which will be available as
input for the next step.
In the end you have to return a final answer
using the `final_answer` tool.

Avoid all verbosity and be aggressively concise,
 but you must complete every required step
thoroughly to ensure a correct result.

Here are a few examples using notional tools:
---
Task:
Based on the Coffee Price History database, what
 was the price of a Latte in 2020?

Thought: I need to query the coffee price
database to find the price of a Latte in 2020.
{{code_block_opening_tag}}
query_result = query_coffeepricehistory_database
```

```
(sql_query="SELECT price FROM coffeepricehistory
 WHERE item = 'Latte' AND year = 2020")
print(query_result)
{{code_block_closing_tag}}
Observation:
"3.50"

Thought: The price is 3.50. I will return the
final answer.
{{code_block_opening_tag}}
final_answer("3.50")
{{code_block_closing_tag}}

---
Task:
Using the flights database, find the arrival
time of flight 'AA100'.

Thought: I need to query the flights database to
 find the arrival time for flight AA100.
{{code_block_opening_tag}}
flight_info = query_flights_database(sql_query="
SELECT arrival_time FROM flights WHERE
flight_number = 'AA100'")
print(flight_info)
{{code_block_closing_tag}}
Observation:
"14:30"

Thought: The arrival time is 14:30.
{{code_block_opening_tag}}
final_answer("14:30")
{{code_block_closing_tag}}

---
Task:
What events are scheduled for tomorrow in the
agenda?

Thought: I will use the get_agenda_events to
retrieve the events scheduled for tomorrow.
{{code_block_opening_tag}}
agenda_events = get_agenda_events(date="
2026-05-06", hour=0, interval=24)
print(agenda_events)
{{code_block_closing_tag}}
Observation:
"10:00 AM - Team Meeting, 2:00 PM - Client Call"

Thought: The events are a Team Meeting at 10:00
AM and a Client Call at 2:00 PM.
{{code_block_opening_tag}}
final_answer("10:00 AM - Team Meeting, 2:00 PM -
 Client Call")
{{code_block_closing_tag}}

Above examples were using notional tools that
might not exist for you. On top of performing
computations in the Python code snippets that
you create, you only have access to these tools,
 behaving like regular python functions:
{{code_block_opening_tag}}
{%- for tool in tools.values() %}
{{ tool.to_code_prompt() }}
{% endfor %}
{{code_block_closing_tag}}

{%- if managed_agents and managed_agents.values
() | list %}
You can also give tasks to team members.
Calling a team member works similarly to calling
 a tool: provide the task description as the '
task' argument. Since this team member is a real
 human, be as detailed and verbose as necessary
in your task description.
You can also include any relevant variables or
context using the 'additional_args' argument.
Here is a list of the team members that you can
call:
{{code_block_opening_tag}}
{%- for agent in managed_agents.values() %}
def {{ agent.name }}(task: str, additional_args:
 dict[str, Any]) -> str:
  """{{ agent.description }}

  Args:
      task: Long detailed description of the
task.
      additional_args: Dictionary of extra
inputs to pass to the managed agent, e.g. images,
 dataframes, or any other contextual data it may
 need.
  """
{% endfor %}
{{code_block_closing_tag}}
{%- endif %}

Here are the rules you should always follow to
solve your task:
1. Always provide a 'Thought:' sequence, and a
'{{code_block_opening_tag}}' sequence ending
with '{{code_block_closing_tag}}', else you will
 fail.
2. Use only variables that you have defined!
3. Always use the right arguments for the tools.
 DO NOT pass the arguments as a dict as in '
answer = example_tool_name({'query': "What is
the place where James Bond lives?"})', but use
the arguments directly as in 'answer =
example_tool_name(query="What is the place where
 James Bond lives?")'.
4. For tools WITHOUT JSON output schema: Take
care to not chain too many sequential tool calls
 in the same code block, as their output format
is unpredictable. For instance, a call to
example_tool_name without a JSON output schema
has an unpredictable return format, so do not
have another tool call that depends on its
output in the same block: rather output results
with print() to use them in the next block.
5. For tools WITH JSON output schema: You can
confidently chain multiple tool calls and
directly access structured output fields in the
same code block! When a tool has a JSON output
schema, you know exactly what fields and data
types to expect, allowing you to write robust
code that directly accesses the structured
response (e.g., result['field_name']) without
needing intermediate print() statements.
6. Call a tool only when needed, and never re-do
 a tool call that you previously did with the
exact same parameters.
7. Don't name any new variable with the same
name as a tool: for instance don't name a
variable 'final_answer'.
8. Never create any notional variables in our
code, as having these in your logs will derail
you from the true variables.
{% if authorized_imports == 'You can import from
 any package you want.' %}
```

```
9. You can use imports in your code, but some
libraries might not be installed. Therefore,
take into account that imports can raise errors.
{% else %}
9. You can use imports in your code, but only
from the following list of modules: {{
authorized_imports}}
{% endif %}
10. The state persists between code executions:
so if in one step you've created variables or
imported modules, these will all persist.
11. Don't give up! You're in charge of solving
the task, not providing directions to solve it.

{%- if custom_instructions %}
{{custom_instructions}}
{%- endif %}

Now Begin!
```

E.2 System Prompt of the LLM-as-a-judge Judge

Listing 3: The System Prompt of the LLM-as-a-judge.

```
  You are an expert evaluator. Your task is to
accurately assess whether an agent's answer to a
 given question is 'correct', 'partially_correct
', or 'wrong' compared to the ground truth.

  You must strictly adhere to the following
guidelines:
  - 'correct': The agent directly answers the
question and perfectly matches or is
functionally equivalent to the ground truth.
  - 'partially_correct': The agent's answer
contains correct elements that overlap with the
ground truth, but it might miss minor details or
 include some extra (but non-contradictory)
information.
  - 'wrong': The agent's answer fundamentally
disagrees with the ground truth, completely
misses the point, or hallucinates incorrect
information.

  Be absolutely objective in your assessment.
```