DS-Agent: A Cognitively-Inspired Multi-Agent Framework for Context-Aware Data Science Automation

Anonymous EMNLP submission

Abstract

LLM-based agent systems have achieved remarkable progress in automatically solving natural language processing tasks, yet they are typically constrained to simpler sequence-to-sequence generation scenarios. Real-world task environments, however, often involve multi-document workspaces requiring agents to explore and achieve specific goals through context-aware information processing. To enhance LLMs' effectiveness in handling end-to-end complex data science tasks, we propose DS-Agent – a novel LLM-based agent framework inspired by human problem-solving cognition. Our architecture enables workspace exploration through external tools while generating code/SQL to fulfill task objectives. Equipped with customized information retrieval tools, the DS-Agent effectively acquires and filters multi-source information from workspaces, significantly improving the quality of contextual information. Furthermore, its multi-agent architecture implements context partitioning and isolation mechanisms that support dynamic pruning during task planning, preventing individual agents from entering ineffective recursive iterations. We showcase the effectiveness of DS-Agent in agent-based data science tasks, where it achieves state-of-the-art accuracy across multiple models. The DS-Agent powered by GPT-4o reaches an accuracy of 42. 26%, representing a 10. 01% improvement over the baseline methods.

1 Introduction

Remarkable progress has been observed in recent Large Language Models (LLMs) for various natural language processing tasks, while LLM-based agent systems further extend these capabilities. However, compared to simply transforming instructions into executable code (Yu et al., 2018; Lin et al., 2018; Chen et al., 2021; Huang et al., 2024a; Lu et al., 2022), research on leveraging LLMs to address complex end-to-end data science tasks in real-world scenarios remains insufficient. In this work, we investigate open-ended automated data science pipelines aiming to democratize data science and improve end-to-end efficiency. Specifically, automated data science requires agents to: (1) accurately comprehend task requirements, (2) generate domain-specific strategies across all pipeline stages, and (3) automate workflow orchestration including preprocessing, analysis, visualization, and execution. As illustrated in Figure 1, the agent must proactively explore multi-source workspace files (databases, datasets, configuration files), autonomously plan solutions, generate code/SQL through natural language interactions, and ultimately deliver required outputs.

Current state-of-the-art LLMs still struggle to achieve high accuracy in end-to-end data science scenarios, primarily due to limitations in exploring local multi-source information and workflow planning. Complex data science tasks often require iterative debugging rather than single-step code generation, while necessitating careful management of intermediate artifacts. While fine-tuning could enhance LLM capabilities in this domain, it demands substantial computational resources and labeled data - constraints that only apply to open-source models. Recent studies have explored methods utilizing LLMs for interactive environment planning and action. In these approaches, environmental outcomes are fed back to the LLMs in text form, enabling LLMs to generate domain-specific actions or plans, which are then executed by a controller (Liu et al., 2024b; Ahn et al., 2022; Nakano et al., 2021; Yao et al., 2020; Huang et al., 2022). Alternative approaches treat code generation as the primary interaction mechanism between agents and environments (Qiao et al., 2023; Wang et al., 2024b), avoiding domain-specific action design while enhancing problem-solving versatility. However, conventional ReAct-based agents suffer from context bloat and

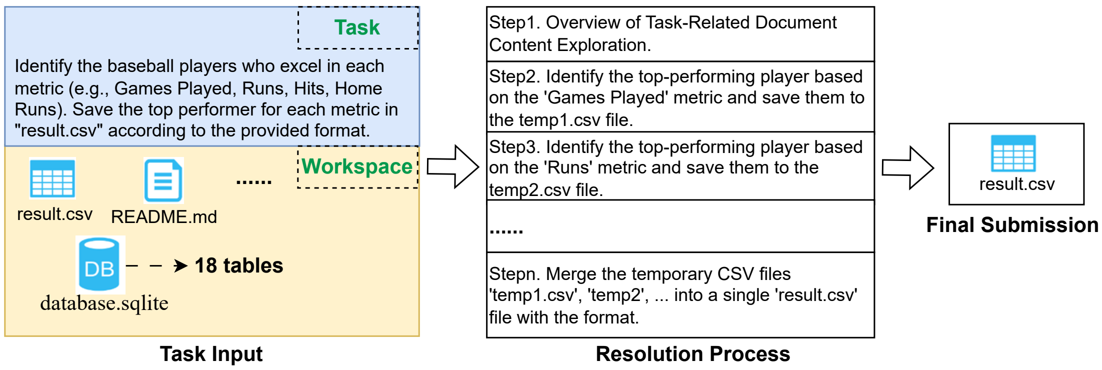

Figure 1: An example of a data wrangling task in data science tasks.

hallucination issues when processing multi-turn dialogue histories, leading to ineffective recursive planning.

When tackling complex real-world data analysis tasks, humans actively utilize existing tools (e.g., Notepad, Excel) to interact with their environment (external world). Through this interaction, they acquire task-relevant information into working memory (Baddeley, 1992) for cognitive reasoning (Alderson-Day and Fernyhough, 2015). The brain filters task-related information, designs task plans based on these filtered data, focuses attention on specific implementations, and self-regulates (Zavershneva and van der Veer, 2018; Luria, 1965; Dick and Overton, 2009) through environmental feedback. We propose DS-Agent - a novel LLM-based agent framework with external tool support for complex data science tasks: (1) Through integrated tool planning and ReAct strategies, DS-Agent proactively retrieves and filters essential workspace content via information retrieval tools, constructing clean reasoning contexts while eliminating irrelevant information from lengthy documents. (2) Our multi-agent architecture extends ReAct with dynamic pruning mechanisms that automatically discard ineffective planning paths after consecutive failures, enabling context-aware error recovery instead of linear recursion. Notably, DS-Agent operates without model training or fine-tuning, relying solely on automated execution with human intervention limited to initial task specification.

We demonstrate DS-Agent's effectiveness on DA-Code (Huang et al., 2024b), a real-world end-to-end dataset containing 100 complex data science tasks requiring advanced coding skills and diverse dataset interactions. Compared to the baseline DA-Agent (Huang et al., 2024b), DS-Agent achieves superior performance across multiple LLMs and task difficulty levels. The GPT-4o-powered DS-Agent reaches an accuracy of 42.26%, outperforming DA-Agent by 10.01%. The results demonstrate that DS-Agent fully unleashes the potential of LLMs in solving complex data analysis tasks.

2 Method

To enable large language models (LLMs) to handle complex data science tasks in end-to-end scenarios, we propose DS-Agent, a novel LLM-based agent framework that leverages the tool invocation capabilities of LLMs to decompose intricate problems into multiple traditional sequence-to-sequence style generation tasks.

2.1 Agent Workflow

There are two common strategies for agent tool invocation. One planning strategy is a variant of Chain-of-Thought (Wei et al., 2022) called tool planning, which leverages LLMs' planning capabilities to formulate plans before task execution. This approach constructs a natural language planner based on LLMs that infers a tool chain-of-thought using tool set descriptions (Lu et al., 2023). The executable tool chain can complete simple one-shot tasks. The second strategy is ReAct (Yao et al., 2023), which prompts LLMs to interactively generate reasoning traces and task-related actions. Based on these actions, ReAct selects appropriate external tools through input provision and continuously adjusts plans according to tool outputs. While suitable for complex tasks requiring dynamic adjustments, this method is vulnerable to performance degradation from dynamically generated action trajectories.

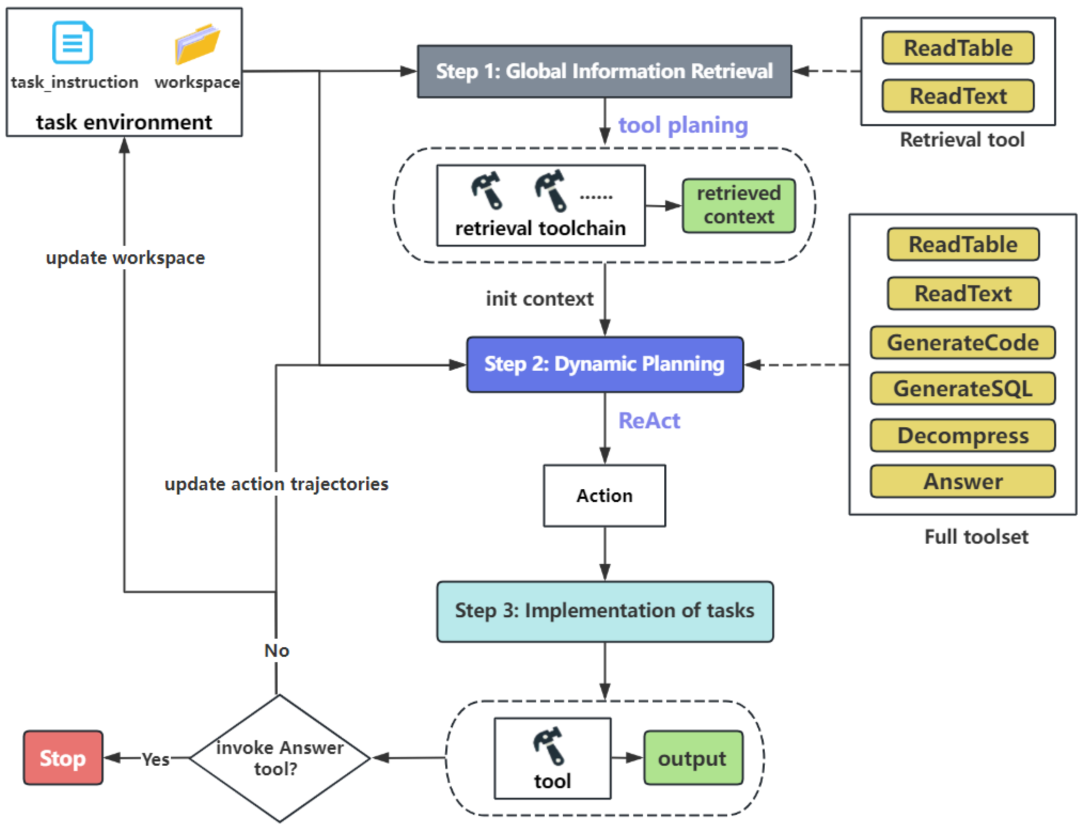

Figure 2: The Operational Workflow of DS-Agent.

As shown in Figure 2, our DS-Agent workflow integrates advantages from both strategies by combining tool planning and ReAct approaches, leveraging their complementary strengths:

Step 1: Global Information Retrieval. Planning phases (Step 2) often suffer from missing critical workspace files, leading to hallucinatory code generation. Since global file identification and retrieval constitute a simple one-shot task, we employ tool planning prior to Step 2 to reduce downstream hallucination. DS-Agent's tool planning generates retrieval chains that fetch task-relevant files from workspaces, forming initial reasoning contexts that persist as input components for subsequent LLM inferences.

Step 2: Dynamic Planning. Complex task execution requires real-time environmental awareness and dynamic adaptation, making ReAct more suitable. The planning module (implemented as a dedicated agent) employs ReAct-inspired reasoning: it predicts subsequent actions by analyzing historical trajectories and task objectives, iteratively selecting tools from the toolset to decompose complex tasks into correct tool sequences for execution.

Step 3: Implementation of subtasks. We design specialized toolsets to implement subtasks generated by the planning module. Tool outputs continuously update action trajectories, maintaining up-to-date system states for subsequent reasoning cycles.

2.2 Designed Tools

2.2.1 Rule-based Tools

Rule-based tools are implemented through Python functions that execute specific operations and return deterministic results when invoked.

- **ReadTable(file_path):** This tool retrieves structural metadata (e.g., column names, data types) and a data preview from structured data files specified by file_path. Unlike existing tools utilized by the agent for reading tabular files, our implementation deliberately excludes raw table rows from the agent's context, thereby mitigating the impact of irrelevant text on agent performance. For details, refer to the Appendix A.
- **Decompress(file_path):** This tool automatically decompresses files using format-specific decompression methods based on file extensions.

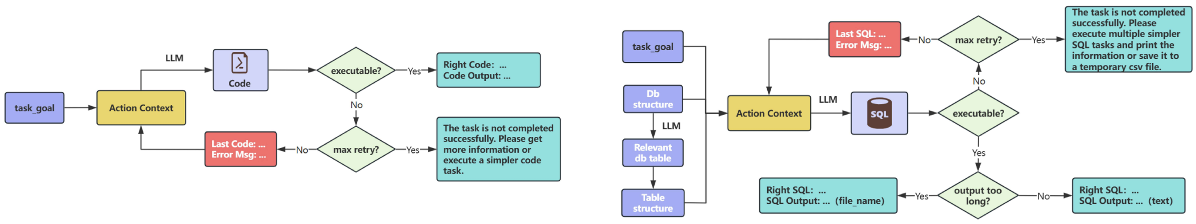

Figure 3: The Operational Workflow of the GenerateCode action (left) and the GenerateSQL action (right).

- **Answer(output):** This tool submits final task results, which may include filenames, text outputs, or failure.

2.2.2 Agent-based Tools

Agent-based tools facilitate multi-agent collaboration through tool invocation, where invoked agents process requests and return results to the planning agent. This architecture decouples agent contexts, significantly reducing individual context length burdens and enhancing system scalability. Key benefits include: (1) Context distribution across agents improves overall performance and task accuracy. (2) Customizable workflows for specialized agent optimization. (3) Hot-swappable agent modules through standardized tool interfaces.

- **ReadText(file_path, task_goal):** Invokes a text-reading agent that extracts task-relevant content from the specified file. The agent loads the file's content as context but filters irrelevant portions using task-specific prompts, minimizing noise.

- **GenerateCode(task_goal):** Invokes an Code agent to generate code through LLMs and executes it within a Docker sandbox to accomplish specified parameterized goal. In Figure 3 (left), when code execution errors occur, this agent loads only the previous code snippet and its corresponding error into context for iterative debugging -— each inference focuses on resolving one error. The action then asks LLMs to debug and regenerate the code, focusing each reasoning cycle on resolving a single error. Upon exceeding a predefined debug threshold, the agent returns guidance prompting the planner to re-plan the task. This prevents deadlock scenarios where overly complex tasks cause LLMs to fail in root-cause analysis.

- **GenerateSQL(file_path, task_goal):** Invokes an SQL agent that: (1) fetches all table names from the target database, (2) retrieves schema details relevant to the task, and (3) generates/executes SQL queries (Figure 3, right). Its workflow mirrors the fault-tolerant design of the code-generation agent.

3 Experiments

3.1 Experimental Setting

Benchmark. DA-Code is a code generation benchmark specifically designed for evaluating LLM-based agents in data science tasks. Distinct from conventional code generation benchmarks, this benchmark is designed to enable agents to explore data and leverage programming capabilities to solve challenging objectives, rather than simply translating explicit natural language instructions into code. Unlike existing benchmarks like DS-1000 (Lai et al., 2023) and HumanEval (Chen et al., 2021), which primarily focus on directly converting natural language instructions into executable code, DA-Code establishes a more realistic scenario that simulates real-world data science tasks under given requirements and workspace constraints. DA-Code tasks not only feature inherently complex solutions but also incorporate diverse data sources (databases, spreadsheets, documents, codebases, etc.) containing multifaceted information and data from authentic programming scenarios. Moreover, these information sources may be saturated with noise and extraneous information. We constructed a subset DA-Code-100 containing 100 randomly sampled tasks for evaluation, with difficulty levels distributed as 23 easy, 60 medium, and 17 challenging tasks.

Model	Method	Easy		Medium		Hard		Total	
		Avg@3	Max@3	Avg@3	Max@3	Avg@3	Max@3	Avg@3	Max@3
Qwen2.5-72B	**DA-Agent**	40.77	44.62	22.94	35.70	12.12	21.69	25.21	35.37
	DS-Agent	**49.11**	55.48	25.09	33.66	15.13	**25.84**	28.92	37.35
DeepSeek-V3	**DA-Agent**	44.50	50.27	25.62	29.12	14.87	18.69	28.13	32.21
	DS-Agent	47.38	**57.66**	**29.05**	39.50	**18.98**	22.69	**31.55**	40.82
GPT-4o	**DA-Agent**	38.66	49.67	21.81	29.19	13.31	19.48	24.24	32.25
	DS-Agent	45.03	54.25	27.95	**42.80**	18.31	24.15	30.24	**42.26**

Table 1: Performance comparison between DS-Agent and baselines on selected LLMs. Avg@3 denotes the agent's mean accuracy rate across three testing trials. Max@3 reflects the peak accuracy rate observed during these trials.

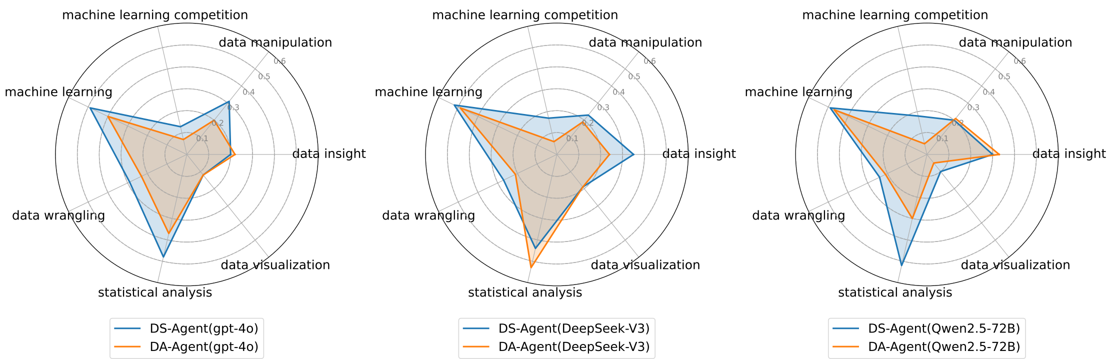

Figure 4: Performance Comparison of DS-Agent and DA-Agent Across Task Categories.

Baselines. To address the challenges posed by the DA-Code benchmark where no existing agent framework has demonstrated sufficient capability, the authors of DA-Code developed DA-Agent, an LLM-based agent framework specialized for complex data analysis through dynamic environment interactions. DA-Agent demonstrates superior performance compared to prevailing agent frameworks including OpenHands (Wang et al., 2024c), AutoGen (Wu et al., 2024), and X-Agent (Team, 2023) in comprehensive evaluations.

Models. We employed two state-of-the-art open-source models, Qwen2.5-72B-Instruct (Yang et al., 2024) and DeepSeek-V3-2024-12-26 (Liu et al., 2024a), as open-source representatives, along with the closed-source model GPT-4o-2024-08-06 (Achiam et al., 2023) as base testing models.

Evaluation Metrics. We evaluate LLM-based agents from diverse foundational models on the DA-Code-100 benchmark through three testing rounds. For each task, we compute both the average score and the best score across these three rounds. Finally, we derive the final Avg@3 and Max@3 metrics by averaging the mean scores and best scores across all 100 tasks respectively.

All models were configured with a temperature of 0, a maximum of 20 action steps, and a 60-second timeout per action execution.

3.2 Experimental Result

3.2.1 Main results

As shown in Table 1, we compared the performance of DS-Agent against baseline methods across various base LLMs. The results demonstrate that DS-Agent achieves superior evaluation metrics across almost all models and difficulty levels, except for a minor 2.04% decrease in maximum score at medium difficulty on Qwen2-72B-Instruct. This indicates our method's enhanced capability to leverage LLMs' reasoning potential in most scenarios. Notably, DS-Agent (GPT-4o) achieves a peak accuracy of 42.26%, representing a 10.01 percentage-point improvement over DA-Agent (GPT-4o), which substantiates that our methodology enables more effective exploitation of LLMs' latent capabilities. Furthermore, DS-Agent (DeepSeek-V3) attains an average accuracy of 31.55%, outperforming its DA-Agent counterpart by 3.42 percentage points, which suggests more consistent performance in complex data analysis tasks.

Figure 4 presents the performance comparison

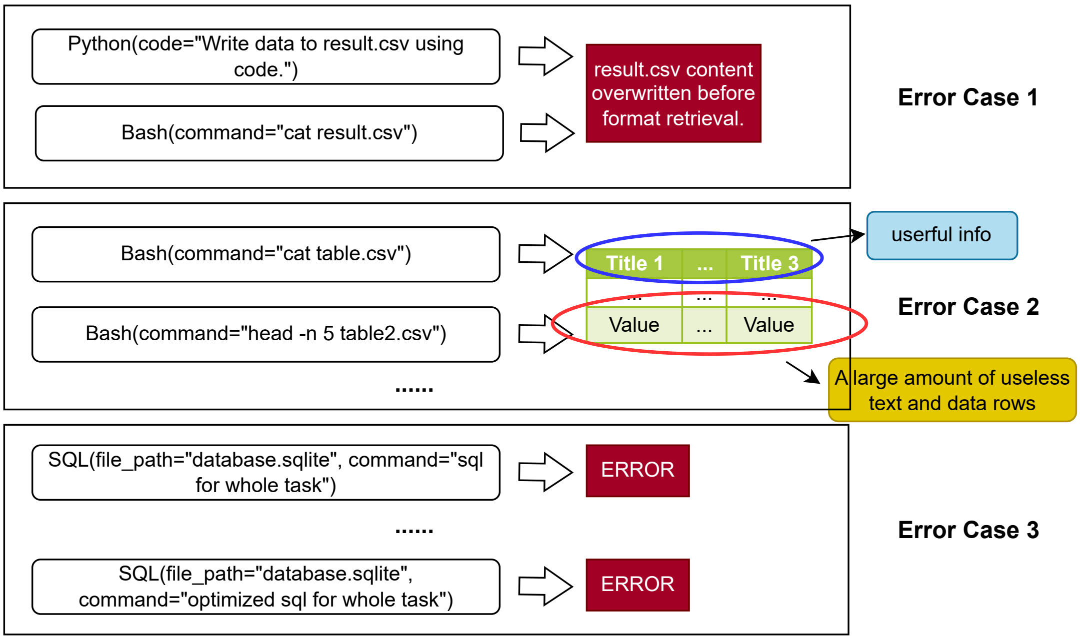

Figure 5: Typical Failure Cases of Existing Agents.

between our DS-Agent and the baseline DA-Agent across various foundation models, with all task categories evaluated using the avg@3 metric. The DS-Agent demonstrates superior performance over the baseline in most task categories, achieving up to several-fold improvements in certain categories. This highlights the generalizability of our agent in data science tasks. For one or two specific task categories where occasional underperformance was observed, we hypothesize this may stem from additional challenges such as requiring extensive domain-specific knowledge or pushing the exploratory capabilities of LLMs beyond their limits.

Model	Method	Round1	Round2	Round3	Total	
		Avg@1	Avg@1	Avg@1	Avg@3	Max@3
GPT-4o	OpenHands	20	38.88	20	26.29	38.88
	DS-Agent	38.38	41.06	32.5	37.31	46.06

Table 2: Comparison with the OpenHands baseline on GPT-4o.

In the DA-Agent study, the authors compared DA-Agent with other agents (OpenHands, AutoGen, X-Agent) on the DA-Code benchmark and achieved the best performance. Due to time constraints, we randomly selected 10 samples from DA-Code for secondary validation comparison with OpenHands (the top-performing agent among the alternatives), conducting three rounds of experiments with the GPT-4o model. As shown in Table 2, OpenHands exhibited significantly lower performance than our proposed DS-Agent on this type of problem.

Experimental Analysis of Framework Limitations and Advantages. In Figure 5, we illustrate classic failure cases of ReAct-based agents represented by DA-Agent:

- **Format Ignorance:** The LLM directly generates data processing code without querying the format of result.csv, leading to final submissions that violate task-specific formatting requirements.
- **Noise Propagation:** When using permissive commands like Bash to retrieve file information, the LLM fails to filter out noisy outputs, overwhelming the agent's context management with redundant text and data lines.
- **Complex Task Handling:** Direct execution of Python/SQL solutions for complex tasks often results in infinite debugging loops when initial holistic attempts fail.

Our multi-agent framework addresses these limitations through three key mechanisms (Figure 6):

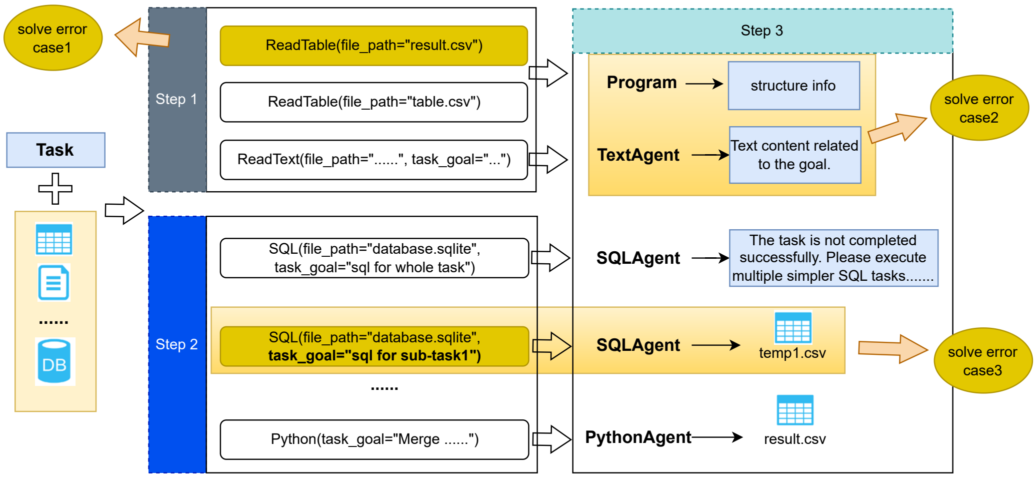

Figure 6: Hierarchical Task Solving Approach of DS-Agent.

- **Global Planning via Tool Strategy:** Step 1 employs a tool planning strategy for systematic retrieval, significantly reducing format-related errors like Case 1.
- **Noise-Resistant Retrieval:** Customized retrieval tools filter irrelevant information, preventing context overload observed in Case 2.
- **Hierarchical Task Decomposition:** The multi-agent architecture enables SQLAgent/PythonAgent to guide the planning module in Step 2 to abandon overly complex tasks, instead pursuing tractable subtasks through feedback loops.

3.3 Ablation Study

3.3.1 Experimental Setup

We investigate the performance degradation in average accuracy and peak accuracy when removing key functional designs of GPT-4o-based Ds-Agent for task processing (Table 3), using three rounds of testing on the DA-Code-100 dataset.

Adequate information enhances the performance of LLMs. As shown in Table 3, removing the global information retrieval step (w/o Global Information Retrieval) while retaining only ReAct-based information acquisition leads to incomplete critical information capture, resulting in performance degradation in both average and peak accuracy. Further removing database information retrieval through SQL agents (w/o Global & Database Information Retrieval) causes additional accuracy reduction.

Information filtering significantly improves LLM performance. While prior studies often use code as the sole interface to avoid tool customization, our experiments reveal critical limitations. In the w/o Information Filtering variant (Table 3), we simulated unconstrained information access (printing first 5 lines for tabular data and full content for text files) while disabling guidance during code/SQL generation failures. This approach causes agents to accumulate excessive irrelevant information in memory contexts, leading to hallucination and catastrophic forgetting. The performance decline stems from LLMs' training on human-written code where print outputs target human readers with inherent "forgetting" mechanisms. Our customized retrieval tools implement artificial memory management by filtering task-irrelevant contexts, thereby enhancing agent performance.

Configuration	ΔAvg@3	ΔMax@3
w/o Global Information Retrieval	-1.44%	-5.02%
w/o Global & Database Information Retrieval	-3.16%	-5.35%
w/o Information Filtering	-3.54%	-3.57%
w/o Muti-Agent Framework & Information Filtering	-3.66%	-7.4%

Table 3: Performance degradation of average and peak accuracy under different ablation configurations.

Multi-agent collaboration proves crucial for complex data science tasks. Removing the multi-agent framework (w/o Multi-Agent Framework & Information Filtering) forces the planning agent to directly generate code/SQL, causing substantial accuracy drops. Decentralizing context through tool-based agent communication allows individual agents to specialize in specific subtasks, demonstrating that distributed contextual management outperforms monolithic processing in complex problem-solving scenarios.

4 Related Work

LLM-based Agent Systems: Agent systems constructed with LLMs have significantly enhanced the performance of LLMs in solving various complex tasks. Currently, there are four primary agent design patterns: (1) Reflection: Enabling agents to review and revise based on self-generated outputs or environmental feedback. SELF-REFINE (Madaan et al., 2023), ReACT (Yao et al., 2023), and Reflexion (Shinn et al., 2023) demonstrate that post-generation reflection effectively improves LLM performance, though single-agent linear planning tends to induce cyclic ineffective recursive iterations. (2) Tool Invocation: Expanding LLM capabilities beyond pure NLP tasks by invoking external APIs. Gorilla (Patil et al., 2024) and ToolLLM (Qin et al., 2024) improve API-calling accuracy through API dataset construction and model fine-tuning, while Chameleon (Lu et al., 2023) enhances LLM performance via plug-and-play module integration. (3) Planning: Leveraging LLMs' reasoning abilities to automate task decomposition and execution planning. Methods like CoT (Wei et al., 2022), PoT (Chen et al., 2023), and SCoT (Li et al., 2025) enhance reasoning performance by generating intermediate reasoning steps before final solutions. (4) Multi-Agent Collaboration: Coordinating multiple role-playing LLMs to accomplish complex tasks. Systems like ChatDev (Qian et al., 2024) and AutoGen (Wu et al., 2024) simulate real-world collaborative environments to solve intricate problems. The DS-Agent architecture mimics human problem-solving patterns in data science tasks through an integrated design combining agent-based reflection mechanisms (error feedback regeneration during code/SQL execution and feedback-guided error correction after repeated errors), tool utilization, strategic planning, and multi-agent coordination. This design ensures sufficient contextual relevance and task-specific validity during LLM inference while strictly adhering to the single-responsibility principle for computational coherence.

Code as Action: LLMs have achieved remarkable results on code generation benchmarks. Given code's universality, many systems employ code as the primary agent-environment interaction medium. VOYAGER (Wang et al., 2024a) enables automated exploration in Minecraft through code-based interactions. CodeAct (Wang et al., 2024b) exclusively uses code for multi-step task solving, while OpenHands (Wang et al., 2024c) extends this paradigm for coding-specific agents. However, prioritizing generality through pure code generation often sacrifices accuracy. DS-Agent employs a tool-planning and ReAct strategy to guide LLMs in acquiring essential task-related information while filtering out redundancies prior to code generation. When tackling complex tasks through code, it steers LLMs to abandon overly intricate tasks in favor of task decomposition. Additionally, it regulates the quality of environmental feedback after LLM-environment interactions. These steps collectively enhance LLMs' performance in solving real-world end-to-end data science tasks.

5 Conclusion

We present DS-Agent, a novel agent framework designed to automate end-to-end data science tasks in real-world scenarios. By integrating tool-augmented planning with the ReAct strategy, DS-Agent acquires critical information from multiple data sources before generating code/SQL and enables inter-agent communication through tool invocation to accomplish complex tasks within limited steps. Our method achieves state-of-the-art accuracy across multiple models and metrics on the challenging DA-Code benchmark. With further optimizations and novel algorithms built upon the current design, we believe DS-Agent can attain even stronger performance in broader application scenarios.

6 Limitations

Domain Limitations: Our approach requires customizing the workflows of agents to replace certain decision-making processes of LLMs, which compromises some degree of generality. However, such trade-offs are inevitable when constructing stable and effective agents, as there exists an inherent tension between generality and determinism.

Inaccuracy Issues: Despite our meticulous design of the agent architecture for accuracy, several failure patterns persist: (1) During two-stage planning, LLMs frequently generate overly complex subsequent action objectives. Even with customized prompting strategies to induce simpler task decomposition, LLMs often reiterate identical problematic goals; (2) When executing ReadText actions, LLMs occasionally extract only partial task-relevant content, with omitted critical information leading to subsequent operations deviating from file-specified requirements; (3) Our current assumption of a compact action space (to preserve LLM context window capacity) may limit handling of complex tasks requiring extensive actions. Potential solutions could involve dynamic retrieval of action sets via RAG (Retrieval-Augmented Generation) (Gao et al., 2023) prior to task planning.

These limitations highlight promising directions for future agent research. While substantial optimization opportunities remain, we maintain these shortcomings do not diminish the significance of our contributions. The current CtxWF framework, despite employing basic code-generated actions and a simplistic error feedback mechanism for LLM reflection, already demonstrates competitive performance. We posit that integrating domain expert workflows (e.g., data scientists' problem-solving patterns) into agent architectures could enable LLMs to generate more effective solutions - a valuable direction for subsequent research.

7 Ethics Statement

The system is designed to augment rather than replace data scientists. By incorporating human cognitive decision-making processes into the agent design architecture, our approach strategically enhances LLM performance in data analysis through controlled restriction of certain decision-making authorities. All datasets employed in this study were sourced from repositories with explicit MIT open-source licenses. The complete implementation codebase, including evaluation datasets, will subsequently be released under the same MIT license to ensure reproducibility and community accessibility.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Ben Alderson-Day and Charles Fernyhough. 2015. Inner speech: Development, cognitive functions, phenomenology, and neurobiology. *Psychological bulletin*, 141(5):931.

Alan Baddeley. 1992. Working memory. *Science*, 255(5044):556–559.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Anthony Steven Dick and Willis F. Overton. 2009. Self- and social-regulation social interaction and the development of social understanding and executive functions.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR.

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, and Weizhu Chen. 2024a. Competition-level problems are effective LLM evaluators. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13526–13544, Bangkok, Thailand. Association for Computational Linguistics.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. 2024b. DA-code: Agent data science code generation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13487–13521, Miami, Florida, USA. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2).

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024b. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*.

Pan Lu, Swaroop Mishra, Tanglin Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *Advances in Neural Information Processing Systems*, volume 35, pages 2507–2521. Curran Associates, Inc.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 43447–43478. Curran Associates, Inc.

Aleksandr Romanovich Luria. 1965. Ls vygotsky and the problem of localization of functions. *Neuropsychologia*, 3(4):387–392.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565. Curran Associates, Inc.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. 2023. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*.

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. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc.

X Team. 2023. Xagent: An autonomous agent for complex task solving. *XAgent blog*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024c. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversation. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Shunyu Yao, Rohan Rao, Matthew Hausknecht, and Karthik Narasimhan. 2020. Keep CALM and explore: Language models for action generation in text-based games. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8736–8754, Online. Association for Computational Linguistics.

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

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Ekaterina Zavershneva and Rene van der Veer. 2018. Thinking and speech.

A Method

A.1 Step 1: Global Information Retrieval

Prompt for the preprocessing steps before proceeding to the next action planning:

```
# ROLE
You are an assistant who evaluates
    whether the current code task
    requires more file information
    according to the rules. If the rules
     are violated, you can only use the
    actions provided in the ACTION SPACE
     to acquire all the necessary info
    that has not been acquired.

# Tools
{retrieval_action_space}

# Rules
1. You need to ensure that I have
    already obtained the necessary file
    information before executing the
    current code task.
2. You should first obtain the relevant
    information about the file before
    saving content to a file.
3. You should ensure that you have
    obtained the format information for
    the specified file.

# Current directory
{files_info}

# Current code task
{current_task}

# RESPONSE FORMAT
1. thought: Based on the information I
    listed above, do reasoning to
    evaluate the code task.
2. actions: All the signature of the
    actions you need.

‘‘‘json
{
    "thought": "thought",
    "actions": ["signature"]
}
‘‘‘
```

A.2 Step 2: Dynamic Planning

A.2.1 Prompt for planning

The prompt used in the first stage to predict the next action:

```
# ROLE
You are a data scientist proficient in
    data analysis, skilled at using code
     to solve data-related problems. You
     can only use the actions provided
    in the ACTION SPACE to determine the
    next action to do. The maximum
    number of the actions you can take
    is {max_steps}.

# ACTION SPACE
{action_space}

# KNOWN FACTS
## Current directory
{files_info}
## Final task
{task}
## Completed action so far
{action_history}

# ATTENTION
1. You need to fully understand the
    action space and its arguments
    before using it.
2. You should first understand the known
     facts before handling the task.
3. You only need to execute the action
    for the same argument once.
4. Before finishing the task, ensure all
     instructions are met and verify the
     existence and correctness of any
    generated files.
5. If a task goal fails multiple times,
    try breaking it down into multiple
    simpler subtasks, and print the
    results of the subtasks or save them
     to a temporary file. Finally, merge
     these files.

# RESPONSE FORMAT
For each task input, your response
    should contain:
1. Based on the information I listed
    above , do reasoning about what the
    next action should be. (prefix "
    Thought: ").
2. One action string in the ACTION SPACE
     (prefix "Action: ").
```

A.2.2 Description of tools

ReadTable

```
## ViewTable Action
* Signature: ViewTable(file_path="path/
    to/table_file")
* Description: This action will get the
    table structure and a portion of the
     data of the table file located at '
    file_path'.
* Constraints:
  - The table file must be accessible
     and in a tabular data format (e.g
    ., .csv, .tsv).
* Example: ViewTable(file_path="info.csv
    ")
```

ReadText

```
## ReadTextFile Action
* Signature: ReadTextFile(file_path="
    path/to/file", task_goal="a detailed
     description of the information you
    want to obtain in the file")
* Description: This action will read the
     file and extract a **relevant
    section of text** from the file
    specified by 'file_path' based on
    the 'task_goal'.
```

```
* Example: ReadTextFile(file_path="info.
    txt", task_goal="the description for
     'money'")
```

GenerateCode

```
## CodeTaskExecutor Action
* Signature: CodeTaskExecutor(task_goal
    ="task_goal")
* Description: This action will generate
     and execute the program code to
    achieve the task goal.
* Example: CodeTaskExecutor(task_goal="
    Print the 'Hello, world!' string.")
```

GenerateSQL

```
## SQLTaskExecutor Action
* Signature: SQLTaskExecutor(file_path="
    path/to/database_file", task_goal="a
     detailed description of the task")
* Description: This action will generate
     and execute the SQL commands on the
     specified database file to achieve
    the task goal.
* Constraints:
  - The database file must be accessible
      and in a format compatible with
     SQLite (e.g., .sqlite, .db).
* Example: SQLTaskExecutor(file_path="
    data.sqlite", task_goal="Calculate
    the average of the quantities.")
```

Decompress

```
## Decompress Action
* Signature: Decompress(file_path="path/
    to/compressed_file")
* Description: This action will extract
    the contents of the compressed file
    located at 'file_path'. It supports
    .zip and .tar and .gz formats.
* Examples:
  - Example1: Decompress(file_path="data
      .zip")
  - Example2: Decompress(file_path="data
      .gz")
```

Answer

```
## Answer Action
* Signature: Answer(output="
    literal_answer_or_output_path")
* Description: This action denotes the
    completion of the entire task and
    returns the final answer or the
    output file/folder path. Make sure
    the output file is located in the
    initial workspace directory.
* Examples:
  - Example1: Answer(output="New York")
  - Example2: Answer(output="result.csv
      ")
  - Example3: Answer(output="FAIL")
```

A.3 Step 3: Implementation of tasks

The implementation details of some tools are as follows:

ReadTable

```
import pandas as pd

pd.set_option('display.max_columns',
    None)
pd.set_option('display.expand_frame_repr
    ', False)

file_path = "{file_path}"
if file_path.endswith('.xlsx') or
    file_path.endswith('.xls'):
    df = pd.read_excel(file_path)
elif file_path.endswith('.tsv'):
    df = pd.read_csv(file_path, sep='\t
        ')
else:
    df = pd.read_csv(file_path)

print(df.head(1))
print('...')
print(f'[{len(df)} rows x {len(df.
    columns)} columns]')
```

ReadText

```
You are a helpful assistant in
    information retrieval. Now I need to
     obtain some information, and you
    should extract the relevant snippets
     from the file content based on the
    descriptions I provide.

The relevant snippets I need to obtain:
‘‘‘
{task_goal}
‘‘‘

The contents of the '{file_path}' file:
‘‘‘
{file_content}
‘‘‘

You should only respond in the format as
     described below:
RESPONSE FORMAT:
For each input, your response should
    contain:
1. One analysis of the query, reasoning
    to determine the required
    information (prefix "Thought: ").
2. One string of the relevant original
    content snippets (prefix "Content:
    ").

Thought: ...
Content:
‘‘‘Plain Text
...
‘‘‘
```

GenerateCode

Prompt for generating code:

```
# ROLE
You are a data scientist proficient in
    data analysis, skilled at using
    Python code to solve data-related
    problems. You can utilize some
    provided APIs to address the current
     task. If you need to print
```

```
    information, please use the print
    function.

# USEFUL APIS
{apis}

# KNOWN FACTS
## Current directory
{files_info}
## Final task
{task}
## Acquired information
{action_history}
## Current task
{current_task}
## Wrong code from the last round
{last_code_info}

# RESPONSE FORMAT
For each task input, your response
    should contain:
1. One analysis of the known facts,
    reasoning to complete the current
    task (prefix "Thought: ").
2. One executable piece of python code
    to achieve the current task (prefix
    "Code: ").
```python
...
```
```

GenerateSQL

Prompt for selecting relevant database table names from the database before generating SQL:

```
# ROLE
You are a database expert, skilled at
    identifying the tables in a database
     that need to be examined further
    based on the current task goal.

# Database table name
```
{tables}
```

# Current task goal
{current_task}

# RESPONSE FORMAT
1. thought: Based on the information I
    listed above, do reasoning to
    evaluate the task.
2. tables: All the name of the tables
    you need to be examined further.

```json
{
    "thought": "thought",
    "tables": []
}
```
```

Prompt for generating SQL:

```
# ROLE
You are a database expert skilled at
    achieving current task goal through
    SQL commands.

# KNOWN FACTS
## Current directory
{files_info}
## Final task
{task}
## Current task
{current_task}
## Acquired information
{action_history}
## Database table names
```
{tables}
```
## Relevant tables structure
{table_columns}
## Wrong sql command from the last round
{last_sql_info}

# RESPONSE FORMAT
1. thought: Based on the information I
    listed above, do reasoning to
    generate the SQL commands to achieve
     the current task goal.
2. sql_command: An SQL command string.
3. output: The file path where the
    results are saved as a CSV file.

```json
{
    "thought": "",
    "sql_command": "",
    "output": ""
}
```
```