Data Interpreter: An LLM Agent For Data Science

Anonymous ACL submission

Abstract

Large Language Model (LLM)-based agents have excelled in various domains but face significant challenges when applied to data science workflows due to their complex, multi-stage nature. Current LLM-based agents struggle with non-linear relationships, recursive dependencies, implicit data- and logic-dependent reasoning, and managing extensive context. In this paper, we introduce **Data Interpreter**, an LLM-based agent that addresses these challenges through hierarchical graph-based modeling to represent the complexity and a progressive strategy for step-by-step verification, refinement, and consistent context management. Extensive experiments confirm the effectiveness of Data Interpreter. On InfiAgent-DABench, it boosts performance by 25% (from 75.9% to 94.9%), and on machine learning and open-ended tasks, it lifts accuracy from 88% to 95% and from 60% to 97%, respectively. Moreover, our method surpasses state-of-the-art baselines by 26% on the MATH dataset. We will release the code upon publication.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in various reasoning tasks (Hong et al., 2023; Wu et al., 2023a; Wang et al., 2023a,b; Chen et al., 2024; Zhang et al., 2024b,a), showcasing their ability to understand complex contexts, generate coherent responses, and even tackle multi-step problem-solving tasks.

Among the many areas where LLMs have been applied, data science stands out as a field of particular importance, but also one that presents unique challenges (Hu et al., 2024; Qin et al., 2020). Data science tasks, including machine learning, data analysis, table-based question answering, and mathematical reasoning, involve multi-stage workflows that require both precise logical and numerical reasoning across various datasets. *These data science workflows are inherently complex and involve multiple steps*, with each task building upon the results of previous ones (Hu et al., 2024; Liu et al., 2024b; Li et al., 2024a). The complexity arises from the interdependencies across different stages, where tasks are not only sequential, but may also involve parallel processes, feedback loops, and recursive relationships. Furthermore, many data science tasks require reasoning that is both *data- and logic-dependent*, introducing implicit dependencies that are not always clearly stated. For example, in machine learning workflows, the transformation of categorical variables across different stages of a pipeline (e.g., encoding methods) may not always be consistent, leading to misalignments that degrade model performance. LLMs may struggle to capture these implicit dependencies, applying different methods inconsistently, which can result in erroneous conclusions and degraded performance.

To address these challenges, several (LLM agent-based) frameworks have been proposed, as shown in Table 1. However, these existing solutions still have significant limitations in tackling the issues faced by LLMs when applied to data science tasks. One major issue is *hallucinations and error propagation*. Errors can compound through dependent tasks, leading to increasingly unreliable results. While most current frameworks include verification mechanisms, as shown in Table 1, their approach of generating complete code at once, rather than step-by-step atomic code, increases the risk of hallucinations propagating through task dependencies. Another challenge is that many data science tasks require reasoning that is both *data- and logic-dependent* as discussed, linear plan structures inadequately capture the often non-linear relationships in data science tasks (Wang et al., 2024d; Rawte et al., 2023) most existing framework, as shown in Table 1. Finally, *contextual memory and long-term dependencies* present a significant challenge. The lengthy steps in data science tasks generate exten-

Table 1: Comparison of DS agent frameworks. **Code Exec.** (Code Execution): indicates how code is executed in real-time; **Memory**: represents the framework's memory structure for storing context and history; **Expandable**: denotes if the framework supports custom extensions and modules; **Domains**: specifies the primary application areas (ML: Machine Learning, DA: Data Analysis, TQA: Table Question Answering, MR: Mathematical Reasoning). * Indicates open-source framework

Framework	Plan Structure	Verification	Code Exec.	Memory	Expandable	Domains
AutoML-GPT (Zhang et al., 2023a)	–	×	×	Raw	×	ML
HuggingGPT* (Shen et al., 2024)	–	×	×	Raw	×	ML, DA, TQA, MR
MLCopilot* (Zhang et al., 2024b)	–	×	×	Raw	×	ML
AutoGen* (Wu et al., 2023a)	Linear	✓	All-at-once	Raw	×	ML, DA, TQA, MR
TaskWeaver* (Qiao et al., 2023)	Linear	✓	Progressive	Raw	✓	ML, DA, TQA, MR
OpenHands* (Wang et al., 2024b)	Linear	✓	All-at-once	Raw	✓	ML, DA, TQA, MR
AIDE* (Schmidt et al., 2024)	Hierarchical	✓	All-at-once	Tree	✓	ML
DS-Agent* (Guo et al., 2024)	Linear	✓	All-at-once	Raw	×	ML
AutoML-Agent (Trirat et al., 2024)	Linear	✓	All-at-once	Raw	×	ML
AutoKaggle* (Li et al., 2024b)	Linear	✓	All-at-once	Raw	✓	ML
Data Interpreter*	Hierarchical	✓	Progressive	Graph	✓	ML, DA, TQA, MR

sive contextual information. However, most current frameworks rely on raw memory structures, which are inadequate for managing relevant context, as shown in Table 1.

To address the above challenges, we propose **Data Interpreter**, a framework that leverages *hierarchical graph-based modeling* to systematically structure and manage task relationships, as shown in Figure 1. By explicitly organizing both high-level task relationships and low-level computational (i.e., action) dependencies into a structured graph format, Data Interpreter ensures a clear representation of the workflow's complexity.

Building on this graph-based structure, Data Interpreter implements a *progressive strategy for managing long-term dependencies*. The framework identifies task dependencies and represents them as a reasoning graph, progressively verifying and refining each node to ensure the continuity of context throughout the process. This progressive verification ensures that earlier steps inform later ones, allowing Data Interpreter to handle complex, multi-step workflows while maintaining coherence and accuracy across extended tasks. This results in a graph-based memory that ensures each task is grounded in a consistent context, minimizing the risk of errors propagating through the workflow.

Our experiments demonstrate that Data Interpreter significantly outperforms existing methods across several benchmarks, achieving a 25% performance boost on the public dataset InfiAgent-DABench (Hu et al., 2024) and a 26% improvement on the MATH dataset (Hendrycks et al., 2021). Compared to other open-source frameworks, Data Interpreter consistently shows notable advancements in machine learning and open-ended tasks.

2 Related Work

LLMs as Data Science Agents LLMs demonstrate expert-level knowledge in machine learning and have made significant progress in automating data science tasks (Xie et al., 2024). Early research focused on using LLMs to write code, aiming to simplify complex computations involved in reasoning processes (Gao et al., 2023; Chen et al., 2022; Zhu et al., 2024). Code interpreters with function-calling mechanisms have become the popular approach for enabling LLMs to handle complex reasoning and scientific tasks (Zhou et al., 2023; Gou et al., 2024; Wang et al., 2024a; Huang et al., 2023b; Hassan et al., 2023; Qiao et al., 2023; Zhang et al., 2024b). Recently, frameworks like AutoML-GPT (Zhang et al., 2023a), MLCopilot (Zhang et al., 2024b), AutoKaggle (Li et al., 2024b), AutoML-Agent (Trirat et al., 2024). Specifically, Zhang et al. (2023b) and Liu et al. (2024a) focus primarily on machine learning tasks but lack comprehensive data science capabilities, particularly in handling multimodal data and automatically detecting and fixing errors in the workflow. Although frameworks such as AutoGen (Wu et al., 2023a), TaskWeaver (Qiao et al., 2023), Agent K (Grosnit et al., 2024), HuggingGPT (Shen et al., 2024), DS-Agent (Guo et al., 2024), and AIDE (Schmidt et al., 2024) support data science scenarios, they face challenges in scalability, sophisticated planning, and effective long context management. End-to-end frameworks tailored for data science tasks are still underdeveloped. To fill this gap, we propose a unified framework designed for data science, thoroughly benchmarked across various tasks and settings, providing key insights

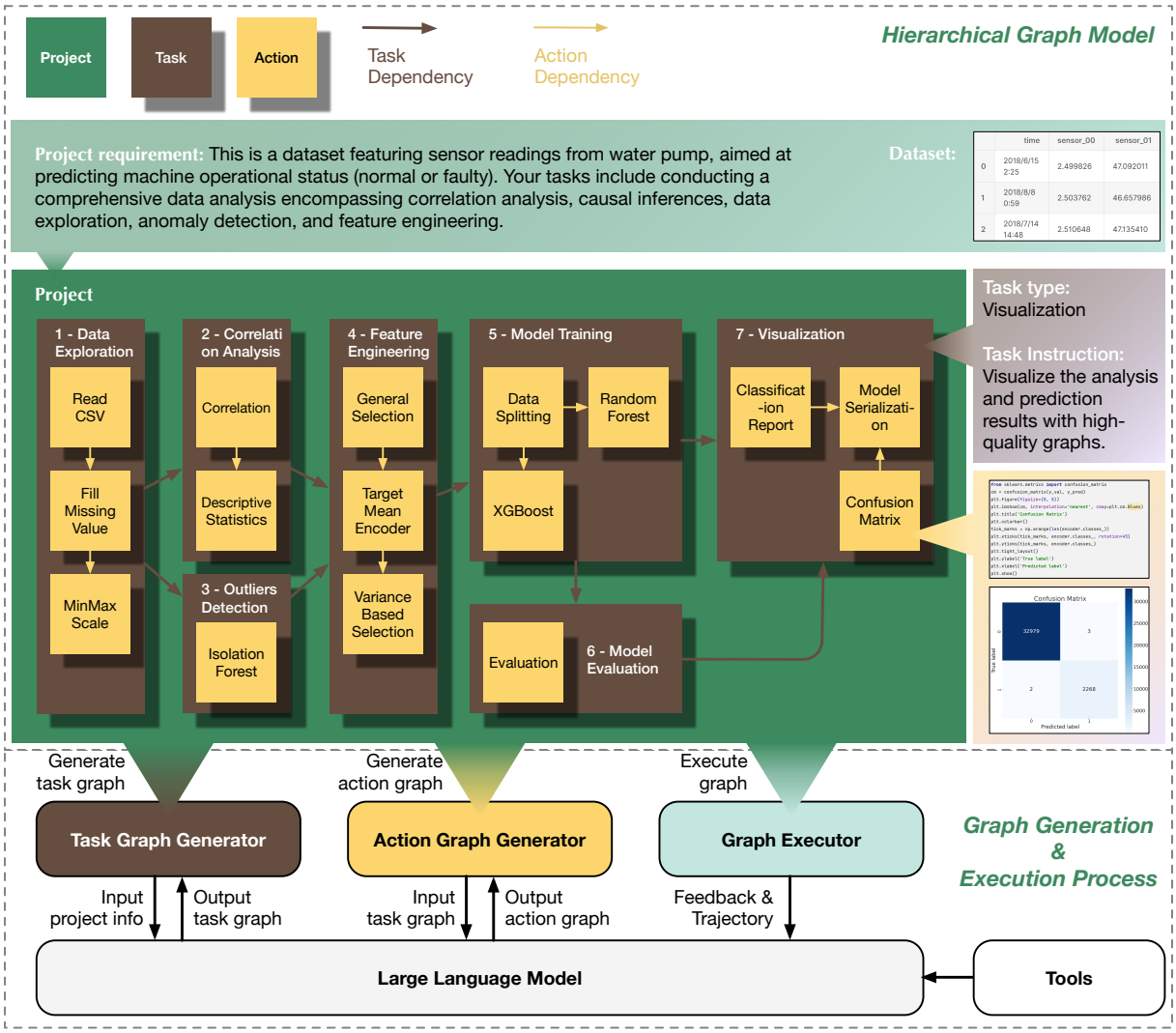


Figure 1: **Data Interpreter Workflow.** The upper section shows how Data Interpreter organizes a data science workflow with a hierarchical structure, starting with decomposing project requirements into a task graph and actions executed via code. The lower section highlights key modules: *task graph generator*, *action graph generator*, and *graph executor*, which work together to manage task execution and provide real-time feedback.

into the effectiveness of LLMs in this field.

Graph-Based Planning for LLM Agents Planning is a crucial capability for LLM-based agents, enabling them to create structured action plans for solving problems (Huang et al., 2024b; Chen et al., 2024). While early approaches like CoT (Wei et al., 2022; Yao et al., 2022) used sequential planning, more recent methods like ToT (Yao et al., 2024) and GoT (Besta et al., 2023) have adopted tree and graph structures to refine LLM prompts. This graph-based paradigm has been further developed in various systems like DSPy (Khattab et al., 2023) and PRODIGY (Huang et al., 2023a), with recent work focusing on enhancing node prompts and agent coordination through graph connectivity (Zhuge et al., 2024; Vierling et al., 2024).

However, these approaches often struggle with multistep, task-dependent problems in data science domains. While OpenHands (Wang et al., 2024b), offers an agent interaction platform with event streaming and sandboxing, it requires improvements in plan management and code verification for complex data science tasks. In this paper, we use a hierarchical structure that adapts to real-time data changes.

3 Methodology

In this section, we first present the foundational formulation of hierarchical graph modeling for data science problems, defining the task graph and action graph in Section 3.1. Next, we detail the iterative process of the hierarchical graph structure in Section 3.2 and illustrate how our Data Interpreter benefits from the graph-based structured memory.

Finally, in Section 3.3, we introduce programmable node generation, explaining how we integrate expertise at different granularities to improve the performance of LLMs.

3.1 Hierarchical Graph Modeling

Data science problems, particularly those involving machine learning, encompass extensive detailing and long-horizon workflows, including data pre-processing, feature engineering, and model training. Drawing inspiration from the application of hierarchical planning in automated machine learning tasks (Mohr et al., 2018; Mubarak and Koeshidayatullah, 2023), we organize the data science workflow via hierarchical structure, which initially decomposes the intricate data science problem into manageable tasks and further breaks down each task into executable code (see Figure 1). Formally, we define the task-solving process as a function P that takes an input x to produce an output $\hat{y} = P(x)$. Our goal is for P to generate solutions that closely approximate or match the anticipated output y. However, due to the complexity of P, which may involve various operations and intermediate data, fully automating the solution to a task is typically challenging (Hutter et al., 2019; Zhuge et al., 2024).

Task Graph. To fully leverage the reasoning capability of LLMs for general task decomposition, our method first decomposes the task-solving process of P into a series of sub-processes $\{p_1, p_2, p_3, \ldots\}$, each of which can be atomic and verifiable. As shown in Figure 1, each sub-process represents a step to complete a specific task. The primary challenge lies in determining the relationships $r = \langle p_i, p_j \rangle \in \mathcal{R}$ between these sub-processes, which define the order of execution: which sub-tasks must be executed first, and which can be executed in parallel or after others.

We represent all sub-processes as *task nodes* within P, where an edge $\langle p_i, p_j \rangle$ indicates that sub-process p_j depends on the output of sub-process p_i, forming a Directed Acyclic Graph (DAG) $\mathcal{G}$ that embodies the entire function P. To execute the task graph, we can compute the task output, which is formally defined as follows:

$$\hat{y} = \mathcal{G}\left(\{p_i(x)\}_{i=1}^{n}, \mathcal{R}\right), \tag{1}$$

where $\mathcal{G}$ represents a DAG composed of the sub-processes $\{p_1, p_2, p_3, \ldots\}$, interconnected through the relationships $\mathcal{R}$, which model the dependencies between tasks.

As shown in Figure 1, for a machine operational status prediction problem, the task graph includes nodes ranging from *data exploration* to *visualization*. The graph topology exhibits complex dependencies that cannot be represented by simple sequential or tree-based structures, as tasks may have multiple predecessors and successors. The detailed task graph representation and the prompt for task decomposition can be found in Appendix C.1.

Action Graph. Each task node expands into an action subgraph within the overall action graph. Specifically, each task node p_i is further decomposed into more granular steps, represented by $\mathcal{A}_i$, forming an implicit graph of atomic operations $\langle o_1, o_2, \ldots \rangle$. These atomic operations correspond to executable code snippets or functions, providing fine-grained control for each task p_i. As illustrated in Figure 1, the visualization task is converted to code snippets, with the confusion matrix calculation handled by *sklearn*. Thus, the complete task-solving process can be expressed as:

$$\hat{y} = \mathcal{G}\left(\{\mathcal{A}_i(x)\}_{i=1}^{n}, \mathcal{R}\right) \tag{2}$$

$\mathcal{A}_i(x) = \langle o_1, o_2, \ldots \rangle$ represents the refined steps for processing input x. Each atomic operation o_j may depend not only on x but also on other parameters or previous operations' outputs. $\mathcal{G}$ connects these atomic action graphs according to the dependency relationships $\mathcal{R}$, forming a comprehensive representation of the entire data science workflow. The dynamic contextual data are automatically managed through inter-task dependencies, making the workflow scalable and flexible for complex applications.

3.2 Iterative Graph Refinement

Graph-based Episodic Memory. As previously discussed, data science tasks generate abundant contextual information due to their lengthy steps. In Data Interpreter, we adopt a graph-based data structure to store context during the reasoning process and provide the memory of reasoning steps with corresponding intermediate results when converting task nodes into action graphs. Specifically, we use the task graph structure to manage agent memory and context. The agent's memory expands and updates along with the task graph refinement, beginning with an initial memory state at task graph

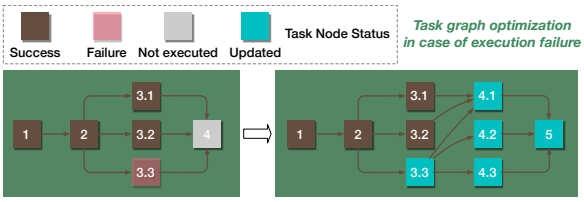


Figure 2: **Task Graph refinement of Data Interpreter.** Task graph refinement for the failed task. After task execution, Task 3.3 fails. The refined task graph integrates existing success tasks, replaces task 3.3 with the updated task 3.3, and introduces new tasks 4.1, 4.2, 4.3, and 5.

initialization. As task nodes are progressively converted into action graphs, Data Interpreter uses a temporary memory to store intermediate data results, generated code, and debugging processes. When a task node's state is updated, the temporary memory is cleared, retaining only the generated code and execution results for the current task node. Consequently, during the problem-solving process, dynamic contextual data is automatically constructed and acquired through task interdependencies. This avoids the need to retrieve the entire context at once, maintaining input relevance and offering flexibility and scalability for broader data science applications.

Iterative Graph Refinement. During task node execution, a task is marked as *Success* if the corresponding code executes successfully. If it fails, Data Interpreter leverages LLMs to debug the code based on runtime errors, making up to a predefined number of attempts to resolve the issue. If the problem persists after the set attempts, the task node is flagged as *Failure*, as shown in Figure 2.

To ensure runtime verification and provide real-time feedback during execution, Data Interpreter incorporates a stateful graph executor that manages both execution and debugging using reflection mechanisms (Shinn et al., 2024). Specifically, if the execution encounters exceptions or fails a verification check, the action graph generator dynamically reflects on the execution results and then regenerates the code to resolve the issue or optimize the output, providing data-driven feedback.

For failed tasks, Data Interpreter regenerates the task graph based on current episodic memory and the execution context, as depicted in Figure 2. Given the task dependencies, the regenerated task graph is sorted topologically and compared to the original using a prefix matching algorithm (Wagner, 2000) to identify differences in task descriptions. This comparison helps identify divergence points (forks), and the final output includes all unchanged tasks before the fork, along with any new or modified tasks after the fork. This approach allows Data Interpreter to efficiently locate the parent node of the failed task and seamlessly integrate the newly generated task and its subsequent tasks into the original graph. It directly leverages the completed memory of all dependent tasks during re-execution, avoiding unnecessary code regeneration or redundant executions.

Using continuous monitoring and iterative updates, Data Interpreter avoids the inefficiencies associated with generating all tasks upfront. This dynamic adjustment of code and planning, based on task outcomes, allows for modifications at various levels of granularity, greatly enhancing overall efficiency.

3.3 Programmable Node Generation

Action Node. As described in Section 3.1, action graph $\mathcal{A}_i(x) = \langle o_1, o_2, \ldots \rangle$, is represented in code format as an implicit graph of various operations. Here, we define the operators as action nodes. An action node encapsulates executable computational logic, integrating both tool-based operations and application programming interface (APIs) into cohesive code snippets.

Programmable Node Generation. Effective tool selection and integration, particularly in the context of task-specific requirements, play a crucial role in the success of task execution, as noted in prior research (Qian et al., 2023; Yuan et al., 2024; Huang et al., 2024a; Liu et al., 2023). In Data Interpreter, we leverage the typology and description of tasks to enrich the task-specific context, thereby enhancing the decision-making process for tool selection and code generation.

Given each task description p_i, Data Interpreter retrieves candidate tools from the toolset $T = \{t_1, t_2, \ldots, t_n\}$, ranks them by functionality relevance, and selects the top-k tools for the task.

Instead of generating isolated function calls, Data Interpreter integrates tools, APIs, and code snippets in context into a context-aware operation. This process can form three levels of advanced operations: 1) Basic tool extension with added functionality, 2) Tool chaining via concatenating their outputs, creating a sequential flow of tools, and 3) Nested tool calls with control logic for complex de-

Table 2: **Performance comparisons on Data Analysis.** Results marked with an asterisk (*) are reported by Hu et al. (2024). Rows marked with a dagger symbol (†) indicate the w/o Agent baseline for comparison. The Δ column represents the accuracy improvement of the agent framework compared to the w/o agent setups. The best results are highlighted in bold. *C.Accuracy* indicates Competition-level Accuracy, and *RPG* refers to the Relative Performance Gap metric.

Methods	Model	Metric	Δ (%)
InfiAgent-DABench		**Accuracy**	
Model-only	*gemini-pro*	56.42*	-
	gpt-3.5-turbo-0613	60.70*	-
	gpt-4-0613	78.99*†	-
	gpt-4-0613	75.21	-
	gpt-4o	75.92†	-
XAgent	*gpt-4-0613*	47.53*	-31.46
AutoGen	*gpt-4-0613*	71.49	-7.50
Data Interpreter	*gpt-4-0613*	73.55	-5.44
Data Interpreter	*gpt-4o*	**94.93**	**+19.01**
DS-Bench Data Modeling		**RPG**	
AutoGen	*gpt-4o*	34.74	-
AutoGen	*gpt-4o-mini*	11.24	-
Data Interpreter	*gpt-4o*	52.43	+50.92
Data Interpreter	*gpt-4o-mini*	**37.84**	**+236.65**
DS-Bench Data Analysis		**C.Accuracy**	
AutoGen	*gpt-4o*	26.72	-
AutoGen	*gpt-4o-mini*	21.01	-
Data Interpreter	*gpt-4o*	**28.75**	+7.60
Data Interpreter	*gpt-4o-mini*	24.46	**+16.42**

pendencies. Programmable node generation relies on in-context learning with retrieved context, ensuring efficient and adaptive tool integration. The prompt for programmable node generation can be found in Figure 6. This can be viewed as developing advanced and composite tool forms.

4 Experiments

4.1 Experimental setup

Data Analysis. For data analysis tasks, we evaluated our approach using two publicly available benchmarks: InfiAgent-DABench (Hu et al., 2024) and DS-Bench (Jing et al., 2024). These benchmarks are specifically designed to comprehensively evaluate LLM performance in real-world data analysis tasks. Following the evaluation setups in these benchmarks, we used accuracy as the primary metric for InfiAgent-DABench and competition-level accuracy, which is calculated by averaging the accuracy scores obtained from each competition for DS-Bench. Notably, data modeling tasks in DS-Bench are also reported in Table 2 with the RPG (Jing et al., 2024) metric. We conducted comparative evaluations mainly against AutoGen (Wu et al., 2023a), utilizing *gpt-4o*, *gpt-4-0613* and *gpt-4o-mini* with temperature set to 0 following the original benchmark configurations. The details of the benchmark are in Appendix B.1.

Machine Learning. For machine learning tasks, we crafted a dataset named ML-Bechmark, consisting of 8 Kaggle machine learning tasks (details in Table 12. We also detail the evaluation metrics on the ML-Benchmark in Appendix B.2. We compared with a broad range of baselines, including XAgent (Team, 2023), AutoGen, OpenInterpreter (Lucas, 2023), TaskWeaver (Qiao et al., 2023), and OpenHands (Wang et al., 2024c). By default, we used *gpt-4-1106-preview* with temperature set to 0.

Mathematical Reasoning. We evaluated four categories (C.Prob, N.Theory, Prealg, Precalc) of level-5 problems from the MATH dataset (Hendrycks et al., 2021), following the setting of (Wu et al., 2023b). The level-5 problems were chosen for their complexity and the challenges in reliable numeric interpretation. We used MathChat (Wu et al., 2023b) and AutoGen (Wu et al., 2023a) as baselines for the MATH benchmark.

Open-ended Task. To verify the capability for dynamic data handling, we also crafted the Open-ended task benchmark comprising 20 tasks. Details about the dataset are in the Appendix B.1. We adopted AutoGen, OpenInterpreter, and OpenHands as baselines, with average results reported over three runs. We adopted *gpt-4-1106-preview* with the temperature set to 0.

4.2 Main Results

Performance on Data Analysis. As demonstrated in Table 2, with *gpt-4-0613*, Data Interpreter achieved a score of 73.55, outperforming AutoGen by 2.9%. In particular, it still did not surpass the performance of directly invoking the LLM. We found that this is primarily due to the growing context overhead in the problem-solving process, where the context length exceeds the maximum window size of *gpt-4-0613*, leading to task failures. However, by incorporating LLMs like *gpt-4o* with longer context windows, Data Interpreter demonstrated outstanding performance, improving results by 25% compared to direct LLM inference. This indicates that Data Interpreter significantly enhances the LLM's multi-step reasoning capabilities across

Table 3: **Performance Comparisons on Machine Learning Task Benchmarks.** This table reports the comprehensive score of each task. "WR", "BCW", "ICR", "SCTP", and "SVPC" represent "Wine recognition", "Breast cancer wisconsin", "ICR - Identifying age-related conditions", "Santander customer transaction prediction", and "Santander value prediction challenge", respectively.

Model / Task	WR	BCW	Titanic	House Prices	SCTP	ICR	SVPC	Avg.	Cost ($)
AutoGen	0.96	**0.99**	0.87	0.86	0.83	0.77	0.73	0.86	-
OpenInterpreter	**1.00**	0.93	0.86	0.87	0.68	0.58	0.44	0.77	-
TaskWeaver	**1.00**	0.98	0.63	0.68	0.34	0.74	0.48	0.69	**0.37**
XAgent	**1.00**	0.97	0.42	0.42	0	0.34	0.01	0.45	20.09
OpenDevin	0.98	0.98	0.87	0.94	0.93	0.73	0.73	0.88	3.01
Data Interpreter	0.98	**0.99**	**0.91**	**0.96**	**0.94**	**0.96**	**0.89**	**0.95**	0.84

Table 4: **Performance Comparisons on Open-ended Task Benchmarks.** This table reports the completion rate of each task. The tested tasks include "OCR" (Optical Character Recognition), "WSC" (Web Search and Crawling), "ER" (Email Reply), "WPI" (Web Page Imitation), "IBR" (Image Background Removal), "T2I" (Text-to-Image), "I2C" (Image-to-Code), and "MGG" (Mini Game Generation).

Model / Task	OCR	WSC	ER	WPI	IBR	T2I	I2C	MGG	Avg.	Cost ($)
AutoGen	0.67	0.65	0.10	0.26	**1.00**	0.10	0.20	0.67	0.46	-
OpenInterpreter	0.50	0.30	0.10	0.36	**1.00**	0.50	0.25	0.20	0.40	-
OpenDevin	0.60	0.87	0.10	0.16	**1.00**	0.50	0.80	0.90	0.60	1.41
Data Interpreter	**0.85**	**0.96**	**0.98**	**1.00**	**1.00**	**1.00**	**1.00**	**0.93**	**0.97**	**0.41**

a wide range of data analysis tasks, especially as the number of interaction rounds increases and the context overhead grows.

For DS-Bench Data Analysis tasks, compared to AutoGen, Data Interpreter showed notable improvements of 7.60% and 16.42% in competition-level accuracy when using *gpt-4o* and *gpt-4o-mini* respectively.

Performance on Machine Learning. As shown in Table 3, Data Interpreter achieved a comprehensive score of 0.95 across tasks, outperforming AutoGen (0.86) and OpenHands (0.88) by 10.3% and 7.9%, respectively. It was the only framework to achieve a score above 0.9 on tasks such as Titanic, House Prices, SCTP, and ICR. Additionally, the Data Interpreter demonstrated a significant advantage over other frameworks, with improvements of 31.5% and 21.9% over OpenHands on the ICR and SVPC tasks, respectively. Notably, Data Interpreter solved the tasks more efficiently, achieving an average score of $ 0.84 while operating at only 27.9% of OpenHands's cost. Data Interpreter consistently completed all mandatory processes in all datasets, maintaining superior performance. Further details can be found in Table 5 in the Appendix.

In data modeling tasks on DS-Bench, Data Interpreter exhibited substantial performance gains compared to AutoGen, with a 50.92% RPG improvement using *gpt-4o* and a remarkable 236.65% improvement using *gpt-4o-mini*, as shown in Table 2. These significant enhancements demonstrate Data Interpreter's superior capabilities in handling complex modeling tasks across different model configurations.

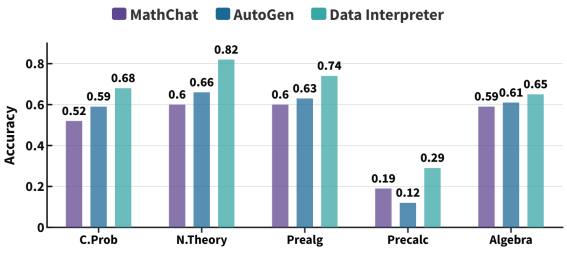


Figure 3: **Performance on the MATH dataset.** We evaluate all the problems with difficulty level 5 from 4 categories of the MATH dataset.

Performance on MATH Problem. As illustrated in Figure 3, Data Interpreter achieved the best results across all tested categories, reaching 0.82 accuracy in the N.Theory category, marking a 0.16 improvement over AutoGen performance. In the category with the most challenging, Precalc, Data Interpreter obtained an accuracy of 0.29, an increase of 0.17 compared to AutoGen. On average, our Data Interpreter showed 26.5% relative improvement compared to AutoGen.

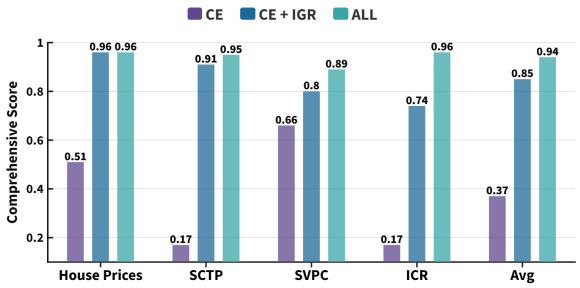


Figure 4: **Ablation on core modules.** Evaluated with Comprehensive Score on ML-Benchmark. "IGR" stands for Iterative Graph Refinement, and "PNG" denotes Programmable Node Generation. "ICR", "SCTP", and "SVPC" represent "ICR - Identifying age-related conditions", "Santander customer transaction prediction", and "Santander value prediction challenge", respectively.

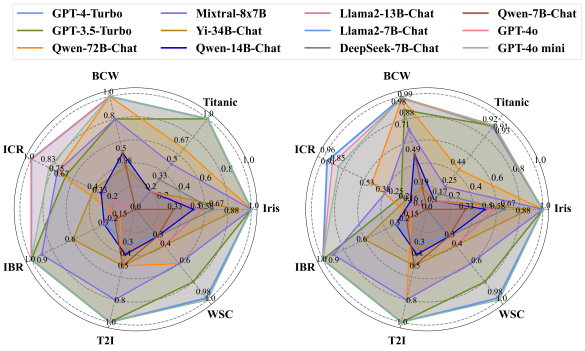


Figure 5: **Evaluation on ML-Benchmark.** Left: completion rate. Right: comprehensive score.

Performance on Open-ended Tasks. Table 4 illustrates that the Data Interpreter achieved a completion rate of 0.97, marking a substantial 110.8% improvement compared to AutoGen and 61.7% improvement compared to OpenHands. In OCR-related tasks, the Data Interpreter maintained an average completion rate of 0.85, outperforming AutoGen, OpenInterpreter, and OpenHands by 26.8%, 70.0%, and 41.7%, respectively. In the tasks requiring multiple steps and utilizing multimodal tools/interfaces, such as WPI, I2C, and T2I, the Data Interpreter emerged as the sole method to execute all steps. Baseline frameworks failed to log in and obtain the status of the ER task, resulting in a lower completion rate. In contrast, Data Interpreter dynamically adjusted to task requirements, achieving a completion rate of 0.97.

4.3 Ablation Study

Ablation on core modules. We conducted ablation experiments with three configurations on the ML-Benchmark. First, we used ReAct (Yao et al., 2022) for code execution with simplified prompts, followed by the addition of iterative graph refinement, and finally, programmable node generation was introduced, using the Data Interpreter as the default. As shown in Figure 4, iterative graph refinement improved performance by 0.48, enhancing dataset preparation and real-time tracking. Programmable node generation further boosted the comprehensive score by 10.6%, reaching 0.94. We detailed the results in Table 11.

Ablation on different base LLMs. Based on GPT-4o and GPT-4o-mini, Data Interpreter shows further improvement in task completion across a wide range of tasks, as illustrated in Figure 5. In machine learning tasks, LLMs like Qwen-72B-Chat (Bai et al., 2023) and Mixtral-8x7B (Jiang et al., 2024) performed comparably to GPT-3.5-Turbo, while smaller LLMs experienced performance degradation. Our Data Interpreter handled data loading and analysis effectively with smaller models but had limitations with tasks requiring advanced coding proficiency. Mixtral-8x7B achieved high completion rates in three tasks, but faced challenges in the WSC task. Smaller LLMs also encountered execution failures due to restricted coding abilities when acquiring images or parsing webpage results, as shown in Figure 5.

5 Conclusion

We introduced Data Interpreter, an LLM-based agent that addresses data science challenges through a novel hierarchical graph representation. By continuously monitoring data changes and adapting to dynamic environments via iterative task refinement and graph optimization, it robustly manages data analysis, machine learning, and reasoning tasks. Leveraging hierarchical decomposition, fine-grained execution, validation, and iterative modifications, Data Interpreter harnesses the LLM's planning and coding abilities to tackle complex multi-step workflows. Extensive experiments confirm its superiority over state-of-the-art open-source frameworks in machine learning, mathematical problem-solving, and real-world applications, marking a significant advance in LLM-driven data science solutions.

6 Limitations

Insufficient diversity and complexity. Our novel framework Data Interpreter outperforms other open-source frameworks on machine learning problems, yet are limited to entry-level Kaggle datasets and benchmarked against the capabilities of a junior human data scientist. These datasets are relatively small (under 500MB), with a limited number of columns (in the hundreds) and rows (in the tens of thousands), and mainly involve classification and regression tasks (as described in Appendix D.2). However, we have not yet evaluated our Data Interpreter on more challenging datasets involving large-scale data or complex tasks such as time series analysis, multi-label classification, or multi-table problems. In our future work, we plan to expand our dataset collection to include these types of problem to thoroughly evaluate our framework's performance and capabilities. **Precise self-improvement.** Human data scientists usually perform multiple experiments on a dataset, focusing on pipeline optimization and hyperparameter tuning (Liu et al., 2021; Hutter et al., 2019). Our Data Interpreter integrates experience to enhance the node generation quality. The experience primarily involves tracking the progress of tasks and code. However, it does not use numerical feedback from multiple experiences to develop and refine specific strategies, such as increasing the learning rate or using an ensemble technique, to continuously improve performance for a given dataset, thus lacking the capability for automatic self-improvement. In the future, we aim to address this limitation by developing mechanisms that allow our model to conduct multiple experiments and derive insights from the numerical feedback for a given dataset on its own. **DAG constraint detection mechanism.** Our current implementation does not include an explicit DAG constraint detection mechanism, we rely on the LLM's inherent ability to avoid cycles during task planning, as observed in our experiments. However, such mechanisms could enhance robustness in handling less structured domains or highly complex dependencies. Incorporating cycle detection and resolution strategies in future iterations would ensure improved reliability and adaptability across diverse applications. **Full-scale evaluation on mathematical problems.** For the MATH problem, our experiments are limited to level-5 problems, primarily due to the budget constraints, we will explore more cost-effective strategies to evaluate our Data Interpreter on a wider range of mathematical problems in future studies.

7 Ethics Statement

This study introduces **Data Interpreter** to systematically structure and manage task relationships, emphasizing legal and ethical compliance throughout its deployment. We utilize only authorized and de-identified data to uphold fairness and inclusivity in training and system design, thereby minimizing bias. Our process is transparent, with detailed sharing of methodology and outcomes to ensure reproducibility. The deployment of this method is conducted responsibly, limiting its application to lawful and beneficial purposes. We encourage active collaboration and feedback to continuously refine our approach, focusing on its fairness, accountability, and positive societal impact.

References

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. 2023. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint*.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering.

Jiaqi Chen, Yuxian Jiang, Jiachen Lu, and Li Zhang. 2024. S-agents: self-organizing agents in open-ended environment.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *ICML*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2024. ToRA: A tool-integrated reasoning agent for mathematical problem solving.

Antoine Grosnit, Alexandre Maraval, James Doran, Giuseppe Paolo, Albert Thomas, Refinath Shahul Hameed Nabeezath Beevi, Jonas Gonzalez, Khyati Khandelwal, Ignacio Iacobacci, Abdelhakim Benechehab, Hamza Cherkaoui, Youssef Attia El-Hili, Kun Shao, Jianye Hao, Jun Yao, Balazs Kegl, Haitham Bou-Ammar, and Jun Wang. 2024. Large language models orchestrating structured reasoning achieve kaggle grandmaster level.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*.

Md Mahadi Hassan, Alex Knipper, and Shubhra Kanti Karmaker Santu. 2023. Chatgpt as your personal data scientist.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks.

Qian Huang, Hongyu Ren, Peng Chen, Gregor Kržmanc, Daniel Zeng, Percy Liang, and Jure Leskovec. 2023a. Prodigy: Enabling in-context learning over graphs.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023b. Benchmarking large language models as ai research agents.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, et al. 2024a. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024b. Understanding the planning of llm agents: A survey.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated machine learning: methods, systems, challenges*. Springer Nature.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. Dsbench: How far are data science agents to becoming data science experts?

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to SQL: are we fully ready? *Proc. VLDB Endow.*, 17(11):3318–3331.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, Wanjun Zhong, Wangchunshu Zhou, Wenhao Huang, and Ge Zhang. 2024b. Autokaggle: A multi-agent framework for autonomous data science competitions.

Siyi Liu, Chen Gao, and Yong Li. 2024a. Large language model agent for hyper-parameter optimization. *arXiv preprint arXiv:2402.01881*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024b. A survey of nl2sql with large language models: Where are we, and where are we going? *Preprint*, arXiv:2408.05109.

Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Zhiheng Li, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, et al. 2023. Controlllm: Augment language models with tools by searching on graphs.

Zhengying Liu, Adrien Pavao, Zhen Xu, Sergio Escalera, Fabio Ferreira, Isabelle Guyon, Sirui Hong, Frank Hutter, Rongrong Ji, Julio CS Jacques Junior, et al. 2021. Winning solutions and post-challenge analyses of the chalearn autodl challenge 2019. *TPAMI*.

Killian Lucas. 2023. GitHub - KillianLucas/open-interpreter: A natural language interface for computers. https://github.com/KillianLucas/open-interpreter.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. 2018. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*.

Yousef Mubarak and Ardiansyah Koeshidayatullah. 2023. Hierarchical automated machine learning (automl) for advanced unconventional reservoir characterization. *Scientific Reports*.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, Minghua Ma, Pu Zhao, Si Qin, Xiaoting Qin, Chao Du, Yong Xu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2023. Taskweaver: A code-first agent framework.

Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. 2020. Making data visualization more efficient and effective: a survey. *VLDB J.*, 29(1):93–117.

Vipula Rawte, Amit Sheth, and Amitava Das. 2023. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922*.

Dominik Schmidt, Zhengyao Jiang, and Yuxiang Wu. 2024. Introducing weco aide.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *NeurIPS*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning.

XAgent Team. 2023. Xagent: An autonomous agent for complex task solving. https://github.com/OpenBMB/XAgent.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. 2024. Automl-agent: A multi-agent llm framework for full-pipeline automl.

Lukas Vierling, Jie Fu, and Kai Chen. 2024. Input conditioned graph generation for language agents.

Marcel Waldvogel. 2000. Fast longest prefix matching: algorithms, analysis, and applications. *Doctoral dissertation, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024b. Openhands: An open platform for ai software developers as generalist agents.

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

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023a. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. In *NeurIPS*.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, Xiaojian Ma, and Yitao Liang. 2023b. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *arXiv preprint arXiv:2311.05997*.

Zihao Wang, Anji Liu, Haowei Lin, Jiaqi Li, Xiaojian Ma, and Yitao Liang. 2024d. Rat: Retrieval augmented thoughts elicit context-aware reasoning in long-horizon generation. *arXiv preprint arXiv:2403.05313*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023a. Autogen: Enabling next-gen llm applications via multi-agent conversation framework.

Yiran Wu, Feiran Jia, Shaokun Zhang, Qingyun Wu, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, and Chi Wang. 2023b. An empirical study on challenging math problem solving with gpt-4.

Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. Haichart: Human and AI paired visualization system. *Proc. VLDB Endow.*, 17(11):3178–3191.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*.

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

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. 2024. Easytool: Enhancing llm-based agents with concise tool instruction.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2024a. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. 2024b. Mlcopilot: Unleashing the power of large language models in solving machine learning tasks.

Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. 2023a. Automl-gpt: Automatic machine learning with gpt.

Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. 2023b. Data-copilot: Bridging billions of data and humans with autonomous workflow. *arXiv preprint arXiv:2306.07209*.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. 2023. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification.

Yizhang Zhu, Shiyin Du, Boyan Li, Yuyu Luo, and Nan Tang. 2024. Are large language models good statisticians? In *NeurIPS*.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jurgen Schmidhuber. 2024. Language agents as optimizable graphs.

A Broader Impact

Our work has the potential to significantly reduce the costs associated with a wide range of customized data science tasks, empowering professionals in the field to enhance their automation capabilities and efficiency. However, the flexibility of tools integration, while convenient for local code snippets integration, comes with potential risks. For example, if users provide malicious code intended for unauthorized system penetration or web attacks, it could lead to security vulnerabilities. In our experiments, we mitigate this risk by prompting our Data Interpreter to check the codes before generating new codes. Additional saftguards against these risks include collaborating exclusively with LLMs that adhere to robust safety policies.

B Experiment Details

B.1 Dataset

InfiAgent-DABench InfiAgent-DABench focuses on evaluating the data analysis capabilities of agents. It comprises 257 data analysis problems, categorized into the following seven areas and their combinations: summary statistics, feature engineering, correlation analysis, machine learning, distribution analysis, outlier detection, and comprehensive data preprocessing. Each category includes problems with varying difficulty levels. In the following, we present some specific prompt cases to provide an intuitive understanding of the task settings in InfiAgent-DABench.

Machine Learning Benchmark. This dataset encompassed eight representative machine learning tasks categorized into three difficulty levels, ranging from easy (level 1) to the most complex (level 3). Each task was accompanied by data, a concise description, standard user requirements, suggested steps, and metrics (see Table 12 in the Appendix). For tasks labeled as “toy", the data were not divided into training and test splits, which required the framework to perform data splitting during modeling.

Open-ended Task Benchmarks. To evaluate the ability to generalize to real-world tasks, we developed the Open-ended task benchmark, comprising 20 tasks. Each task required the framework to understand user needs, break down complex tasks, and execute code. They delineated their requirements, foundational data or sources, completion steps, and specific metrics. The scope was broad, encompassing common needs like Optical Character Recognition (OCR), web search and crawling (WSC), automated email replies (ER), web page imitation (WPI), text-to-image conversion (T2I), image-to-HTML code generation (I2C), image background removal (IBR), and mini-game generation (MGG). We present about these tasks in Figure 15 and Figure 16 in the Appendix.

MATH Dataset. The MATH dataset (Hendrycks et al., 2021) comprises 12,500 problems, with 5,000 designated as the test set, covering various subjects and difficulty levels. These subjects include Prealgebra (Prealg), Algebra, Number Theory (N.Theory), Counting and Probability (C.Prob), Geometry, Intermediate Algebra, and Precalculus (Precalc), with problems categorized from levels "1" to "5" based on difficulty. Following the setting of Wu et al. (Wu et al., 2023b), we evaluated four typical problem types (C.Prob, N.Theory, Prealg, Precalc), excluding level-5 geometry problems from the test set.

B.2 Evaluation Metrics

In the MATH benchmark (Hendrycks et al., 2021), accuracy served as the chosen evaluation metric, aligning with the setting proposed in (Wu et al., 2023b; Hendrycks et al., 2021).

For the ML-Benchmark, three evaluation metrics were utilized: completion rate (CR), normalized performance score (NPS), and comprehensive score (CS). These metrics provided comprehensive insights into the model performance and were defined as follows:

Completion rate (CR): In the task requirements description, there were T steps, and the task completion status of each step was denoted by a score s_t, with a maximum score s_{max} of 2 and a minimum score s_{min} of 0. The task completion status categories were defined as follows: missing (score of 0), fail (score of 0), success - non-compliant (score of 1), success-compliant (score of 2), and optional step (not involved in scoring). To measure the completion level, we proposed a completion ratio where the numerator was the sum of scores s_t for each step, and the denominator was the sum of the maximum possible scores for all steps ($s_{max} \times T$):

$$\text{CR} = \frac{\sum_{t=1}^{T} s_t}{s_{max} \times T}. \tag{3}$$

Normalized performance score (NPS): In our ML-Benchmark, each task was associated with its

```
# Capabilities
- You can utilize pre-defined tools in any code lines from 'Available Tools' in the
    form of Python Class.
- You can freely combine the use of any other public packages, like sklearn, numpy,
    pandas, etc..

# Available Tools:
Each Class tool is described in JSON format. When you call a tool, import the tool
    from its path first.
{tool_schemas}

# Output Example:
when the current task is "do data preprocess, like fill missing value, handle
    outliers, etc.", the code can be like:
```python
# Step 1: fill missing value
# Tools used: ['FillMissingValue']
from metagpt.tools.libs.data_preprocess import FillMissingValue

train_processed = train.copy()
test_processed = test.copy()
num_cols = train_processed.select_dtypes(include='number').columns.tolist()
if 'label' in num_cols:
    num_cols.remove('label')
fill_missing_value = FillMissingValue(features=num_cols, strategy='mean')
fill_missing_value.fit(train_processed)
train_processed = fill_missing_value.transform(train_processed)
test_processed = fill_missing_value.transform(test_processed)

# Step 2: handle outliers
for col in num_cols:
    low, high = train_processed[col].quantile([0.01, 0.99])
    train_processed[col] = train_processed[col].clip(low, high)
    test_processed[col] = test_processed[col].clip(low, high)
```end

# Constraints:
- Ensure the output new code is executable in the same Jupyter notebook with the
    previous tasks code has been executed.
- Always prioritize using pre-defined tools for the same functionality.
- Always copy the DataFrame before processing it and use the copy to process.
```

Figure 6: One-shot tool usage prompt

evaluation metric, which may vary between tasks, including metrics such as accuracy, F1, AUC, RMSLE, etc. For metrics such as accuracy, F1, and AUC, we presented the raw values to facilitate comparison across identical data tasks. We normalize all performance values s:

$$\text{NPS} = \begin{cases} \dfrac{1}{1+s}, & \text{if } s \text{ is smaller the better} \\ s, & \text{otherwise.} \end{cases} \tag{4}$$

This transformation ensured that loss-based metrics like RMSLE are scaled from 0 to 1, with higher normalized performance score values indicating better performance.

Comprehensive score (CS): To simultaneously assess both the completion rate of task requirements and the performance of generated machine learning models, we calculated the weighted sum of CR and NPS as follows:

$$\text{CS} = 0.5 \times \text{CR} + 0.5 \times \text{NPS}. \tag{5}$$

Considering the lack of unified performance standards for Open-ended tasks, we default to NPS $= 0$ and directly equate CS to CR.

DS-Bench. DS-Bench (Jing et al., 2024), a comprehensive benchmark with 466 data analysis and 74 data modeling tasks from Eloquence and Kaggle competitions, designed to evaluate data science agents in realistic settings involving long contexts, multimodal tasks, large data files, multi-table structures, and end-to-end data modeling. We followed

the DS-Bench evaluation setup, randomly sampling 64 tasks for data analysis and 10 tasks for data modeling in our experiments.

MLE-Benchmark. MLE-Benchmark (Chan et al., 2024) assesses AI agents' capabilities in machine learning engineering through 75 curated Kaggle competitions, focusing on model training, dataset preparation, and experimental execution. In this paper, we designed MLE-Bench-Lite for efficient evaluation, selecting eight random tasks from MLE-Bench and testing them with a 3-hour time limit per task, in contrast to the 24-hour limit for AIDE (Schmidt et al., 2024) and OpenDevin (Wang et al., 2024c). The experimental setup utilized a single 24GB GPU, 125GB memory, and a 36-core CPU, running `gpt-4o` with temperature set to 0.

Table 5: **Additional performance comparisons on ML benchmark.** "WR", "BCW", "ICR", "SCTP", and "SVPC" represent "Wine recognition"", "Breast cancer wisconsin", "ICR - Identifying age-related conditions", "Santander customer transaction prediction", and "Santander value prediction challenge", respectively. "Avg." denotes "Average".

Model / Task	WR	BCW	Titanic	House Prices	SCTP	ICR	SVPC	Avg.
Completion rate								
AutoGen	0.92	**1.00**	0.92	0.83	0.83	0.83	0.83	0.88
OpenInterpreter	**1.00**	0.90	0.92	0.88	0.85	0.91	0.88	0.90
TaskWeaver	**1.00**	**1.00**	0.83	0.88	0.67	0.83	0.80	0.86
XAgent	**1.00**	**1.00**	0.83	0.83	0	0.67	0	0.62
OpenDevin	**1.00**	**1.00**	0.92	1.00	1.00	0.83	1.00	0.96
Data Interpreter	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Normalized performance score								
AutoGen	**1.00**	0.97	0.82	0.88	0.82	0.71	0.63	0.83
OpenInterpreter	**1.00**	0.96	0.81	0.87	0.52	0.25	0	0.63
TaskWeaver	**1.00**	0.96	0.43	0.49	0	0.65	0.17	0.53
XAgent	**1.00**	0.94	0	0	0	0	0	0.28
OpenDevin	**0.96**	0.96	0.81	0.87	0.86	0.62	0.45	0.79
Data Interpreter	0.96	**0.99**	**0.82**	**0.91**	**0.89**	**0.91**	**0.77**	**0.89**

B.3 Additional Results

B.3.1 Additional results of ML-benchmark and Math dataset

For a deeper understanding, Table 5 presents the results on the ML benchmark for both Completion Rate and Normalized Performance Score metrics. Additionally, Table 11 showcases the results of ablation experiments on the ML benchmark, focusing on the completion rate (CR) and the normalized performance score (NPS).

B.4 Overhead Analysis

We compared our token cost (average per task) and inference time (average per task) across the ML-Benchmark, Open-ended Task Benchmark, MATH Dataset, and InfriAgent-DABench, while also reporting our performance. Our framework demonstrates state-of-the-art performance with competitive efficiency.

Table 6: **Additional performance comparisons on MATH dataset.** "Avg." and "Std." denotes "Average", "Standard Deviation" respectively.

Category	MathChat	AutoGen	Data Interpreter				
			Avg.	Trial1	Trail2	Trail3	Std.(%)
C.Prob	0.52	0.59	0.68	0.70	0.66	0.68	2.05
N.Theory	0.60	0.66	0.82	0.81	0.82	0.82	0.99
Prealg	0.60	0.63	0.74	0.73	0.75	0.75	1.20
Precalc	0.19	0.12	0.29	0.28	0.30	0.29	1.13

Table 7: **Overhead analysis on MATH Dataset.**"Cost" represents the total cost in USD, "Time" indicates the total execution time in seconds, "Avg." denotes "Average".

Model / Metric	Cost ($)↓	Time (s)↓	Accuracy↑
AutoGen	**0.242**	**120.99**	0.500
Data Interpreter	0.336	211.57	**0.633**

Table 8: **Overhead analysis on InfriAgent-DABench.**"Cost" represents the total cost in USD, "Time" indicates the total execution time in seconds, "Avg." denotes "Average".

Model / Metric	Cost ($)↓	Time (s)↓	Accuracy↑
AutoGen (gpt-4o)	0.112	**42.42**	88.72
AutoGen (gpt-4-0613)	0.423	45.69	71.49
Data Interpreter (gpt-4o)	**0.017**	49.44	**94.93**
Data Interpreter (gpt-4-0613)	0.311	51.09	73.55

In specific domains such as MATH Dataset (See Table 7) and InfriAgent-DABench (See Table 8), Data Interpreter consistently shows superior accuracy (63.3% and 94.93% respectively) while maintaining competitive efficiency, as demonstrated in Table 7 and Table 8. Notably, on InfriAgent-DABench, our approach achieves better performance with lower cost (0.017 USD vs. 0.112 USD) compared to AutoGen.

On ML-Benchmark (See Table 9), Data Interpreter achieves the highest comprehensive score (0.95) among all frameworks, though with moderate cost (0.84 USD) and inference time (237.31s), as shown in table 9. While frameworks like OpenInterpreter achieve lower costs (0.21 USD) through one-time code generation, they show inferior performance (0.77).

In Table 10, for Open-ended tasks, Data Interpreter significantly outperforms baselines with a compre-

Table 9: **Overhead analysis on ML Benchmark.** “SCTP", and “SVPC" represent “ICR - Identifying age-related conditions", “Santander customer transaction prediction", and “Santander value prediction challenge", respectively. “Cost" represents the total cost in USD, “Time" indicates the total execution time in seconds, “Avg." denotes “Average".

Model / Task	Titanic	House	ICR	SCTP	SVPC	Avg.
Cost ($)↓						
AutoGen	0.08	0.25	0.19	0.48	0.58	**0.32**
OpenInterpreter	**0.26**	**0.15**	0.27	**0.18**	**0.21**	**0.21**
OpenDevin	2.66	3.01	3.35	3.24	2.78	3.01
TaskWeaver	0.35	0.38	0.36	0.29	0.48	0.37
XAgent	21.15	17.16	27.81	14.12	20.23	20.09
Data Interpreter	0.65	0.84	**0.76**	0.54	1.41	0.84
Time (s)↓						
AutoGen	**124.71**	**84.11**	**136.91**	280.60	**244.04**	**174.07**
OpenInterpreter	116.66	132.00	170.00	239.00	296.00	190.73
OpenDevin	164.00	133.00	148.00	282.00	212.00	187.80
TaskWeaver	109.76	279.25	151.97	**182.13**	119.62	168.55
XAgent	5400.00	5107.00	5400.00	6023.00	9000.00	6186.00
Data Interpreter	168.01	193.21	184.77	244.39	396.17	237.31
Comprehensive Score↑						
AutoGen	0.87	0.86	0.83	0.77	0.73	0.86
OpenInterpreter	0.86	0.87	0.68	0.58	0.44	0.77
OpenDevin	0.87	**0.94**	**0.93**	0.73	0.73	0.88
TaskWeaver	0.63	0.68	0.34	0.74	0.48	0.69
XAgent	0.42	0.42	0.00	0.34	0.01	0.45
Data Interpreter	**0.91**	0.96	0.94	**0.96**	**0.89**	**0.95**

Table 10: **Overhead comparison on Open-ended Tasks.** “OCR”, “WSC", “WPI”, and “IBR” represent “Optical Character Recognition”, “Web Search and Crawling”, “Web Page Imitation”, and “Image Background Removal”, respectively. “Cost” represents the total cost in USD, “Time” indicates the total execution time in seconds, “Avg.” denotes “Average”.

Model / Task	OCR	WSC	WPI	IBR	Avg.
Cost ($)↓					
AutoGen	0.10	0.18	0.43	0.48	0.30
OpenInterpreter	**0.28**	**0.08**	**0.15**	**0.07**	**0.15**
OpenDevin	1.27	1.88	1.26	1.24	1.41
Data Interpreter	0.275	0.69	0.23	0.18	0.34
Time (s)↓					
AutoGen	**68.85**	57.28	154.46	79.26	90.05
OpenInterpreter	133.00	109.00	102.00	68.00	**103.00**
OpenDevin	190.00	196.00	94.00	146.00	156.50
Data Interpreter	77.00	**293.00**	65.00	34.00	117.25
Comprehensive Score↑					
AutoGen	0.67	0.65	0.26	**1.00**	0.65
OpenInterpreter	0.50	0.30	0.36	**1.00**	0.54
OpenDevin	0.60	0.87	0.16	**1.00**	0.66
Data Interpreter	**0.85**	**0.96**	**1.00**	**1.00**	**0.95**

hensive score of 0.953, maintaining reasonable cost (0.34 USD) compared to OpenDevin (1.41 USD) and AutoGen (0.30 USD).

B.4.1 Ablation Study

Here we provide detailed ablation study results on core modules.

Table 11: **Ablation on core modules.** Evaluated with CR, NPS and CS on ML-Benchmark. “IGR" stands for Iterative Graph Refinement, and “PNG" denotes Programmable Node Generation. “ICR", “SCTP", and “SVPC" represent “ICR - Identifying age-related conditions", “Santander customer transaction prediction", and “Santander value prediction challenge", respectively.

Code execution	IGR	PNG	House Prices	SCTP	SVPC	ICR	Avg.
Completion rate							
✓			0.58	0.33	0.67	0.33	0.48
✓	✓		1.00	1.00	0.92	0.88	0.95
✓	✓	✓	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Normalized performance score							
✓			0.43	0	0.64	0	0.27
✓	✓		0.91	0.82	0.68	0.60	0.75
✓	✓	✓	**0.91**	**0.89**	**0.77**	**0.91**	**0.87**
Comprehensive score							
✓			0.51	0.17	0.66	0.17	0.37
✓	✓		0.96	0.91	0.80	0.74	0.85
✓	✓	✓	**0.96**	**0.95**	**0.89**	**0.96**	**0.94**

C Additional Examples

C.1 An Example of Task Graph

Here is the prompt used to generate the task graph.

Here is an example of a task graph. The user requirement is: “This is a dataset featuring sensor readings from industrial machines, aimed at predicting machine operational status (normal or faulty). Visualize the analysis and prediction results with high-quality graphs. Train data path: {train_path}, eval data path: {eval_path}.”

C.2 Prompts for Action Graph

Data Interpreter utilizes LLMs to generate an action graph for each task. For each task node, we maintain the execution context and task graph state via plan status, and generate executable code using the following prompt:

C.3 Example of Dynamic Task Graph Refinement

This section details how Data Interpreter resolves task failures and refines the task graph dynamically. Initially, the task graph is created as described in Appendix C.1. When encountering task execution failures (e.g., Task 4: feature engineering), Data Interpreter utilizes a reflection-based debugging prompt (REFLECTION_PROMPT) to iteratively analyze errors and propose improved implementations.

```
PLAN_PROMPT = """
# Context:
{context}
# Available Task Types:
{task_type_desc}
# Task:
Based on the context, write a plan or modify an existing plan of what you should do
    to achieve the goal. A plan consists of one to {max_tasks} tasks.
If you are modifying an existing plan, carefully follow the instruction, don't make
    unnecessary changes. Give the whole plan unless instructed to modify only one
    task of the plan.
If you encounter errors on the current task, revise and output the current single
    task only.
Output a list of jsons following the format:
[
    {{
        "task_id": str = "unique identifier for a task in plan, can be an ordinal",
        "dependent_task_ids": list[str] = "ids of tasks prerequisite to this task",
        "instruction": "what you should do in this task, one short phrase or
    sentence",
        "task_type": "type of this task, should be one of Available Task Types",
    }},
    ...
]
"""
```

Figure 7: Prompt for task graph generator

After repeated failures (e.g., three unsuccessful attempts to execute the action graph), Data Interpreter restructures the task graph: Tasks 1-3 remain unchanged, but Task 4 is simplified to basic feature creation, a new Task 5 for feature selection is introduced, and subsequent tasks (e.g., original Task 5 becoming Task 6) are automatically reindexed with updated dependencies, as shown below:

C.4 Runtime Results of Task Graph

We provide three distinct runtime results of our Data Interpreter, to offer an in-depth demonstration of its capabilities. These results meticulously showcase the intricacies of the task graph, action graph, and the overall graph typology, as shown in Figure 12.

C.5 Additional Results of Open-ended Tasks

We present the results by Data Interpreter of several Open-ended tasks in: tasks 4, 14, and 15 in Figure 13.

C.6 Result of Data Visualization

Figure 14 illustrates the results of the data analysis and visualization using Data Interpreter.

D Details of Datasets

D.1 Open-ended Task Details

Figures 15 and 16 showcase several typical Open-ended tasks in the following illustrations. For each task, we include the necessary data, user requirements, and the assessment pipeline.

D.2 ML-Benchmark Dataset Description

Here are the details of the ML-Benchmark dataset. We collect several typical datasets from Kaggle[1] and machine learning. Details are in Table 12

[1] https://www.kaggle.com/

```
[
    {
        "task_id": "1",
        "dependent_task_ids": [],
        "instruction": "Perform data loading and preliminary exploration of the
    train and eval datasets. Fill missing values and apply MinMax scaling.",
        "task_type": "eda"
    },
    {
        "task_id": "2",
        "dependent_task_ids": [
            "1"
        ],
        "instruction": "Conduct correlation analysis and provide descriptive
    statistics.",
        "task_type": "eda"
    },
    {
        "task_id": "3",
        "dependent_task_ids": [
            "1"
        ],
        "instruction": "Perform outlier detection using Isolation Forest to identify
     and handle anomalies.",
        "task_type": "eda"
    },
    {
        "task_id": "4",
        "dependent_task_ids": [
            "2",
            "3"
        ],
        "instruction": "Execute feature engineering, including General Selection,
    Target Mean Encoding, and Variance Based Selection to prepare features for model
     training.",
        "task_type": "feature_engineering"
    },
    {
        "task_id": "5",
        "dependent_task_ids": [
            "4"
        ],
        "instruction": "Split the data and train predictive models using Random
    Forest and XGBoost.",
        "task_type": "model_train"
    },
    {
        "task_id": "6",
        "dependent_task_ids": [
            "5"
        ],
        "instruction": "Evaluate the model's performance and generate an evaluation
    report.",
        "task_type": "model_evaluate"
    },
    {
        "task_id": "7",
        "dependent_task_ids": [
            "5",
            "6"
        ],
        "instruction": "Visualize the analysis and prediction results, including
    classification reports and confusion matrix, and serialize the model.",
        "task_type": "visualization"
    }
]
```


Figure 8: Task graph example

```
GRAPH_STATUS = """
## Finished Tasks
### code
```python
{code_written}
```

execution result
{task_results}

Current Task
{current_task}

Task Guidance
Write complete code for 'Current Task'. And avoid duplicating code from 'Finished
    Tasks', such as repeated import of packages, reading data, etc.
Specifically, {guidance}
"""

Action_Graph_Prompt = """
# User Requirement
{project_requirement}

Plan Status
{plan_status}

Tool Info
{tool_info}

Constraints
- Take on Current Task if it is in Plan Status, otherwise, tackle User Requirement
    directly.
- Ensure the output new code is executable in the same Jupyter notebook as the
    previous executed code.
- Always prioritize using pre-defined tools for the same functionality.

Output
While some concise thoughts are helpful, code is absolutely required. Always output
    one and only one code block in your response. Output code in the following
    format:
```python
your code
```
"""
```

Figure 9: Prompt for action graph generator

```
REFLECTION_PROMPT = """
[example]
Here is an example of debugging with reflection.
{debug_example}
[/example]

[context]
{context}

[previous impl]:
{previous_impl}

[instruction]
Analyze your previous code and error in [context] step by step, provide me with
    improved method and code. Remember to follow [context] requirement. Don't forget
     to write code for steps behind the error step.
Output a json following the format:
```json
{{
    "reflection": str = "Reflection on previous implementation",
    "improved_impl": str = "Refined code after reflection.",
}}
```
"""
```

Figure 10: Prompt for reflection and debugging

```
...
{
    "task_id": "4",
    "dependent_task_ids": [
            "2",
            "3"
        ],
    "instruction": "Create engineered features from sensor readings",
    "task_type": "feature_engineering"
},
{
    "task_id": "5",
    "dependent_task_ids": [
            "4",
        ],
    "instruction": "Perform feature selection using statistical methods and
    importance analysis",
    "task_type": "feature_engineering"
},
 {
        "task_id": "6",
        "dependent_task_ids": [
            "4",
            "5"
        ],
        "instruction": "Train a predictive model to determine machine status",
        "task_type": "model_train"
    },
  ...
```

Figure 11: Example of refined task graph

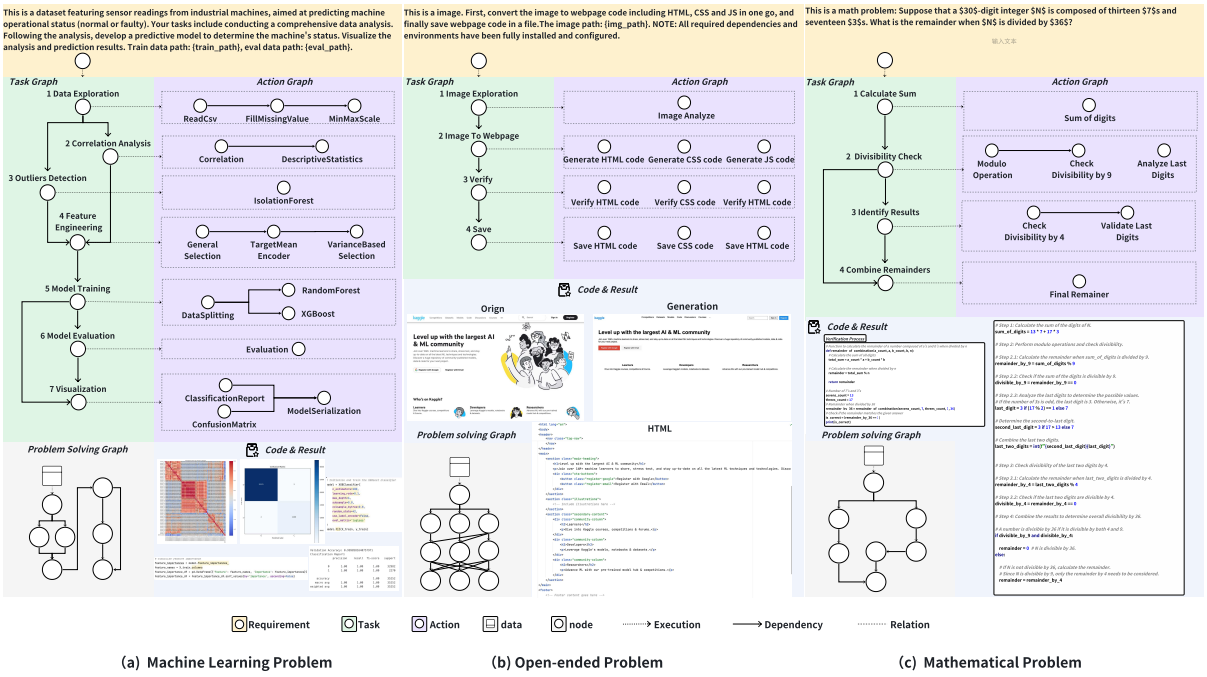


Figure 12: **Runtime examples of Data Interpreter**: machine learning, webpage imitation, and math problem solving

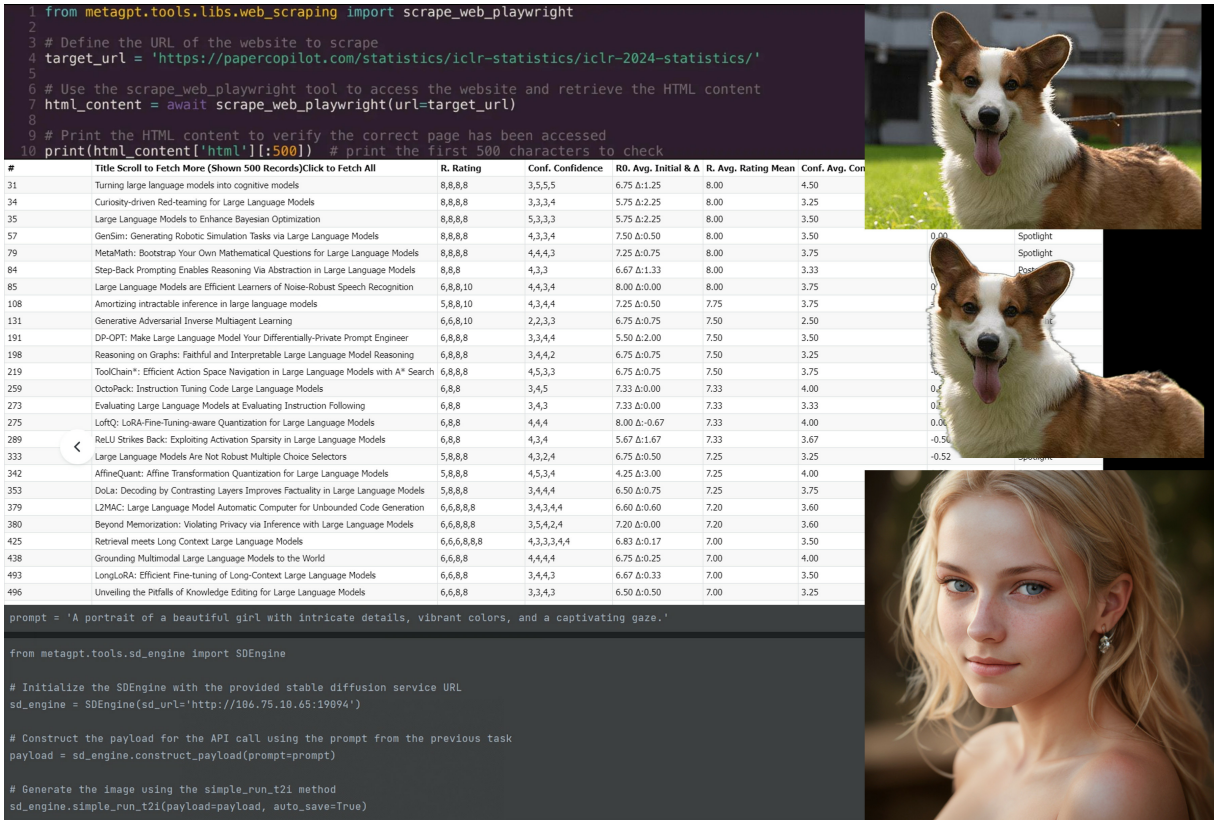


Figure 13: Image background removal / text-to-image / web search and crawling by Data Interpreter

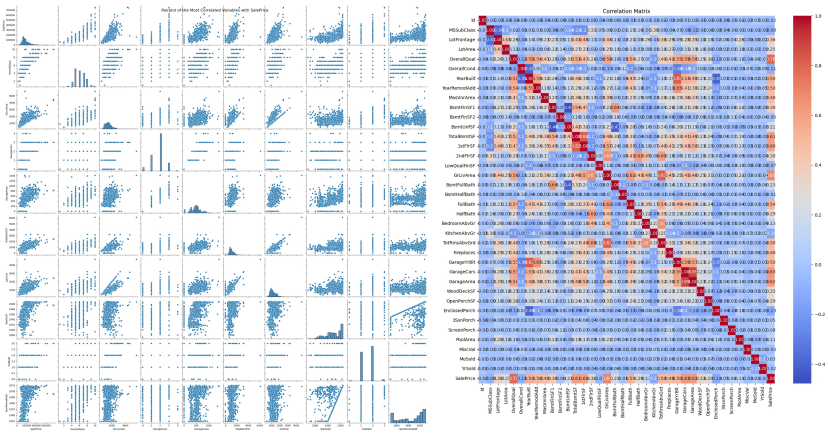


Figure 14: Data analysis and visualization capabilities of Data Interpreter

(1) OCR (Task 1-3)

Scenario Description: Scan all the necessary fields and amounts from the given file and then create an Excel sheet with the extracted data

User Requirement: This is an English invoice image.
Your goal is to perform OCR on the image, extract the total amount from ocr result and save as table, using PaddleOCR.
The PaddleOCR environment has been fully installed, try to use Paddleocr as much as possible.
Image path: ./workspace/CORD_test/image/receipt_00001.png

Pipeline Requirement:
1.Load and read images from a given folder/path
2.Install OCR tools/software
3.Using OCR tools/software to extract necessary fields and amounts
4.Collect results and convert them to a DataFrame
5.Save the result in a csv/xlsx forma

Performance Requirement: Recall / Precision / Accuracy

Data:
- Task 1:

- Task 2:

- Task 3:

(2) Web search and crawling (Task 4-7)

Scenario Description: Crawling and organizing web form information

Data: -

Pipeline Requirement:
1.Open target URL
2.Select and filter the required information
3.Download or transform the data, convert them into a specified format
4.Output in a tabular form

Performance Requirement: Recall / Precision / Accuracy

User Requirement:
- Task 4:
Get data from `paperlist` table in https://papercopilot.com/statistics/iclr-statistics/iclr-2024-statistics/, and save it to a csv file. paper title must include `multiagent` or `large language model`.
notice: print key variables

Figure 15: **Open-ended task cases (OCR and web search and crawling)**. We present task 4, omitting similar tasks for brevity.

(5) Image Background Removal (Task 14)

Scenario Description: Remove the background of a given image

User Requirement: This is an image, you need to use python toolkit rembg remove the background of the image. image path:'./data/lxt.jpg'; save path:'./data/lxt_result.jpg'

Data:

Pipeline Requirement:
1. Read a local image
2. Install image background removal tools/software
3. Using background removal tools/software to remove the background of the target image
4. Save the new image

Performance Requirement: Correctness

(6) Text2Img (Task 15)

Scenario Description: Use SD tools to generate images

User Requirement: I want to generate an image of a beautiful girl using the stable diffusion text2image tool, sd_url=""

Data: -

Pipeline Requirement: -

Performance Requirement: -

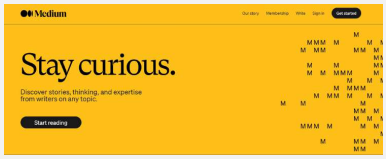
(7) Image2Code (Task 16-17)

Scenario Description: Web code generation

User Requirement:
- Task 16:
This is a image. First, check if the path exists, then convert the image to webpage code including HTML, CSS and JS in one go, and finally save webpage code in a file.The image path: ./medium.png .NOTE: All required dependencies and environments have been fully installed and configured.

- Task 17:
This is a image. First, check if the path exists, then convert the image to webpage code including HTML, CSS and JS in one go, and finally save webpage code in a file.The image path: ./gemini.png .NOTE: All required dependencies and environments have been fully installed and configured.

Data: (Task 16-17 in order)



Pipeline Requirement: -

Performance Requirement: -

Figure 16: **Open-ended task cases (image background removal, text-to-image, and image-to-code)**

Table 12: Details of the ML-Benchmark dataset, including dataset name, description, standard user requirements, dataset type, task type, difficulty, and metric used.

ID	Dataset Name	User Req.	Dataset Type	Dataset Description	Task Type	Difficulty	Metric
01	Iris	Run data analysis on sklearn Iris dataset, including a plot	Toy	Suitable for EDA, simple classification and regression	EDA	1	
02	Wine recognition	Run data analysis on sklearn Wine recognition dataset, include a plot, and train a model to predict wine class with 20% as test set, and show prediction accuracy	Toy	Suitable for EDA, simple classification and regression	Classification	1	ACC
03	Breast Cancer	Run data analysis on sklearn Wisconsin Breast Cancer dataset, include a plot, train a model to predict targets (20% as validation), and show validation accuracy	Toy	Suitable for EDA, binary classification to predict benign or malignant	Classification	1	ACC
04	Titanic	This is a Titanic passenger survival dataset, and your goal is to predict passenger survival outcomes. The target column is Survived. Perform data analysis, data preprocessing, feature engineering, and modeling to predict the target. Report accuracy on the eval data. Train data path: 'dataset\titanic\split_train.csv', eval data path: 'dataset\titanic\split_eval.csv'.	Beginner	Binary classification of survival, single table	Classification	2	ACC
05	House Prices	This is a house price dataset, and your goal is to predict the sale price of a property based on its features. The target column is SalePrice. Perform data analysis, data pre-processing, feature engineering, and modeling to predict the target. Report RMSE between the logarithm of the predicted value and the logarithm of the observed sales price on the eval data. Train data path: 'dataset\house-prices-advanced-regression-techniques\split_train.csv', eval data path: 'dataset\house-prices-advanced-regression-techniques\split_eval.csv'.	Beginner	Predicting house prices through property attributes, regression, single table	Regression	2	RMSLE
06	Santander Customer	This is a customer's financial dataset. Your goal is to predict which customers will make a specific transaction in the future. The target column is the target. Perform data analysis, data preprocessing, feature engineering, and modeling to predict the target. Report AUC on the eval data. Train data path: 'dataset\santander-customer-transaction-prediction\split_train.csv', eval data path: 'dataset\santander-customer-transaction-prediction\split_eval.csv' .	Industry	Binary classification to predict customer transactions, single table	Classification	2	AUC
07	ICR - Identifying	This is a medical dataset with over fifty anonymized health characteristics linked to three age-related conditions. Your goal is to predict whether a subject has or has not been diagnosed with one of these conditions. The target column is Class. Perform data analysis, data preprocessing, feature engineering, and modeling to predict the target. Report F1 Score on the eval data. Train data path: 'dataset\icr-identify-age-related-conditions\split_train.csv', eval data path: 'dataset\icr-identify-age-related-conditions\split_eval.csv' .	Industry	Binary classification of health symptoms, single table	Classification	2	F1
08	Santander Value	This is a customer's financial dataset. Your goal is to predict the value of transactions for each potential customer. The target column is the target. Perform data analysis, data preprocessing, feature engineering, and modeling to predict the target. Report RMSLE on the eval data. Train data path: 'dataset\santander-value-prediction-challenge\split_train.csv', eval data path: 'dataset\santander-value-prediction-challenge\split_eval.csv' .	Industry	Predicting transaction values, regression, single table, 5k columns, suitable for complex algorithms	Regression	3	RMSLE