DATA INTERPRETER: AN LLM AGENT FOR DATA SCIENCE

Sirui Hong[1*], **Yizhang Lin**[1*], **Bangbang Liu**[1], **Binhao Wu**[1], **Danyang Li**[1], **Jiaqi Chen**[2], **Jiayi Zhang**[3], **Jinlin Wang**[1], **Lingyao Zhang**, **Mingchen Zhuge**[4], **Taicheng Guo**[5], **Tuo Zhou**[6], **Wei Tao**[2], **Wenyi Wang**[4], **Xiangru Tang**[7], **Xiangtao Lu**[1], **Xinbing Liang**[1,8], **Yaying Fei**[9], **Yuheng Cheng**[10], **Zongze Xu**[1,11], **Chenglin Wu**[1†], **Li Zhang**[2], **Min Yang**[12], **Xiawu Zheng**[13]

[1]DeepWisdom, [2]Fudan University [3]Renmin University of China
[4]AI Initiative, King Abdullah University of Science and Technology
[5]University of Notre Dame [6]The University of Hong Kong [7]Yale University
[8]East China Normal University [9]Beijing University of Technology
[10]The Chinese University of Hong Kong, Shenzhen [11]Hohai University
[12]Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences [13]Xiamen University

ABSTRACT

Large Language Model (LLM)-based agents have demonstrated remarkable effectiveness. However, their performance can be compromised in data science scenarios that require real-time data adjustment, expertise in optimization due to complex dependencies among various tasks, and the ability to identify logical errors for precise reasoning. In this study, we introduce the Data Interpreter, a solution designed to solve with code that emphasizes three pivotal techniques to augment problem-solving in data science: 1) dynamic planning with hierarchical graph structures for real-time data adaptability; 2) tool integration dynamically to enhance code proficiency during execution, enriching the requisite expertise; 3) logical inconsistency identification in feedback, and efficiency enhancement through experience recording. We evaluate the Data Interpreter on various data science and real-world tasks. Compared to open-source baselines, it demonstrated superior performance, exhibiting significant improvements in machine learning tasks, increasing from 0.86 to 0.95. Additionally, it showed a 26% increase in the MATH dataset and a remarkable 112% improvement in open-ended tasks. The solution will be released at https://github.com/geekan/MetaGPT.

1 INTRODUCTION

Large Language Models (LLMs) have enabled agents to excel in a wide range of applications, demonstrating their adaptability and effectiveness (Guo et al., 2024; Wu et al., 2023a; Zhou et al., 2023b). These LLM-powered agents have significantly influenced areas like software engineering (Hong et al., 2023), navigating complex open-world scenarios (Wang et al., 2023; Chen et al., 2024a), facilitating collaborative multi-agent structures for multimodal tasks (Zhuge et al., 2023), improving the responsiveness of virtual assistants (Lu et al.), optimizing group intelligence (Zhuge et al., 2024), and contributing to scientific research (Tang et al., 2024).

Recent studies focused on improving the problem-solving capabilities of these agents by improving their reasoning process, aiming for increased sophistication and efficiency (Zhang et al., 2023; Besta et al., 2023; Sel et al., 2023; Yao et al., 2024; Wei et al., 2022). However, data-centric scientific problems, including machine learning, data analysis, and mathematical problem-solving, present unique challenges that remain to be addressed. The machine learning process involves complex, lengthy task handling steps, characterized by intricate dependencies among multiple tasks. This requires expert intervention for process optimization and dynamic adjustment in the event of failure or data updates. It is often challenging for LLMs to provide the correct solution in a single attempt.

[*]These authors contributed equally to this work.
[†]Chenglin Wu (E-mail: alexanderwu@deepwisdom.ai), is the corresponding author.

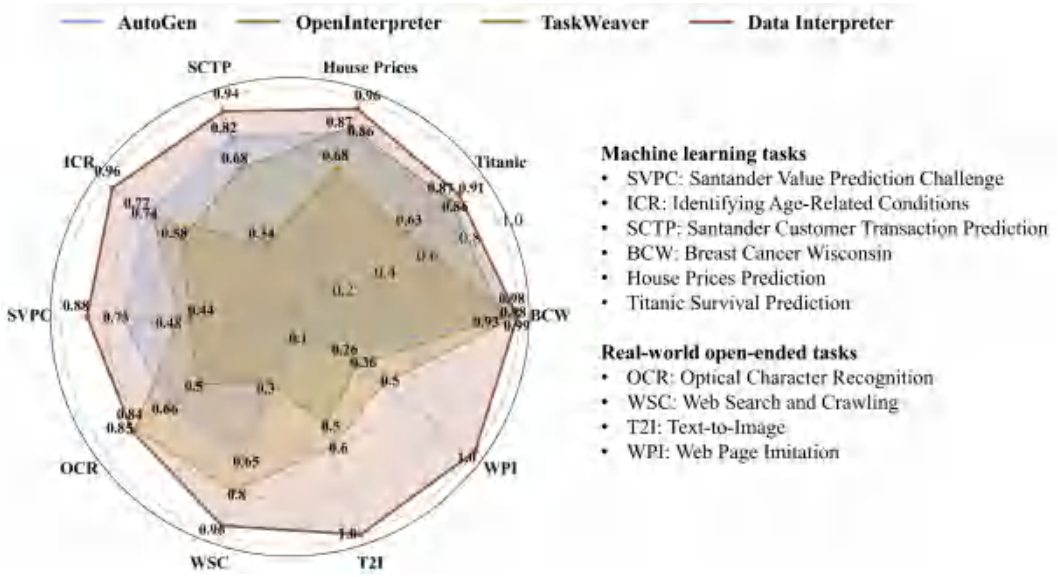


Figure 1: Comparison with various open-source frameworks on machine learning tasks and real-world open-ended tasks.

Furthermore, these problems demand precise reasoning, and thorough data verification (Romera-Paredes et al., 2023), which poses additional challenges to the LLM-based agent framework.

Moreover, existing works such as (Qiao et al., 2023; OpenAI, 2023; Lucas, 2023) address data-centric problems through code-based problem-solving methods, known as the interpreter paradigm, which combines static requirement decomposition with code execution. However, several key challenges arise when employing these frameworks in practical data science tasks: ***1) Data dependence intensity:*** The complexity inherent in data science arises from the intricate interplay among various steps, which are subject to real-time changes (Liu et al., 2021). For accurate results, data cleaning and comprehensive feature engineering are prerequisites before developing any machine learning model. Therefore, it is critical to monitor data changes and dynamically adjust to the transformed data and variables. The machine learning modeling process, encompassing feature selection, model training, and evaluation, involves a broad spectrum of processing operators and search spaces (Zheng et al., 2021). The challenge lies in generating and resolving the entire process code simultaneously. ***2) Refined domain knowledge***: The specialized knowledge and coding practices of data scientists are pivotal in addressing data-related challenges. Typically embedded in proprietary code and data, this knowledge often remains inaccessible to current LLMs. For instance, generating code for data transformation in specific domains such as energy or geology may pose a challenge for LLMs without the requisite domain expertise. Existing methodologies predominantly depend on LLMs, a reliance that may streamline the process but potentially compromise performance. ***3) Rigorous logic requirements***: Currently, interpreters such as (Qiao et al., 2023; OpenAI, 2023; Lucas, 2023) incorporate code execution and error capturing capabilities to enhance problem-solving performance. However, they often neglect error-free execution, erroneously considering it as correct. While basic programming tasks can be streamlined and depend on immediate execution feedback when requirements are clearly delineated, data science problems often pose ambiguous, irregular, and not well-defined requirements, making it difficult for LLMs to understand. Consequently, LLM-generated code solutions for task resolution may contain ambiguities that necessitate rigorous validation of logical soundness, extending beyond mere execution feedback.

To address the aforementioned challenges, we propose ***1) Dynamic planning with hierarchical structure***: Our framework employs hierarchical graph structures to comprehend the inherent complexities of data science more effectively. A dynamic planning approach equips it with the adaptability to task variations, proving especially efficient in monitoring data changes and managing intricate variable dependencies inherent in data science problems. ***2) Tool utilization and generation***: We enhance coding proficiency by integrating various human-authored code snippets, creating custom tools for specific tasks beyond mere API-focused capabilities. This process involves the automatic combination of diverse tools with self-generated code. It utilizes task-level execution to

independently build and expand its tool library, simplify tool usage, and perform code restructuring as needed. ***3) Enhancing reasoning with logic bug aware***: This is based on the confidence score obtained from execution results and test-driven validations. It detects inconsistencies between the code solution and test code execution and compares multiple trials to reduce logic errors. Throughout the execution and reasoning process, task-level experiences, primarily comprising metadata and runtime trajectory, which include both successes and failures, are recorded.

As depicted in Figure 1, Data Interpreter significantly surpasses existing open-source frameworks. In comparison to these baselines, Data Interpreter exhibits superior performance, with 10.3% (from 0.86 to 0.95) improvement in machine learning tasks and 26% enhancement on the MATH dataset, demonstrating robust problem-solving capabilities. In open-ended tasks, its performance has more than doubled, marking a 112% increase, showcasing its efficacy in tackling a wide spectrum of challenges.

We summarize our contributions as follows:

- We propose a dynamic planning framework with hierarchical structures, enhancing adaptability and problem-solving capabilities in data science tasks.
- We improve the proficiency and efficiency of coding in LLMs by introducing automated tool integration for tool utilization and generation.
- We improve reasoning by integrating verification and experience, thereby enhancing the accuracy and efficiency of problem-solving.
- Our experiments demonstrate that our framework surpasses existing benchmarks across machine learning tasks, mathematical problems, and open-ended tasks, thereby setting a new standard for performance.

2 Related Work

LLMs as data scientist agents Cutting-edge Large Language Models (LLMs), pre-trained on diverse natural and programming data, exhibit strong interpretation abilities. Like, (Gao et al., 2023) (Chen et al., 2022) leverages program interpreters to decouple complex computation, (Zhou et al., 2023a) boost their performance on the MATH dataset, and (Hendrycks et al., 2021), (Li et al., 2023), (Liang et al., 2023) enable code-based reasoning capabilities in embodied agents. Building on code interpretation capabilities, researchers are exploring ways to leverage LLMs to address data science challenges(Bordt et al., 2024; Chen et al., 2024b; Yang et al., 2024; Hassan et al., 2023; Sänger et al., 2023) and integrate LLMs with specialized machine learning pipelines. For instance, (Huang et al., 2023) develops or enhances Machine Learning models from data and task descriptions autonomously. In addition, (Romera-Paredes et al., 2023) pairs LLMs with systematic evaluation to discover solutions to open problems by evolving executable programs describing solution methods. However, there is a lack of datasets and evaluation methods designed to assess the abilities of LLM-based methods in this field. We benchmark our work and various open-source frameworks on machine learning problem-solving to provide more insight and understanding of this research area.

Planning Planning is the critical capability of LLM-based agents. Planning capability emphasizes the generation of logically structured actions or thoughts roadmap for specific problems (Huang et al., 2024; Chen et al., 2024a). For the planning capability of LLM-based agents, earlier work such as CoT (Wei et al., 2022) and ReAct (Yao et al., 2022) focus on the decomposition of complicated tasks and perform sequential planning for subtasks. Due to the complexity of tasks, one single plan generated by the LLM-based agent is sometimes infeasible. Hence, some kinds of work, such as ToT (Yao et al., 2024) and GoT (Besta et al., 2023), are designed to generate multiple plans and select one plan to execute. Although these previous planning approaches demonstrate impressive performance, they struggle to address multi-step problems with strong task dependencies, a common occurrence in data science tasks. Alternatively, we utilize dynamic hierarchical planning to enhance the capability, allowing for the decomposition of complex problems into task and action graphs, commonly encountered in data science scenarios.

Tools Recent research has focused on improving the capabilities of LLMs by creating and integrating external tools (Schick et al., 2024), (Paranjape et al., 2023). (Zhuge et al., 2023) (Shen

et al., 2024) proposes multiple agents to solve multimodal tasks. (Liu et al., 2023) proposed an automatic tool selection mechanism based on LLM decision-making rather than statically assigning specific tools for certain tasks. In the field of self-creation of tools, (Cai et al., 2023) transformed the role of LLM from a tool user to a creator, achieving self-sufficiency in tool creation. (Qian et al., 2023) presented a framework that combines the creation and use of tools to solve problems. In this paper, we have expanded the types and range of tools usage. We not only implemented the two types of tools proposed in their future work, named "Upgrade of Existing Tool" and "Combination of Multiple Tools", but also improved tool generation efficiency and practicality. We achieve this by leveraging execution experience instead of relying on Few-Shot Prompts. Furthermore, this study supports creating various private tool libraries and allows LLMs to independently select and combine multiple tools as needed.

Reasoning Reasoning capability emphasizes understanding and processing of information to make decisions (Huang et al., 2024), which is another key strength of LLM-based agents. For the reasoning capability, previous works such as Reflexion (Shinn et al., 2024), Self-Refine (Madaan et al., 2024), CRITIC (Gou et al., 2023) focus on encouraging LLM-based agents to reflect on failures and refine the reasoning process. Moreover, (Gao et al., 2023) is pioneering work in leveraging code to improve the accuracy of LLM to solve mathematical, symbolic, and algorithmic reasoning problems. (Chen et al., 2022) decouples complex computation from language understanding and reasoning by using a program interpreter to solve numeric reasoning tasks. (Wang et al., 2023) leverages an iterative prompting mechanism to enhance programs used as actions by agents in Minecraft based on feedback from the environment and self-verification. Unlike prior approaches that primarily focused on general language feedback or execution feedback, our work tackles the unique challenges posed by data science problems that require advanced logical reasoning. Specifically, we propose novel automated confidence-based verification mechanisms to improve reasoning capability.

3 METHODOLOGY

The methodology adopted in this paper, as depicted in Figure 2, employs a plan-code-review paradigm, integrating dynamic planning within a hierarchical structure to monitor and adjust goals in real time. Further details are elaborated in Section 3.1.

Our interpreter uses code to accomplish each task. Tasks are decomposed following the plan, with code generated through LLMs based on these tasks. Tools are incorporated as needed to augment proficiency. The executor carries out code execution, producing traceable runtime results. Detailed explanations are provided in Section 3.2. Moreover, each task is subjected to a validation process to ensure its reliability. The process of executing a task is then characterized and analyzed as an experience, which can be retrieved for similar tasks in the future. More information on this mechanism is provided in Section 3.3.

3.1 DYNAMIC PLANNING WITH HIERARCHICAL STRUCTURE

The intensive data dependence complicates modeling and orchestrating data science pipelines. In this section, we outline these pipelines naturally organized by the graph, modeling them with the hierarchical structure, and introduce dynamic plan management for effective orchestration.

3.1.1 HIERARCHICAL GRAPH FOR DATA SCIENCE PROBLEMS

Data science projects encompass extensive detailing and long-range pipelines, complicating the direct planning of all detailed tasks and coding. This complexity necessitates careful planning, execution, and management (Biswas et al., 2022). Drawing inspiration from the application of hierarchical planning in automated machine learning tasks (Mohr et al., 2018; Mubarak & Koeshidayatullah, 2023), we organize the data science pipelines via hierarchical structure, which initially decomposes the intricate data science problem into manageable tasks and further break down each task into specific actions executed through code.

Figure 3 illustrates the hierarchical data science task pipelines composed of preprocessing, feature construction, feature selection, hyperparameter tuning, model training, model evaluation, ensemble, and model prediction tasks (green circle in Figure 3). Alongside, an execution graph, referred to as

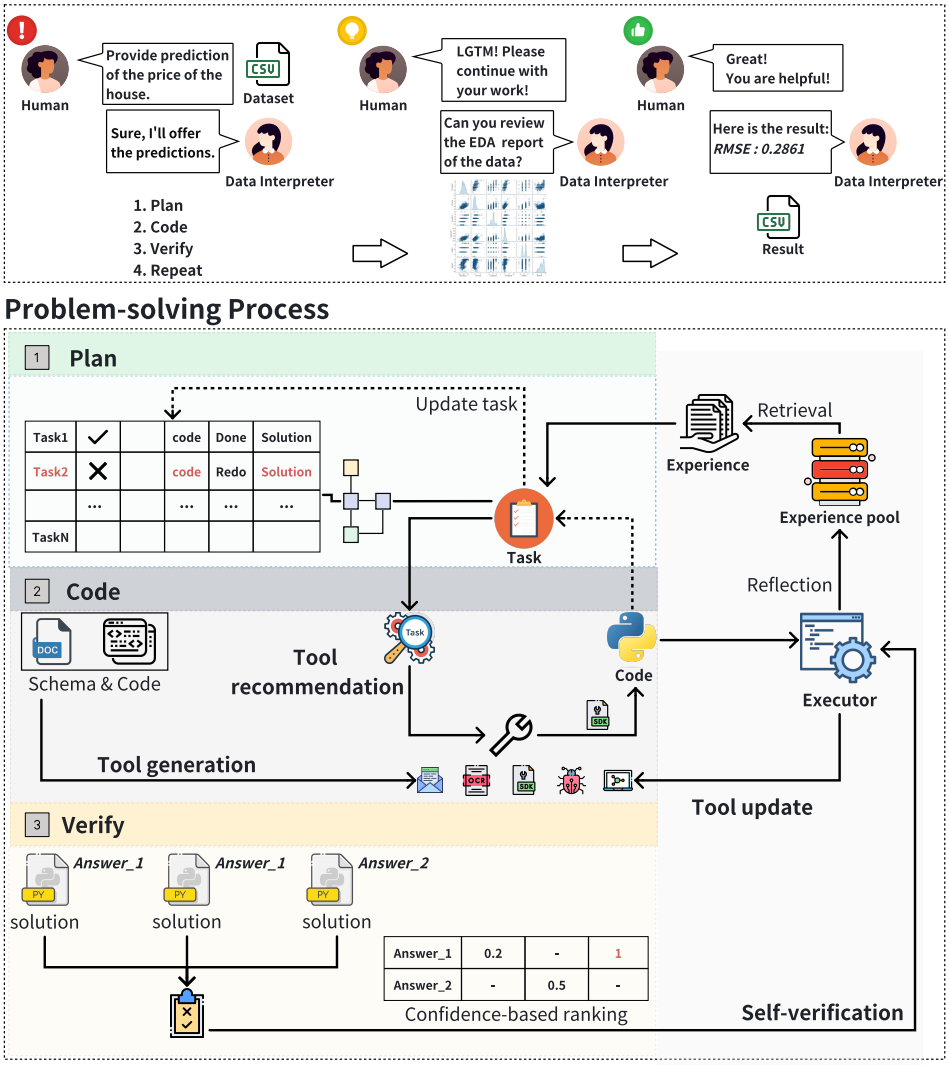


Figure 2: **The overall design of Data Interpreter.** This model consists of three stages: dynamic plan graph and management, wherein a plan is generated for data-centric tasks, and the state of each step is managed during execution; tool utilization and evolution, involving the selection or creation of suitable tools to solve problems, continually evolving these tools; and automated confidence-based verification, which examines and votes on logically sound solutions.

the action graph, represents the corresponding execution actions (purple circle in Figure 3). This hierarchical graph structure facilitates structured problem-solving for our interpreter and adeptly captures both sequential task relationships (such as from model evaluation to ensemble and model prediction) and parallel task relationships (such as model training and hyperparameter tuning). Therefore, we propose structuring data science workflows as a hierarchical directed acyclic graph (DAG), aptly representing data science pipelines at both task and coding levels. Our interpreter leverages the advanced planning capabilities of LLMs to decompose the complex data science problem into multiple tasks consistent with the problem goal and express their executing dependencies through a graph structure. We design the metadata for each task node, including task description, completion status as well as code. More details about the task node are described in Appendix A.1.1.

3.1.2 Dynamic Plan Management

Leveraging a hierarchical graph structure, tasks are executed automatically. Unlike previous methods (Wei et al., 2022; Besta et al., 2023; Yao et al., 2022) that create and execute plans once before

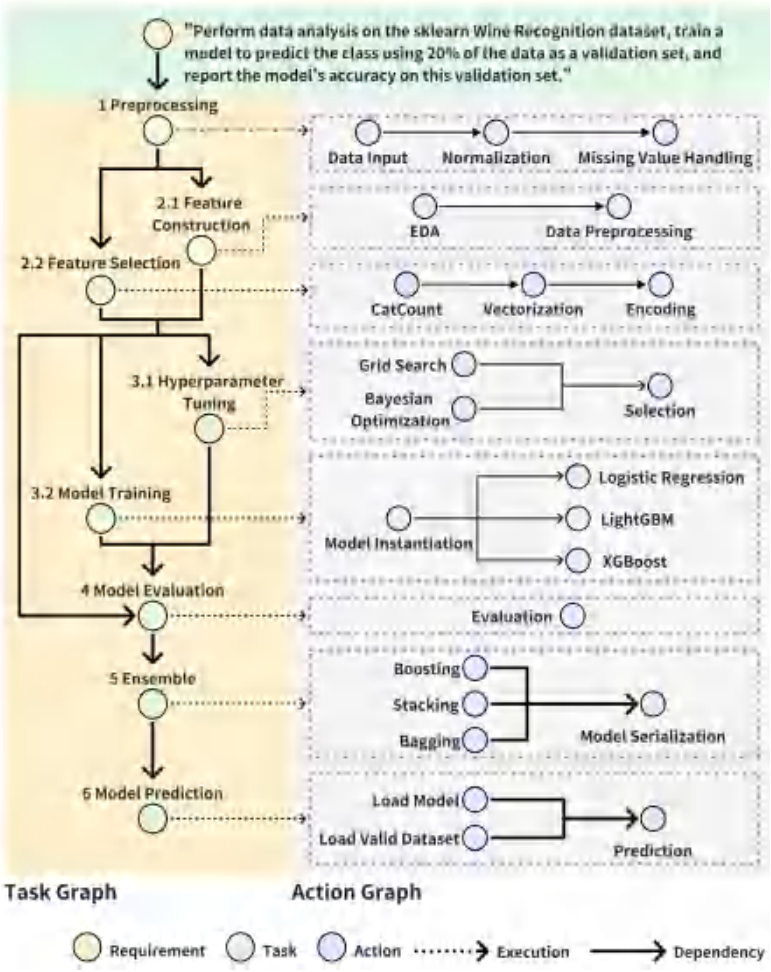


Figure 3: **Directed acyclic graph (DAG) of tasks**: Take wine recognition problem as an example. The task graph is an expression of the disassembled planned tasks. The action graph also referred to as the execution graph, executes the node based on the planned task graph. The execution code of each node is converted by LLM.

execution for static problems, we investigate that in intensive data dependence scenario, the intermediate data among tasks will be dynamically changed during execution due to tool operations or new information in the workflow, which can lead to runtime errors if the data does not match the pre-defined plan. To tackle this, we introduce a dynamic plan management, detailed in Figure 4.

To ensure efficient progress execution and facilitate plan modifications, our interpreter dynamically updates the corresponding code, execution result, and status of each node in the task graph following each task execution. A task is considered completed by successfully executing the corresponding code. Once completed, the task is marked as "Success" and added to a completed tasks list, proceeding to the next task according to the plan. On the contrary, the task is marked as "Failure" if it fails.

We have designed two strategies: *Self-debugging* and *Human editing*, aimed at enhancing the autonomous completeness and correctness of our interpreter. In the event of task failure, *Self-debugging* is enabled, utilizing LLMs to debug the code based on runtime errors, up to a predefined number of attempts. If the task remains unresolved, it is flagged as "Failure". Due to the high logic requirements of data science problems, an additional human-in-the-loop approach, human editing, is introduced to ensure code precision. When *Human editing* is activated, our interpreter holds the task until it is manually modified, upon which it is rerun based on human input.

For failed or manually edited tasks, our interpreter will regenerate the plan based on current episodic memory and the context of execution. Specifically, the regenerated task graph is sorted in topological

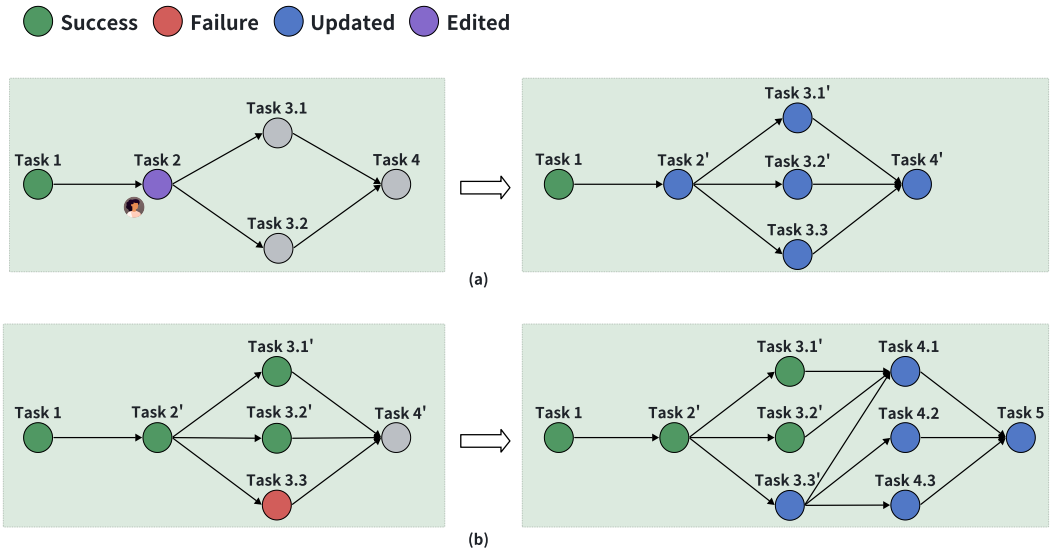


Figure 4: **Dynamic Plan management of Data Interpreter.** (a) Plan refinement using *Human editing*. The left image illustrates a human-edited task on the graph, and the refined plan with updated tasks 3.1', 3.2', along with newly added task 3.3 is delineated in the right image. (b) Plan refinement for the failed task. After task execution, Task 3.3 fails. The refined plan consists of existing success tasks and updated task 4.1 from task 4', as well as newly added tasks 4.2, and 4.3.

order and then compared to the original task graph using a prefix matching algorithm (Waldvogel, 2000) to identify any differences in instructions. Based on this comparison, the fork can be identified. The final output of this process includes all unchanged tasks existing before the fork and any new tasks added or modified after the fork.

Throughout execution, our interpreter monitors the dynamic task graph, promptly removing failed tasks, generating refined tasks, and updating the graph. This avoids the inefficiency of generating fine-grained planning tasks at once and improves the success rate of plans requiring multi-step execution, making it better suited for scenarios where the data flow constantly changes in data science problems.

3.2 Tool Utilization and Generation

To address the intricate nature of tasks that are too complex to be entirely coded from scratch, utilizing existing toolkits or integrating existing code snippets becomes essential. Take, for example, feature engineering, which demands domain-specific expertise for data transformation. In such cases, using tools crafted by experts can be significantly more effective, as generating this type of code directly through LLMs poses considerable challenges. Similarly, email processing involves orchestrating different code snippets to establish efficient workflows. To improve the efficiency of using these tools, we suggest a two-pronged method: one focuses on recommending or generating the most suitable tools, while the other organizes these tools effectively. This approach offers clear advantages over previous methods (Schick et al., 2024; Hong et al., 2023), which relied on mere library calls or did not incorporate tools using clear modularization in the code. By combining the strengths and mitigating the weaknesses of these methods, our approach presents a more balanced and efficient solution. Notice that tool usages follow the principles and procedures of the task graph described in Section 3.1.1; the use of tools itself would be considered one of the tasks in the graph.

3.2.1 Tool Recommendation and Organization

In tool recommendations, our interpreter classifies tools based on task descriptions and types. This process effectively narrows down the pool of potential tools, making the selection process more efficient for subsequent tasks. Our interpreter then identifies the top-k tools that best fit the tasks by evaluating the compatibilities of the candidate tools with one task. Additionally, we incorporate

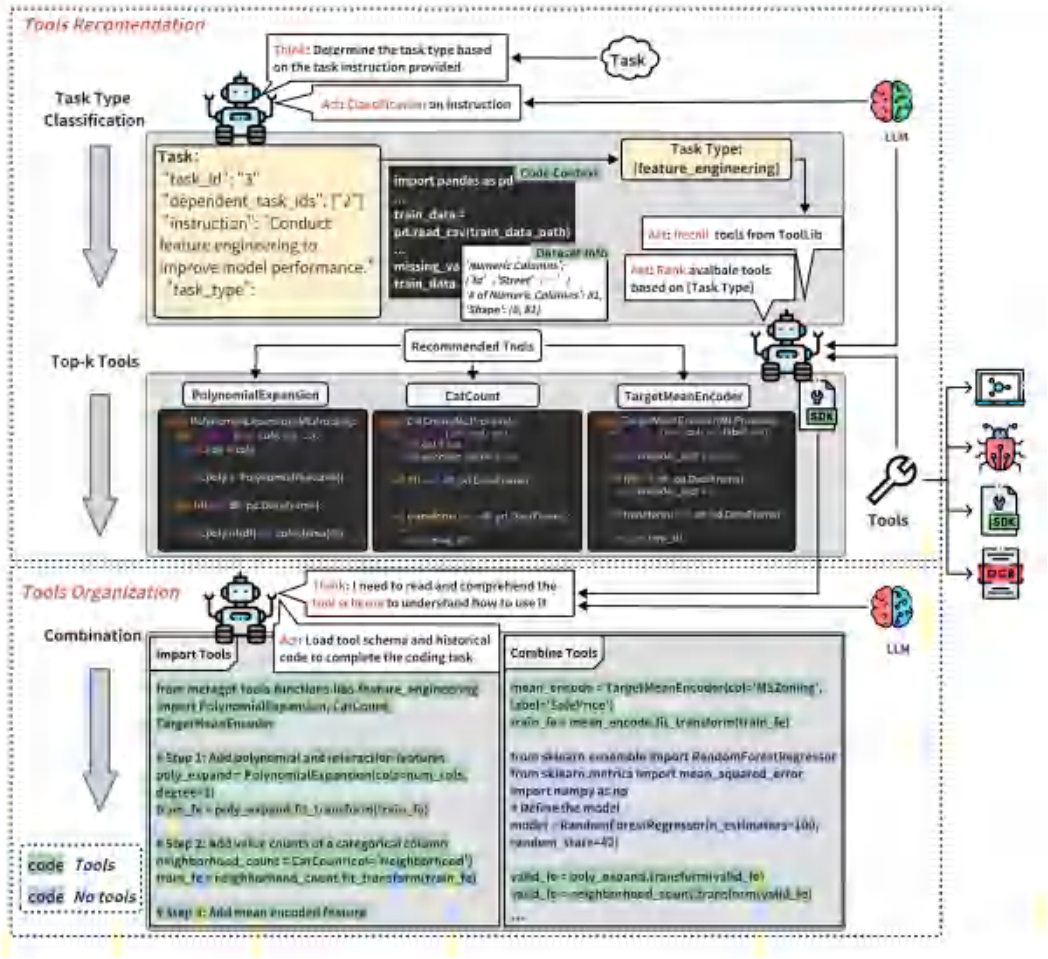


Figure 5: **Tool usage pipeline in Data Interpreter.** The tool recommendation initially selects tools based on task classification. Then the tool organization combines multiple tools as needed to accomplish tasks.

a tool schema to help LLMs understand these tools' functionalities and use cases, embedding this schema during execution phases as outlined in Appendix A.3. This schema-guided understanding enables more accurate tool selection and application. Besides, during execution, the algorithm dynamically adjusts tool parameters using LLMs, considering the code context and task requirements. This dynamic parameter adjustment improves tool adaptability to the tasks at hand.

In tool organizations, our method employs LLMs to seamlessly integrate tools into the code, optimally positioning them based on a thorough analysis of the tool functions. This is particularly useful for complex tasks such as feature engineering, facilitating a process that is efficient and adaptable to the integration of tools. We refine this process by considering the context of the current task and the tools at our disposal. The LLM is directed to craft code that not only invokes the required tool functions but also seamlessly integrates these calls with other aspects of the code. This allows for dynamic orchestration of various tools tailored to the specific requirements of the encoding process. A prime example of this adaptability is demonstrated with the CatCount tool in our deployment pipeline (Figure 5), showcasing the dynamic use of its fit and transform functions according to the task context. This strategy ensures that tool integration is automated and precisely aligned with task demands, significantly boosting coding efficiency and flexibility.

3.2.2 Continuous Tool Evolution

To minimize the frequency of debugging and improve execution efficiency, our model learns from experience during task execution. After each task, it abstracts tools by distilling their core function-

alities, stripping away any sample-specific logic. This creates versatile, generic tool functions that are added to the library for future use.

In addition, Data Interpreter automatically ensures the reliability of these tools by conducting rigorous unit tests and leveraging its self-debugging capabilities through LLMs. Consequently, Data Interpreter facilitates rapidly transforming sample-specific code snippets into reusable tool functions, continuously improving its toolkit and coding expertise over time.

3.3 Enhancing Reasoning with Verification and Experience

Our designed task graph, dynamic plan management, and tool utilization can improve task planning and tool mastery. However, relying only on error detection or capturing exceptions is inadequate feedback to complete a task. For complex reasoning problems, even if the code runs without errors, it can still contain logical flaws (Wang et al., 2023; Zhou et al., 2023a).Therefore, in this section, we introduce automated confidence-based verification and leverage experience further to improve the correctness and efficiency of the reasoning results.

3.3.1 Automated Confidence-based Verification

To address this issue, we propose a simple yet effective technique, **Automated Confidence-based Verification (ACV)** , which introduces an interpretation layer between the environment and the interpreter. This approach allows LLMs to evaluate code execution results and determine if the code solution is mathematically rigorous or logically correct. Specifically, once a code solution for the task starts to be executed, our interpreter is required to generate a validation code to ensure that the output result complies with the task requirement. The validation code is designed to simulate the logical process according to the task description and to verify the correctness of the result generated by the code.

This process is similar to performing white-box testing on each task, guaranteeing that the code produces the expected output.

Our interpreter returns a confidence score that indicates how likely the output will pass the verification. Formally, in the first verification, given the initial code $\boldsymbol{C_1}$, its execution result (i.e, candidate answer) $\boldsymbol{A_1}$ and the task description $\boldsymbol{T}$, validation code is generated by LLM, and this process is denoted as $\mathcal{V}$. The validation code $\boldsymbol{V_k}$ in k-th verification is generated as follows:

$$\boldsymbol{V_k} = \mathcal{V}(\boldsymbol{T}, \boldsymbol{C_k}, \boldsymbol{A_k}), \tag{1}$$

where k denotes the k-th verification and it starts from 1. After that, the validation code (i.e., $\boldsymbol{V_k}$) is executed and its result is denoted as $\boldsymbol{R_k}$. N represents the maximum verification process that we allow.

For each verification, the confidence score is calculated based on the validation result $\boldsymbol{R_k}$ as follows,

$$\text{Confidence} = \begin{cases} 1, & \text{if } \boldsymbol{R_k} = \text{True}, \\ 0.2, & \text{if } \boldsymbol{R_k} = \text{False}, \\ 0.5, & \text{otherwise.} \end{cases} \tag{2}$$

The confidence score helps the interpreter select a more accurate result as the final answer. It helps the interpreter choose a more accurate result as the final answer by ranking the average confidence scores corresponding to different execution results.

A specific example of this automated confidence-based verification process from the MATH dataset is shown in 6. The validation code takes into account both the task, the code, and its execution result. The function is_prime is to check the code, the $probability$ is generated from the task description, and $given\_answer$ is the candidate answer. In this example, the process undergoes five separate verifications. Specifically, the results of the code execution (that is, the candidate response) for the first and fifth validations are $1/108$. For the remaining verifications, the results consistently are $56/219$. The candidate answer (i.e. $1/108$) gets two confidence scores (0.2 and 1), with an average of 0.6. Another candidate answer (i.e. $56/219$) gets three confidence scores (0.2, 0.5, and 0.2), with an average of 0.3. As the former gets a higher average confidence score, our interpreter selects $1/108$ as the final answer, and it is correct. In contrast, simply using the majority voting strategy will choose the latter, which is wrong.

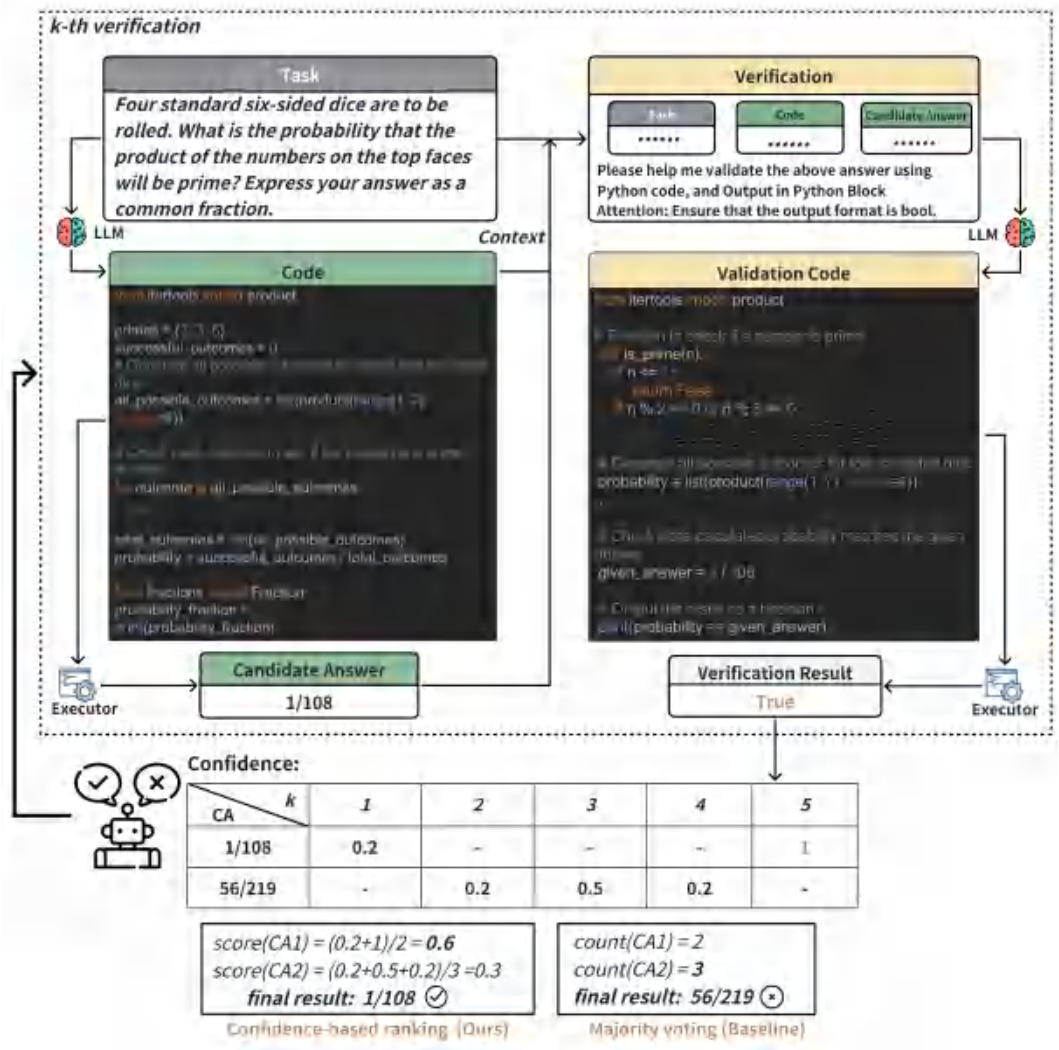


Figure 6: **Example for Automated Confidence-based Verification**. Inside the dotted box is the verification process, and below the dotted box is the final answer based on verification.

3.3.2 Experience-Driven Reasoning

As the automated confidence-based verification makes the task-solving process more transparent and reliable, the data generated in the verification can be reused as experience for other tasks. Therefore, we improve our interpreter's adaptability through a reflective analysis that allows tasks to be reviewed, updated, and confirmed. This process is called Experience-Driven Reasoning.

Specifically, our interpreter integrates an external repository designated as the 'experience pool' to archive essential elements of each task, including task description, final version code, and final answer. In the pool, all archived data is reorganized into reusable experiences based on the reflective mechanism (Zhao et al., 2023; Shin et al., 2023). These experiences, including both failed and successful attempts, can provide a comprehensive context for a task.

This pool functions as a valuable resource, enabling the retrieval of past experiences to inform and optimize new task executions. An experience can be reused if it is found to be one of the nearest neighbors of a new task from the vector store, which is generated through task-level reflective analysis. Specifically, for a certain task, top-k experiences are retrieved as the context of the current task, which can improve the accuracy and efficiency of its reasoning. This approach mirrors the fundamental principles of human cognition, where individuals take advantage of past experiences

to enhance decision-making and problem-solving. More experimental evaluations that validate this approach can be found in Section 4.3.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

4.1.1 DATASET

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.

ML-Benchmark Given the absence of datasets and evaluation metrics for assessing capabilities in the machine learning domain, we developed a benchmark dataset and corresponding evaluation method known as ML-Benchmark. This dataset encompassed eight representative machine learning tasks categorized into three difficulty levels, ranging from easy (level 1) to most complex (level 3). Each task was accompanied by data, a concise description, standard user requirements, suggested steps, and metrics (see Table 8 in the Appendix). For tasks labeled as "toy", the data was not divided into training and test splits, which required the framework to perform data splitting during modeling.

Open-ended task benchmark 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, steps for completion, 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 showcase about these tasks in Figure 10, Figure 12, and Figure 13 in the Appendix.

4.1.2 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). Considering the variability in interpreting test results, we manually reviewed the outputs generated by all methods to determine the count of accurate responses. 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's 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 evaluation metric, which may vary between tasks, including metrics such as accuracy, F1, AUC and RMSLE, etc. For metrics such as accuracy, F1, and AUC, we presented the raw values to facilitate

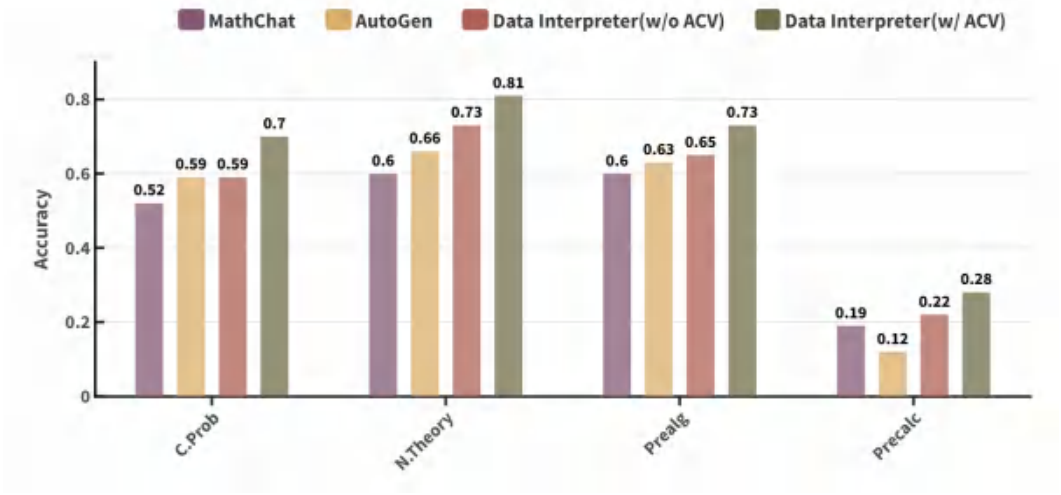


Figure 7: **Performance on the MATH dataset.** We evaluate all the problems with difficulty level 5 from 4 categories of the MATH dataset. We set N=3 for ACV.

comparison across identical data tasks. We normalize all performance values s:

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

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.

4.1.3 Baselines and Implementation Details

GPT-4-Turbo (gpt-4-1106-preview) was used in all frameworks to ensure an impartial performance evaluation. To ensure a fair comparison with other frameworks, we kept our experience pool empty to eliminate any prior knowledge. The effect of experience learning is reported in Table 3. ***MATH dataset:*** We adopted zero-shot baselines, including MathChat (Wu et al., 2023b) and AutoGen (Wu et al., 2023a) with GPT-4-Turbo as the baseline for a fair comparison. We set N=3 for ACV in the MATH dataset. Considering the variability in interpreting test results, we manually reviewed the outputs generated by all methods to determine the count of accurate responses (Wu et al., 2023b). ***ML-Benchmark:*** We selected four typical open-source LLM-based agent frameworks that support data analysis and modeling as baselines: XAgent (Team, 2023), AutoGen (Wu et al., 2023a), Open-Interpreter (Lucas, 2023), and TaskWeaver (Qiao et al., 2023). By default, we set $N = 1$ for ACV in ML-Benchmark and conducted the experiments before January 2024 for all baseline frameworks. ***Open-ended task benchmark:*** We employed AutoGen (Wu et al., 2023a) and OpenInterpreter (Lucas, 2023) as baseline models. Each framework underwent three experiments per task, and we reported the average completion rate. We also set $N = 1$ for ACV in the open-ended task benchmark by default.

4.2 Main Result

Performance on math problem solving As illustrated in the Figure 7, the Data Interpreter achieved the best results across all tested categories, reaching 0.81 accuracy in the N.Theory cat-

Table 1: **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.

Model / Task	WR	BCW	Titanic	House Prices	SCTP	ICR	SVPC	Avg.
AutoGen	0.96	**0.99**	0.87	0.86	0.83	0.77	0.73	0.86
Open Interpreter	**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
XAgent	**1.00**	0.97	0.42	0.42	0	0.34	0.01	0.45
Data Interpreter	0.98	**0.99**	**0.91**	**0.96**	**0.94**	**0.96**	**0.89**	**0.95**

Table 2: **Performance comparisons on Open-ended task benchmark.** The tested tasks include "OCR" (Optical Character Recognition), "WSC" (Web Search and Crawling), and "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.
AutoGen	0.67	0.65	0.10	0.26	**1.00**	0.10	0.20	0.67	0.46
Open Interpreter	0.50	0.30	0.10	0.36	**1.00**	0.50	0.25	0.20	0.40
Data Interpreter	**0.85**	**0.96**	**0.98**	**1.00**	**1.00**	**1.00**	**1.00**	**0.93**	**0.97**

egory, which was a 0.15 improvement over AutoGen. In the most challenging category, Precalc, the Data Interpreter obtained an accuracy of 0.28, an increase of 0.16 compared to AutoGen. Notably, the inclusion of ACV resulted in significant improvements across all task categories, with an average improvement of 17.29% relative improvement compared to the version without ACV. On average, the ACV strategy showed 26% relative improvement compared to AutoGen.

Performance on machine learning In Table 1, Data Interpreter achieved a comprehensive score of 0.95 across the seven tasks, compared to an average score of 0.86 by AutoGen, marking a significant 10.3% improvement. It was the only framework with a comprehensive score exceeding 0.9 on Titanic, House Prices, SCTP, and ICR. Data Interpreter outperformed other frameworks and gained a significant advantage on corresponding datasets, showing a notable improvement of 24.7% and 21.2% over AutoGen in ICR and SVPC, respectively. The Data Interpreter completed all mandatory processes on every dataset and consistently maintained superior performance, more details can be found in Table 6 in the Appendix.

Performance on open-ended tasks Table 2 illustrates that Data Interpreter achieved a completion rate of 0.97, marking a substantial 112% improvement compared to AutoGen.

For the IBR task, all three frameworks achieved a 1.0 completion score. In OCR-related tasks, Data Interpreter achieved an average completion rate of 0.85, outperforming AutoGen and OpenInterpreter by 26.8% and 70.0%, respectively. In 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. AutoGen and OpenInterpreter failed to log in and obtain the status for the ER task, resulting in a lower completion rate. The Data Interpreter can dynamically adjust the task and achieve a 0.98 score in completion rate.

4.3 Ablation Study

Ablation on core modules To assess the performance of various modules, we conducted ablation experiments with three additional configurations on the ML-Benchmark. The initial setup entailed the ReAct (Yao et al., 2022) framework with simplified prompt phrases that allow code execution. The second configuration integrated dynamic planning, encompassing hierarchical planning and dynamic plan management following each step to facilitate real-time adjustments. The third config-

Table 3: **Ablation on core modules.** Evaluated with comprehensive score on ML-Benchmark. "ICR", "SCTP", and "SVPC" represent "ICR - Identifying age-related conditions", "Santander customer transaction prediction", and "Santander value prediction challenge", respectively.

Code execution	Dynamic plan	Tool	House Prices	SCTP	SVPC	ICR	Avg.
✓			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**

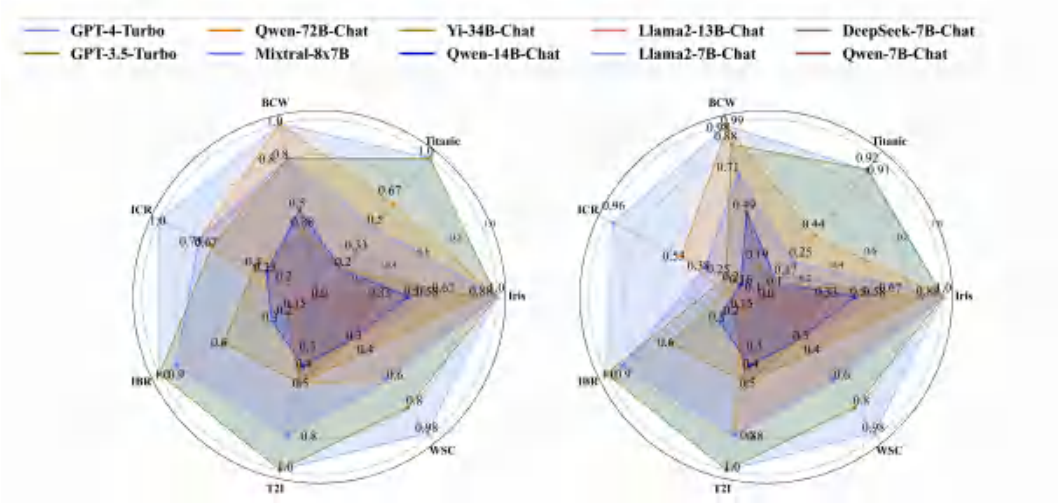


Figure 8: **Evaluation on ML-Benchmark with different LLMs.** Left: completion rate. Right: comprehensive score.

uration incorporated the utilization and generation functionalities of tools, which defaulted to the Data Interpreter.

As indicated by Table 3, dynamic planning yielded a significant improvement of 0.48. It helped prepare the dataset and track changes to the data in real-time, resulting in better performance, especially in terms of completion rate. Furthermore, using tools resulted in an additional improvement of 9.84%, bringing the comprehensive score to 0.94.

Ablation on LLM backbones In machine learning tasks, more extensive LLM backbones such as Qwen-72B-Chat (Bai et al., 2023) and Mixtral-8x7B (Jiang et al., 2024) exhibited performance comparable to GPT-3.5-Turbo, while smaller LLMs experienced performance degradation.

As shown in Figure 8, Data Interpreter, when paired with smaller models such as Yi-34B-Chat (01-ai, 2023), Qwen-14B-Chat (Bai et al., 2023), Llama2-13B-Chat (Touvron et al., 2023), and even DeepSeek-7B-Chat (Bi et al., 2024), effectively handled tasks such as data loading and analysis. However, these models faced limitations when executing tasks requiring advanced coding proficiency, which can lead to incomplete processes. In open-ended tasks, Mixtral-8x7B achieved high completion rates in three tasks but encountered challenges in the WSC task due to difficulty accurately outputting complete results to CSV files. Similar to machine learning tasks, smaller LLMs encountered execution failures due to their restricted coding abilities while acquiring images or parsing webpage results. (See Figure 8).

Ablation on experience learning To evaluate experience learning, we conducted experiments on five tasks with varying experience pool sizes, measuring task efficiency by debugging attempts and cost. Increasing the pool size from 0 to 200 significantly reduced debugging attempts from 1.48 to 0.32 per task, with costs decreasing from $0.80 to $0.24. This highlights substantial efficiency gains from experience learning. Notably, at a pool size of 80, debugging attempts decreased, especially in ER, Titanic, and House Prices tasks, by 1.25, 1, and 1, respectively. This underscored the

Table 4: **Ablation on experience pool size.** We evaluate the efficiency of completing tasks by taking the average of the number of debugging attempts (NDA) required and the total cost, excluding experience retrieval and usage. We take the mean of multiple attempts to arrive at the final value. Lower values indicate better performance. "Avg." and "Std." denote "Average" and "Standard deviation", respectively.

Experience pool size	size=0		size=80		size=200	
Metric	NDA	Cost ($)	NDA	Cost ($)	NDA	Cost ($)
WSC	1.50	0.69	0.80	0.33	**0.40**	**0.29**
ER	1.50	0.81	0.25	0.27	**0**	**0.13**
Titanic	1.40	0.65	0.40	0.35	**0.40**	**0.31**
House Prices	1.60	1.01	0.60	0.15	**0.40**	**0.15**
ICR	1.40	0.85	0.60	0.44	**0.40**	**0.37**
Avg. / Std.	1.48 / 0.08	0.80 / 0.14	0.53 / 0.21	0.31 / 0.11	**0.32** / 0.18	**0.24** / 0.10

efficiency enhancement even with a modest pool size, indicating the sensitivity of LLMs to context and effectiveness in code-centric problem-solving.

5 Conclusion

While current LLM-based agents have proven effective in managing static and straightforward tasks, they lose competency when confronted with intricate multi-step challenges, particularly evident in tasks such as machine learning. Our analysis reveals that the complex dependencies among various tasks pose critical challenges, often involving task decomposition where the failure of a single task disrupts the entire process. Additionally, the dynamic nature of these tasks introduces variability in specific problems, demanding a nuanced understanding of coding tools beyond the current API-focused capabilities. Existing methods, primarily designed for static tasks, often overlook these challenges common in complex data science scenarios. In addition, LLM-based agents, with the aid of code execution feedback, can enhance their problem-solving abilities. However, their capacity is limited when interpreting error feedback for task completion evaluation. In data science scenarios, LLM-based agents must distinguish logical errors from error-free feedback, thereby verifying the reliability of their code solutions and providing more accurate results.

To address these challenges, we introduce the Data Interpreter, a solution for data science problem-solving through dynamic planning with hierarchical graphs, tools integration and evolution, and automated confidence-based verification. Our Data Interpreter is meticulously designed to enhance reliability, automation, and reasoning capability in managing complex data science tasks. Through extensive evaluations, our Data Interpreter outperforms various open-source frameworks in machine learning tasks, mathematical problems, and real-world task performance, signifying a substantial advancement in the capabilities of LLM-based agents for data science.

References

01-ai. Yi-34B-Chat. `https://huggingface.co/01-ai/Yi-VL-34B`, 2023.

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

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

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint*, 2024.

Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *ICSE*, 2022.

Sebastian Bordt, Ben Lengerich, Harsha Nori, and Rich Caruana. Data science with llms and interpretable models. *arXiv preprint*, 2024.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint*, 2023.

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

Kexin Chen, Hanqun Cao, Junyou Li, Yuyang Du, Menghao Guo, Xin Zeng, Lanqing Li, Jiezhong Qiu, Pheng Ann Heng, and Guangyong Chen. An autonomous large language model agent for chemical literature data mining. *arXiv preprint*, 2024b.

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

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

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint*, 2023.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint*, 2024.

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

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

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint*, 2023.

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

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

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. Mixtral of experts. *arXiv preprint*, 2024.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint*, 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *ICRA*, 2023.

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

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

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *NeurIPS*.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *NeurIPS*, 2024.

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

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

OpenAI. GPT-4-Code-Interpreter. `https://chat.openai.com/?model=gpt-4-code-interpreter`, 2023.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint*, 2023.

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

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. Taskweaver: A code-first agent framework. *arXiv preprint*, 2023.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 2023.

Mario Sänger, Ninon De Mecquenem, Katarzyna Ewa Lewińska, Vasilis Bountris, Fabian Lehmann, Ulf Leser, and Thomas Kosch. Large language models to the rescue: Reducing the complexity in scientific workflow development using chatgpt. *arXiv preprint*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *NeurIPS*, 2024.

Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Lu Wang, Ruoxi Jia, and Ming Jin. Algorithm of thoughts: Enhancing exploration of ideas in large language models. *arXiv preprint*, 2023.

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

Seunggyoon Shin, Seunggyu Chang, and Sungjoon Choi. Past as a guide: Leveraging retrospective learning for python code completion. *arXiv preprint*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 2024.

Xiangru Tang, Qiao Jin, Kunlun Zhu, Tongxin Yuan, Yichi Zhang, Wangchunshu Zhou, Meng Qu, Yilun Zhao, Jian Tang, Zhuosheng Zhang, et al. Prioritizing safeguarding over autonomy: Risks of llm agents for science. *arXiv preprint*, 2024.

XAgent Team. Xagent: An autonomous agent for complex task solving, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint*, 2023.

Marcel Waldvogel. Fast longest prefix matching: algorithms, analysis, and applications. 2000.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint*, 2023.

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

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

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

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, et al. Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. *arXiv preprint*, 2024.

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

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

Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, et al. Igniting language intelligence: The hitchhiker's guide from chain-of-thought reasoning to language agents. *arXiv preprint*, 2023.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *arXiv preprint*, 2023.

Xiawu Zheng, Yang Zhang, Sirui Hong, Huixia Li, Lang Tang, Youcheng Xiong, Jin Zhou, Yan Wang, Xiaoshuai Sun, Pengfei Zhu, et al. Evolving fully automated machine learning via life-long knowledge anchors. *TPAMI*, 2021.

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

Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, et al. Agents: An open-source framework for autonomous language agents. *arXiv preprint*, 2023b.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. *arXiv preprint*, 2023.

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

A APPENDIX

A.1 PLAN EXAMPLE

A.1.1 NODE COMPONENTS IN THE TASK GRAPH

Figure 9 illustrates the structure of each task. Each task is structured to include instructions, dependencies array, code, and a flag. Specifically, dependencies array and flag are designed to maintain and manage the node's dependency and runtime status, while instructions and code describe tasks in natural and coding languages respectively. Since the code of each task is automatically executed in the code interpreter, the corresponding code is directly entered into the code interpreter used by its predecessor task to ensure the consistency of code variables between sequential tasks. During the task execution process, the execution results will be stored as runtime results.

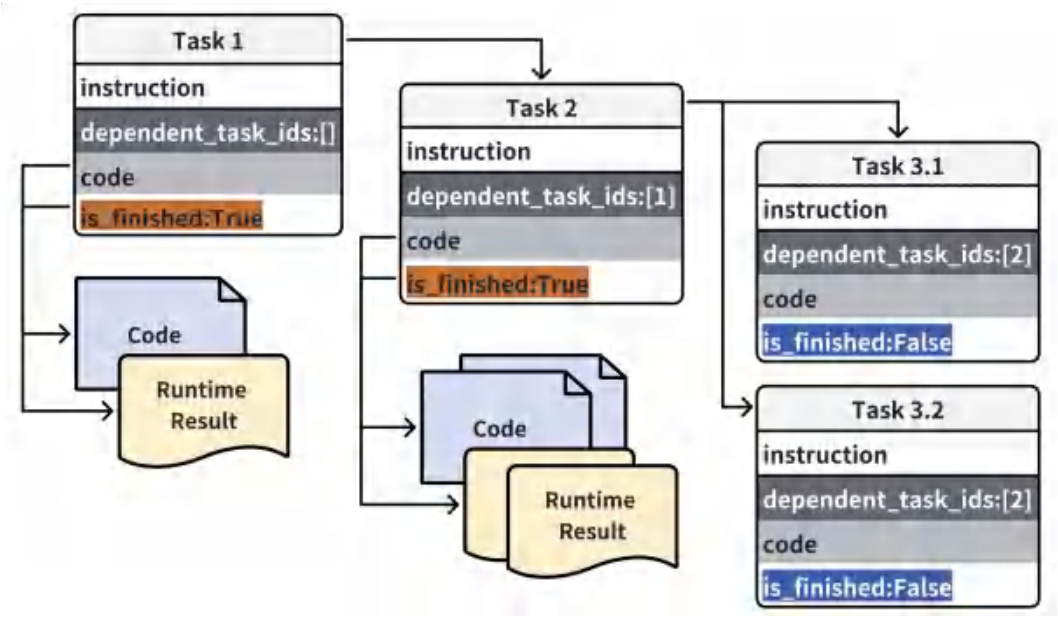


Figure 9: The components of a task in task graph

A.1.2 AN EXAMPLE OF PLAN

Below is an example of a typical plan for a machine-learning task.

Plan example

```
```[
    {
        "task_id": "1",
        "dependent_task_ids": [],
        "instruction": "Conduct data exploration on the train_sett.csv to understand its structure, missing values , and basic statistics."
    },
    {
        "task_id": "2",
        "dependent_task_ids": ["1"],
        "instruction": "Perform correlation analysis and causal inferences to identify relationships between variables."
    },
    {
        "task_id": "3",
        "dependent_task_ids": ["1"],
        "instruction": "Implement anomaly detection to identify and handle outliers in the dataset."
    },
    {
        "task_id": "4",
        "dependent_task_ids": ["2", "3"],
        "instruction": "Carry out feature engineering to prepare the dataset for predictive modeling."
    },
    {
        "task_id": "5",
        "dependent_task_ids": ["4"],
        "instruction": "Develop a predictive model using the processed dataset to determine machine operational status. "
    },
    {
        "task_id": "6",
        "dependent_task_ids": ["5"],
        "instruction": "Evaluate the model's performance using the eval_set.csv"
    },
    {
        "task_id": "7",
        "dependent_task_ids": ["5", "6"],
        "instruction": "Visualize the analysis and prediction results with high-quality graphs."
    },
    {
        "task_id": "8",
        "dependent_task_ids": ["2", "3", "4", "6", "7"],
        "instruction": "Write a detailed report covering methodologies, findings, and model performance, and save it as a text file."
    }
] ```
```

### A.2 TOOLS DETAILS

The tools of our Data Interpreter are listed in Table 5
```

Table 5: **Tools of our Data Interpreter.**

Tool name	Tool type	Functions	Domain
FillMissingValue	Class	4	Machine learning
MinMaxScale	Class	4	Machine learning
StandardScale	Class	4	Machine learning
MaxAbsScale	Class	4	Machine learning
LabelEncode	Class	4	Machine learning
OneHotEncode	Class	4	Machine learning
OrdinalEncode	Class	4	Machine learning
RobustScale	Class	4	Machine learning
CatCount	Class	4	Machine learning
TargetMeanEncoder	Class	4	Machine learning
KFoldTargetMeanEncoder	Class	4	Machine learning
CatCross	Class	4	Machine learning
SplitBins	Class	4	Machine learning
GeneralSelection	Class	4	Machine learning
TreeBasedSelection	Class	4	Machine learning
VarianceBasedSelection	Class	4	Machine learning
PolynomialExpansion	Class	3	Machine learning
GPT-4V	Class	3	Multimodal
SD-T2I	Class	3	Multimodal
Web Scraping	Function	1	Common

A.3 AN EXAMPLE OF TOOL SCHEMA

Tool schema for a feature engineering tool

```
type: class
description: Add value counts of a categorical column as new feature.
methods:
  __init__:
    type: function
    description: Initialize self.
    parameters:
      properties:
        col:
          type: str
          description: Column for value counts.
      required:
      - col
  fit:
    type: function
    description: Fit a model to be used in subsequent transform.
    parameters:
      properties:
        df:
          type: pd.DataFrame
          description: The input DataFrame.
      required:
      - df
  fit_transform:
    type: function
    description: Fit and transform the input DataFrame.
    parameters:
      properties:
        df:
          type: pd.DataFrame
          description: The input DataFrame.
      required:
      - df
    returns:
    - type: pd.DataFrame
      description: The transformed DataFrame.
  transform:
    type: function
    description: Transform the input DataFrame with the fitted model.
    parameters:
      properties:
        df:
          type: pd.DataFrame
          description: The input DataFrame.
      required:
      - df
    returns:
    - type: pd.DataFrame
      description: The transformed DataFrame.
```

A.4 TOOL USAGE PROMPTS

We use two types of prompts for tool utilization. For open-ended tasks, we use zero-shot prompts, and for machine-learning tasks, we use one-shot prompts as illustrated below.

Zero-shot tool usage prompt

```
# Instruction
Write complete code for 'Current Task'. And avoid duplicating code from finished tasks, such as repeated import of packages, reading data, etc.
Specifically, {tool_type_usage_prompt}

# 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 (can be empty):
Each Class tool is described in JSON format. When you call a tool, import the tool first.
{tool_schemas}

# Constraints:
- Ensure the output new code is executable in the same Jupyter notebook with the previous tasks code have been executed.
- Always prioritize using pre-defined tools for the same functionality.
"""
```

One-shot tool usage prompt

```
# 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 have 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.
"""
```

A.5 Additional results

For a deeper understanding, Table 6 presents the results on the ML-benchmark for both Completion Rate and Normalized Performance Score metrics. Additionally, Table 7 showcases the results of ablation experiments on the ML-benchmark, focusing on the Completion Rate and Normalized Performance Score metrics.

A.6 ML-Benchmark dataset description

Here are the details about the ML-Benchmark dataset. We collect several typical datasets from Kaggle[1] and machine learning. Details are in Table 8

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

Table 6: **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	0.91
Open Interpreter	**1.00**	0.90	0.92	0.88	0.85	0.91	0.88	0.93
TaskWeaver	**1.00**	**1.00**	0.83	0.88	0.67	0.83	0.80	0.89
XAgent	**1.00**	**1.00**	0.83	0.83	0	0.67	0	0.70
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
Open Interpreter	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
Data Interpreter	0.96	**0.99**	**0.82**	**0.91**	**0.89**	**0.91**	**0.77**	**0.89**

Table 7: **Additional ablation on core modules.** Evaluated with CR, NPS and CS on ML-Benchmark. "ICR", "SCTP", and "SVPC" represent "ICR - Identifying age-related conditions", "Santander customer transaction prediction", and "Santander value prediction challenge", respectively.

Code execution	Dynamic plan	Tool	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	0.8	0.74	0.86
✓	✓	✓	0.96	0.95	0.89	0.96	0.94

Table 8: Details of the ML-Benchmark dataset, including dataset name, brief 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 preprocessing, 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

A.7 OPEN-ENDED TASK DETAILS

We are showcasing several typical Open-Ended Tasks in the following illustrations. For each task, we include the necessary data, user requirements, and assessment pipeline.

(1) Invoice 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 10: Open-ended task cases (invoice OCR and web search and crawling)

A.8 RESULTS OF OPEN-ENDED TASKS

We present the results by the Data Interpreter of several open-ended tasks in two figures: tasks 8, 9, 10, and 13 in Figure 14, and tasks 4, 14, and 15 in Figure 15.

A.9 VISUALIZATION RESULT

Visualization capability can be demonstrated in Figure 16

(3) Email reply (Task 8)

Scenario Description: Filter through my emails and respond to them as necessary

User Requirement: You are an agent that automatically reads and replies to emails. I will give you your Outlook email account and password. You need to check the content of the latest email and return it to me. If the email address suffix of this email is @communication.microsoft.com, please automatically reply with "I've received your email and will reply as soon as possible. Thank you!"
Email account: englishgpt@outlook.com
Email Password: xxxx

Data: -

Pipeline Requirement:
1. Login to the target email account
2. Summarize and filter the email content accordingly.
3. set up an automatic reply to the sender with an email address that ends with a specific domain name.

Performance Requirement: -

(4) Web page imitation (Task 9-13)

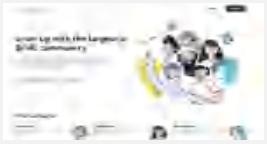
Scenario Description: Using Selenium and WebDriver to access a webpage and convert it to an image, with the assistance of GPT-4V to mimic the creation of a one-page website.

- Task 10:
This is a URL of webpage: https://pytorch.org/. Firstly, utilize Selenium and WebDriver for rendering. Secondly, convert image to a webpage including HTML, CSS and JS in one go. Finally, save webpage in a file.
NOTE: All required dependencies and environments have been fully installed and configured.

- Task 11:
This is a URL of webpage: https://www.kaggle.com/. Firstly, utilize Selenium and WebDriver to render the webpage, ensuring the browser window is maximized for an optimal viewing experience. Secondly, convert image to a webpage including HTML, CSS and JS in one go. Finally, save webpage in a file. NOTE: All required dependencies and environments have been fully installed and configured.

- Task 12:
This is a URL of webpage: https://chat.openai.com/auth/login. Firstly, utilize Selenium and WebDriver to render the webpage, ensuring the browser window is maximized for an optimal viewing experience. Secondly, convert image to a webpage including HTML, CSS and JS in one go. Finally, save webpage in a file. NOTE: All required dependencies and environments have been fully installed and configured.

Data: (Task 10-12 in order)


Pipeline Requirement:
1. Open a target Web URL
2. Transform the Website into an image
3. Send the image to GPT-4V via API
4. Request a similar website generation using the code.

Performance Requirement: Similarity/Correctness

Figure 11: Open-ended task cases (email reply and web page imitation)

(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: -

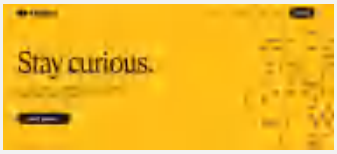
(7) Image2Code (Task 16-17)

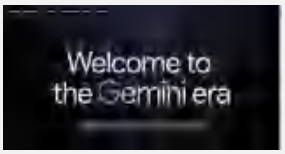
Scenario Description: Web code generation

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

(5)Mini game generation (Task 18 &20)

Scenario Description: Game tool usage (pyxel)

User Requirement:
- Task 18:
Create a Snake game. Players need to control the movement of the snake to eat food and grow its body, while avoiding the snake's head touching their own body or game boundaries. Games need to have basic game logic, user interface. During the production process, please consider factors such as playability, beautiful interface, and convenient operation of the game.
Note: pyxel environment already satisfied

- Task 20:
Make a mouse click game that click button as many times as possible in 30 seconds using pyxel.
Note: pyxel environment already satisfied

Data: -

Pipeline Requirement: -

Performance Requirement: -

Figure 13: Open-ended task cases (mini-game generation)

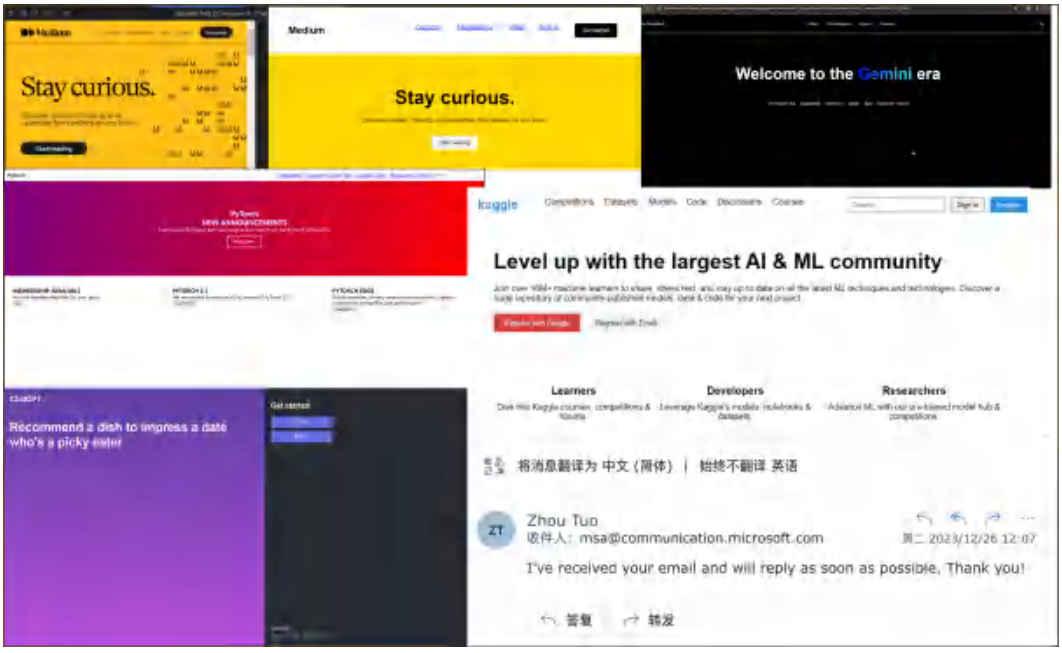


Figure 14: Web page imitation / automated email replies by Data Interpreter

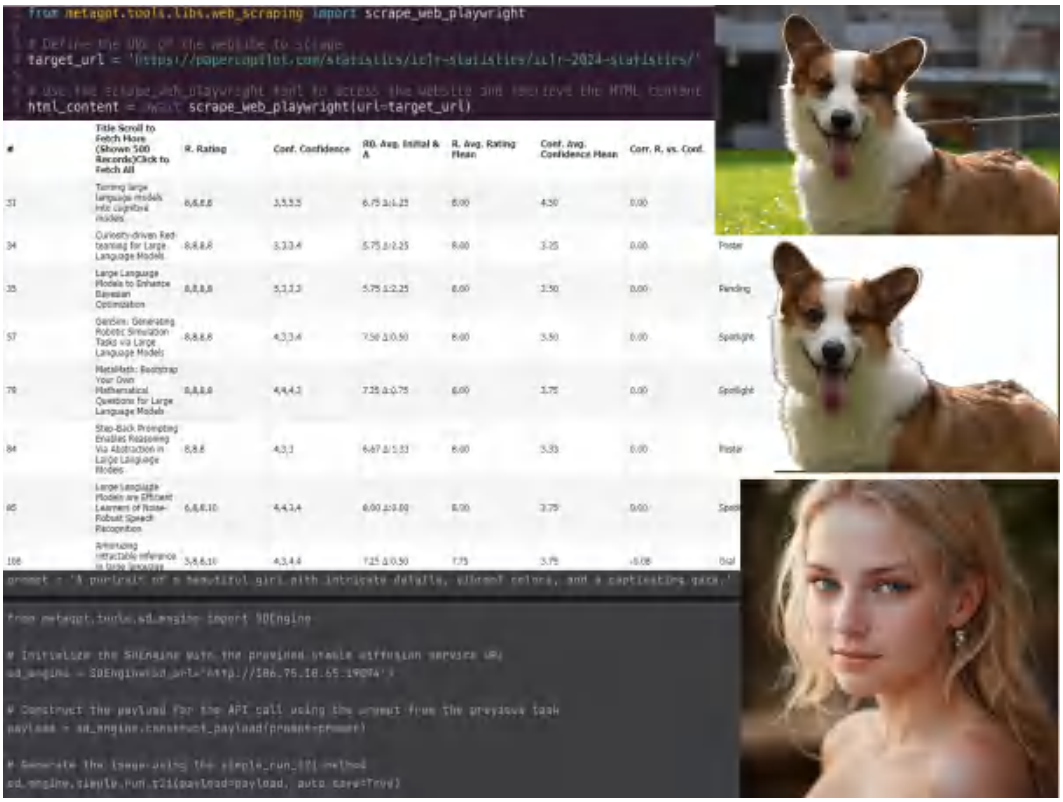


Figure 15: Image background removal / text-to-image / web search and crawling by Data Interpreter

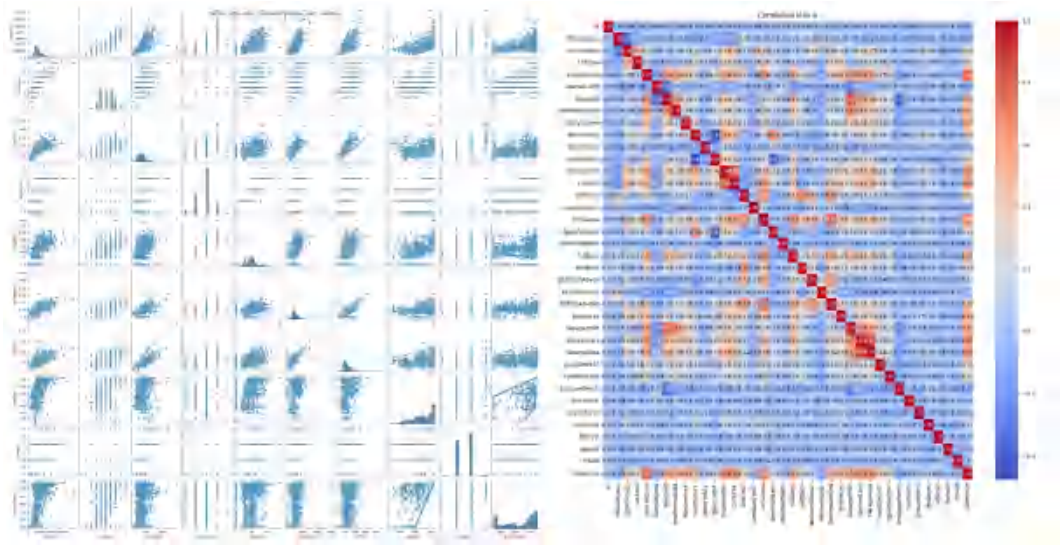


Figure 16: Enhancing data analysis and visualization capabilities of Data Interpreter