DatawiseAgent: A Notebook-Centric LLM Agent Framework for Adaptive and Robust Data Science Automation

Ziming You[1], **Yumiao Zhang**[2], **Dexuan Xu**[3], **Yiwei Lou**[3],
Yandong Yan[3], **Wei Wang**[4], **Huamin Zhang**[5], **Yu Huang**[1] *
[1] National Engineering Research Center for Software Engineering, Peking University
[2] School of Software & Microelectronics, Peking University
[3] School of Computer Science, Peking University
[4] Xi'an Jiaotong University
[5] Institute of Basic Theory of Chinese Medicine, China Academy of Chinese Medical Sciences
`zimingyou@stu.pku.edu.cn,hy@pku.edu.cn`

Abstract

Existing large language model (LLM) agents for automating data science show promise, but they remain constrained by narrow task scopes, limited generalization across tasks and models, and over-reliance on state-of-the-art (SOTA) LLMs. We introduce **DatawiseAgent** [1], a notebook-centric LLM agent framework for adaptive and robust data science automation. Inspired by how human data scientists work in computational notebooks, DatawiseAgent introduces a unified interaction representation and a multi-stage architecture based on finite-state transducers (FSTs). This design enables flexible long-horizon planning, progressive solution development, and robust recovery from execution failures. Extensive experiments across diverse data science scenarios and models show that DatawiseAgent consistently achieves SOTA performance by surpassing strong baselines such as AutoGen and TaskWeaver, demonstrating superior effectiveness and adaptability. Further evaluations reveal graceful performance degradation under weaker or smaller models, underscoring the robustness and scalability.

1 Introduction

Data science, the practice of extracting knowledge and insights from data, spans a broad spectrum of processes from data gathering and interpretation to model building and decision making (Donoho, 2017; Zhang et al., 2024b). As demand for data-driven decision-making continues to grow, automating data science has become a longstanding and critical challenge. Although traditional efforts such as AutoML (He et al., 2021; Jin et al., 2023) have achieved success in well-defined stages such as model selection and hyperparameter tuning, broader tasks remain difficult to formalize or mechanize due to their inherently exploratory, interdependent, and context-independent nature (Bie et al., 2022).

Recent advances in Large Language Models (LLMs) and LLM-based agents (Xue et al., 2023; Cheng et al., 2023; Dibia, 2023; Hollmann et al., 2024; Zhang et al., 2023a) have opened new possibilities for automating data science. LLMs demonstrate strong zero/few-shot generalization, in-context reasoning, code generation, and tool use capabilities, enabling a new line of research on *data science agents* (Zhang et al., 2024b), which are autonomous systems that perform data science tasks through natural language interaction.

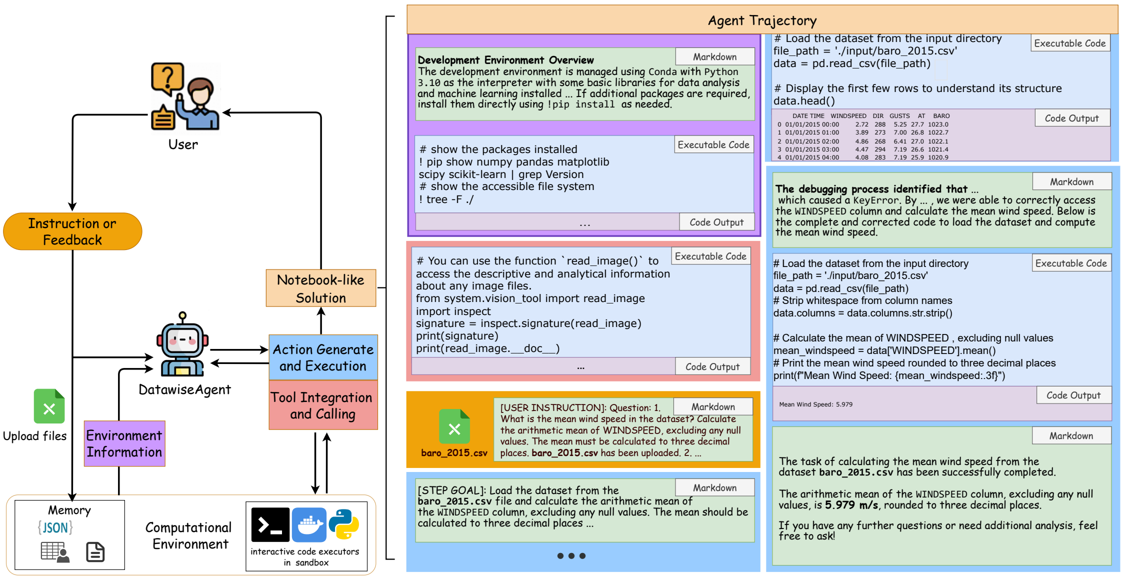
However, current data science agents face three key limitations: (1) **Focus on Isolated Phases.** Many existing agents target specific stages of the data science pipeline, such as feature engineering (Hollmann et al., 2024), model selection (Shen et al., 2024), or hyperparameter tuning (Zhang et al., 2023a) while overlooking the interdependent nature of real-world workflows. As a result, they fall short of supporting comprehensive end-to-end automation. (2) **Limited Task and Model Adaptability.** Agents designed for broader workflows often struggle to generalize across diverse task types or model configurations (Qiao et al., 2023; Hong et al., 2024; Hu et al., 2024b). While general-purpose frameworks such as ReAct (Yao et al., 2023; Wang et al., 2024) and AutoGen (Wu et al., 2023) offer cross-domain applicability, they tend to exhibit suboptimal performance in specialized scenarios such as exploratory analysis or predictive modeling, particularly under constrained model capacities. (3) **Over-Reliance on SOTA LLMs in Agent Design.** The majority of current data science agents are designed under the assumption of access to SOTA LLMs (Hong et al., 2024; Qiao et al., 2023), such as GPT-4o. These systems often lack scalability and robustness when

---

[*] Corresponding author: Yu Huang

[1] Our data and code are open-sourced at `https://github.com/Luffyzm3D2Y/DatawiseAgent`.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 1099–1123
November 4-9, 2025 ©2025 Association for Computational Linguistics



Development Environment Overview
The development environment is managed using `Conda` with `Python 3.10` as the interpreter with some basic libraries for data analysis and machine learning installed ... If additional packages are required, install them directly using `!pip install` as needed.

```
# show the packages installed
! pip show numpy pandas matplotlib
scipy scikit-learn | grep Version
# show the accessible file system
! tree -F ./
```

...

```
# You can use the function `read_image()` to
access the descriptive and analytical information
about any image files.
from system.vision_tool import read_image
import inspect
signature = inspect.signature(read_image)
print(signature)
print(read_image.__doc__)
```

...

[USER INSTRUCTION]: Question: 1. What is the mean wind speed in the dataset? Calculate the arithmetic mean of WINDSPEED, excluding any null values. The mean must be calculated to three decimal places. **baro_2015.csv** has been uploaded. 2. ...

[STEP GOAL]: Load the dataset from the **baro_2015.csv** file and calculate the arithmetic mean of the `WINDSPEED` column, excluding any null values. The mean should be calculated to three decimal places ...

```
# Load the dataset from the input directory
file_path = './input/baro_2015.csv'
data = pd.read_csv(file_path)

# Display the first few rows to understand its structure
data.head()
```

```
        DATE TIME  WINDSPEED  DIR  GUSTS    AT    BARO
0  01/01/2015 00:00      2.72  288   5.25  27.7  1023.0
1  01/01/2015 01:00      3.89  273   7.00  26.8  1022.7
2  01/01/2015 02:00      4.86  268   6.41  27.0  1022.1
3  01/01/2015 03:00      4.47  294   7.19  26.6  1021.4
4  01/01/2015 04:00      4.08  283   7.19  25.9  1020.9
```

The debugging process identified that ...
which caused a `KeyError`. By ... , we were able to correctly access the `WINDSPEED` column and calculate the mean wind speed. Below is the complete and corrected code to load the dataset and compute the mean wind speed.

```
# Load the dataset from the input directory
file_path = './input/baro_2015.csv'
data = pd.read_csv(file_path)
# Strip whitespace from column names
data.columns = data.columns.str.strip()

# Calculate the mean of WINDSPEED , excluding null values
mean_windspeed = data['WINDSPEED'].mean()
# Print the mean wind speed rounded to three decimal places
print(f"Mean Wind Speed: {mean_windspeed:.3f}")
```

```
Mean Wind Speed: 5.979
```

The task of calculating the mean wind speed from the dataset **baro_2015.csv** has been successfully completed.

The arithmetic mean of the `WINDSPEED` column, excluding any null values, is **5.979 m/s**, rounded to three decimal places.

If you have any further questions or need additional analysis, feel free to ask!

Figure 1: **DatawiseAgent performs diverse data science tasks across various models by operating entirely within a computational notebook.** The unified interaction representation expresses all agent–user–environment communication. Tool integration involves importing external APIs or libraries via code cells, with tool descriptions provided in markdown; environment information, such as system details or resource status, is either proactively injected as markdown at initialization or obtained through code execution during task progress.

deployed with smaller or open-source models, limiting their applicability in resource-constrained or privacy-sensitive settings.

To address these limitations, we draw inspiration from the exploratory, progressive, iterative workflows that human data scientists follow in computational notebooks (Head et al., 2019). As the *de facto* interface for data science, notebooks integrate natural language, code, and real-time feedback (Rule et al., 2018; Chattopadhyay et al., 2020; Wang et al., 2022, 2021). We posit that this paradigm provides a natural foundation for building adaptive and robust data science agents.

To this end, we propose **DatawiseAgent**, a notebook-centric LLM agent framework designed for **adaptive** and **robust** data science automation (see Figure 1). DatawiseAgent combines two key components: (1) a *unified interaction representation* that expresses all agent–user–environment communication as interleaved markdown and code cells within computational notebooks; and (2) a *finite-state transducer (FST)-based multi-stage architecture* that governs agent behavior across four functional stages, including DFS-like planning, incremental execution, self-debugging, and post-filtering. This design enables flexible long-horizon planning, progressive solution development, and robust recovery from execution failures, making DatawiseAgent suitable for deployment with LLMs of varying capacities and capabilities.

We evaluate DatawiseAgent on three representative data science scenarios, namely data analysis, scientific visualization, and predictive modeling, across both proprietary (GPT-4o, GPT-4o mini) and open-source (Qwen2.5 at multiple scales) LLMs. Experimental results show that DatawiseAgent consistently achieves SOTA performance under comparable evaluation conditions, surpassing strong baselines such as ReAct (Yao et al., 2023; Hu et al., 2024b), MatplotAgent (Yang et al., 2024b), AutoGen (Wu et al., 2023), and Taskweaver (Qiao et al., 2023). Notably, on the challenging DS-Bench data modeling benchmark, DatawiseAgent achieves over 90% task success and more than 40 Relative Performance Gap (RPG) across all LLMs, including surpassing prior SOTA results even when using GPT-4o mini. Further evaluation shows that DatawiseAgent maintains strong performance on weaker LLMs and widens the performance gap with baseline methods, highlighting its robustness and scalability. In summary, these results demonstrate that DatawiseAgent provides a practical and scalable foundation for robust, end-to-end data science automation across diverse tasks and LLM configurations.

2 Related Work

LLMs for Code Generation. Large Language Models (LLMs) have achieved strong performance across a range of code-related tasks (Jiang et al., 2024), including completion (Li et al., 2023; Roziere et al., 2023), translation (Chen et al., 2021), and repair (Anthropic, 2024; Achiam et al., 2023). However, generating correct code in a single attempt remains challenging, particularly for complex or interactive tasks (Chen et al., 2024). Recent studies show that external feedback and iterative refinement can significantly improve code generation (Zhou et al., 2024; Zhong et al., 2024; Madaan et al., 2024; Shinn et al., 2024). Building on these findings, we focus on data science tasks and investigate how to leverage diverse LLMs' limited reasoning and coding capabilities, along with feedback, to enable adaptive end-to-end automation.

LLM-based Data Science Agents. LLM-based agents have shown promise in automating various stages of the data science pipeline, such as feature engineering (Hollmann et al., 2024), model selection (Shen et al., 2024), and hyperparameter tuning (Zhang et al., 2023a). To support broader workflows, a range of frameworks have been proposed for machine learning pipelines (Guo et al., 2024; Jiang et al., 2025; Zhang et al., 2023b; Li et al., 2024; Trirat et al., 2024; Zhang et al., 2024a), data analysis (Qiao et al., 2023; OpenAI, 2023), and visualization (Yang et al., 2024b). While effective within specific scopes, many agents lack adaptability across tasks and models. In particular, most ML-focused agents (Guo et al., 2024; Jiang et al., 2025) adopt single-turn paradigms, limiting support for multi-turn interaction and human involvement. Furthermore, recent end-to-end systems (Hong et al., 2024) often rely on powerful proprietary LLMs, such as GPT-4o, limiting deployment with smaller or open-source models. In contrast, our agent framework supports robust, adaptive automation across diverse data science tasks and LLMs.

3 DatawiseAgent

We present DatawiseAgent, a novel notebook-centric LLM agent framework for effective, adaptive, and robust data science automation. Inspired by how human data scientists work, through *exploratory*, *progressive*, and *iterative* strategies within computational notebooks, DatawiseAgent comprises two key components: (1) *unified interaction representation* that captures all agent–user–environment communication via interleaved markdown and code cells; (2) *finite-state transducer (FST)-based multi-stage architecture* that governs agent behavior via transitions across four core stages. This architecture supports flexible planning, progressive solution development, and robust recovery from execution failures, making DatawiseAgent well-suited for models with varying reasoning and coding capabilities.

3.1 Unified Interaction Representation

Computational notebooks are central to data science practice, seamlessly integrating natural language, code, and execution feedback. Inspired by this paradigm, DatawiseAgent operates entirely within a notebook environment, enabling agents to reason, act, and revise solutions in a format familiar to practitioners. To support this design, we define a *unified* interaction representation (see Figure 1) in which all agent–user–environment communication, including task instructions, environment information, tool integration and calling, and observations, is expressed as a sequence of **markdown** and **executable code cells**. Agents incrementally construct solutions by generating and updating cells over multiple rounds, producing an interpretable execution trace that supports user feedback and follow-up interaction.

Unlike prior systems that adopt stage-specific task formats (e.g., JSON-based graph planning or mixed-format tool calls) (Qiao et al., 2023; Hong et al., 2024; Zhang et al., 2023c), DatawiseAgent maintains a structurally unified interaction mode, where context and actions are represented as cell sequences. We posit that this cell-level consistency reduces cognitive load and enhances in-context reasoning, particularly for models with constrained capabilities, while also enabling transparent oversight and unified multi-turn interaction.

3.2 FST-Based Multi-Stage Architecture

To govern the agent's behavior in a structured, modular, and extensible manner, DatawiseAgent adopts a finite-state transducer (FST)-based multi-stage architecture. Rather than defining a rigid workflow or static pipeline, DatawiseAgent organizes the problem-solving process into four core stages, including *DFS-like planning*, *incremental execution*, *self-debugging*, and *post-filtering* (detailed in Section 3.3), and employs a finite-state transducer (Hopcroft et al., 2001; Carroll and Long, 1989) to orchestrate autonomous transitions among them. Notably, this design facilitates modular extension with new stages and supports fine-grained ablation of individual components.

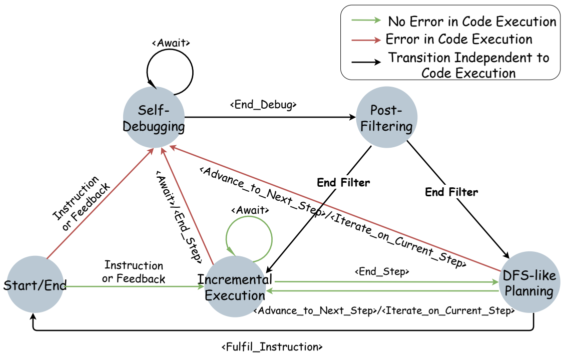


Figure 2: **State transition diagram of the FST-based multi-stage architecture**, modeled as a non-deterministic finite-state transducer (NFST). Transitions are driven by user instructions or feedback, agent-generated action signals, and execution feedback from the environment. At each state, the agent generates and executes actions based on the current context before proceeding to the next state.

We conceptualize the agent as a non-deterministic finite-state transducer (NFST), where the state space $Q = \{q_{\text{plan}}, q_{\text{inc}}, q_{\text{debug}}, q_{\text{filter}}, q_0\}$ corresponds to the four functional stages and a special start/end state. The idle state q_0 denotes either task completion or readiness for new instructions. State transitions are driven by internally generated *action signals* and external inputs, including user instructions and environment feedback (i.e., execution success or failure). At each state, DatawiseAgent produces two outputs: an *action*, uniformly represented as markdown and executable code cells, and an *action signal* indicating the intended next state. The action is executed in the notebook environment, yielding external feedback. The agent determines its next state via the transition function $\delta(q, \sigma, f)$, which takes as input the current state q, the generated action signal σ, and the feedback f from the environment or user.

The runtime logic of the FST-based architecture is formalized in Algorithm 1. The agent starts in an idle state and, upon receiving a user instruction, autonomously transitions across functional stages, generating and executing actions, processing feedback, and updating context, until the task is completed. It then returns to the idle state, awaiting further user instructions or feedback. The state transition process is visualized in Figure 2 using an NFST formulation for clarity. The corresponding deterministic FST, stage-wise action signal spaces, and implementation details are provided in Section C.

3.3 Detailed Explanation of Each Stage

To operationalize FST-based multi-stage architecture, DatawiseAgent organizes the agent's behavior into four functional stages: DFS-like planning, incremental execution, self-debugging, and post-filtering. These stages are inspired by how data scientists work in notebooks, collectively supporting flexible planning, progressive solution development, and robust recovery from execution failures.

DFS-like Planning and Incremental Execution. DatawiseAgent introduces two tightly coupled stages: DFS-like planning and incremental execution. Together, they form a tree-structured task-completion process (see Figure 3), enabling flexible exploration and progressive problem solving under constrained reasoning and coding capabilities.

In the DFS-like planning stage, the agent dynamically selects one of three actions based on task progress and feedback: (1) *advance* to the next subgoal; (2) *backtrack* to revise the current subtask by replacing it with a newly proposed one; or (3) *terminate* when the objective is satisfied. This non-linear planning strategy departs from static or sequential pipelines, enabling adaptive exploration of alternative solution paths. During incremental execution, instead of one-shot generation followed by iterative refinement, each subtask is completed step by step through interleaved markdown and code cells, leveraging fine-grained feedback. This progressive strategy exploits limited model capabilities while improving robustness against execution failures.

By coordinating planning and execution via transitions between q_{plan} and q_{inc}, DatawiseAgent enables models of varying capabilities to perform

---

Algorithm 1 FST-based Multi-Stage Architecture

```
Require: $I$: task input, $\mathcal{H}$: context history, Agent$_{\mathcal{P}}$: LLM
    agent with language model $\mathcal{P}$
Initialize context: $\mathcal{H} \leftarrow$ environment info and tools
$\mathcal{H}.update(I)$,  $q \leftarrow q_0$,  $\sigma \leftarrow I$,  $f \leftarrow$ no_error
while True do
    Generate action and action signal: A, signal $\leftarrow$ Agent$_{\mathcal{P}}(q, \mathcal{H})$
    Execute action $A$ and receive feedback $f \in \{$error, no_error$\}$
    Determine the next state: $q \leftarrow \delta(q, \sigma, f)$
    Update context with executed $A$: $\mathcal{H}.update(A, f)$
    Update action signal: $\sigma \leftarrow$ signal
    if $q = q_0$ then
        Exit loop (Task complete or waiting for new instructions)
    end if
end while
return $\mathcal{H}$
```

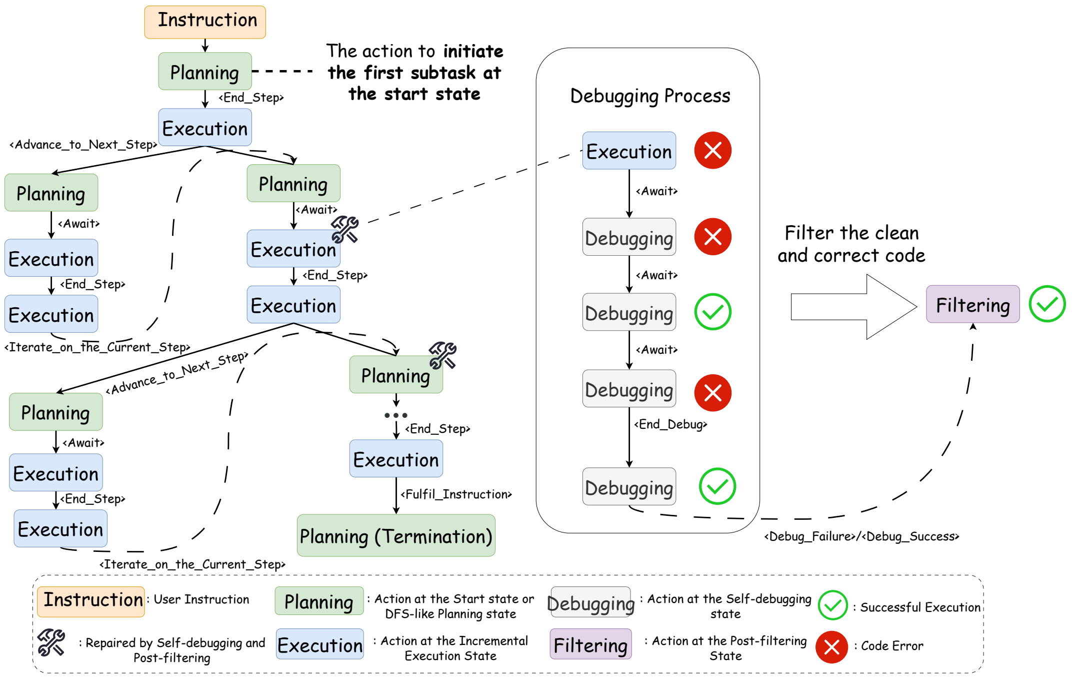

Instruction
Planning
The action to initiate the first subtask at the start state
<End_Step>
Execution
<Advance_to_Next_Step>
Planning
<Await>
Execution
<End_Step>
Execution
<Iterate_on_the_Current_Step>
Planning
<Await>
Execution
<End_Step>
Execution
<Advance_to_Next_Step>
Planning
<Await>
Execution
<End_Step>
Execution
<Iterate_on_the_Current_Step>
Planning
<End_Step>
Execution
<Fulfil_Instruction>
Planning (Termination)
Debugging Process
Execution
<Await>
Debugging
<Await>
Debugging
<Await>
Debugging
<End_Debug>
Debugging
Filter the clean and correct code
Filtering
<Debug_Failure>/<Debug_Success>
Instruction : User Instruction
Planning : Action at the Start state or DFS-like Planning state
Debugging : Action at the Self-debugging state
: Successful Execution
: Repaired by Self-debugging and Post-filtering
Execution : Action at the Incremental Execution State
Filtering : Action at the Post-filtering State
: Code Error

Figure 3: **Illustration of DatawiseAgent 's task-completion process.** Left: tree-structured trajectory from DFS-like planning and incremental execution. Right: code repair via self-debugging and post-filtering.

long-horizon reasoning and adaptively solve complex data science tasks through progressive strategies.

Code Repair through Self-Debugging and Post-Filtering. To ensure robust recovery from execution failures and prevent the accumulation of misleading traces in the context, DatawiseAgent introduces a code repair module implemented via FST transitions across two stages: self-debugging and post-filtering (see Figure 3).

In the self-debugging stage, the agent analyzes and iteratively refines faulty code using execution feedback. This stage is designed to be extensible, allowing integration of advanced LLM-based repair techniques (Hu et al., 2024a; Chen et al., 2024; Zhong et al., 2024) to further enhance correction performance. The post-filtering stage then assesses whether the error has been resolved in the debugging process: if successful, the agent extracts the clean and corrected code from the debugging trace; otherwise, it generates a concise diagnostic report in markdown, distilling key failure insights to prevent context pollution and guide future decisions.

This code repair module is triggered by execution errors during DFS-like planning or incremental execution. Upon completion, the agent replaces the original faulty code and debugging traces with post-filtered output, and resumes the ongoing task-solving process.

4 Experiments

In this section, We evaluate DatawiseAgent across three key dimensions: its *effectiveness* and *adaptability* across tasks and LLMs (Section 4.2), its *robustness* under varying model capabilities and scales (Section 4.3), and the contributions of its planning and code repair modules through an ablation study (Section 4.4).

4.1 Experimental Setup

Benchmarks, Evaluation Metrics, and Baselines. We evaluate DatawiseAgent on three public benchmarks covering core data science scenarios, each with tailored metrics and established baselines:

(1) **Data Analysis:** InfiAgent-DABench (Hu et al., 2024b) contains 257 challenges with CSV inputs and multi-level (easy/medium/hard) analysis questions. We report *Accuracy by Questions* (ABQ), i.e., the proportion of correctly answered questions. Baselines include ReAct (Yao et al., 2023; Hu et al., 2024b), AutoGen (Wu et al., 2023), TaskWeaver (Qiao et al., 2023), and Data Interpreter (Hong et al., 2024).

(2) **Scientific Visualization:** We use MatplotBench (Yang et al., 2024b), comprising 100 expert-verified cases involving input data, user queries, and reference plots. A vision model assigns a 0–100 score based on alignment with ground truth. We use GPT-4o as a unified scoring model across all settings. Baselines include Direct De-

Model	Method	ABQ/% ↑
GPT-4o mini	ReAct	80.08
	AutoGen	70.04
	Taskweaver	76.65
	Data Interpreter	67.70
	DatawiseAgent (Ours)	**82.88**
GPT-4o	ReAct	81.32
	AutoGen	73.54
	Taskweaver	**85.99**
	Data Interpreter*	94.93*
	Data Interpreter	75.78
	DatawiseAgent (Ours)	**85.99**
Qwen2.5-72B -Instruct	ReAct	75.88
	AutoGen	70.04
	Taskweaver	74.71
	DatawiseAgent (Ours)	**81.71**

Table 1: **Performance comparison on InfiAgent-DABench.** Asterisked (*) result is from Hong et al. (2024) and could not be reproduced in our setting; it is shown for reference only and excluded from SOTA comparison. Best results in bold; second-best underlined.

coding, where the model generates code in a single pass; MatplotAgent (Yang et al., 2024b), a vision-augmented agent specialized in plotting tasks; and AutoGen (Wu et al., 2023).

(3) **Predictive Modeling:** We use the data modeling part from DSBench (Jing et al., 2024), which includes 74 real-world Kaggle competitions. Each task requires predictive modeling based on training/testing data, a sample submission file, and a detailed description. Following DSBench (Jing et al., 2024), we report *Task Success Rate*, *Relative Performance Gap* (RPG).RPG serves as a standardized score that reflects the agent's overall performance across tasks by directly evaluating the performance of the resulting models on testing datasets. A task is marked incomplete if it exceeds the 3600-second time limit. We compare DatawiseAgent with results reported for AutoGen (Wu et al., 2023) and Code Interpreter[2].

Further details on the benchmarks, metric definitions, and method configurations are provided in Section D.

Model	Framework	Avg. Score ↑	Δ Score ↑
GPT-4o mini	Direct Decoding	38.09	-
	MatplotAgent	51.44	+13.35
	AutoGen	51.82	+13.73
	w/ visual tool	52.07	+13.98
	DatawiseAgent	55.85	+17.76
	w/ visual tool	**58.60**	**+20.51**
GPT-4o	Direct Decoding	45.28	-
	MatplotAgent	57.86	+12.58
	AutoGen	60.42	+15.14
	w/ visual tool	63.60	+18.32
	DatawiseAgent	61.22	+15.94
	w/ visual tool	**64.33**	**+19.05**
Qwen2.5 -72B-Instruct	Direct Decoding	47.54	-
	AutoGen	40.80	-6.74
	w/ visual tool	53.72	+6.18
	DatawiseAgent	56.41	+8.87
	w/ visual tool	**61.88**	**+14.34**

Table 2: **Performance comparison on MatplotBench.** **Δ Score** denotes the score gain over Direct Decoding. Bold and underline highlight the best and second-best results, respectively. Visual tool rows indicate integration with the GPT-4o mini-based visual tool.

Framework	Model	Task Success/%	RPG
AutoGen	Llama3-8B	5.41	1.55
	Llama3-70B	16.22	7.79
	GPT-4	87.84	45.52
	GPT-4o	71.62	34.74
	GPT-4o mini	22.97	11.24
Code Interpreter	GPT-4	54.05	26.14
	GPT-4o	44.59	19.87
	GPT-4o mini	39.19	16.90
Human*	Human*	100.00*	65.02*
DatawiseAgent	GPT-4o	**98.64**	**53.18**
	GPT-4o mini	**98.64**	46.61
	Qwen2.5-72B	91.89	42.90

Table 3: **Performance comparison on 74 Data Modeling tasks from DSBench.** Bold and underline indicate best and second-best results. *Human performance is from (Jing et al., 2024), based on evaluations across 22 competitions.

Model Configurations. To assess the adaptability and generalization across diverse LLMs, we evaluate DatawiseAgent using both proprietary and open-source models: GPT-4o, GPT-4o mini (Hurst et al., 2024), and Qwen2.5-72B-Instruct (Yang et al., 2024a) [3]. To further examine robustness under varying model capacities, we conduct additional experiments on InfiAgent-DABench (Hu et al., 2024b) using Qwen2.5 instruction-tuned models of different sizes: 7B, 14B, 32B, and 72B.

4.2 Effectiveness and Adaptability Across Tasks and LLMs

We evaluate DatawiseAgent's effectiveness and adaptability across three representative data science scenarios: data analysis, scientific visualization, and predictive modeling, using three distinct LLMs: GPT-4o, GPT-4o mini, and Qwen2.5-72B-Instruct. The comparative performance with existing agent frameworks is presented in Tables 1, 2, and 3.

Data Analysis. In data analysis, as shown in Table 1, DatawiseAgent achieves strong performance

---

[2] `https://platform.openai.com/docs/assistants/tools/code-interpreter`

[3] `gpt-4o-2024-08-06` and `gpt-4o-mini-2024-07-18`.

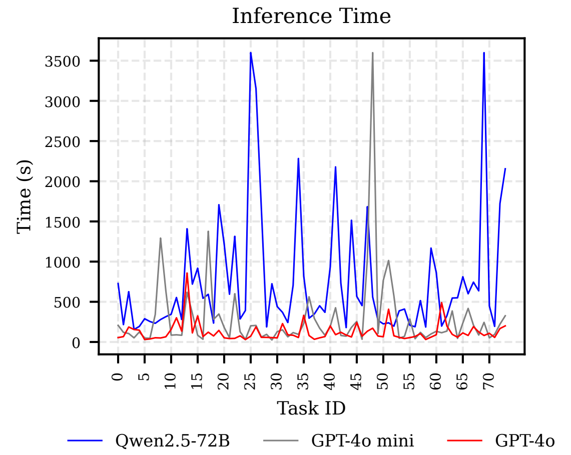


Figure 4: Inference time of DatawiseAgent on 74 data modeling tasks from DSBench.

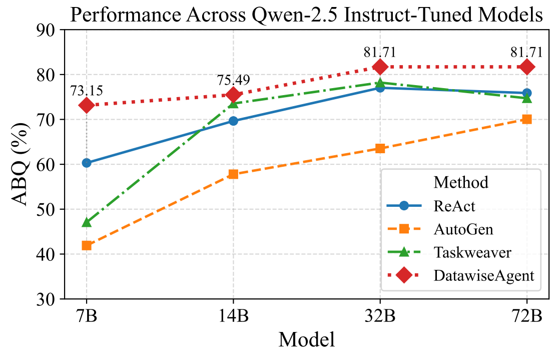


Figure 5: **Performance across Qwen2.5 models on InfiAgent-DABench.** DatawiseAgent demonstrates strong robustness across models of different sizes, maintaining top performance while the gap over competing methods becomes more pronounced on smaller models.

across all model settings, highlighting its strong capability in executing accurate and reliable data analyses. On GPT-4o mini and Qwen2.5-72B-Instruct, DatawiseAgent outperforms all baselines, achieving SOTA results. On GPT-4o, DatawiseAgent matches Taskweaver, a framework specifically designed for data analysis tasks, while surpassing AutoGen and ReAct. Notably, although Data Interpreter is reported to reach 94.93% on GPT-4o by Hong et al. (2024), our best-effort replication under comparable conditions yields significantly lower scores (75.78% on GPT-4o and 67.7% on GPT-4o mini), which we include for fair comparison. This discrepancy may be due to differences in evaluation settings not fully specified in the original paper. We include both results for transparency.

Scientific Visualization. In scientific visualization, DatawiseAgent consistently achieves the best performance across models (as shown in Table 2), highlighting its ability to produce high-quality scientific visual output. On GPT-4o, DatawiseAgent obtains the highest average score of 64.33, both with and without the visual tool (61.22 without, 64.33 with), establishing a new SOTA. We also observe that DatawiseAgent leads by a clear margin in completion rate and high-quality output proportion across all settings, demonstrating strong robustness in producing valid and reliable figures. These auxiliary metrics are reported in Appendix Table 11.

To further assess tool usage impact, we incorporate a GPT-4o mini-based visual tool into AutoGen and DatawiseAgent, enabling iterative figure refinement via visual-textual feedback. Implementation details are provided in Section D.4. Tool-integrated variants consistently outperform their non-tool counterparts, aligned with findings from (Yang et al., 2024b). Notably, DatawiseAgent with visual tool integration achieves the best results in all three model configurations, suggesting the effectiveness and sound design of our tool integration.

Model	Avg. LLM Calls	Planning	Code Repair
GPT-4o	12.31	4.72	0.62
GPT-4o-mini	18.80	4.15	1.39
Qwen2.5-72B-Instruct	16.91	5.05	1.18

Table 4: **Average number of transitions per task in DatawiseAgent on 74 Data Modeling tasks from DSBench.** "Avg. LLM Calls" counts the number of LLM calls, while "Planning" and "Code Repair" refer to transitions into the respective modules.

Method	**Inf. /s**	**Suc. /% ↑**	**RPG ↑**	**ABQ /% ↑**
DatawiseAgent	291.57	**98.64**	**46.61**	**77.14**
w/o planning	529.99	77.03	38.35	70.86
w/o code repair	429.18	87.84	43.80	75.43

Table 5: **Ablation results of DatawiseAgent on GPT-4o mini.** Metrics are reported for 74 data modeling tasks from DSBench and 175 medium-/hard-level data analysis tasks from InfiAgent-DABench. **Inf. /s** = Inference time, which measures the average time taken to complete a single task; **Suc. /%** = Task Success rate.

Predictive Modeling. In predictive modeling, DatawiseAgent achieves SOTA performance across all model settings, as shown in Table 3, demonstrating strong capability in solving comprehensive and complex end-to-end data-centric prediction tasks. It consistently obtains high Task Success Rates ($\geq$90%) and strong RPG values, with GPT-4o reaching the best overall performance (RPG 53.18). We further observe that DatawiseAgent with the weaker GPT-4o mini outperforms AutoGen with GPT-4, suggesting the potential for achieving competitive performance with smaller models.

Together, the results demonstrate the strong performance and adaptability of DatawiseAgent across diverse tasks and LLMs. It achieves SOTA results in both scientific visualization and predictive modeling, while maintaining strong performance in data analysis. Additionally, these results reinforce the effectiveness of DatawiseAgent's tool integration design, particularly in scientific visualization.

We also observe high task completion rates across domains (see Table 3; Appendix Table 11), which we attribute to DatawiseAgent 's FST-based multi-stage architecture orchestrating DFS-like planning, incremental execution, and code repair. This design supports flexible long-horizon planning, progressive solution building, and robust recovery from failures.

4.3 Robustness to Model Capability and Scale Variations

Robustness to Model Capability. Despite substantial differences in model capability, DatawiseAgent achieves comparable performance across GPT-4o, GPT-4o mini, and Qwen2.5-72B-Instruct on predictive modeling tasks. To understand this robustness, we analyze per-task inference time (i.e., time to complete a task) across models (Figure 4), finding that Qwen2.5-72B-Instruct incurs significantly longer durations despite strong performance. Manual inspection reveals that this discrepancy primarily stems from inefficient code execution—especially in data preprocessing and model training. This suggests that stronger models like GPT-4o tend to generate more efficient code, leading to faster execution. To explain how weaker models nonetheless maintain performance, we examine DatawiseAgent's internal state transitions (Table 4), including average LLM calls, planning steps, and code repair attempts per task. We find that weaker models invoke these modules more frequently, indicating that DatawiseAgent dynamically adaptively increases reasoning depth and self-correction to compensate for limited model capability. These results highlight DatawiseAgent 's robustness in adapting to varying model capabilities while maintaining competitive performance.

Robustness to Model Scale. We further test robustness by evaluating DatawiseAgent on Qwen2.5 instruct-tuned models of varying sizes (7B, 14B, 32B, 72B) using InfiAgent-DABench. As shown in Figure 5, although all agent frameworks degrade with smaller models, DatawiseAgent consistently outperforms all baselines on all scales. Notably, the performance gap between DatawiseAgent and other methods **widens substantially** as model size decreases, demonstrating DatawiseAgent 's robustness to model scale variation and its superior scalability compared to existing frameworks.

4.4 Ablation Study of Planning and Code Repair Modules

One key contribution of DatawiseAgent is its FST-based multi-stage architecture. To assess the impact of key components, we ablate two modules, DFS-like planning and code repair (self-debugging and post-filtering), on 175 medium- and hard-level cases from InfiAgent-DABench and the Data Modeling tasks in DSBench, using GPT-4o mini. We compare three variants: (1) **DatawiseAgent**: full system; (2) **w/o planning**: removes DFS-like planning, enforcing linear execution; (3) **w/o code repair**: disables code repair by removing self-debugging and post-filtering.

As shown in Table 5, performance consistently declines when either module is removed, with a more pronounced drop in predictive modeling, a more challenging task. These results underscore the importance of both flexible planning and robust recovery from failures in enabling DatawiseAgent to solve complex data science tasks. Due to space limitations, we defer cost analysis and case study of DatawiseAgent to Section A and Section B.

5 Conclusion

We propose DatawiseAgent, a notebook-centric LLM agent framework for adaptive and robust data science automation. By combining a unified interaction representation with an FST-based multi-stage architecture, DatawiseAgent supports flexible long-horizon planning, progressive solution development, and robust recovery from execution failures within computational notebooks. Experiments across diverse tasks and LLMs demonstrate its strong performance, adaptability, robustness across domains and models, establishing a notebook-centric paradigm for adaptive and robust data science automation.

6 Limitations

While DatawiseAgent demonstrates effectiveness, adaptability, and robustness across multiple tasks and LLMs, several limitations remain that suggest directions for future work. First, our evaluation of tool integration is limited to a single visual feedback tool in scientific visualization; broader assessment in domains with proprietary or complex toolchains (e.g., healthcare or finance) is needed. Second, although DatawiseAgent is naturally suited for integration into computational notebooks (e.g., Jupyter or Colab), we do not evaluate human-in-the-loop collaboration. This omission reflects our focus on autonomous task completion, but evaluating interactive workflows remains a valuable and methodologically challenging direction for future work. These limitations point to promising directions for expanding DatawiseAgent toward broader applicability in real-world and collaborative data science workflows.

7 Ethics Statement

This work does not involve human subjects, personal data, or proprietary user information. All datasets used in our experiments are publicly available. As DatawiseAgent is designed to automate data science workflows through LLM-based agents, we acknowledge potential risks related to error propagation, unintended behavior, or unsafe code execution in autonomous settings. We encourage responsible and lawful use of such systems and recommend incorporating appropriate safeguards before real-world deployment. Our methodology is fully transparent and reproducible, and we support continued dialogue around fairness, accountability, and reliability in LLM-based automation.

8 Acknowledgement

This paper was supported by National Key R&D Program of China (No. 2023YFC3502902, 2021YFF1201100), National Natural Science Foundation of China under Grants (62436006), Sanya Science and Technology Special Fund (No. 2024KFJX04) and Beijing Natural Science Foundation (No. L257018, No. L246024).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. `https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf`.

T De Bie, LD Raedt, J Hernández-Orallo, HH Hoos, P Smyth, and CKI Williams. 2022. Automating data science: Prospects and challenges. *Communications of the ACM*, 65(2):76–87.

John Carroll and Darrell Long. 1989. Theory of finite automata with an introduction to formal languages.

Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–12.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Liying Cheng, Xingxuan Li, and Lidong Bing. 2023. Is gpt-4 a good data analyst? In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9496–9514.

Victor Dibia. 2023. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 113–126.

David Donoho. 2017. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766.

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. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16813–16848. PMLR.

Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622.

Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

Noah Hollmann, Samuel Müller, and Frank Hutter. 2024. Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. *Advances in Neural Information Processing Systems*, 36.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Chenxing Wei, Danyang Li, Jiaqi Chen, Jiayi Zhang, and 1 others. 2024. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. 2024a. Leveraging print debugging to improve code generation in large language models. *arXiv preprint arXiv:2401.05319*.

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. 2024b. InfiAgent-DABench: Evaluating agents on data analysis tasks. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 19544–19572. PMLR.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*.

Haifeng Jin, François Chollet, Qingquan Song, and Xia Hu. 2023. Autokeras: An automl library for deep learning. *Journal of machine Learning research*, 24(6):1–6.

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? *arXiv preprint arXiv:2409.07703*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, and 1 others. 2024. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

OpenAI. 2023. Advanced data analysis (code interpreter). `https://platform.openai.com/docs/assistants/tools/code-interpreter`. Accessed: 2025-05-19.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, and 1 others. 2023. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

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. *Advances in Neural Information Processing Systems*, 36.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. 2024. Automl-agent: A multi-agent llm framework for full-pipeline automl. *arXiv preprint arXiv:2410.02958*.

April Yi Wang, Dakuo Wang, Jaimie Drozdal, Xuye Liu, Soya Park, Steve Oney, and Christopher Brooks. 2021. What makes a well-documented notebook? a case study of data scientists' documentation practices in kaggle. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–7.

April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2022. Documentation matters: Human-centered ai system to assist data science code documentation in computational notebooks. *ACM Transactions on Computer-Human Interaction*, 29(2):1–33.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232. PMLR.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, and 1 others. 2023. Db-gpt: Empowering database interactions with private large language models. *arXiv preprint arXiv:2312.17449*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, Zhiyuan Liu, Xiaodong Shi, and Maosong Sun. 2024b. MatPlotAgent: Method and evaluation for LLM-based agentic scientific data visualization. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 11789–11804, Bangkok, Thailand. Association for Computational Linguistics.

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

Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. 2024a. Mlcopilot: Unleashing the power of large language models in solving machine learning tasks. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2931–2959.

Michael Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. 2023a. Using large language models for hyperparameter optimization. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. 2023b. Automl-gpt: Automatic machine learning with gpt. *arXiv preprint arXiv:2305.02499*.

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

Yuge Zhang, Qiyang Jiang, Xingyu Han, Nan Chen, Yuqing Yang, and Kan Ren. 2024b. Benchmarking data science agents. *arXiv preprint arXiv:2402.17168*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 851–870.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning acting and planning in language models. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

A Cost Analysis

Model	Framework	Cost/$	ABQ/% ↑
GPT-4o	ReAct	4.79	81.32
	AutoGen	4.71	73.54
	TaskWeaver	16.19	85.99
	DatawiseAgent	10.60	85.99
GPT-4o mini	ReAct	0.26	80.08
	AutoGen	0.44	70.04
	TaskWeaver	1.64	76.65
	DatawiseAgent	1.14	82.88

Table 6: Total cost comparison on GPT-4o and GPT-4o mini on InfiAgent-DABench across 257 cases.

We record the total cost of DatawiseAgent across different model settings, as illustrated in Table 7. Compared to the previous best method, AutoGen with GPT-4, which incurs a cost of \$19.34, DatawiseAgent outperforms it with a cost of only \$2.13, achieving superior performance. Additionally, when DatawiseAgent with GPT-4o achieves the best performance of 53.18 in RPG, it incurs a cost of \$18.49. This demonstrates that DatawiseAgent achieves strong performance more cost-effectively, delivering impressive results without incurring the high costs associated with previous approaches.

In addition to DSBench, we further evaluate the total cost on InfiAgent-DABench, which consists of 257 decision-making tasks. As shown in Table 6, DatawiseAgent consistently demonstrates high cost-efficiency across different model settings.

Under the GPT-4o configuration, DatawiseAgent achieves an ABQ score of 85.99%, matching the best-performing baseline, but at a significantly lower cost (\$10.60 vs. \$16.19). Similarly, under the GPT-4o mini setting, DatawiseAgent achieves the highest ABQ score of 82.88%, while incurring only \$1.14 in total cost—substantially cheaper than TaskWeaver (\$1.64) and considerably more accurate than AutoGen.

These results highlight that DatawiseAgent not only performs competitively or better in terms of quality, but also offers a favorable cost-performance trade-off, especially in scenarios requiring high scalability or low-latency inference.

B Case Study on Predictive Modeling

We present a case example corresponding to the data modeling task with index 48 from DSBench (Jing et al., 2024). The task's instruction is shown in Figure 8, and the final agent trajectory of DatawiseAgent with GPT-4o is illustrated in Figure 7. As demonstrated by this example, DatawiseAgent utilizes DFS-like planning and incremental execution to dynamically decompose and execute the task. In the process, the framework performed multiple rounds of interactive data exploration, dataset partitioning, model design, training, and prediction. During execution, several code errors occurred; these were resolved through code repair module which is implemented by transitions between self-debugging and post-filtering, with the framework effectively consolidating past mistakes into the final context history. This example highlights the efficacy and flexibility of the FST-based multi-stage architecture in unified interaction representation, which leverages the reasoning and coding capabilities of large language models alongside dynamic environmental interactions to accomplish complex and multifaceted data science tasks.

C Details of DFST-based Multi-Stage Architecture

C.1 State Transition of DFST and Action Signal Space

Built on a finite state transducer (FST), DatawiseAgent orchestrates four distinct stages—DFS-like planning, incremental execution, self-debugging, and post-filtering. At each stage, DatawiseAgent samples an action signal from the predefined action signal space (see Table 8), which participates in driving the state transition while triggering the generation and execution of the corresponding markdown and code cells. We model this multi-stage architecture as a deterministic FST as illustrated in Figure 6. In the event of an execution error during either the DFS-like planning or incremental execution stage, DatawiseAgent transitions to the self-debugging and post-filtering stage for code repair. After post-filtering, the flow returns to the subsequent stage that would normally follow if no error had occurred.

C.2 Prompts for Each Stage

To give readers a clearer understanding of the agent's behavior at each stage, we detail the prompts used by the agent to generate actions in different states in Figs. 10 to 14.

Framework	Model	Cost/$	Inference Time/s	Task Success/% ↑	RPG ↑
AutoGen	Llama3-8b	-	50.9	5.41	1.55
	Llama3-70b	-	158.4	16.22	7.79
	GPT-4	19.34	77.4	87.84	45.52
	GPT-4o	12.27	104.1	71.62	34.74
	GPT-4o mini	0.10	26.7	22.97	11.24
Code Interpreter	GPT-4	38.81	237.6	54.05	26.14
	GPT-4o	19.26	268.6	44.59	19.87
	GPT-4o mini	2.70	199.6	39.19	16.90
Human*	Human*	-	-	100.00	65.02
DatawiseAgent (Ours)	GPT-4o	18.49	123.86	**98.64**	**53.18**
	GPT-4o mini	2.13	291.57	**98.64**	<u>46.61</u>
	Qwen2.5-72B-Instruct	-	760.25	<u>91.89</u>	42.9

Table 7: **Performance comparison on 74 Data Modeling tasks from DSBench.** We report **Cost**, **Inference Time**, **Task Success**, and **Relative Performance Gap (RPG)**, with best and second-best results in bold and underline, respectively. *Human performance is from (Jing et al., 2024), based on evaluations across 22 competitions.

Stage	Action Signal Space
DFS-like Planning	{<Advance_to_Next_Step>, <Iterate_on_the_Current_Step>, <Fulfil_Instruction>}
Incremental Execution	{<Await>, <End_Step>}
Self-debugging	{<Await>, <End_Debug>}
Post-filtering	{<Debug_Failure>, <Debug_Success>}

Table 8: Action signal space for each stage of the DatawiseAgent framework. At every stage, DatawiseAgent generates an action and selects a corresponding signal from the defined signal space.

C.3 Implementation Details

To prevent the FST from entering an infinite loop, we count and limit the number of transitions across three stages: DFS-like planning, incremental execution, and self-debugging. We introduce the following hyperparameters: (1) `max_planning_number`: the maximum number of transitions into the DFS-like Planning stage; (2) `max_execution_number`: the maximum number of transitions into the Incremental Execution stage for a given subtask; (3) `max_debug_number`: the maximum number of consecutive transitions into the Self-Debugging stage during code repair, representing the upper bound on debugging attempts for a single error; and (4) `max_planning_execution_number`: the maximum number of non-root nodes in the agent trajectory tree, where actions from both the DFS-like Planning and Incremental Execution phases are considered as nodes. Notably, `max_planning_execution_number` serves to constrain the overall search cost of the solution space. Those hyperparameters are uniformly configured in our experiments.

D Experiments Details

D.1 Datasets

InfiAgent-DABench. We use InfiAgent-DABench (Hu et al., 2024b), a benchmark specifically designed to evaluate agent performance on data analysis tasks. It comprises 257 real-world challenges, each accompanied by a CSV input file and one or more questions related to the data. The challenges span various categories such as summary statistics, feature engineering, and correlation analysis, and are labeled with one of three difficulty levels: easy, medium, or hard.

MatplotBench. We adopt MatplotBench (Yang et al., 2024b), a benchmark for the automatic and quantitative evaluation of AI methods in scientific data visualization. It contains 100 curated test cases, each consisting of a user query, an associated input dataset, and a ground-truth figure verified by human experts. The benchmark enables rigorous assessment of plotting accuracy and visual reasoning capabilities.

DSBench. We further utilize the data modeling part from DSBench (Jing et al., 2024), designed to assess agents on complex, real-world data science problems. DSBench includes 74 predictive modeling tasks derived from competitive platforms such

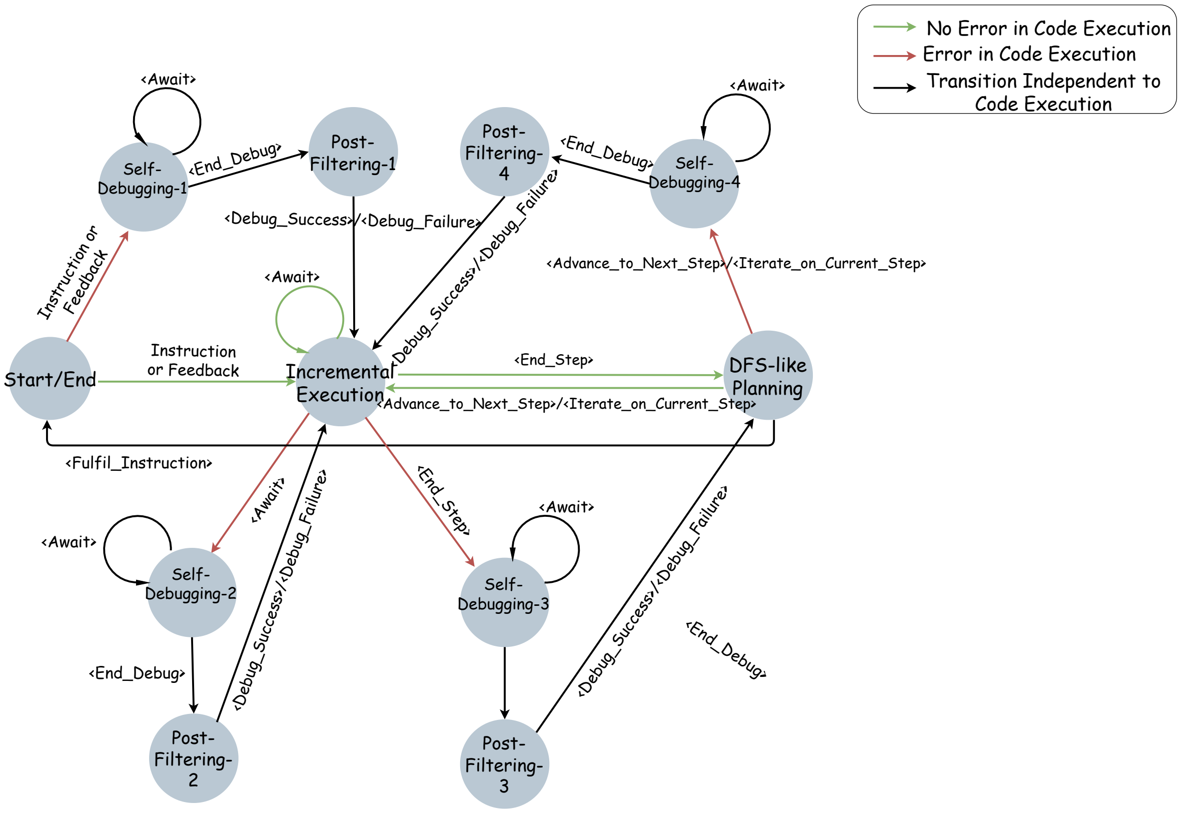


Figure 6: State transition diagram in the FST-based multi-stage design of DatawiseAgent, represented as a Non-deterministic Finite State Transducer (NFST). State transitions are driven by instructions or user feedback, action signals from the agent, and code execution feedback from the environment. Before each state transition, the agent generates and executes actions based on the current state.

as ModelOff[4] and Kaggle[5]. Each task provides a large-scale training/test dataset, sample submission file, and a detailed problem description, requiring agents to build end-to-end modeling solutions.

D.2 Metrics

InfiAgent-DABench. InfiAgent-DABench (Hu et al., 2024b) comprises 258 challenges, each paired with a corresponding CSV input file. These challenges are categorized into three difficulty levels (easy, medium, or hard) and include one or more questions about the data. For close-form questions, we define the following metrics:

- **Proportional Accuracy by Subquestions (PASQ)**:

$$\text{PASQ} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{M_i}\sum_{j=1}^{M_i} I_{ij}\right) \quad (1)$$

Here, N denotes the total number of questions, M_i is the number of subquestions in the i-th question, and I_{ij} is the indicator function for the j-th subquestion of the i-th question.

- **Accuracy by Questions (ABQ)**

$$\text{ABQ} = \frac{1}{N}\sum_{i=1}^{N}\left(\prod_{j=1}^{M_i} I_{ij}\right) \quad (2)$$

The product $\prod_{j=1}^{M_i} I_{ij}$ equals 1 if all subquestions of the i-th question are answered correctly, and 0 otherwise.

- **Uniform Accuracy by Subquestions (UASQ)**

$$\text{UASQ} = \frac{1}{\sum_{i=1}^{N} M_i}\sum_{i=1}^{N}\sum_{j=1}^{M_i} I_{ij} \quad (3)$$

DSBench. The 74 data modeling tasks from DSBench (Jing et al., 2024) are sourced from real-world Kaggle competitions and feature large-scale

---

[4] https://corporatefinanceinstitute.com/resources/financial-modeling/modeloff-guide/

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

training and testing datasets along with complex instructions, making them particularly challenging for data science agents. For evaluation, DSBench first adopts **Task Success Rate**, which measures whether the agent successfully builds a machine learning model and generates a bug-free submission. However, due to the inconsistency of metric scales and evaluation dimensions across different tasks, directly comparing performance is non-trivial. To address this, DSBench introduces the **Relative Performance Gap (RPG)** as an additional metric to normalize results across diverse tasks. RPG measures the agent's relative improvement over a baseline, scaled by the gap between the baseline and the best-known performance, and is defined as:

$$\text{RPG} = \frac{1}{N}\sum_{i=1}^{N} \max\left(\frac{p_i - b_i}{g_i - b_i}, 0\right) \tag{4}$$

where N is the total number of competitions, p_i is the performance of the agent's submission for the i-th competition, g_i is the highest known performance for the i-th competition, and b_i is the performance of a baseline. DSBench (Jing et al., 2024) uses the performance of the original submission file in the competition as the baseline performance in the RPG computation process.

D.3 DatawiseAgent and Baselines Configurations

DatawiseAgent Configuration. For the experiments on **data analysis** and **scientific visualization**, we set `max_planning_number` = 7, `max_execution_number` = 6, and `max_debug_number` = 8. For **predictive modeling**, we configured the hyperparameters as `max_planning_number` = 7, `max_execution_number` = 6, `max_debug_number` = 8, and `max_planning_execution_number` = 15. These predefined hyperparameters act as guardrails to ensure robust performance and maintain a consistent experimental environment. They do not constrain the generality of our approach but rather provide necessary safeguards against unexpected failures.

To investigate the degree to which DatawiseAgent adheres to the designed state machine during the experiments, we recorded the average number of LLM calls made by DatawiseAgent with GPT-4o. The results, as illustrated in Table 9, indicate that DatawiseAgent, through its FST-based multi-stage architecture, effectively orchestrates the transitions among the four key stages.

Avg. LLM calls	Benchmark
6.42	InfiAgent-DABench
6.41	MatplotBench
7.56	MatplotBench(**w/ visual tool**)
12.31	Data Modeling

Table 9: Average number of LLM calls across benchmarks made by DatawiseAgent using GPT-4o.

Details of Experimental Setups. (1) **Data Analysis.** We benchmark DatawiseAgent in InfiAgent-DABench against several state-of-the-art agent systems (SoTA), including ReAct(Hu et al., 2024b), AutoGen(Wu et al., 2023), Taskweaver(Qiao et al., 2023) and Data Interpreter(Hong et al., 2024). For model configuration, we set the temperature to 0 for all agents, except for ReAct, where the temperature is set to 0.2, as required by Hu et al., 2024b.

(2) **Scientific Visualization.** We benchmark DatawiseAgent against three baselines in different model settings: Direct Decoding, MatplotAgent, and AutoGen. MatplotBench employs a vision-based scoring mechanism aligned with human assessment (Yang et al., 2024b), where an advanced multi-modal LLM, such as GPT-4V(Achiam et al., 2023), is prompted to score the generated figure on a scale from 0 to 100, comparing it with the ground truth figure. Since OpenAI deprecated GPT-4V during our experiments, we adopt GPT-4o, a more powerful version with enhanced vision capabilities, as the recommended replacement by OpenAI(Hurst et al., 2024), to serve as the scoring model. The temperature is set to 0 in all methods.

(3) **Predictive Modeling.** We evaluate DatawiseAgent using the experimental setup described in DSBench (Jing et al., 2024) and compare its performance with the results reported for AutoGen (Wu et al., 2023) and Code Interpreter[6]. The primary metrics include **Task Success Rate**, which measures whether the data science agent successfully completes the predictive task, and the **Relative Performance Gap (RPG)**, which quantifies the overall performance of a data science agent across different competitions. We also record **Inference Time**, the average time taken to complete a task. Each task is assigned a maximum time limit of 3600 seconds, as some competitions involve

---

[6] https://platform.openai.com/docs/assistants/tools/code-interpreter

Model	Framework	PASQ/% ↑	ABQ/% ↑	UASQ/% ↑
GPT-4o mini	ReAct	85.30	80.08	84.55
	AutoGen	74.68	70.04	77.41
	Taskweaver	81.95	76.65	81.34
	Data Interpreter	73.85	67.7	72.15
	DatawiseAgent (Ours)	**88.39**	**82.88**	**87.06**
GPT-4o	ReAct	87.48	81.32	86.62
	AutoGen	76.43	73.54	79.39
	Taskweaver	89.35	85.99	**90.24**
	Data Interpreter*	-	**94.93**	-
	Data Interpreter	79.97	75.78	79.59
	DatawiseAgent (Ours)	**89.95**	85.99	89.91
Qwen2.5-72B-Instruct	ReAct	82.39	75.88	78.73
	AutoGen	73.87	70.04	75.22
	DatawiseAgent (Ours)	**87.27**	**81.71**	**85.09**

Table 10: **Performance comparison on InfiAgent-DABench across various model settings.** The result marked with an asterisk (*) is reported by Hong et al. (2024). Best results are in bold; second-best are underlined.

Model	Framework	Comp. Rate/%	Scores $\geq$ 80/%	Avg. Score↑	Δ Avg. Score
GPT-4o mini	Direct Decoding	61	24	38.09	-
	MatplotAgent	94	33	51.44	+13.35
	AutoGen	92	32	51.82	+13.73
	w/ visual tool	90	32	52.07	+13.98
	DatawiseAgent (Ours)	**99**	34	55.85	+17.76
	w/ visual tool	**99**	**39**	**58.60**	**+20.51**
GPT-4o	Direct Decoding	68	32	45.28	-
	MatplotAgent	95	41	57.86	+12.58
	AutoGen	97	39	60.42	+15.14
	w/ visual tool	99	36	63.60	+18.32
	DatawiseAgent (Ours)	**100**	43	61.22	+15.94
	w/ visual tool	99	**44**	**64.33**	**+19.05**
Qwen2.5-72B -Instruct	Direct Decoding	73	35	47.54	-
	AutoGen	65	26	40.80	-6.74
	w/ visual tool	85	32	53.72	+6.18
	DatawiseAgent (Ours)	98	37	56.41	+8.87
	w/ visual tool	**99**	**42**	**61.88**	**+14.34**

Table 11: **Performance comparison on MatplotBench.** We report three metrics: **Completion Rate** (valid output rate), **Scores $\geq$ 80** (proportion of high-quality completions), and **Average Score** (0–100). The last column (Δ Avg. Score) denotes the score gain over Direct Decoding. Bold and underline highlight the best and second-best results, respectively. Visual tool rows indicate integration with the GPT-4o mini-based visual tool.

large datasets or complex tasks that could require extensive computation time. If the task exceeds this limit, it is marked as incomplete, and the time is recorded as 3600 seconds. The temperature of DatawiseAgent is set to 0.

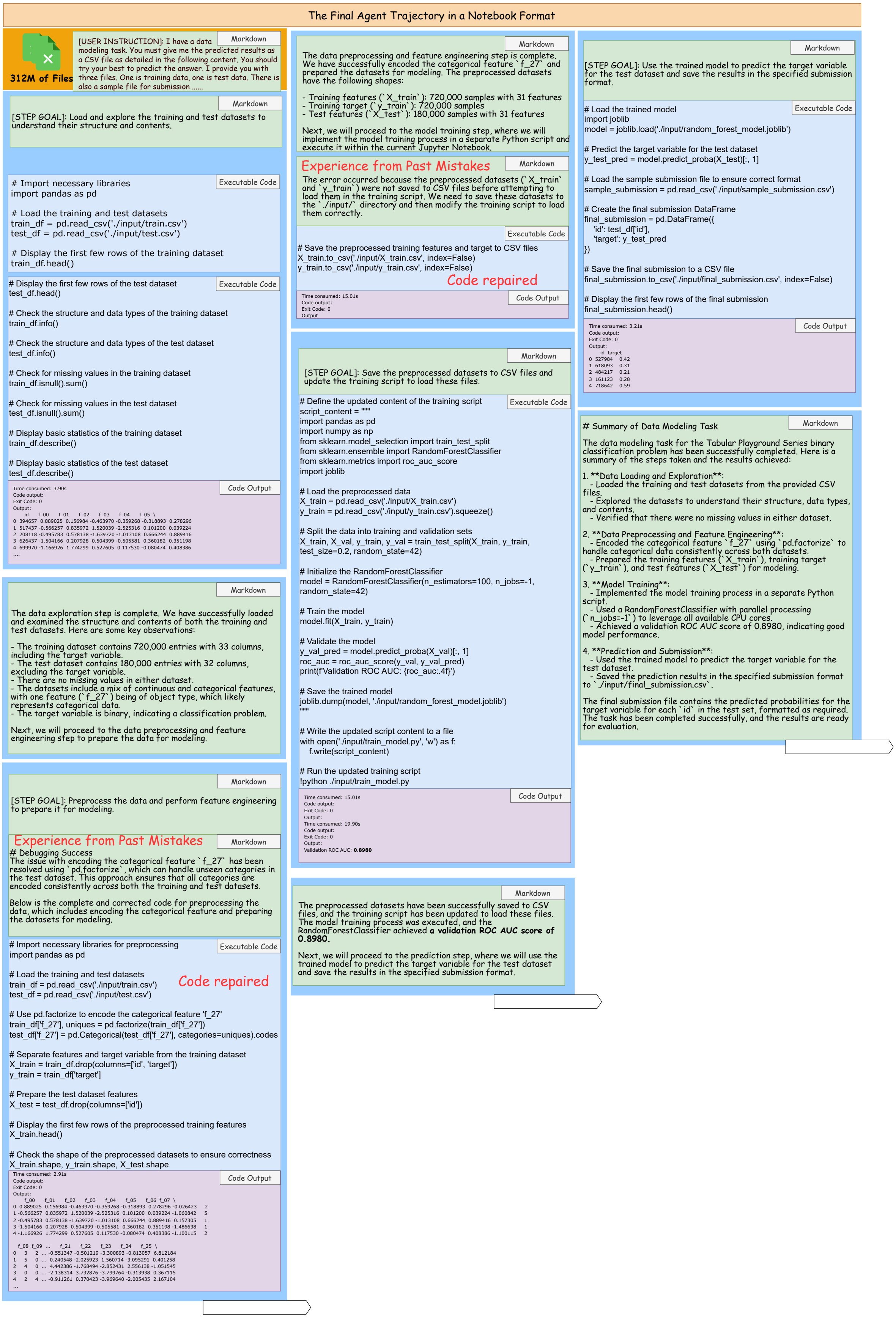
In Jing et al. 2024, detailed specifications of the experimental environment are not provided, and it is challenging to control for resources and environmental factors across different methods. Moreover, since inference time is affected by numerous factors, comparing the inference time of DatawiseAgent with that of AutoGen and Code Interpreter may not yield meaningful insights. Nevertheless, for the experiments of DatawiseAgent on the data modeling tasks from DSBench, we conducted all evaluations under a consistent environment to ensure fairness and reproducibility. Specifically, the experiments were run on a machine with 80 CPU cores, 512 GB of RAM. The operating system is Ubuntu 24.04.1 LTS, and the software environment is managed via Conda with Python 3.10. Core libraries for predictive modeling, such as NumPy (v2.2.1), Pandas (v2.2.3), Matplotlib (v3.10.0), SciPy (v1.15.1), Scikit-learn (v1.6.1), and PyTorch (v2.5.1+cu121), were preinstalled to support the experiments. Additionally, DatawiseAgent is capable of dynamically installing required packages during task execution by executing command-line installation commands within code cells.

D.4 Visual Tool for Scientific Visualization

We implement a visual tool based on GPT-4o mini to evaluate DatawiseAgent 's capability in completing scientific visualization tasks through the integration of visual feedback tools. In our experiments, each test case can call the visual tool at most **four** times for both AutoGen and DatawiseAgent. Figure 9 illustrates the details of the implementation and integration of this tool.

D.5 Full Experimental Results on Data Analysis, Scientific Visualization, and Predictive Modeling

In addition to the main results reported in Section 4, we present the complete experimental results for **Data Analysis**, **Scientific Visualization**, and **Predictive Modeling** in Tables Table 10, Table 11, and Table 7, respectively. These tables include additional evaluation metrics (with definitions provided in Section D.2), offering a more comprehensive assessment of DatawiseAgent 's performance across different tasks. As shown in Table 11, DatawiseAgent achieves substantially higher scores than all baselines in both *Completion Rate* and the *Proportion of Scores* ≥ 80. These results highlight DatawiseAgent 's robustness in task completion as well as its ability to generate consistently high-quality visual outputs.



The Final Agent Trajectory in a Notebook Format

[USER INSTRUCTION]: I have a data modeling task. You must give me the predicted results as a CSV file as detailed in the following content. You should try your best to predict the answer. I provide you with three files. One is training data, one is test data. There is also a sample file for submission ......

[STEP GOAL]: Load and explore the training and test datasets to understand their structure and contents.

```
# Import necessary libraries
import pandas as pd

# Load the training and test datasets
train_df = pd.read_csv('./input/train.csv')
test_df = pd.read_csv('./input/test.csv')

# Display the first few rows of the training dataset
train_df.head()
```

```
# Display the first few rows of the test dataset
test_df.head()

# Check the structure and data types of the training dataset
train_df.info()

# Check the structure and data types of the test dataset
test_df.info()

# Check for missing values in the training dataset
train_df.isnull().sum()

# Check for missing values in the test dataset
test_df.isnull().sum()

# Display basic statistics of the training dataset
train_df.describe()

# Display basic statistics of the test dataset
test_df.describe()
```

```
Time consumed: 3.90s
Code output:
Exit Code: 0
Output:
       id      f_00      f_01      f_02      f_03      f_04      f_05  \
0  394657  0.889025  0.156984 -0.463970 -0.359268 -0.318893  0.278296
1  517437 -0.566257  0.835972  1.520039 -2.525316  0.101200  0.039224
2  208118 -0.495783  0.578138 -1.639720 -1.013108  0.666244  0.889416
3  626437 -1.504166  0.207928  0.504399 -0.505581  0.360182  0.351198
4  699970 -1.166926  1.774299  0.527605  0.117530 -0.080474  0.408386
....
```

The data exploration step is complete. We have successfully loaded and examined the structure and contents of both the training and test datasets. Here are some key observations:

- The training dataset contains 720,000 entries with 33 columns, including the target variable.
- The test dataset contains 180,000 entries with 32 columns, excluding the target variable.
- There are no missing values in either dataset.
- The datasets include a mix of continuous and categorical features, with one feature (`f_27`) being of object type, which likely represents categorical data.
- The target variable is binary, indicating a classification problem.

Next, we will proceed to the data preprocessing and feature engineering step to prepare the data for modeling.

[STEP GOAL]: Preprocess the data and perform feature engineering to prepare it for modeling.

Experience from Past Mistakes

# Debugging Success
The issue with encoding the categorical feature `f_27` has been resolved using `pd.factorize`, which can handle unseen categories in the test dataset. This approach ensures that all categories are encoded consistently across both the training and test datasets.

Below is the complete and corrected code for preprocessing the data, which includes encoding the categorical feature and preparing the datasets for modeling.

Code repaired

```
# Import necessary libraries for preprocessing
import pandas as pd

# Load the training and test datasets
train_df = pd.read_csv('./input/train.csv')
test_df = pd.read_csv('./input/test.csv')

# Use pd.factorize to encode the categorical feature 'f_27'
train_df['f_27'], uniques = pd.factorize(train_df['f_27'])
test_df['f_27'] = pd.Categorical(test_df['f_27'], categories=uniques).codes

# Separate features and target variable from the training dataset
X_train = train_df.drop(columns=['id', 'target'])
y_train = train_df['target']

# Prepare the test dataset features
X_test = test_df.drop(columns=['id'])

# Display the first few rows of the preprocessed training features
X_train.head()

# Check the shape of the preprocessed datasets to ensure correctness
X_train.shape, y_train.shape, X_test.shape
```

```
Time consumed: 2.91s
Code output:
Exit Code: 0
Output:
      f_00      f_01      f_02      f_03      f_04      f_05      f_06  f_07  \
0  0.889025  0.156984 -0.463970 -0.359268 -0.318893  0.278296 -0.026423     2
1 -0.566257  0.835972  1.520039 -2.525316  0.101200  0.039224 -1.060842     5
2 -0.495783  0.578138 -1.639720 -1.013108  0.666244  0.889416  0.157305     1
3 -1.504166  0.207928  0.504399 -0.505581  0.360182  0.351198 -1.486638     1
4 -1.166926  1.774299  0.527605  0.117530 -0.080474  0.408386 -1.100115     2

   f_08  f_09  ...      f_21      f_22      f_23      f_24      f_25  \
0     3     2  ... -0.551347 -0.501219 -3.300893 -0.813057  6.812184
1     5     0  ...  0.240548 -2.025923  1.560714 -3.095291  0.401258
2     4     0  ...  4.442386 -1.768494 -2.852431  2.556138 -1.051545
3     0     0  ... -2.138314  3.732876 -3.799764 -0.313938  0.367115
4     2     4  ... -0.911261  0.370423 -3.969640 -2.005435  2.167104
...
```

The data preprocessing and feature engineering step is complete. We have successfully encoded the categorical feature `f_27` and prepared the datasets for modeling. The preprocessed datasets have the following shapes:

- Training features (`X_train`): 720,000 samples with 31 features
- Training target (`y_train`): 720,000 samples
- Test features (`X_test`): 180,000 samples with 31 features

Next, we will proceed to the model training step, where we will implement the model training process in a separate Python script and execute it within the current Jupyter Notebook.

Experience from Past Mistakes

The error occurred because the preprocessed datasets (`X_train` and `y_train`) were not saved to CSV files before attempting to load them in the training script. We need to save these datasets to the `./input/` directory and then modify the training script to load them correctly.

```
# Save the preprocessed training features and target to CSV files
X_train.to_csv('./input/X_train.csv', index=False)
y_train.to_csv('./input/y_train.csv', index=False)
```

Code repaired

```
Time consumed: 15.01s
Code output:
Exit Code: 0
Output
```

[STEP GOAL]: Save the preprocessed datasets to CSV files and update the training script to load these files.

```
# Define the updated content of the training script
script_content = """
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
import joblib

# Load the preprocessed data
X_train = pd.read_csv('./input/X_train.csv')
y_train = pd.read_csv('./input/y_train.csv').squeeze()

# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)

# Initialize the RandomForestClassifier
model = RandomForestClassifier(n_estimators=100, n_jobs=-1,
random_state=42)

# Train the model
model.fit(X_train, y_train)

# Validate the model
y_val_pred = model.predict_proba(X_val)[:, 1]
roc_auc = roc_auc_score(y_val, y_val_pred)
print(f'Validation ROC AUC: {roc_auc:.4f}')

# Save the trained model
joblib.dump(model, './input/random_forest_model.joblib')
"""

# Write the updated script content to a file
with open('./input/train_model.py', 'w') as f:
    f.write(script_content)

# Run the updated training script
!python ./input/train_model.py
```

```
Time consumed: 15.01s
Code output:
Exit Code: 0
Output:
Time consumed: 19.90s
Code output:
Exit Code: 0
Output:
Validation ROC AUC: 0.8980
```

The preprocessed datasets have been successfully saved to CSV files, and the training script has been updated to load these files. The model training process was executed, and the RandomForestClassifier achieved **a validation ROC AUC score of 0.8980.**

Next, we will proceed to the prediction step, where we will use the trained model to predict the target variable for the test dataset and save the results in the specified submission format.

[STEP GOAL]: Use the trained model to predict the target variable for the test dataset and save the results in the specified submission format.

```
# Load the trained model
import joblib
model = joblib.load('./input/random_forest_model.joblib')

# Predict the target variable for the test dataset
y_test_pred = model.predict_proba(X_test)[:, 1]

# Load the sample submission file to ensure correct format
sample_submission = pd.read_csv('./input/sample_submission.csv')

# Create the final submission DataFrame
final_submission = pd.DataFrame({
    'id': test_df['id'],
    'target': y_test_pred
})

# Save the final submission to a CSV file
final_submission.to_csv('./input/final_submission.csv', index=False)

# Display the first few rows of the final submission
final_submission.head()
```

```
Time consumed: 3.21s
Code output:
Exit Code: 0
Output:
       id  target
0  527984    0.42
1  618093    0.31
2  484217    0.21
3  161123    0.28
4  718642    0.59
```

# Summary of Data Modeling Task

The data modeling task for the Tabular Playground Series binary classification problem has been successfully completed. Here is a summary of the steps taken and the results achieved:

1. **Data Loading and Exploration**:
   - Loaded the training and test datasets from the provided CSV files.
   - Explored the datasets to understand their structure, data types, and contents.
   - Verified that there were no missing values in either dataset.

2. **Data Preprocessing and Feature Engineering**:
   - Encoded the categorical feature `f_27` using `pd.factorize` to handle categorical data consistently across both datasets.
   - Prepared the training features (`X_train`), training target (`y_train`), and test features (`X_test`) for modeling.

3. **Model Training**:
   - Implemented the model training process in a separate Python script.
   - Used a RandomForestClassifier with parallel processing (`n_jobs=-1`) to leverage all available CPU cores.
   - Achieved a validation ROC AUC score of 0.8980, indicating good model performance.

4. **Prediction and Submission**:
   - Used the trained model to predict the target variable for the test dataset.
   - Saved the prediction results in the specified submission format to `./input/final_submission.csv`.

The final submission file contains the predicted probabilities for the target variable for each `id` in the test set, formatted as required. The task has been completed successfully, and the results are ready for evaluation.

Figure 7: The case example of DatawiseAgent for the data modeling task with index = 48

PLAIN TEXT

~~~~
I have a data modeling task. You must give me the predicted results as a CSV file as detailed in
     the following content. You should try your best to
predict the answer. I provide you with three files. One is training data, one is test data. There
     is also a sample file for submission.
Description
The May edition of the 2022 Tabular Playground series binary classification problem includes a
     number of different feature interactions. This
competition is an opportunity to explore various methods for identifying and exploiting these
     feature interactions.
About the Tabular Playground Series
Kaggle competitions are incredibly fun and rewarding, but they can also be intimidating for people
     who are relatively new to their data science journey.
In the past, we've launched many Playground competitions that are more approachable than our
     Featured competitions and thus, more beginner-
friendly.
The goal of these competitions is to provide a fun and approachable-for-anyone tabular dataset to
     model. These competitions are a great choice for
people looking for something in between the Titanic Getting Started competition and the Featured
     competitions. If you're an established competitions
master or grandmaster, these probably won't be much of a challenge for you; thus, we encourage you
     to avoid saturating the leaderboard.
For each monthly competition, we'll be offering Kaggle Merchandise for the top three teams. And
     finally, because we want these competitions to be
more about learning, we're limiting team sizes to 3 individuals.
Getting Started
For ideas on how to improve your score, check out the Intro to Machine Learning and Intermediate
     Machine Learning courses on Kaggle Learn.
We've also built a starter notebook for you that uses TensorFlow Decision Forests, a TensorFlow
     library that matches the power of XGBoost with a
friendly, straightforward user interface.
Good luck and have fun!
Acknowledgments
Photo by Clarisse Croset on Unsplash.
Evaluation
Submissions are evaluated on the area under the ROC curve between the predicted probability and the
      observed target.
Submission File
For each id in the test set, you must predict a probability for the target variable. The file
     should contain a header and have the following format:
```
id, target
900000, 0.65
900001, 0.97
900002, 0.02
etc.
```
Dataset Description
For this challenge, you are given (simulated) manufacturing control data and are tasked to predict
     whether the machine is in state 0 or state 1. The
data has various feature interactions that may be important in determining the machine state. Good
     luck!
Files
- train.csv: the training data, which includes normalized continuous data and categorical data
- test.csv: the test set; your task is to predict the binary target variable which represents the
     state of a manufacturing process
- sample_submission.csv: a sample submission file in the correct format
All three data files can be found in the folder `./input/`. **You should use sklearn or pytorch to
     complete the task.** Any training scripts, models, and
experiment log should be saved in `./input/`.
After data modeling, provide the prediction results for the test file in the format specified by
     the sample submission file. Save the final submission to
`./input/final_submission.csv`.
~~~~

Figure 8: The complete instruction of the data modeling task with index = 48.

PSEUDOCODE

```
GLOBAL_CNT <- 4
EVALUATION_CNT <- 0

function evaluate_image(image_path, requirements, query):
    if EVALUATION_CNT >= GLOBAL_CNT:
        return "Usage limit reached. Please manually evaluate."

    if image_path is invalid or does not exist:
        raise error

    if requirements or query is empty:
        raise error

    encoded_image <- encode_image_to_base64(image_path)

    prompt <- "Expected Requirements:\n" + requirements
    prompt += "\nQuery:\n" + query
    prompt += "\nYour response:\n"

    message <- [
        {"type": "text", "text": prompt},
        {"type": "image_url", "image_url": {"url": encoded_image}}
    ]

    try:
        response <- call_chat_completion(model="gpt-4o-mini", message)
        EVALUATION_CNT += 1
        return response.content
    except:
        raise runtime_error
```

Figure 9: Pseudocode of the GPT-4o mini-based visual tool. This tool generates a textual response to a given query by analyzing the provided image in light of the specified requirements.

PROMPT

```
The current [USER INSTRUCTION]:
{{the description of user instruction}}

The current [STEP GOAL]:
[STEP GOAL]: {{the description of current step}}

The current STEP has been finished as above. Currently in the Planning Stage
    .
Available Action Space: {<Iterate on Current STEP>, <Advance to Next STEP>,
    <Fulfill USER INSTRUCTION>}

Your response MUST start with **exactly one** of the action signals, and
    then generate the corresponding action:

1. ‘<Iterate on Current STEP>‘:
    When to choose: Select this if the current STEP was incorrect or
        requires replacement.
    Action content: Write observations from the wrong current STEP when
        needed. Then reinitiate a NEW AND DISTINCT [STEP GOAL] and write
        cells incrementally for kernel execution to REPLACE the current one.
    Response Format:
    <Iterate on Current STEP>
    ‘‘‘markdown
    (observations in detailed from the replaced STEP when needed)
    ‘‘‘
    ‘‘‘markdown
    [STEP GOAL]: (the description of [STEP GOAL])
    ‘‘‘
    ‘‘‘markdown/python
    # several markdown and code cells
    ‘‘‘

2. ‘<Advance to Next STEP>‘:
    When to choose: Select this when the current STEP was successful and
        provides a correct foundation for further progress.
    Action content: Define the next [STEP GOAL] to **progress towards
        fulfilling the [USER INSTRUCTION]**. Write cells to implement the
        next step incrementally.

    One [STEP GOAL] could be initiated in one markdown cell labeled with ‘[
        STEP GOAL]: ‘.

3. ‘<Fulfill USER INSTRUCTION>‘:
    When to choose: Select this if the current [USER INSTRUCTION] has been
        fully satisfied, and no further STEPs are necessary.
    Action content: Conclude the process by providing a thorough and
        structured summary that encapsulates all key aspects of the completed
         [USER INSTRUCTION]. The summary should be clear, concise, and
        organized to ensure the user fully understands the results and
        implications of the task.

Selecting <Iterate on Current STEP> or <Advance to Next STEP> transitions
    the workflow to the Incremental Execution Stage to implement the new or
    next STEP, while selecting <Fulfill USER INSTRUCTION> concludes the
    workflow and no further stages are required.

Specifically, the response format is below:

<Iterate on Current STEP>/<Advance to Next STEP>/<Fulfill USER INSTRUCTION>
‘‘‘markdown/python
# several markdown and code cells
‘‘‘

Your response:
```

Figure 10: The prompt of the DFS-like planning stage.

PROMPT

```
The current [STEP GOAL]:
[STEP GOAL]: {{the description of current step}}
Currently in the Incremental Execution Stage.
Available Action Space: {<await>, <end_step>}
Your response MUST start with **exactly one** of the action
   signals and then generate the corresponding action:
1. ‘<await>‘:
When to choose: Select this when you need code to be executed
   in the Jupyter kernel.
Action content: Writing several markdown and code cells.
2. ‘<end_step>‘:
When to choose: Select this when the current [STEP GOAL] has
   been fully completed.
Action content: Indicate that the step is finished and write
   any final cells needed to finalize the STEP.
Selecting ‘<end_step>‘ transitions the workflow to the **
   Planning Stage** to evaluate the next step or finalize the
   response to the ‘[USER
INSTRUCTION]‘. Otherwise, stay in the **Incremental Execution
   Stage** to continue working on the current STEP.
Specifically, the response format is below:
<await>/<end_step>
‘‘‘markdown/python
# several markdown and code cells
‘‘‘
Your response:
```

Figure 11: The prompt in the incremental execution stage.

PROMPT

```
The current [STEP GOAL]:
[STEP GOAL]: {{the description of current step}}

Currently in the Debugging Stage.
Available Action Space: {<await>, <end_debug>}

Your response MUST start with **exactly one** of the action
   signals, and then generate the corresponding action:

1. ‘<await>‘:
    When to choose: Select this when you need code cells to be
        executed in the Jupyter kernel to diagnose issues,
       test fixes, or inspect intermediate results.
    Action content: Writing several markdown and code cells
       containing your debugging attempts, such as code
       modifications, print statements, or variable
       inspections.
2. ‘<end_debug>‘:
    When to choose: Select this when **all bugs are fixed**
       and you are ready to exit the debugging phase and
       advance towards the ‘[STEP GOAL]‘.
    Action content: Indicate that debugging is complete and
       write any final cells needed.

Selecting ‘<end_debug>‘ transitions the workflow to the **Post
   -Debugging Stage** to evaluate and clean up the debugging
   artifacts. Otherwise, stay in the **Debugging Stage** to
   continue diagnosing and fixing issues.

Specifically, the response format is below:
<await>/<end_debug>
‘‘‘markdown/python
# several markdown and code cells
‘‘‘

Your response:
```

Figure 12: The prompt in the self-debugging stage.

PROMPT

```
Currently in the Post-Debugging Stage.
Available Action Space: {<debug_success>, <debug_failure>}

Your response MUST start with **exactly one** of the action
   signals, and then generate the corresponding action:

1. ‘<debug_success>‘:
    When to choose: Select this when the debugging phase
       succeeded in fixing all bugs.
    Action content:
        - Record any valuable information from the debugging
           process for future reference in a markdown cell.
        - Write one or more **fully cleaned and complete code
           cells** that **include all necessary steps** to
           replace the entire debugging process. The provided
           code must be **self-contained and ready for
           execution** without requiring any external context
           or prior cells in the debugging process.
2. ‘<debug_failure>‘:
    When to choose: Select this when the debugging phase
       failed to fix the bugs.
    Action content: Provide a credible **diagnostic report**
       based on the debugging process in a markdown cell,
       explaining what was attempted, why it failed, and any
       insights from the debugging trace. No code cells should
        be provided.

Specifically, the response format is below:

<debug_success>/<debug_failure>
‘‘‘markdown/python
# several markdown and code cells
‘‘‘

Your response:
```

Figure 13: The prompt in the post-filtering stage.

PROMPT

```
Let’s address the [USER INSTRUCTION] STEP by STEP. You should
   initiate ONE [STEP GOAL] first and write markdown and code
   cells incrementally for kernel execution.

Specifically, the response format is below:

‘‘‘markdown
[STEP GOAL]: (the description of [STEP GOAL])
‘‘‘
‘‘‘markdown/python
# several markdown and code cells
‘‘‘

Your response:
```

Figure 14: The prompt at the start state.