Tree-Notebook: A Context-Aware Agent with Tree Search and Entropy-Aware Data Shadow for Interactive Data Science

Junkun Qiu, Min Huang, Qinghai Miao*
School of Artificial Intelligence
University of Chinese Academy of Sciences
qiujunkun24@mails.ucas.ac.cn
{huangm, miaoqh}@ucas.ac.cn

Abstract

While LLM-based agents have emerged as a focal point for automating data science tasks, they continue to grapple with inefficient context management, "silent failures" (where code executes correctly but fails the task objectives), and error propagation inherent in sequential generation. In this paper, we propose **Tree-Notebook**, an agentic framework designed to mimic the iterative cognitive process of human data scientists. At its core, Tree-Notebook conceptualizes Jupyter Notebook cells as nodes within a tree structure, facilitating organized and efficient context retrieval. We formalize the task-solving process as a Partially Observable Markov Decision Process (POMDP) over a dynamic tree, utilizing an entropy-based information gain function for path evaluation to enhance adaptability in real-world environments. Furthermore, we introduce the "Data Shadow" system, which resolves silent failures by performing real-time tracking of data distributions, provenance, and semantic constraints. Experimental results demonstrate that Tree-Notebook achieves state-of-the-art (SOTA) performance on both InfiAgent-DABench and DS-Bench. To further evaluate robustness, we introduce an augmented version of InfiAgent-DABench to simulate complex environments, where Tree-Notebook consistently maintains its SOTA standing. Code is available at: https://github.com/QJK-BUAA/Tree-Notebook

1 Introduction

Data science is an intricate discipline centered on extracting actionable insights from data, encompassing a broad spectrum of tasks such as data preprocessing, aggregation, and modeling(Donoho, 2017; Zhang et al., 2024). Traditionally, Python-based programmatic manipulation has been the cornerstone of these workflows. However, driven by the rapid advancements in Large Language Models (LLMs), the field is undergoing a transition toward an agentic paradigm, where LLM-powered agents autonomously synthesize code to execute complex data science tasks(Hong et al., 2023; Wu et al., 2023; Xue et al., 2023; Dibia, 2023).

AutoKaggle and DS-Agent established the foundation for full-lifecycle automation through multi-agent collaboration and Case-Based Reasoning (CBR)(Li et al., 2024; Guo et al., 2024b). However, these systems often struggle with the non-linear exploration required in large search spaces. Subsequent works like Data Interpreter and Datawise Agent improve hierarchical planning but remain "data-decoupled," creating a gap between planning and actual data states(Hong et al., 2025; You et al., 2025). While JUPITER introduces inference-time search, it prioritizes code logic over the data-driven adjustments essential for complex DS pipelines(Li et al., 2025b). Synthesizing the above analysis and insights from recent surveys (Rahman et al., 2025; Zhang et al., 2025b), we argue that current linear frameworks struggle to align with the **exploratory and trial-and-error nature** of real-world data science. We identify **three critical misalignments:**

(1) **The "Silent Failure" in State Tracking.** Existing agents suffer from *state invisibility*. They rely on textual outputs (stdout) but lack perception of the underlying data distribution in the kernel. For instance, an operation like *df.dropna()* might silently discard 90% of the dataset. A linear agent, seeing no execution error, proceeds on compromised data, leading to "silent failures" that traditional text-based observations miss(Hong et al., 2025; Munkhdalai et al., 2024). (2) **Rigidity in Non-Linear Exploration.** Real-world tasks involve branching hypotheses (e.g., comparing imputation strategies). Linear frameworks force a single execution path. Once an agent commits to a suboptimal feature engineering path, it often lacks the mechanism to cleanly backtrack to a historical state without polluting the context window(Guo et al.,

*Corresponding author

Findings of the Association for Computational Linguistics: ACL 2026, pages 15957–15970

2024b; Hassan et al., 2023).(3) **Insufficiency of existing benchmarks.** Mainstream benchmarks are often too sanitized to expose these fragilities. In realistic workflows involving environmental noise (e.g., irrelevant files) or ambiguous schemas, linear agents prone to hallucination fail to distinguish signal from noise, rendering simple evaluations insufficient(Hu et al., 2024; Jing et al., 2024).

To address these limitations—specifically the risks of silent data corruption and the inability to recover from dead ends—we propose **Tree-Notebook**. This architecture formulates code generation as a Partially Observable Markov Decision Process (POMDP) over a tree structure(Janner et al., 2022; Weir et al., 2024). By treating Jupyter notebook cells as nodes, we replace error-prone linear planning with a non-linear dynamic search guided by entropy-regularized trajectory optimization(Wen et al., 2024; Vanlioglu, 2025).

Key contributions include: (1) *Entropy-Regularized POMDP Tree-Framework.* We propose Tree-Notebook, utilizing a dual-objective evaluation of task utility and subjective semantic entropy to effectively balance exploration efficiency with execution risk in uncertain environments. (2) *Data Shadow and Isomorphic Merging.* To resolve state invisibility, we introduce the **Data Shadow** to track semantic data evolution (e.g., detecting silent data loss). Coupled with isomorphic merging, this compresses the search space into a Directed Acyclic Graph (DAG) by identifying observationally equivalent states. (3) *Non-linear Backtracking and Enhanced Benchmarking.* We enable non-linear spatio-temporal backtracking using system checkpoints to perform state rollbacks, preventing error propagation. Furthermore, we establish a noise-augmented InfiAgent-DABench with hierarchical perturbations (e.g., irrelevant files, data noise)(Mao et al., 2025; Siqueira de Cerqueira et al., 2024) to rigorously evaluate agent robustness in complex, realistic scenarios.

In summary, we propose Tree-Notebook, a POMDP-based framework that structures notebook execution as an optimized tree search using Data Shadow, alongside an enhanced InfiAgent-DABench tailored to realistic scenarios. Our approach achieves SOTA performance on public benchmarks for data analysis and modeling, with robustness further validated via a noise-augmented dataset.

2 Related Work

2.1 LLM-Based Reasoning and Search

While Chain-of-Thought (CoT) enhances LLM reasoning, its reliance on linear greedy decoding limits its error correction(Wei et al., 2022). To address this, non-linear architectures have emerged: Tree of Thoughts (ToT) introduces tree-based search for backtracking and state evaluation(Yao et al., 2023), while Graph of Thoughts (GoT) enables complex topologies via node aggregation and looping(Besta et al., 2024). Building on these, RAP frames reasoning as planning, employing Monte Carlo Tree Search (MCTS) with a learned World Model to prune low-quality branches, achieving superior long-range reasoning compared to ToT(Hao et al., 2023).

Search strategies are equally critical in code generation. AlphaCode(Li et al., 2022) demonstrated the power of large-scale sampling, while Language Agent Tree Search (LATS) combines MCTS with compiler feedback to guide generation(Zhou et al., 2023). Similarly, Reflexion optimizes reasoning trajectories through linguistic reinforcement learning based on past errors(Shinn et al., 2023). However, these methods predominantly target closed domains (arithmetic, puzzles) with discrete, well-defined state spaces, lacking the capacity to perceive complex data states (e.g., variable distributions) inherent in data science tasks(Aouini and Loubani, 2025).

2.2 Evolution of Data Science Agents

The evolution of DS agents has moved from simple execution to complex orchestration. Early tools like PandasAI and ReAct-based agents(Yao et al., 2022) relied on linear "Think-Act-Observe" loops(Rahman et al., 2025). CodeAct later demonstrated that using executable code as a unified action space offers better generalization than JSON or text instructions(Guo et al., 2024a). However, these linear approaches often fail in long-horizon tasks due to a lack of global planning(Zhang et al., 2025a; Wang et al., 2025b).

To handle complex workflows, recent frameworks adopt multi-agent and hierarchical strategies. AutoGen facilitates role-based collaboration(Wu et al., 2024), while MetaGPT encodes Standard Operating Procedures (SOPs) into agent workflows to reduce hallucination(Hong et al., 2023). TaskWeaver(Qiao et al., 2023) and Data Interpreter(Hong et al., 2025) emphasize code-first

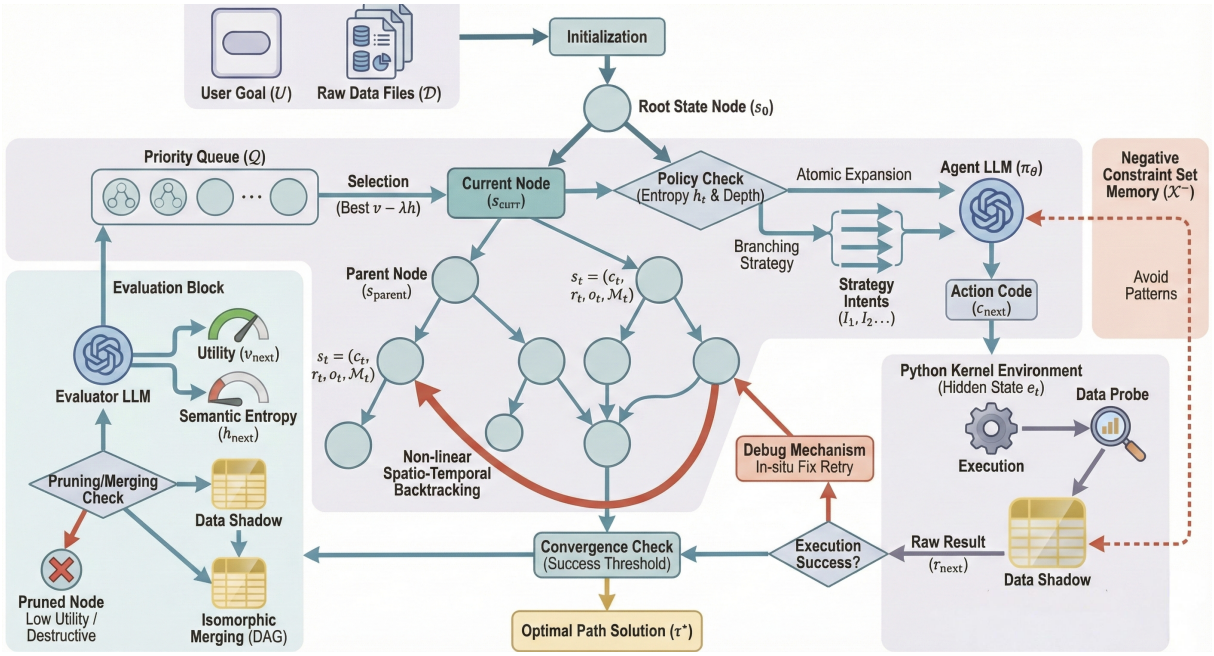

Figure 1: **The Tree-Notebook Workflow.** The diagram illustrates the evolution from the root node (s_0) to the optimal path (τ^*). (1) **Tree Growth**: Semantic entropy (h) triggers dynamic branching or atomic expansion, transforming linear reasoning into a multi-branch search tree. (2) **Tree Restructuring**: The topology is optimized via non-linear backtracking, pruning, and isomorphic merging to ensure convergence to high-completion nodes.

planning and dynamic tool integration. Specific implementations like AutoKaggle(Li et al., 2024), Automind(Ou et al., 2025), CAAFE(Hollmann et al., 2023), and InsightPilot(Ma et al., 2023) address distinct DS lifecycle stages. Despite new benchmarks like MLAgentBench(Huang et al., 2023; Bai et al., 2025) revealing limitations in authentic research tasks, most existing systems remain linear executors. They lack the ability to actively explore "data states" or perform counterfactual reasoning during execution(Zhang et al., 2026).

2.3 Notebook and Search-Enhanced Agents

Recognizing the exploratory nature of data science, research is shifting toward interactive Jupyter environments that persist memory state. OpenCodeInterpreter leverages execution feedback to iteratively refine code, proving the value of environmental interaction(Zheng et al., 2024). Jupybara(Wang et al., 2025c) introduces planning-verification mechanisms within notebooks(Wang et al., 2024), while Datawise Agent models tasks as Finite State Machines (FST), achieving SOTA performance via active "Self-Debugging"(You et al., 2025).

Most recently, JUPITER formalizes data analysis as a search problem, using MCTS and a trained Value Model to guide exploration in code space(Li et al., 2025b; Wang et al., 2025d), yet it only deploys tree search for transient trajectory optimization. Moreover, while JUPITER relies primarily on textual feedback (e.g., error logs, stdout) to infer state, Tree-Notebook not only leverages structured state projection to maintain deep semantic awareness of the underlying data (e.g., DataFrame transformations) but also fundamentally restructures the interaction context into a persistent tree topology—enabling parallel experimental branches without context pollution(Li et al., 2025a), a critical edge over linear frameworks.

Method	Context Topo.	Planning (Tracking)
Datawise Agent	Lin. Seq.	FST (Text/Std)
JUPITER	Search Tree	VG-MCTS (Text Traj.)
Data Interp.	Dyn. Graph	Graph Ref. (Graph Mem.)
Tree-Notebook	**Tree Cells**	**E-POMDP (Data Shadow)**

Table 1: Comparison with SOTA agents. Abbreviations: **Topo.**: Topology; **Lin. Seq.**: Linear Sequence; **Dyn.**: Dynamic; **Std**: Stdout; **VG**: Value-Guided; **Traj.**: Trajectory; **Ref.**: Refinement; **Mem.**: Memory; **E-POMDP**: Entropy-POMDP. JUPITER uses trees only during search, not for context retention. **Tree-Notebook** uniquely structures the notebook as a tree to manage diverse exploration branches and employs E-POMDP for robust planning.

3 Methodology

We formalize the problem of automatic data science code generation in Tree-Notebook as an **Entropy-Regularized Trajectory Optimization** task within a **Partially Observable Markov Decision Process (POMDP)** framework. Unlike traditional linear code generation, we treat data science tasks as a navigation problem within a vast, dynamic latent state space. The core challenge in this scenario is *State Invisibility*: the LLM cannot directly perceive the full memory state of the DataFrames within the Python kernel. To address this perception bottleneck, our framework introduces a proxy observation mechanism named "Data Shadow" and circumvents execution risks by evaluating semantic entropy on a dynamic search tree.

3.1 Problem Definition: Environment and State Space

We define the interaction environment as a tuple $\langle \mathcal{S}, \mathcal{A}, T, \mathcal{R}, \Omega, \mathcal{O} \rangle$. Specifically, $\mathcal{S}$ represents the set of latent physical states (memory snapshots), $\mathcal{A}$ denotes the action space of executable code, and $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the deterministic transition function governed by the Python interpreter. The reward function $\mathcal{R}$ evaluates the utility of transitions, while Ω represents the space of observations. Since the physical state is strictly invisible, the agent relies on the observation function $\mathcal{O} : \mathcal{S} \to \Omega$ to perceive the environment.

At time step t, the system resides in a latent physical state $e_t \in \mathcal{S}$ (i.e., the actual memory snapshot). The agent cannot access e_t directly but receives an observation $o_t = \mathcal{O}(e_t)$ (Data Shadow) and maintains an observable state s_t (a node in the Notebook). This node s_t is defined as an enriched encapsulation object designed for complex reasoning and backtracking:

$$s_t = (c_t, r_t, o_t, \mathcal{M}_t) \tag{1}$$

The components are defined as follows: c_t **(Action Code):** The Python logical code unit generated at the current step, constituting the agent's action space ($a_t \equiv c_t$). r_t **(Execution Result):** Direct feedback from the physical environment via transition T, including Stdout, Traceback, and a system checkpoint to support rollback operations. o_t **(Observation / Data Shadow):** The critical bridge between the LLM and the physical environment, generated by $\mathcal{O}$. Since the LLM's context window cannot process raw large-scale tabular data, we inject a "Data Probe" after each execution. The probe extracts DataFrame meta-information (dimensions, schema semantics, type distribution, statistical summaries, and key slices) to reconstruct a lightweight state representation, o_t. This serves as the sole evidence for the model to understand environmental transitions ($e_t \to e_{t+1}$)(Zhao and Krishnan, 2025). $\mathcal{M}_t$ **(Node Metrics):** A tuple $\langle v_t, h_t \rangle$ guiding the tree search policy. It comprises the **Task Completion Score** (v_t), which acts as the immediate reward (derived from $\mathcal{R}$) quantifying how well c_t satisfies user requirements, and the **Subjective Semantic Entropy** (h_t), which quantifies the model's cognitive uncertainty regarding the logical plausibility of the current state (see Section 3.2).

3.2 Optimization Objective Function

The agent's ultimate goal is to construct an optimal trajectory $\tau^* = (s_0, s_1, \ldots, s_T)$ within the search space $\mathcal{T}$. To balance "efficient task progression" and "risk avoidance," we propose a **Dual-Objective Evaluation Framework** that maximizes the following compound utility function:

$$\tau^* = \underset{\tau \in \mathcal{T}}{\operatorname{argmax}} \sum_{t=0}^{|\tau|} \left(\underbrace{v_t(s_t)}_{\text{Utility}} - \lambda \cdot \underbrace{h_t(s_t)}_{\text{Uncertainty}} \right) \tag{2}$$

Here, $v_t(s_t)$ represents the utility gain of the current state, while $\lambda \cdot h_t(s_t)$ acts as a regularization term penalizing high-risk uncertainty (Sun et al., 2025; Wang et al., 2025a). λ is a hyperparameter controlling the agent's risk aversion (Zhang et al., 2025a; Wang et al., 2025b).

Task Utility Definition (v_t) and Convergence. The Task Completion Score (v_t) serves as the primary driver for trajectory optimization. We define $v_t \in [0, 1]$ as a normalized probability score estimated by the Evaluator LLM $\mathcal{E}$, measuring the alignment between the current observable state s_t and the User Goal $\mathcal{U}$:

$$v_t(s_t) = \mathbb{I}_{\text{exec}} \cdot P_{\mathcal{E}}(\text{success} \mid \mathcal{U}, o_t, r_t) \tag{3}$$

where $\mathbb{I}_{\text{exec}} \in \{0, 1\}$ is an indicator function that is 1 if the code executes without fatal errors and 0 otherwise. The definition of success varies by task type: **For Data Analysis QA (e.g., InfiAgent-DABench):** v_t measures *Answer Confidence*. The Evaluator checks if the final output in r_t strictly follows the required format and logically answers the

query $\mathcal{U}$ based on the data profile in o_t. If a direct answer match is found, $v_t \to 1$. **For Data Modeling (e.g., DSBench):** Since final model metrics are unknown during intermediate steps, v_t measures *Process Completeness*. We decompose the pipeline into milestones (e.g., Data Loading $\to$ Cleaning $\to$ Feature Engineering $\to$ Model Training). v_t is assigned incrementally (e.g., 0.2, 0.5, 0.9) as the agent successfully generates valid artifacts corresponding to these milestones. **Termination Criteria:** In Algorithm 1, the threshold τ_{success} acts as a confidence gate (set to 0.95). When $v_t(s_t) > \tau_{\text{success}}$, the agent deems the current trajectory optimal and terminates the search early.

Computation of Subjective Semantic Entropy (h_t). Traditional code generation focuses on syntactic correctness. However, in data science, code may run without errors yet violate semantic integrity (Kuhn et al., 2023) (e.g., erroneously dropping key columns). To quantify the risk of such "implicit hallucinations," we introduce an independent LLM **Evaluator** as a second-order observer. Utilizing the same backbone model as the main workflow, the Evaluator is prompted to output confidence scores within the range $[0, 1]$ for three dimensions. Based on the input $\langle \mathcal{U}, c_t, o_t \rangle$, these scores constitute a probability distribution P over three mutually exclusive semantic categories: *Effective* (y_1, logical convergence), *Ineffective* (y_2, no substantive change), and *Destructive* (y_3, information loss). The subjective semantic entropy is calculated via the Shannon entropy formula:

$$h_t(s_t) = -\sum_{i=1}^{3} P(y_i) \log P(y_i) \tag{4}$$

Physical Interpretation: Low entropy indicates high system confidence. Note that if the system is certain the action is destructive ($P(y_3) \approx 1$), the node is discarded due to low v_t. High entropy implies ambiguity regarding the consequences, triggering defensive branching search strategies via the penalty term λ.

3.3 Information-Theoretic Dynamic Tree Search

We model the inference process as an iterative expansion on a state space graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Unlike static graph search, our structure is an "Evolutionary Tree" that grows dynamically based on information gain and physical feedback(Farquhar et al., 2024; Nguyen et al., 2025).

3.3.1 Initialization and Atomic Expansion

Search begins at root s_0, initialized with a fixed template to load user data and establish the initial Data Shadow o_0. A global **Negative Constraint Set** $\mathcal{K}_0^- = \emptyset$ is initialized to record invalid path patterns. For any non-terminal node s_t in a low-entropy state, the agent performs **Atomic Expansion**. This autoregressive process generates the subsequent code unit c_{t+1} based on history H_t, observation o_t, and explicitly references $\mathcal{K}_t^-$ to avoid repeating known errors:

$$a_t = \underset{a}{\operatorname{argmax}}\, \pi_\theta(a \mid H_t, o_t, \mathcal{K}_t^-) \tag{5}$$

Following the action selection, the state transitions via execution: $s_{t+1} \leftarrow \text{Exec}(s_t, a_t)$. Here, $\mathcal{K}_t^-$ serves as an implicit penalty, suppressing token sequences associated with prior physical failures or semantic illegality.

3.3.2 Structural Mutation: Branching

To prevent greedy search from trapping in local optima, we introduce a dynamic branching mechanism(Li et al., 2022). When a node enters a decision-sensitive zone (e.g., depth $d \in \{2, 3\}$) or semantic entropy exceeds a threshold ($h_t(s_t) > \delta$), the **Branching Strategy** is triggered. The model enters "Planning Mode," first generating K mutually exclusive **Strategy Intents** $I = \{i_1, \ldots, i_K\}$ (e.g., {"Drop NaNs", "Impute Mean", "KNN Imputation"}). Subsequently, child nodes are conditionally generated for each intent:

$$\text{Children}(s_t) = \{\text{Exec}(s_t, a^{(k)})\}_{k=1}^{K} \tag{6}$$

where each action is sampled conditionally as $a^{(k)} \sim \pi_\theta(\cdot \mid s_t, i_k)$. This mimics human expert "multi-hypothesis testing" under uncertainty.

3.3.3 State Space Compression: Merging

To eliminate computational redundancy, we employ **Observational Equivalence**. Two nodes s_i, s_j are observationally isomorphic if their Data Shadows are identical:

$$s_i \cong s_j \iff o_i \equiv o_j \tag{7}$$

Upon detection, the graph performs a **Merge** operation, collapsing nodes into a representative and retaining only the parent pointer with the optimal cumulative utility. This transforms the search space from an exponentially growing Tree into a compact Directed Acyclic Graph (DAG)(Chivukula et al., 2025).

Algorithm 1 TREE-NOTEBOOK Optimization

Require: Inputs $(U, \mathcal{D})$, Params $(\lambda, \delta, \xi, K)$; **Ensure** Trace τ^*

Init s_0, $\mathcal{K}^- \leftarrow \emptyset$, $\mathcal{Q} \leftarrow \{s_0\}$, $\mathcal{V}_{seen} \leftarrow \{o_0\}$
while $\mathcal{Q} \neq \emptyset$ **do**
  $s_t \leftarrow$ PopBest($\mathcal{Q}$)
  if $v(s_t) > \tau_{success}$ **then return** $(s_0, \ldots, s_t)$ ▷ Success
  if $h(s_t) > \delta$ **then** $\mathcal{A} \leftarrow \{\text{Sample}(s_t, i_k)\}_{k=1}^{K}$ ▷ Structural
  else $\mathcal{A} \leftarrow \{\text{Sample}(s_t, \mathcal{K}^-)\}$ ▷ Atomic
  for $c \in \mathcal{A}$ **do**
    $r, s_{new} \leftarrow \text{Execute}(s_t, c)$
    if r is error **then**
      $s_{new} \leftarrow \text{Refine}(s_t, c, r)$; **if** fails **then** $\mathcal{K}^-$.add(ErrPat); **continue**
    end if
    Calc $o_{new}, v_{new}, h_{new}$; **if** $v_{new} < v(s_t) - \xi$ **then** $\mathcal{K}^-$.add(c)
    else if $o_{new} \in \mathcal{V}_{seen}$ **then** Merge s_{new}
    else Push s_{new} to $\mathcal{Q}$; $\mathcal{V}_{seen}$.add(o_{new})
  end for
end while
return Failure

3.3.4 Survival of the Fittest: Pruning

Nodes satisfying any of the following conditions are pruned: (1) **Physical Failure**: Execution error persists after self-correction; (2) **Utility Collapse**: Significant negative gradient in v_t; (3) **Semantic Illegality**: High probability of destructive action.

3.4 Non-linear Spatio-Temporal Backtracking

Traditional "error → in-situ fix → retry" loops are insufficient for data science, where erroneous code often pollutes the global state. Tree-Notebook proposes **Non-linear Spatio-Temporal Backtracking**, treating debugging as time travel and branch reconstruction.

Phase I: In-situ Refinement. For minor errors or assertion failures, the system attempts low-cost in-situ repair. The model π_θ generates a corrected code c_t' based on the error traceback r_{error}: $c_t' \sim \pi_\theta(\cdot \mid H_t, c_t, r_{error})$. If unsuccessful after $N_{retry} = 2$, the system escalates to Phase II.

Phase II: Backtracking Reconstruction. If in-situ repair fails, the agent performs a **Rollback**: it abandons s_t and forces the physical environment to revert to the parent state s_{t-1} using the checkpoint mechanism ($s_{curr} \leftarrow \text{Parent}(s_t)$). Subsequently, it triggers **Branch Rebirth**: a new child branch s_t' is sampled from s_{t-1}. To ensure orthogonality to the failed path, the failed action c_{fail} is added to the Negative Constraint Set $\mathcal{K}^-$, explicitly warning the model:

$$\pi_\theta(c_t' \mid H_{t-1}, o_{t-1}, \mathcal{K}^-) \tag{8}$$

This mechanism enables the agent to learn from trial-and-error, proactively avoiding logic that led to dead ends in previous attempts.

4 Experiments

4.1 Experimental Setup

Given that data analysis and predictive modeling are core application scenarios of data science, we selected two authoritative and public benchmarks for extensive comparative experiments: **InfiAgent-DABench** (Hu et al., 2024), which covers data analysis QA tasks, and **DSBench** (Jing et al., 2024), which covers end-to-end data modeling tasks. Furthermore, to evaluate the reasoning capabilities of agents under extreme difficulty, we introduced the **InfiAgent-DABench (Enhanced)** setting, an enhanced evaluation designed for Hard-level difficulty and multi-step reasoning scenarios.

(1) Baselines To comprehensively evaluate the performance of our method, we compare it against five representative baselines: ReAct(Yao et al., 2022), AutoGen(Wu et al., 2024), TaskWeaver(Qiao et al., 2023), Data Interpreter(Zheng et al., 2024), Jupiter(Li et al., 2025b)and Datawise Agent(You et al., 2025).

(2) InfiAgent-DABench This dataset contains a total of 257 questions, categorized by difficulty into Easy (82), Medium (87), and Hard (88). The primary inputs for the task include CSV data files, user questions, and output format requirements. We employ **Accuracy** as the evaluation metric, defined as the proportion of questions correctly answered by the model.

(3) DSBench We focus on the modeling tasks within DSBench, selecting 74 tasks derived from Kaggle competitions. The evaluation utilizes two core metrics:

- **Task Success Rate:** Determines whether the agent can complete the end-to-end predictive modeling task.
- **Relative Performance Gap (RPG):** Quantifies the gap between the agent's performance and the best-known performance. The formula is as follows:

$$RPG = \frac{1}{N} \sum_{i=1}^{N} \frac{p_i - b_i}{g_i - b_i} \tag{9}$$

where N is the total number of competitions (tasks); p_i is the actual performance score of the agent in the i-th competition; g_i is the known highest performance score (Gold standard); and b_i is the baseline performance score for the i-th competition.

(4) InfiAgent-DABench Robustness Setup In the original InfiAgent-DABench dataset, Easy and Medium samples account for approximately 2/3 of the data. Our model achieved a 92.90% one-pass generation success rate on these 169 simple samples. While this demonstrates the powerful capabilities of the backbone LLM API, it also suggests that simple tasks fail to fully stimulate the potential of the search or backtracking mechanisms designed within the Agent. Therefore, we augmented the data from two dimensions: environmental noise and data noise. **Environmental Noise:** We inject files with similar filenames but incorrect content into the working directory (randomly selecting irrelevant CSVs from the dataset and renaming them) to interfere with the Agent's file retrieval process. **Data Noise:** We populate correctly named files with random columns from irrelevant CSVs or randomly insert blank rows into the correct CSVs to test the Agent's data cleaning and anti-interference capabilities.

4.2 InfiAgent-DABench Benchmarking

Framework	Model	Acc (%)
ReAct	GPT-4o	81.32
AutoGen	GPT-4o	73.54
TaskWeaver	GPT-4o	85.99
Data Interpreter*	GPT-4o*	75.78*
Datawise Agent	GPT-4o	85.99
Tree-Notebook	**GPT-4o**	**89.49 (SOTA)**
ReAct	GPT-4o-mini	80.08
AutoGen	GPT-4o-mini	70.04
TaskWeaver	GPT-4o-mini	76.65
Data Interpreter	GPT-4o-mini	67.70
Datawise Agent	GPT-4o-mini	82.88
Tree-Notebook	**GPT-4o-mini**	**85.60**
ReAct	Qwen2.5-72B-Ins	75.88
AutoGen	Qwen2.5-72B-Ins	70.04
TaskWeaver	Qwen2.5-72B-Ins	74.71
Datawise Agent	Qwen2.5-72B-Ins	81.71
Jupiter (w/ VM)*	Qwen2.5-14B-Ins*	86.38*
Tree-Notebook	**Qwen2.5-72B-Ins**	**86.38**

Table 2: Accuracy comparison of various Agents on InfiAgent-DABench. * Note: The best result in the original Jupiter paper(Li et al., 2025b) was achieved with the 14B version, and the results for Data Interpreter are taken from You et al. (2025); thus, we retain its original data for comparison with Qwen2.5-72B-Instruct.

Under the GPT-4o setting, Tree-Notebook achieved a SOTA accuracy of 89.49%, surpassing the previous best performers, TaskWeaver (85.99%) and Datawise Agent (85.99%). This result proves the superiority of the Tree-Notebook structure in handling complex data analysis tasks.

4.3 DSBench Benchmarking

This set of quantitative results presents the success rate and RPG metrics of various agents on DSBench modeling tasks.

Framework	Model	Success Rate (%) / RPG
AutoGen	GPT-4o	71.62 / 34.74
	GPT-4	87.84 / 45.52
	GPT-4o mini	22.97 / 11.24
Datawise Agent	GPT-4o	98.64 / 53.18
	GPT-4o mini	98.64 / 46.61
Jupiter (Search)	Qwen2.5-14B	98.64 / 51.35
Tree-Notebook	**GPT-4o**	**98.64 / 54.20(SOTA)**
	GPT-4	98.64 / 52.56
	GPT-4o mini	98.64 / 49.37

Table 3: Task Success Rate and Relative Performance Gap (RPG) on DSBench.

As seen in Table 3, most advanced models have reached a plateau in Task Success Rate at 98.64%, indicating that this metric's discriminative power is saturated under current model capabilities. In contrast, the RPG metric better reflects modeling quality. Tree-Notebook demonstrates exceptional stability in complex modeling tasks, achieving an RPG score of 54.20 with GPT-4o support, significantly leading Datawise Agent (53.18).

4.4 Robustness Analysis on Noisy Datasets

To verify the model's anti-interference capabilities, we selected Data Interpreter and Datawise Agent, which performed well in the benchmark experiments, for comparison. Table 4 presents the accuracy of each Agent on the enhanced InfiAgent-DABench (noisy dataset) and its retention ratio relative to baseline performance.

Framework	Model	Acc. (Noise / Base)	Retention
Data Interpreter	GPT-4o	61.48 / 75.78	0.8113
	GPT-4o-mini	52.53 / 67.70	0.7759
Datawise Agent	GPT-4o	70.04 / 85.99	0.8145
	GPT-4o-mini	66.54 / 82.88	0.8028
Tree-Notebook	GPT-4o	**80.54** / 89.49	**0.9000**
	GPT-4o-mini	75.10 / 85.60	0.8773

Table 4: Comparison of Accuracy and Robustness (Retention Ratio) on the Noisy Dataset.

On the noise-injected dataset, Tree-Notebook maintained SOTA performance. We achieved the highest accuracy on noisy data and the largest Retention Ratio relative to the benchmark, demonstrating the model's robustness against complex data. Compared to Jupiter's brute-force search, we utilized Notebook State Caching to avoid redundant execution, drastically reducing time costs; compared to Datawise, although we increased exploration overhead, we achieved extremely high robustness. This also corrects the previous assumption that a single linear path is sufficient to cope with complex environments.

4.5 Ablation Study

To validate the effectiveness of the core components in the Tree-Notebook framework, we conducted a detailed ablation study on the InfiAgent-DABench validation set. We compared the full model with three variants to deconstruct the marginal contributions of **Dynamic Branching**, **Spatio-Temporal Backtracking**, and **Data Shadow**.

4.5.1 Experimental Setup and Variants

We used Tree-Notebook (GPT-4o) as the Full Model and established the following variants:

w/o Dynamic Branching (Linear Mode): Removes the dynamic branching mechanism. Regardless of whether the semantic entropy h_t exceeds the threshold, the model is forced to generate only a unique atomic expansion path. The search structure degenerates into a unidirectional structure, consistent with Datawise Agent.

w/o Spatio-Temporal Back (Linear Debug): Removes the Checkpoint rollback and branch respawn mechanisms. When an error occurs, only in-situ refinement is allowed, adopting the traditional "Error $\rightarrow$ Fix $\rightarrow$ Continue" linear debugging flow.

w/o Data Shadow (Text-only Obs.): Removes the structured Data Shadow o_t. The model relies solely on default text output from pandas (e.g., plain text from `df.head()`) to perceive the environment, and node merging based on observational equivalence is disabled.

4.5.2 Results and Analysis

Table 5 presents the specific results of the ablation study. Experiments show that all three components contribute positively to model performance. Notably, Dynamic Branching exhibits the most significant impact on both handling complex tasks (Hard subset) and maintaining system robustness against environmental noise. Meanwhile, the spatio-temporal backtracking mechanism serves as a crucial error-recovery layer, preventing the propagation of logical failures in long-horizon reasoning.

Method / Variant	Acc. (Base) %	Acc. (Noise) %
Tree-Notebook (Full)	**89.49**	**80.54**
w/o Dynamic Branching	85.99	73.15
w/o Spatio-Temporal Back	87.55	77.82
w/o Data Shadow	86.38	76.65

Table 5: Ablation Study Results on InfiAgent-DABench. All variants are evaluated using GPT-4o as the base model.

(1) Impact of Dynamic Branching: Removing dynamic branching caused the sharpest accuracy drop. Unlike linear methods prone to error propagation from early failures, dynamic branching enables parallel exploration under high-entropy states. This transforms code generation into an entropy-regularized trajectory optimization problem, which is essential for escaping local optima.

(2) Impact of Spatio-Temporal Backtracking: Ablating this mechanism significantly reduced success in long-horizon tasks. Linear debugging cannot resolve "state pollution" (logical memory corruption). In contrast, spatio-temporal backtracking uses Checkpoints for physical "time reversal," ensuring new branches s'_t respawn in clean environments. This prevents the Agent from looping in contaminated contexts and boosts self-healing.

(3) Impact of Data Shadow: Without Data Shadow o_t, code generation accuracy declined due to hallucinations caused by truncated data in limited contexts. Data Shadow acts as a bridge between the physical environment e_t and cognitive state s_t. Crucially, it enables isomorphic merging to compress the search space into a DAG, thereby avoiding redundant computation and improving reasoning precision.

4.6 Cost and Efficiency Analysis

To provide a comprehensive evaluation of our proposed framework, we analyze the computational cost and efficiency of Tree-Notebook. Given that LLM invocations and token consumption constitute the primary overhead in LLM-based data science agents, we compare our method against several baselines.

First, we evaluated the average token consumption per data entry across different frameworks on the InfiAgent-DABench dataset (using GPT-4o), as shown in Table 6. While basic linear frameworks like ReAct and AutoGen exhibit lower token consumption, they yield significantly lower accuracy. TaskWeaver incurs the highest token cost (15,069 tokens). Tree-Notebook requires a comparable but slightly lower token consumption (14,977 tokens) than TaskWeaver, yet it achieves the highest accuracy (89.49%). This demonstrates that the token expenditure in Tree-Notebook is effectively translated into performance gains.

Framework	Avg. Tokens Cost	Accuracy (%)
ReAct	3,937	81.32
AutoGen	3,850	73.54
TaskWeaver	15,069	85.99
Datawise Agent	8,557	85.99
Tree-Notebook	**14,977**	**89.49**

Table 6: Average token consumption and accuracy comparison across baseline methods on InfiAgent-DABench (Model: GPT-4o).

We further break down the invocation counts , and token usage of Tree-Notebook against the Jupyter-based baseline Datawise Agent on InfiAgent-DABench and DSBench (Table 7). While the dynamic tree structure and LLM evaluator add extra interactions, Tree-Notebook's Data Shadow mechanism fully avoids redundant LLM-generated data exploration code (e.g., df.info(), df.describe()), yielding substantial exploration cost savings. This is especially pronounced on DSBench, where Tree-Notebook (w/o Eval, i.e., excluding the semantic evaluator) reduces invocation counts to 5.73, versus 10.19 for Datawise Agent. Although tree search and the semantic evaluator introduce extra steps, the overall cost increase is mild and well justified by the state-of-the-art robustness and accuracy.

Method	InfiAgent (Invoc. / Tokens)	DSBench (Invoc. / Tokens)
Datawise	3.25 / 8,557	10.19 / 23,737
Ours (w/o Eval)	3.65 / 9,841	5.73 / 17,652
Ours (Full)	5.66 / 14,977	10.37 / 24,873

Table 7: Efficiency comparison. We report LLM invocations and token consumption (Invoc. / Tokens) per task.

5 Conclusion

We propose Tree-Notebook, a framework that reformulates data science code generation as an entropy-regularized trajectory optimization problem within a POMDP, innovatively modeling the linear interaction process as a dynamically growing tree structure where each node represents a Notebook state. By utilizing the Data Shadow mechanism to overcome the invisibility of the Python kernel state, combined with non-linear spatio-temporal backtracking and isomorphic merging strategies, our approach effectively resolves the issues of local optima entrapment and error propagation found in traditional linear agents. Consequently, Tree-Notebook achieves state-of-the-art performance across InfiAgent-DABench, DSBench, and complex noisy environments, demonstrating that this non-linear tree search paradigm, mimicking human iterative reasoning, is a pivotal direction for next-generation data science agents.

Limitations

Although the framework exhibits superior robustness in complex tasks, the introduction of dynamic tree search and multi-branch exploration, coupled with the repetitive extraction of Data Shadows at every reasoning step, significantly increases inference computational overhead and time costs compared to traditional linear models. Furthermore, the branching strategy heavily relies on the accuracy of the underlying LLM evaluator's semantic entropy estimation, where insufficient model capability may lead to ineffective exploration. Additionally, the current Data Shadow mechanism is primarily designed for tabular metadata, and its perception capabilities and potential performance bottlenecks when handling unstructured data (e.g., images, text) or ultra-large-scale distributed datasets remain to be further validated and optimized in future work.

Acknowledgments

This work was supported by the National Key R&D Program of China (No. 2025ZD0122005) and the National Natural Science Foundation of China (No. 62271485).

Here, we sincerely thank the reviewers and ACs for their valuable input, which has greatly improved our work.

References

Mourad Aouini and Jinan Loubani. 2025. Towards more contextual agents: An extractor-generator optimization framework. *arXiv preprint arXiv:2502.12926*.

Jincheng Bai, Zhenyu Zhang, Jennifer Zhang, and Jason Zhu. 2025. Insight agents: An llm-based multi-agent system for data insights. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 4335–4339.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17682–17690.

Abhi Chivukula, Jay Somasundaram, and Vijay Somasundaram. 2025. Agint: Agentic graph compilation for software engineering agents. *arXiv preprint arXiv:2511.19635*.

Victor Dibia. 2023. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. *arXiv preprint arXiv:2303.02927*.

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

Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630.

Chengquan Guo, Xun Liu, Chulin Xie, Andy Zhou, Yi Zeng, Zinan Lin, Dawn Song, and Bo Li. 2024a. Redcode: Risky code execution and generation benchmark for code agents. *Advances in Neural Information Processing Systems*, 37:106190–106236.

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

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173.

Md Mahadi Hassan, Alex Knipper, and Shubhra Kanti Karmaker Santu. 2023. Chatgpt as your personal data scientist. *arXiv preprint arXiv:2305.13657*.

Noah Hollmann, Samuel Müller, and Frank Hutter. 2023. Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. *Advances in Neural Information Processing Systems*, 36:44753–44775.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, et al. 2025. Data interpreter: An llm agent for data science. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19796–19821.

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

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, et al. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507*.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*.

Michael Janner, Yilun Du, Joshua B Tenenbaum, and Sergey Levine. 2022. Planning with diffusion for flexible behavior synthesis. *arXiv preprint arXiv:2205.09991*.

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 from becoming data science experts? *arXiv preprint arXiv:2409.07703*.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. *arXiv preprint arXiv:2302.09664*.

Jiaqi Li, Xinyi Dong, Yang Liu, Zhizhuo Yang, Quansen Wang, Xiaobo Wang, Song-Chun Zhu, Zixia Jia, and Zilong Zheng. 2025a. Reflectevo: Improving meta introspection of small llms by learning self-reflection. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 16948–16966.

Shuocheng Li, Yihao Liu, Silin Du, Wenxuan Zeng, Zhe Xu, Mengyu Zhou, Yeye He, Haoyu Dong, Shi Han, and Dongmei Zhang. 2025b. Jupiter: Enhancing llm data analysis capabilities via notebook and inference-time value-guided search. *arXiv preprint arXiv:2509.09245*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

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

Pingchuan Ma, Rui Ding, Shuai Wang, Shi Han, and Dongmei Zhang. 2023. Demonstration of insightpilot: An llm-empowered automated data exploration system. *arXiv preprint arXiv:2304.00477*.

Junyuan Mao, Fanci Meng, Yifan Duan, Miao Yu, Xiaojun Jia, Junfeng Fang, Yuxuan Liang, Kun Wang, and Qingsong Wen. 2025. Agentsafe: Safeguarding large language model-based multi-agent systems via hierarchical data management. *arXiv preprint arXiv:2503.04392*.

Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. 2024. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 101.

Dang Nguyen, Ali Payani, and Baharan Mirzasoleiman. 2025. Beyond semantic entropy: Boosting llm uncertainty quantification with pairwise semantic similarity. *arXiv preprint arXiv:2506.00245*.

Yixin Ou, Yujie Luo, Jingsheng Zheng, Lanning Wei, Zhuoyun Yu, Shuofei Qiao, Jintian Zhang, Da Zheng, Yuren Mao, Yunjun Gao, et al. 2025. Automind: Adaptive knowledgeable agent for automated data science. *arXiv preprint arXiv:2506.10974*.

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

Mizanur Rahman, Amran Bhuiyan, Mohammed Saidul Islam, Md Tahmid Rahman Laskar, Ridwan Mahbub, Ahmed Masry, Shafiq Joty, and Enamul Hoque. 2025. Llm-based data science agents: A survey of capabilities, challenges, and future directions. *arXiv preprint arXiv:2510.04023*.

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

José Antonio Siqueira de Cerqueira, Mamia Agbese, Rebekah Rousi, Nannan Xi, Juho Hamari, and Pekka Abrahamsson. 2024. Can we trust ai agents? an experimental study towards trustworthy llm-based multi-agent systems for ai ethics. *arXiv e-prints*, pages arXiv–2411.

Jia Ao Sun, Hao Yu, Fabrizio Gotti, Fengran Mo, Yihong Wu, Yuchen Hui, and Jian-Yun Nie. 2025. Search-on-graph: Iterative informed navigation for large language model reasoning on knowledge graphs. *arXiv preprint arXiv:2510.08825*.

Abdullah Vanlioglu. 2025. Entropy-guided sequence weighting for efficient exploration in rl-based llm fine-tuning. *arXiv preprint arXiv:2503.22456*.

Ante Wang, Linfeng Song, Ye Tian, Dian Yu, Haitao Mi, Xiangyu Duan, Zhaopeng Tu, Jinsong Su, and Dong Yu. 2025a. Don't get lost in the trees: Streamlining llm reasoning by overcoming tree search exploration pitfalls. *arXiv preprint arXiv:2502.11183*.

Chenglong Wang, Bongshin Lee, Steven M Drucker, Dan Marshall, and Jianfeng Gao. 2025b. Data formulator 2: Iterative creation of data visualizations, with ai transforming data along the way. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–17.

Huichen Will Wang, Larry Birnbaum, and Vidya Setlur. 2025c. Jupybara: Operationalizing a design space for actionable data analysis and storytelling with llms. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–24.

Xiaobo Wang, Zixia Jia, Jiaqi Li, Qi Liu, and Zilong Zheng. 2025d. Adaptive preference optimization with uncertainty-aware utility anchor. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 19204–19225.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Nathaniel Weir, Muhammad Khalifa, Linlu Qiu, Orion Weller, and Peter Clark. 2024. Learning to reason via program generation, emulation, and search. *Advances in Neural Information Processing Systems*, 37:36390–36413.

Muning Wen, Cheng Deng, Jun Wang, Weinan Zhang, and Ying Wen. 2024. Entropy-regularized token-level policy optimization for large language models. *CoRR*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2023. An empirical study on challenging math problem solving with gpt-4. *arXiv e-prints*, pages arXiv–2306.

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Ziming You, Yumiao Zhang, Dexuan Xu, Yiwei Lou, Yandong Yan, Wei Wang, Huaming Zhang, and Yu Huang. 2025. Datawiseagent: A notebook-centric llm agent framework for automated data science. *arXiv preprint arXiv:2503.07044*.

Xiaokang Zhang, Sijia Luo, Bohan Zhang, Zeyao Ma, Jing Zhang, Yang Li, Guanlin Li, Zijun Yao, Kangli Xu, Jinchang Zhou, et al. 2025a. Tablellm: Enabling tabular data manipulation by llms in real office usage scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10315–10344.

Yuge Zhang, Qiyang Jiang, XingyuHan XingyuHan, Nan Chen, Yuqing Yang, and Kan Ren. 2024. Benchmarking data science agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5677–5700.

Zhenkui Zhang, Wendong Bu, Kaihang Pan, Bingchen Miao, Wenqiao Zhang, Guoming Wang, Wei Ji, Rui Tang, Juncheng Li, and Siliang Tang. 2026. Evolving generalist virtual agents with generative and associative memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 13006–13014.

Zhenliang Zhang, Xinyu Hu, Huixuan Zhang, Junzhe Zhang, and Xiaojun Wan. 2025b. Icr probe: Tracking hidden state dynamics for reliable hallucination detection in llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17986–18002.

Jinjin Zhao and Sanjay Krishnan. 2025. Metadata management for ai-augmented data workflows. *arXiv preprint arXiv:2508.06814*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

A Appendix

A.1 LLM Usage

The authors of this paper utilized the web-based version of Gemini for linguistic polishing and drawing.

A.2 LLM Api Usage

The versions of the GPT API employed in the experiments of this paper are gpt-4o-2024-08-06 and gpt-4o-mini-2024-07-18.

A.3 Data Shadow

The concrete implementation of Data Shadow invokes the Jupyter kernel at each node (Jupyter cell) to extract metadata, as shown below.

```
def data_probe(local_vars):
    """
    Extract metadata of all DataFrames in memory
    """
    shadow = {}
    for var_name, var_value in local_vars.items
    ():
        if isinstance(var_value, pd.DataFrame):
            shadow[var_name] = {
                "rows": var_value.shape[0],
                "cols": var_value.shape[1],
                "columns": list(var_value.
    columns),
                "dtypes": {k: str(v) for k, v in
                           var_value.dtypes.
    items()},
                "sample": var_value.head(2).
    to_dict()  # Simple sampling
            }
    return shadow
```

This method directly obtains data information from the Jupyter kernel, whose efficiency is negligibly affected by data scale (an inherent property of the Jupyter kernel). Independent of dataset row count, Data Shadow acts as a real-time updated external knowledge base, enabling the LLM to perceive the latest DataFrame changes. Thus, there is no need to generate data exploration code such as `df.head()` and `df.info()`, making the overhead from Data Shadow very small.

A.4 POMDP Modeling Explanation

We model data science programming as a **POMDP (Partially Observable Markov Decision Process)**: LLMs are "blind" to complete memory data and rely on probes/inference to proceed step-by-step. Key components:

- **Hidden State:** Actual memory snapshot/-data distribution in Python kernel (invisible to LLMs).
- **Observation:** "Data Shadow" via projection, extracting metadata for LLM reasoning.
- **Action:** Python code cell driving state transition.
- **Optimization Objective:** Balance task progression against uncertainty by maximizing task utility while penalizing semantic entropy.

A.5 Basic System Configuration Prompt

```
[Role Definition]
You are an intelligent assistant proficient in
    Python and data science, working in a
    Jupyter Notebook environment.
Your goal is to solve the user's problem step-by
    -step by writing code cells (Cells).
```

A.6 Code Generation Prompt

```
[Role Definition]
You are a senior data scientist working in the
    Jupyter Notebook environment, adept at
    continuing to write Python code cells based
    on existing analytical workflows.

[Current Task Information]
User's Core Requirement (Question): {question}
Core Knowledge Points to Apply (Concepts): {
    concepts}
Mandatory Constraints to Follow (Constraints): {
    constraints}
Output Format Requirements: {format}
  - If information output is required, use the
    print() function to display key results;
  - If a CSV path is specified, save the results
     as a CSV file at the corresponding path;
  - In the absence of explicit format
    requirements, use print() to output core
    conclusions by default.
Current Analytical Strategy: {level}

[Historical Context]
Executed code cells and their output results:
{history}

[Current Data Status (Data Shadow)]
Metadata of key variables in memory (type,
    dimension, core fields, etc.):
{data_shadow}

[Negative Constraints (Validated Infeasible
    Solutions)]
{negative_constraints}

[Core Instructions]
Please write the **Next Cell** for Jupyter
    Notebook, strictly adhering to the following
     rules:
1. Continue writing based on historical code and
     current data status; do not re-import
    already imported libraries or re-define
    already defined variables;
2. The code must be directly executable,
    avoiding basic issues such as syntax errors
    and undefined variables;
```

```
3. Key analytical results must be output in
    accordance with the specified output format
    (print/CSV saving);
4. Output only Python code blocks without any
    additional textual explanations;
5. If it is determined that the task has been
    fully completed based on historical context,
     directly output the Chinese phrase [Task
    Completed] (ensure the final output is
    generated in compliance with the output
    format requirements first).

Output Example:
‘‘‘python
# Your code

‘‘‘
```

A.7 Strategy Branching Prompt

```
[Task Objective]
We are currently at a critical node in data
    analysis. It is necessary to conduct
    divergent thinking based on existing
    information and formulate 2-3 differentiated
     strategies for the next step of analysis.

[Basic Information]
User’s Core Requirement: {question}
Historical Execution Context (completed code and
     outputs): {history_context}
Current Data Status (metadata of variables/
    datasets): {data_shadow}

[Output Requirements]

1. Output strictly in JSON list format without
    any additional text;
2. Each strategy shall contain two fields:
* "strategy_name": A concise English name (e.g.,
     DropMissing/FillMean) that reflects the
    core of the strategy;
* "intent": A detailed Chinese description
    explaining the basis for formulating the
    strategy, specific operations, and expected
    objectives.



Format Example:
[
{"strategy_name": "DropMissing", "intent": "
    Missing values detected in the ’Age’ column;
     it is decided to directly delete rows
    containing missing values..."},
{"strategy_name": "FillMean", "intent": "Missing
     values detected in the ’Age’ column; it is
    decided to fill the missing values with the
    mean value..."}
]

Please output the JSON:
```

A.8 Error Fixing Prompt

```
[Error Report]
The code generated in the previous step failed
    to execute.


[Historical Jupyter Information]
{history}

[Error Code]
{code}

[Error Message (Traceback)]
{error_msg}

[Instructions]
Please analyze the cause of the error, combine
    it with the historical context, and re-
    output the corrected code.
Please output in a markdown code block,
    containing only Python code without any
    other text explanation.
```

A.9 Scoring and Evaluation Prompt

```
[Evaluation Task]
You need to act as an "observer" to evaluate
    whether the current code cell completes the
    task and assess its risk.
If you feel this code block solves the problem
    and outputs according to [User Output
    Constraints], boldly give a score of 1.0,
    which is greater than 0.9;
{question}
[User Output Constraints]
{constraints}
[Current Code]
{code}

[Current Code Output]
{output}

[Output Requirements]
Please output the following metrics in JSON
    format:
1. "completion_score": (0.0 - 1.0) task
    completion score.
2. "status_probs": {{ "Effective": float, "
    Ineffective": float, "Destructive": float }}
   (Probability distribution of the three states,
     sum must be 1.0)

Effective: Data changes are logical and move
    closer to the target.
Ineffective: The operation is meaningless or
    introduces noise.
Destructive: Logical error resulting in critical
     data loss or error.

JSON Example:
{{
    "completion_score": 0.8,
    "status_probs": {{ "Effective": 0.7, "
    Ineffective": 0.2, "Destructive": 0.1 }}
}}
```