DataSight: Learning to Predict Before Executing for Data Analysis Agents

Anonymous ACL submission

Abstract

Data-analytic agents must iteratively generate and execute code while interpreting real-time feedback to refine their analytical strategies. Existing training paradigms decouple the agent's policy from the underlying environment dynamics, producing agents that learn to recover from errors rather than to avoid them: failure rates rise alongside recovery rates, and interaction turns inflate as agents resort to trial-and-error exploration. We introduce DataSight, a framework that internalizes environment dynamics by making the model's anticipation of execution outcomes an explicit, structured output accompanying every generated action. DataSight first enriches expert trajectories with foresight prediction fields grounded in real execution feedback, then applies a dual-stage RL curriculum that calibrates prediction fidelity before optimizing long-horizon task performance, and at inference time intercepts anticipated failures before physical execution. Trained on only 7,273 trajectories, DataSight-4B surpasses both DataMind-7B and DeepAnalyse-8B in average accuracy, while DataSight-8B achieves 64.13% Avg@3 across four benchmarks with Pass@3 of 74.81%, approaching DeepSeek-V3.2 at 75.56%. Analysis confirms that foresight training generalizes across non-reasoning backbones and SQL-based tasks, improving both single-step code quality and multi-turn interaction efficiency.

1 Introduction

Data-analytic agents must iteratively generate code, execute it against structured data files, and interpret real-time execution feedback to progressively refine their analytical strategies (Zeng et al., 2026; Team et al., 2025). Unlike single-turn code generation, this multi-turn interaction requires the agent to maintain coherent analytical plans while responding to unpredictable execution outcomes across successive turns.

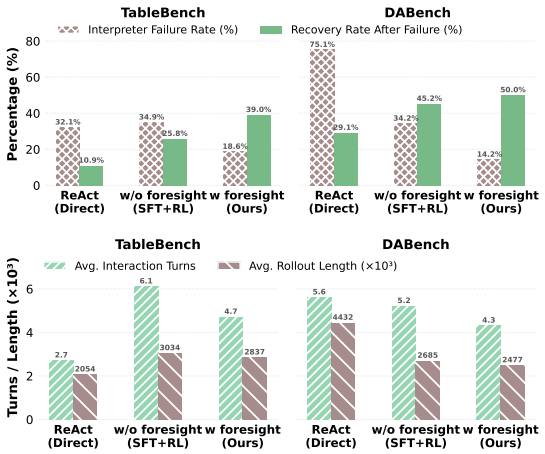

Figure 1: Behavioral analysis of three training paradigms on Qwen3-1.7B. Top: training without foresight exhibits weak failure-avoidance capability. Bottom: this inflates interaction turns and rollout length. DataSight reduces failures while achieving high recovery and efficient interaction with less tokens.

Existing training paradigms, whether through imitation of expert trajectories or constrained action exploration, decouple the agent's policy from the underlying environment dynamics: agents learn *what to do next* given what they have observed, but never internalize *what will happen* as a result of their actions. As illustrated in Figure 1, untuned ReAct agents exhibit weak performance in both action generation and environment perception. Training without foresight (SFT followed by RL) only strengthens action generation. This asymmetry leaves them deficient in both error avoidance and error resolution: on TableBench, the failure rate even rises slightly (32.11%→34.87%) despite improved recovery; on DABench, over a third of execution steps still fail. Each avoidable failure injects erroneous observations into the context, compounding errors across subsequent turns.

Recent efforts have begun to address this gap by embedding environment dynamics implicitly into

model parameters, for instance training agents to predict next states as a warm-up before policy optimization (Zhang et al., 2025a; Yu et al., 2026). While such approaches provide useful initialization, the model's anticipatory reasoning becomes latent once training transitions to the policy phase, where it can neither be directly shaped by reward signals nor inspected at inference time. We take a different approach: rather than treating environmental understanding as a latent prior, we make the model's anticipation of execution outcomes an explicit, structured output accompanying every generated action. This unlocks two capabilities that latent approaches cannot support: the alignment between predicted and actual outcomes becomes a directly optimizable training signal during RL, and the model's own predictions become actionable during inference, enabling the agent to intercept likely failures before code is submitted to the interpreter.

We introduce **DataSight**, a framework that internalizes environment dynamics into the agent's policy. DataSight first enriches expert trajectories with structured foresight fields grounded in real execution feedback, training the agent to predict execution outcomes alongside its actions. A dual-stage RL process then calibrates this predictive capability: the first stage jointly rewards action correctness and prediction fidelity, while the second stage drops the prediction constraint, allowing the policy to freely optimize long-horizon task performance atop the calibrated world model. At inference time, the agent evaluates its own foresight after generating each action; when failure is anticipated, the action is intercepted and regenerated before any interpreter call is made.

Trained on 7,273 trajectories, DataSight-8B achieves 64.13% Avg@3 across four benchmarks, outperforming DeepAnalyse-8B by 7.41% with Pass@3 of 74.81% approaching DeepSeek-V3.2 at 75.56%. Compared to an action-only baseline on the same data, DataSight reduces interaction turns by 2.82 while improving accuracy by 13.33% on TableBench and DABench.

Our contributions are threefold:

- We empirically reveal across multiple benchmarks that training without foresight yields agents with limited error-avoidance and error-resolution capabilities: without learning to anticipate execution consequences, agents can neither effectively preempt failures nor efficiently recover from them.
- We propose a foresight-augmented training paradigm that requires the agent to explicitly predict execution outcomes alongside its actions. A dual-stage RL curriculum first calibrates prediction fidelity against real interpreter feedback, then optimizes long-horizon task performance upon the calibrated world model.
- We propose the Predict-and-Intercept inference framework, which leverages internalized foresight to intercept potentially erroneous actions before physical execution, reducing context contamination and improving reasoning robustness under residual inference-time uncertainty.

2 Related Work

2.1 Data Analysis Agents

The development of LLM-based data analysis agents has evolved through three paradigms (Gu et al., 2025). **Prompt-based approaches** rely on architectural design and workflow orchestration, including case-based reasoning (Guo et al., 2024), multi-agent decomposition (Li et al., 2024), predefined workflows (Zhang et al., 2023; Nam et al., 2025; Sun et al., 2025; Liu et al., 2025b), and dynamic graph-based planning (Hong et al., 2025). **Fine-tuning-based methods** instill domain knowledge through specialized datasets, such as Jupyter notebooks (Li et al., 2026b) and table-specific corpora (Wu et al., 2025; Hu et al., 2024). **RL-based methods** further enable self-exploration through sandbox interaction: DeepAnalyse (Zhang et al., 2025b) performs cold-start SFT on ∼500k trajectories before RL, while DataMind (Qiao et al., 2025) jointly optimizes SFT and RL losses to mitigate exploration collapse. Despite these advances, existing methods train agents to map observed states to actions without internalizing the causal link between actions and their environmental consequences.

2.2 Learning from Environment Feedback

Agent interaction with environmental feedback is central to autonomous data analysis. Existing approaches follow three directions: **prompt-based** frameworks use environmental feedback as in-context guidance for error correction (Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2022); **World model** methods train separate predictive models for trajectory simulation (Li et al., 2025; Liu et al., 2026) or rollout guidance (Qiao et al.,

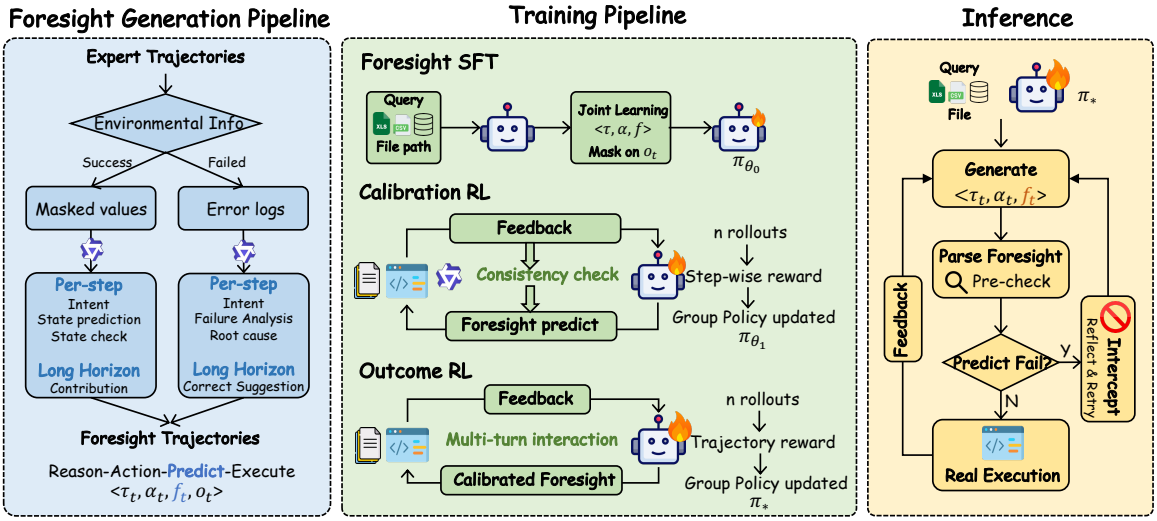

Figure 2: Overview of the DataSight framework. Left: foresight trajectory construction with dual-track annotation. Middle: foresight-augmented training paradigm. Right: Predict-and-Intercept (PnI) inference, where the agent's foresight is parsed before execution and likely-failing actions are intercepted and regenerated.

2024; Gu et al., 2024); **Parametric integration** methods embed environmental knowledge into policy parameters as a cold-start prior (Yu et al., 2026; Zhang et al., 2025a) or through on-policy distillation (Hübotter et al., 2026; Wang et al., 2026; Li et al., 2026a; Ye et al., 2026). However, these methods either decouple environmental prediction from policy execution or implicitly compress it into parameters, making anticipatory reasoning neither directly optimizable nor accessible at inference time. DataSight addresses both limitations by externalizing foresight as an explicit prediction field f_t jointly generated with each action, enabling direct training via a step-level consistency reward and direct inference-time interception of predicted failures.

3 Method

3.1 Problem Formulation

We formulate the data analysis task as a goal-oriented interaction $u = \langle q, \mathcal{F}, a \rangle$, where q is the user query, $\mathcal{F}$ is a set of data files, and a is the target answer. An LLM agent π_θ interacts with a Python interpreter through Thought-Action-Observation cycles, generating reasoning τ_t and code action α_t at each step t, with observation o_t as execution feedback. We denote the input context as $x = (q, \mathcal{F})$ and the complete trajectory as $y \sim \pi_\theta(\cdot \mid x)$.

Standard training operates over trajectories of (τ_t, α_t, o_t) tuples, optimizing the agent to produce correct actions conditioned on prior observations. DataSight extends this by introducing a foresight field f_t at each step, forming $(\tau_t, \alpha_t, f_t, o_t)$. The foresight field f_t is generated by the policy *before* execution occurs, anticipating the execution outcome that o_t will reveal. It encapsulates both step-wise execution feasibility and long-horizon impact assessment of the current action, serving three roles: as a training target during SFT, as a reward anchor during RL, and as a decision signal during inference. An overview of the framework is shown in Figure 2.

3.2 Foresight-Augmented Trajectory Construction

The foresight field must satisfy three constraints: it must be *verifiable* against actual execution outcomes to enable reward computation, *informative* enough to anticipate failures for inference-time interception, and free of *information leakage* of post-execution values to prevent hallucination. We construct foresight-augmented trajectories from 7,273 expert segments in the DataMind dataset (Qiao et al., 2025), using Qwen3-30B-A3B (Yang et al., 2025a) as the teacher model conditioned on the interaction history and real execution feedback.

Successful Steps. We first mask specific numerical values and column names in the interpreter output to prevent information leakage, then provide the masked result to the teacher model. The generated foresight field contains four structured sub-fields: `intent` (action purpose), `state_prediction` (predicted logical state changes in future tense, without concrete values), `contribution` (how this step

advances the overall goal), and `status_check` (explicit confirmation of execution feasibility). The `status_check` field serves as an anchor for downstream consistency evaluation during RL and inference-time interception.

Failed Steps. The teacher model receives the actual error log and generates a foresight field with: `intent` (intended action logic), `failure_analysis` (categorized diagnosis mapping errors to specific code lines), `root_cause` (concise failure summary), and `correction_suggestion` (actionable fix for the next step).

3.3 Foresight Training

Foresight SFT. We fine-tune a base LLM on the constructed expert demonstrations $\mathcal{D}_{\text{foresight}}$ to obtain an initial policy π_0. During training, we apply a completion mask to the actual interpreter returns, forcing the model to concurrently learn executable code generation and environmental impact prediction. This warm-up phase establishes the foundational capability to produce functional code within structured tags, followed by a foresight prediction field, and to manage multi-turn interaction between internal reasoning, code generation, foresight, and environmental feedback.

Dual-Stage Predictive Calibration. Since SFT data primarily consists of expert demonstrations, the policy needs to align its own generated actions with corresponding state predictions. A direct transition to outcome-only RL would leave this miscalibration uncorrected, as the reward provides no gradient on the quality of individual predictions. We therefore implement a two-stage RL process using GRPO (Guo et al., 2025) to progressively calibrate foresight and task performance.

The first stage (**Calibration RL**) jointly optimizes action quality and step-wise prediction fidelity. We design a logic consistency reward R_{cons} that evaluates the alignment between the model's foresight f_i and the actual interpreter observation o_i at each turn:

$$R_{\text{cons}} = \frac{\sum_{i=1}^{T} \mathbb{I}(\text{logic_match}(f_i, o_i))}{T} \tag{1}$$

where $\mathbb{I}(\cdot)$ is the indicator function and T is the total number of interaction turns. $\texttt{logic_match}(f_i, o_i)$ is implemented as a binary classification by an external LLM judge. The Stage 1 reward combines both signals: $R_1 = R_{\text{acc}} + R_{\text{cons}}$.

The second stage (**Outcome RL**) retains only the accuracy reward $R_2 = R_{\text{acc}}$, releasing the prediction constraint to allow the policy to pursue long-horizon task success upon the calibrated world model. The accuracy reward incorporates a length penalty to encourage concise trajectories:

$$R_{\text{acc}} = \begin{cases} 1, & \text{if correct and } l \leq L \\ \max\left(0.5, 1 - \frac{l-L}{L}\right), & \text{if correct and } l > L \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where l is the trajectory length and L is a predefined length bound. We update the policy parameters θ by optimizing the GRPO objective with a clipped objective:

$$L(\theta) = -\mathbb{E}\left[\min\left(\rho_\theta A, \text{clip}(\rho_\theta, 1-\epsilon, 1+\epsilon)A\right)\right] \tag{3}$$

where $\rho_\theta = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ is the importance sampling ratio, and A represents the relative advantage calculated within each group of sampled outputs.

3.4 Predict-and-Intercept Framework

During inference, after the agent generates an action α_t, the system parses the accompanying foresight field f_t to evaluate predicted executability. If f_t indicates a potential failure or does not confirm successful execution, the action is intercepted before any interpreter call and the agent regenerates with additional reflection. This loop continues until either a success is predicted or a maximum of k regeneration attempts is reached.

Unlike standard retry mechanisms, interception happens *before* execution, blocking erroneous observations from polluting dialogue history. This preemptive correction avoids the reactive pathology of executing, failing, observing and then recovering.

4 Experiments

4.1 Experimental Setup

Implementation Details. We implement DataSight on Qwen3-1.7B, 4B, and 8B (Yang et al., 2025a) backbones using the DataMind-12k (Qiao et al., 2025) dataset. SFT is conducted with LlamaFactory (Zheng et al., 2024) and RL with verl (Sheng et al., 2024). Following DAPO (Yu et al., 2025), we disable KL divergence and set $\epsilon_{\text{high}} = 0.28$, $\epsilon_{\text{low}} = 0.2$. Inference is capped at 10 interaction rounds with $top_p = 0.95$ and $temperature = 0.7$. Full hyperparameters

Method	TableBench		DABench		FinQA		DSbench		Average	
	Avg@3	Pass@3	Avg@3	Pass@3	Avg@3	Pass@3	Avg@3	Pass@3	Avg@3	Pass@3
Deepseek-V3.2	67.75	75.96	90.91	96.89	70.79	80.03	39.02	49.36	67.12	75.56
Kimi-K2	72.13	79.57	92.04	96.89	53.27	74.54	36.27	46.50	63.43	74.38
Qwen-2.5-7B-Coder	43.75	62.08	49.81	78.99	17.79	30.60	12.30	18.67	30.90	47.59
TableLLM-8b	3.87	9.37	6.61	14.40	10.70	23.45	11.44	24.46	8.16	17.92
TableGPT2-7B	37.02	62.08	59.15	83.27	21.47	44.64	14.16	26.80	32.95	54.20
Table-R1-zero-8B	19.41	35.78	61.48	72.76	19.58	30.17	16.74	27.04	29.30	41.44
DataMind-7B	66.85	78.56	81.43	95.33	48.10	74.46	16.53	29.83	53.23	69.55
DeepAnalyse-8B	62.98	76.41	77.47	91.05	49.91	66.87	36.52	45.71	56.72	70.01
Base: Qwen3-1.7B										
ReAct	18.45	36.57	31.39	59.53	21.42	39.41	9.86	19.20	20.28	38.68
Foresight-SFT	45.90	63.77	65.89	81.71	24.82	48.47	10.30	22.10	36.73	54.01
Calibration-RL	52.60	69.41	70.69	86.38	36.59	62.69	15.67	28.33	43.89	61.70
Outcome-RL	54.97	69.41	74.65	89.11	40.01	65.65	17.81	30.26	46.86	63.61
Outcome-RL + PnI	**55.42**	**69.53**	**75.62**	**89.49**	**41.18**	**66.00**	**20.10**	**30.69**	**48.08**	**63.93**
Base: Qwen3-4B										
ReAct	40.58	62.64	62.78	86.38	41.35	60.07	12.96	24.47	39.42	58.39
Foresight-SFT	58.56	71.14	82.75	91.83	43.16	67.31	21.46	31.33	51.48	65.40
Calibration-RL	64.48	76.07	85.08	93.00	50.51	72.80	25.46	38.41	56.38	70.07
Outcome-RL	65.69	77.43	86.14	93.77	54.81	75.76	27.77	40.77	58.60	71.93
Outcome-RL + PnI	**67.30**	**78.44**	**86.27**	**93.77**	**56.18**	**76.29**	**30.54**	**42.49**	**60.07**	**72.75**
Base: Qwen3-8B										
ReAct	63.66	72.23	80.54	88.01	45.91	64.86	23.82	35.83	53.48	65.23
Foresight-SFT	67.23	77.54	87.55	95.33	56.06	77.24	24.39	34.33	58.81	71.11
Calibration-RL	68.80	78.33	**89.62**	95.33	57.75	76.63	25.25	35.84	60.36	71.53
Outcome-RL	69.26	79.46	88.33	95.33	63.27	**79.08**	27.92	39.27	62.20	73.29
Outcome-RL + PnI	**69.42**	**79.68**	88.46	**96.11**	**64.17**	78.38	**34.48**	**45.06**	**64.13**	**74.81**

Table 1: Main results on four benchmarks. Avg@3: average accuracy over 3 runs; Pass@3: pass rate with at least one correct answer over 3 runs. All the results are reproduced in our unified execution environment using officially released model weights. All methods without PnI operate under the identical ReAct interaction protocol: the agent observes execution feedback (including errors) at each turn and may attempt corrective actions in subsequent turns.

and data preparation details are provided in Appendix B.

Evaluation Protocol. We evaluate DataSight across four data-analytic benchmarks: TableBench (Wu et al., 2025), DABench (Hu et al., 2024), FinQA (Chen et al., 2021), and DSBench (Jing et al., 2024), spanning tasks from financial reports to Kaggle competitions. To rigorously assess file-interaction capabilities, we extract tables originally embedded in prompt contexts into standalone data files, requiring the model to retrieve information via code execution. We employ DeepSeek-V3.2 (Liu et al., 2025a) for final answer verification and report Avg@3 and Pass@3 to measure both average accuracy and generative robustness.

Baselines. We compare DataSight against a diverse set of state-of-the-art methodologies: (1) Generalist Frontiers, including Kimi-K2 (Team et al., 2025) and DeepSeek-V3.2 (Liu et al., 2025a) ; (2) Specialized Small Models specifically fine-tuned for data-analytic task: Qwen2.5-7B-coder-Instruct (Hui et al., 2024),TableLLM (Zhang et al., 2025c),TableGPT2 (Su et al., 2024),TableR1 (Yang et al., 2025b),DataMind (Qiao et al., 2025) and DeepAnalyse (Zhang et al., 2025b);

4.2 Main Results

Table 1 summarizes results across four benchmarks. DataSight yields consistent improvements at every training stage and model scale. Trained on merely 7k curated trajectories, DataSight-4B surpasses both DataMind-7B and DeepAnalyse-8B in average Avg@3 by 6.84% and 3.35% respectively. Scaling to 8B, DataSight achieves 64.13% average Avg@3, outperforming DeepAnalyse-8B by 7.41%, with its Pass@3 of 74.81% approaching DeepSeek-V3.2 at 75.56%. On TableBench, DataSight-8B attains a Pass@3 of 79.68%, surpassing all models including frontier APIs. At the 1.7B scale, DataSight elevates the average Avg@3

from 20.28% under untuned ReAct to 48.08%, a gain of over 27 percentage points that demonstrates the effectiveness of internalizing environment dynamics under severely constrained model capacity. Each training stage contributes consistent gains: Foresight-SFT establishes a strong base, Calibration-RL improves prediction fidelity, and Outcome-RL pushes task accuracy to its peak.

PnI yields further improvements on top of Outcome-RL across all scales, with the most pronounced gains on DSBench: +2.29% (1.7B), +2.77% (4B), and +6.56% (8B) in Avg@3. On average across four benchmarks, PnI improves Avg@3 by 1.22%, 1.47%, and 1.93% for 1.7B, 4B, and 8B respectively, with the scaling trend indicating that stronger internalized world models produce more reliable foresight for interception.

4.3 Ablation Study

Model	Config	TableBench		DABench	
		Avg@3	Turns	Avg@3	Turns
1.7B	DataSight	**54.97**	**4.73**	**74.65**	4.25
	w/o Foresight	49.17	6.09	72.91	5.08
	w/o Calibration	52.53	5.03	71.44	**4.20**
4B	DataSight	**65.69**	**4.47**	**86.14**	**3.98**
	w/o Foresight	52.93	7.30	72.24	6.79
	w/o Calibration	60.03	5.15	83.06	4.33

Table 2: Ablation study on foresight prediction field and Calibration-RL, evaluated on Qwen3-1.7B and 4B with the same ReAct interaction protocol without PnI.

Effect of Foresight Prediction Field. As shown in Table 2, removing the foresight prediction field (w/o Foresight) causes the most severe degradation: Avg@3 drops by 5.80% (1.7B) and 12.76% (4B) on TableBench, alongside a marked rise in interaction turns (4.73→6.09 and 4.47→7.30), indicating that without explicit environment-dynamics modeling, the model resorts to inefficient trial-and-error exploration. Notably, in Stage 2 where the reward only reflects task accuracy(Figure 3), our method still achieves significantly higher reward than variants without foresight. This indicates that foresight is not merely an auxiliary signal, but contributes to learning a stronger action policy.

Effect of Calibration-RL. Removing Calibration RL (w/o Calibration) incurs a milder but consistent drop of 2.44% and 5.66% on 1.7B and 4B respectively on TableBench. Figure 4 reveals the underlying cause: without the foresight consistency reward, per-step predict scores remain consistently low throughout training, whereas DataSight maintains high predict scores from Stage 1 and sustains them into Stage 2. This persistence indicates that step-level prediction fidelity, once established, is preserved even after the explicit prediction reward is removed. Table 3 further reveals a critical trade-off. Retaining R_{cons} throughout both stages yields higher action-level F1 (93.14 vs. 88.23) but lower trajectory-level F1 (62.58 vs. 68.45) and task accuracy (Avg@3: 51.27 vs. 54.81). Persistent prediction supervision drives the model toward conservative, easily-predictable actions, sacrificing the exploratory steps necessary for long-horizon task success. Releasing R_{cons} in Stage 2 frees the policy to leverage its calibrated world model for more ambitious exploration, recovering both trajectory-level coherence and downstream performance.

Metric	Stage 2	Precision	Recall	F1
Trajectory level	Retain R_{cons}	45.83	98.61	62.58
	Release R_{cons}	52.05	99.94	68.45
Action level	Retain R_{cons}	88.40	98.42	93.14
	Release R_{cons}	79.24	99.52	88.23

Table 3: Foresight prediction quality under two calibration strategies with Qwen3-4B on FinQA. Trajectory level measures alignment between predictions and final task outcomes; Action level measures per-step prediction consistency with interpreter feedback.

5 Analysis

Having established that DataSight improves accuracy and efficiency, we now ask three deeper questions about what foresight training learns.

SFT Configuration	TableBench	DABench	Avg	Tokens
(a) Action only	47.62	64.33	55.98	2,188
(b) + Interpreter	45.67	**66.01**	55.84	2,030
(c) + Think + Pred.	47.63	65.50	56.57	1,987
(d) + Pred. (Ours)	**50.34**	64.98	**57.66**	2,033
(e) +Shuffled Pred.	46.82	61.07	53.95	2,195
(f) status only	48.29	62.36	55.33	2,067
(g) w/o contribution	48.01	65.11	56.56	2,141

Table 4: SFT output structure and foresight sub-field ablation on Qwen3-1.7B. Predict fields are suppressed at inference for fair comparison.

How is foresight internalized into parameters? Table 4 reports results of Qwen3-1.7B fine-tuned on 7,273 trajectories for one epoch. Adding raw interpreter outputs (b) introduces hallucination

Figure 3: Training dynamics of entropy, mean reward (Stage II only), and rollout token length under different training configurations. DataSight maintains higher entropy, higher reward, and shorter rollout length across stages.

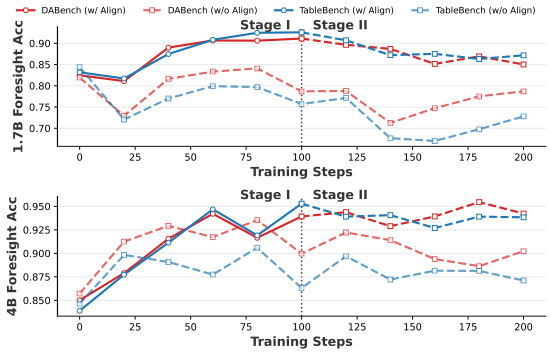

Figure 4: Foresight dynamics during two-stage RL training on Qwen3-1.7B and Qwen3-4B. w/ Align: full DataSight pipeline; w/o Align: without Calibration-RL.

DS-1000	Matplotlib	Numpy	Pandas	Scipy
1.7B SFT w/ Action	34.2	2.7	5.2	8.5
1.7B DataSight	$51.0^{+16.8}$	$5.5^{+2.8}$	$3.8^{-1.4}$	$6.6^{-1.9}$
4B SFT w/ Action	45.2	21.4	13.4	15.1
4B DataSight	$58.1^{+12.9}$	$21.8^{+0.4}$	$17.9^{+4.5}$	$20.8^{+5.7}$
8B SFT w/ Action	56.8	30.5	18.6	19.8
8B DataSight	$60.6^{+3.8}$	$27.3^{-3.3}$	$19.2^{+0.6}$	$28.3^{+8.5}$

Table 5: DS-1000 single-step code generation results.

on unseen data, slightly degrading performance. Foresight-SFT (d) outperforms the think+predict variant (c) by 1.09%, suggesting that the verbose think field dilutes foresight learning.Shuffling foresight labels across samples (e) drops to 53.95%, confirming the model leverages semantic content rather than structural regularization. Sub-field ablations show that both status alone (f, 55.33%) and removing contribution (g, 56.56%) degrade performance, indicating each sub-field contributes. Notably, all foresight variants maintain comparable token consumption to the action-only baseline, confirming that environment dynamics are internalized into parameters rather than encoded as inference-time overhead. After RL, this translates to lower inference cost: DataSight reduces rollout length from 3,034 to 2,837 tokens (TableBench) and 2,685 to 2,477 (DABench) despite generating additional foresight fields, as fewer interaction turns more than compensate for per-step overhead.

To further test whether this internalization improves fundamental code quality, we evaluate on DS-1000 (Lai et al., 2023), where each task requires a standalone code snippet without multi-turn interaction (Table 5). DataSight achieves cumulative gains of +16.3%, +23.5%, and +9.6% across data science libraries for 1.7B, 4B, and 8B models respectively, confirming that foresight training improves the quality of individual code generation, not only multi-turn strategy.

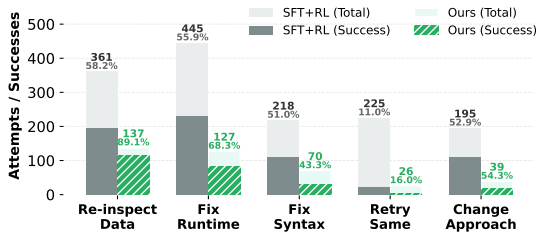

Figure 5: Recover Strategy Analysis on Qwen3-1.7B. Foresight training encourages more informed recovery strategies rather than blind retries.

Beyond single-turn code quality, we examine how foresight training improves robustness in multi-turn interactions. Using DeepSeek-V3 (Liu et al., 2025a), we classify recovery behaviors into five categories (Figure 5). Foresight training leads to more effective self-correction. Notably, DataSight strongly favors *Re-inspect Data*, achieving a success rate of **89.1%** (vs. 58.2% for SFT+RL). Meanwhile, reliance on *Retry Same* is substantially reduced (16% → 6%), indicating fewer redundant retries. These results suggest that modeling environment dynamics enables the agent to adopt more informed recovery strategies, leading to more robust and efficient error resolution.

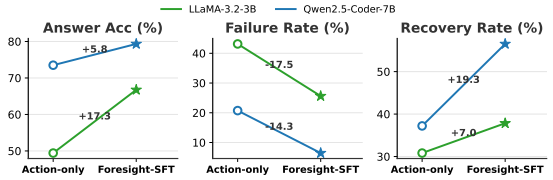

Figure 6: Foresight-SFT generalization to non-thinking models on DABench with Avg@3 metric. ◯ and ★ denote action-only SFT and Foresight-SFT respectively.

Does Foresight Training Generalize Across Backbones? We apply Foresight-SFT to LLaMA-3.2-3B-Instruct (Grattafiori et al., 2024) and Qwen2.5-7B-Coder (Hui et al., 2024) using trajectories constructed by Qwen3-30B-A3B (Yang et al., 2025a). As shown in Figure 6, both models yield more substantial accuracy gains than thinking-capable backbones, with LLaMA-3.2-3B-Instruct and Qwen2.5-7B-Coder improving by +17.33% and +5.83% on DABench Avg@3 respectively. Interpreter failure rate drops by 17.53 and 14.28 percentage points while recovery rate increases by 7.02 and 19.31 percentage points, confirming that foresight training effectively instills environment dynamics understanding independent of long chain-of-thought pretraining, and that Qwen3-constructed trajectories transfer well across diverse backbone families.

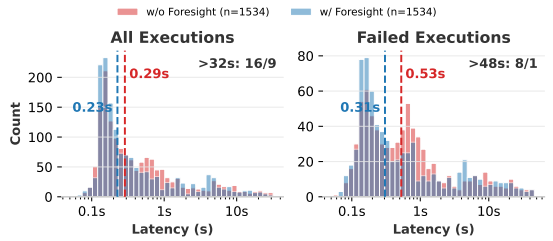

Figure 7: Interpreter execution latency distribution on BIRD (Qwen3-8B). Left: all executions; Right: failed executions. Dashed lines indicate median latency.

Does Foresight Training Transfer Across Execution Environments? We construct 4,721 trajectories for SQLite-based tasks and evaluate on the BIRD (Li et al., 2023) benchmark, where the agent autonomously retrieves database schemas, executes multi-turn SQL queries and saves final results to CSV files for evaluation. We fine-tune Qwen3-8B with foresight and action-only trajectories respectively, yielding average accuracy of 54.23% and 51.17%. As shown in Figure 7, foresight training reduces median execution latency from 0.29s to 0.23s and substantially eliminates high-latency outlier executions (16 vs. 9 exceeding 32s). The effect is most pronounced for failed executions (median: 0.53s → 0.31s, outliers exceeding 48s: 8 → 1), indicating that internalized environment dynamics help the model avoid issuing costly ill-formed queries. This demonstrates that foresight training transfers beyond Python-based data analysis to SQL environments, with benefits in both accuracy and computational efficiency.

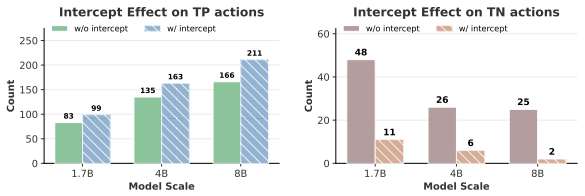

Figure 8: PnI impact on DSBench. TP: actions with correctly predicted success in tasks ultimately solved. TN: actions with correctly predicted failure in tasks ultimately unsolved.

How Effective Is Predict-and-Intercept? As shown in Figure 8, PnI increases TP count from 83 to 99 on 1.7B, 135 to 163 on 4B, and 166 to 211 on 8B, while TN count drops from 48 to 11, 26 to 6, and 25 to 2 respectively. Interception effectively shifts trajectory outcomes toward success by catching likely-failing actions before execution. More analysis is provided in Appendix F.

6 Conclusion

We presented DataSight, a training and inference framework that internalizes environment dynamics into data-analytic agents through foresight-augmented trajectory construction, dual-stage predictive calibration, and predict-and-intercept inference. Behavioral analysis reveals that conventional action-only training produces agents that recover better from more frequent failures rather than preempting them, a pathology that DataSight directly addresses by making anticipatory reasoning an explicit, trainable, and actionable output. Experiments across four benchmarks demonstrate substantial improvements over base models, with gains of over 27 percentage points on Qwen3-1.7B and 10 percentage points on Qwen3-8B relative to untuned ReAct baselines. Foresight training further transfers effectively to non-reasoning backbones and SQL-based tasks, confirming the generality of the approach.

Limitations

Our work has several limitations. First, the foresight consistency reward R_{cons} relies on an LLM-based judge to evaluate alignment between predictions and execution outcomes, introducing both computational overhead and potential evaluation noise during RL training. To assess judge reliability, we manually verified 500 randomly sampled (f_i, o_i) pairs and observed 97% agreement with human annotations (Cohen's κ = 0.94), confirming that judge noise is negligible for reward signal construction. Developing more efficient and deterministic consistency metrics would reduce this dependency. Second, our evaluation focuses on data analysis tasks with Python and SQL execution environments; whether foresight training yields similar benefits in other agentic domains such as web navigation or API orchestration remains to be validated. Third, our trajectory construction relies on a strong teacher model (Qwen3-30B-A3B) to generate foresight annotations, and the quality of these annotations likely bounds the effectiveness of downstream training. Investigating self-generated foresight without teacher supervision is a promising direction for future work.

Ethics Statement

All datasets used in this work are derived from publicly available benchmarks and open-source resources. Our framework does not involve human subjects, private data, or sensitive information. We acknowledge that AI-assisted tools were used for grammar checking and formatting during the writing process; all technical content, analysis, and conclusions represent the original work of the authors.

References

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, and 1 others. 2025. Scaling agent learning via experience synthesis. *arXiv preprint arXiv:2511.03773*.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan R Routledge, and 1 others. 2021. Finqa: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3697–3711.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, and 1 others. 2025. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Yang Gu, Hengyu You, Jian Cao, Muran Yu, Haoran Fan, and Shiyou Qian. 2025. Large language models for constructing and optimizing machine learning workflows: A survey. *ACM Transactions on Software Engineering and Methodology*.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, and 1 others. 2024. Is your llm secretly a world model of the internet? model-based planning for web agents. *arXiv preprint arXiv:2411.06559*.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nat.*, 645(8081):633–638.

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

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, and 1 others. 2025. Data interpreter: An llm agent for data science. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19796–19821.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, and 1 others. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507*.

Jonas Hübotter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, and 1 others. 2026. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Jiaze Li, Hao Yin, Haoran Xu, Boshen Xu, Wenhui Tan, Zewen He, Jianzhong Ju, Zhenbo Luo, and Jian Luan. 2026a. Video-opd: Efficient post-training of multimodal large language models for temporal video grounding via on-policy distillation. *arXiv preprint arXiv:2602.02994*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

Shuocheng Li, Yihao Liu, Silin Du, Wenxuan Zeng, Zhe Xu, Mengyu Zhou, Yeye He, Haoyu Dong, Shi Han, and Dongmei Zhang. 2026b. Jupiter: Enhancing llm data analysis capabilities via notebook and inference-time value-guided search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 31726–31733.

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Pony Ma, Guanhua Chen, and Heng Ji. 2025. From word to world: Can large language models be implicit text-based world models? *arXiv preprint arXiv:2512.18832*.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, Wanjun Zhong, Wangchunshu Zhou, Wenhao Huang, and Ge Zhang. 2024. Autokaggle: A multi-agent framework for autonomous data science competitions. *CoRR*, abs/2410.20424.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Youwei Liu, Jian Wang, Hanlin Wang, Beichen Guo, and Wenjie Li. 2026. Imagine-then-plan: Agent learning from adaptive lookahead with world models. *arXiv preprint arXiv:2601.08955*.

Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Siheng Chen, and 1 others. 2025b. Ml-master: Towards ai-for-ai via integration of exploration and reasoning. *arXiv preprint arXiv:2506.16499*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, and Tomas Pfister. 2025. Ds-star: Data science agent via iterative planning and verification. *arXiv preprint arXiv:2509.21825*.

Shuofei Qiao, Runnan Fang, Ningyu Zhang, Yuqi Zhu, Xiang Chen, Shumin Deng, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. 2024. Agent planning with world knowledge model. *Advances in Neural Information Processing Systems*, 37:114843–114871.

Shuofei Qiao, Yanqiu Zhao, Zhisong Qiu, Xiaobin Wang, Jintian Zhang, Zhao Bin, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and 1 others. 2025. Scaling generalist data-analytic agents. *arXiv preprint arXiv:2509.25084*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

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.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Gang Chen, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, and 1 others. 2024. Tablegpt2: A large multimodal model with tabular data integration. *arXiv preprint arXiv:2411.02059*.

Ji Sun, Guoliang Li, Peiyao Zhou, Yihui Ma, Jingzhe Xu, and Yuan Li. 2025. Agenticdata: An agentic data analytics system for heterogeneous data. *arXiv preprint arXiv:2508.05002*.

Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. 2026. Openclaw-rl: Train any agent simply by talking. *arXiv preprint arXiv:2603.10165*.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, and 1 others. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25497–25506.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Zheyuan Yang, Lyuhao Chen, Arman Cohan, and Yilun Zhao. 2025b. Table-r1: Inference-time scaling for table reasoning. *arXiv preprint arXiv:2505.23621*.

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*.

Tianzhu Ye, Li Dong, Xun Wu, Shaohan Huang, and Furu Wei. 2026. On-policy context distillation for language models. *arXiv preprint arXiv:2602.12275*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Xiao Yu, Baolin Peng, Ruize Xu, Yelong Shen, Pengcheng He, Suman Nath, Nikhil Singh, Jiangfeng Gao, and Zhou Yu. 2026. Reinforcement world model learning for llm-based agents. *arXiv preprint arXiv:2602.05842*.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, and 1 others. 2026. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, and 1 others. 2025a. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*.

Shaolei Zhang, Ju Fan, Meihao Fan, Guoliang Li, and Xiaoyong Du. 2025b. Deepanalyze: Agentic large language models for autonomous data science. *arXiv preprint arXiv:2510.16872*.

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

Xiaokang Zhang, Sijia Luo, Bohan Zhang, Zeyao Ma, Jing Zhang, Yang Li, Guanlin Li, Zijun Yao, Kangli Xu, Jinchang Zhou, and 1 others. 2025c. 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.

Jingsheng Zheng, Jintian Zhang, Yujie Luo, Yuren Mao, Yunjun Gao, Lun Du, Huajun Chen, and Ningyu Zhang. 2026. Can we predict before executing machine learning agents? *arXiv preprint arXiv:2601.05930*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Appendix

A Algorithm Pipeline

The complete procedure is summarized in Algorithm 1.

Algorithm 1 DataSight Training

Require: Base LLM π_{base}, Expert dataset $\mathcal{D}_{\text{foresight}}$, Environment $\mathcal{E}$, Input File $\mathcal{F}$
Ensure: Optimized Policy π_*

```
// Phase 1: Foresight SFT
π_{θ_0} ← SFT(π_base, D_foresight)   ▷ Joint learning of ⟨τ, α, f⟩
// Phase 2: Dual-Stage Reinforcement Learning
for stage s ∈ {1, 2} do
    for iteration t = 1, ..., T do
        H_0 ← {q, F}, C_0 ← ∅
        for step k = 1, ..., K do
            (τ_k, α_k, f_k) ~ π_{θ_t}(· | H_k)   ▷ Generate reasoning, action, foresight
            C_k ← C_{k−1} ⊕ α_k
            o_k ← E(C_k)   ▷ Execute and observe
            H_{k+1} ← H_k ∪ {(τ_k, α_k, f_k, o_k)}
            if o_k is Failure then
                C_k ← C_{k−1}   ▷ Rollback erroneous code
            end if
            if α_k ∈ FinalAnswer or k = K then
                break
            end if
        end for
    // Reward Computation
        R_cons ← (1/K) Σ_{k=1}^{K} I(logic_match(f_k, o_k))   ▷ Eq. 1
        R_acc ← I(correct) · r_len   ▷ Eq. 2
        if s = 1 then   ▷ Calibration RL
            R ← R_acc + R_cons
        else   ▷ Outcome RL
            R ← R_acc
        end if
        θ_{t+1} ← GRPO(θ_t, R)   ▷ Eq. 3
    end for
end for
π_* ← π_{θ_T}
```

B Implementation Details

Data Preparation. We extract 7,273 multi-turn interaction trajectories from the DataMind-12k dataset (Qiao et al., 2025), retaining the original Kaggle-sourced files and corresponding questions. For the dual-stage RL training, we filter out samples whose ground-truth answers are descriptive long-form reports, as these are prone to misjudgment by LLM-based reward models. This yields 6,328 trajectories for RL training. The remaining 945 filtered samples are used exclusively for warm-up SFT on the Qwen3-1.7B model (2 epochs), while the 4B and 8B models are trained on the full 7,273 trajectories for 2 epochs to better learn the interaction format.

SFT Configuration. We use LlamaFactory (Zheng et al., 2024) for SFT training with a learning rate of 1×10^{-5}, warmup ratio of 0.1, and cosine decay schedule.

Hyperparameter	Value
RL Training	
Train Batch Size	64
PPO Mini-Batch Size	16
PPO Micro-Batch Size (per GPU)	4
Rollout Group Size N	4
Rollout Engine	sglang
Training Epochs	2
Learning Rate	1×10^{-6}
LR Warmup Steps	20
Weight Decay	0.1
Grad Clip	1.0
Loss Aggregation	token-mean
Clip Ratio Low	0.2
Clip Ratio High	0.28
Use KL Loss	False
KL Coef	0.0
Max Turns	10
Sequence Lengths	
Max Prompt Length	2048
Max Response Length	8192
Max Observation Length	2048
Max Answer Length(length bound L)	1024
Sampling	
Training Temperature	0.7
Training top-p	1.0
Validation Temperature	0.7
Validation top-p	0.95

Table 6: RL training hyperparameters for Qwen3-1.7B and Qwen3-4B. For Qwen3-8B, batch size is increased to 128.

RL Configuration. We use the VeRL framework (Sheng et al., 2024) for reinforcement learning. Following DAPO (Yu et al., 2025), we set clipping thresholds $\epsilon_{\text{high}} = 0.28$ and $\epsilon_{\text{low}} = 0.2$, with KL divergence disabled to encourage exploration. Each prompt generates $N = 4$ rollout trajectories for group-relative advantage estimation. Training proceeds for one epoch for each stage(99 steps) with a learning rate of 1×10^{-6}, warmup of 20 steps, weight decay of 0.1, and train batch size of 64. For both training and validation, we use temperature = 0.7 and validation top-p = 0.95. Maximum interaction turns are capped at 10. Full hyperparameters are listed in Table 6.

Hardware. For Qwen3-1.7B and 4B models, we train on 2× NVIDIA RTX PRO 6000 (96GB) Blackwell Server Edition GPUs. For the 8B model, we use 8× Ascend 910B (64GB) with batch size increased to 128 for improved GPU utilization.

Inference Metrics. Let $\mathcal{F}_\tau = \{t \in \tau : \texttt{exec}(a_t) \neq \texttt{success}\}$ denote the failed steps in trajectory τ. We define:

$$\text{Failure Rate} = \frac{\sum_\tau |\mathcal{F}_\tau|}{\sum_\tau |\tau|} \tag{4}$$

$$\text{Recovery Rate} = \frac{|\{\tau : \mathcal{F}_\tau \neq \emptyset \ \wedge \ \texttt{correct}(\tau)\}|}{|\{\tau : \mathcal{F}_\tau \neq \emptyset\}|} \tag{5}$$

Agentic Code-based Rollout To ensure stability and efficiency during online rollout, we implement an asynchronous execution which prevents synchronized spikes in file I/O and computational load. For each task, we maintain an isolated execution directory that is purged upon completion to ensure environment cleanliness.

In terms of trajectory management, we only append successfully executed code snippets to the historical code context. If the agent fails to produce valid code or a final answer, we trigger a regenerated attempt via prompting.

C Dataset and Benchmarks

DataMind-12K DataMind-12K is a high-quality trajectory dataset for data-analytic agent training (Qiao et al., 2025). It is constructed through a scalable synthesis pipeline that first collects diverse data files (.csv, .xlsx, .sqlite) from public sources such as Kaggle and existing Text-to-SQL benchmarks, then generates queries across 18 fine-grained task categories using a recursive easy-to-hard composition mechanism. Expert trajectories are sampled using DeepSeek-V3.1 (Liu et al., 2024), followed by self-consistency filtering with GPT-4o-mini and rule-based quality checks. The final dataset comprises 11,707 high-quality trajectories spanning multiple domains, task types, and file formats. Experiments confirm that DataMind-12K has no overlap with any of the test benchmarks.

TableBench TableBench (Wu et al., 2025) is a real-world table question answering benchmark containing 886 human-annotated samples across 18 subcategories. In our evaluation, we extract the tabular data from each instance and save it as a separate file, allowing the agent to autonomously explore the table through code-based environment interactions rather than relying on prompt-embedded table content.

DABench InfiAgent-DABench (Hu et al., 2024) is the first benchmark designed to evaluate LLM-based agents on data analysis tasks. It contains DAEval, a dataset comprising 257 data analysis questions derived from 52 real-world CSV files, along with an agent framework for evaluation.

FinQA FinQA (Chen et al., 2021) is a dataset for numerical reasoning over financial reports, containing 8,281 expert-annotated QA pairs (1,147 in the test set). In our evaluation, we extract each accompanying table from the financial report and save it as a separate CSV file, allowing the agent to autonomously explore and query the tabular data through code-based environment interactions.

DSBench DSBench (Jing et al., 2024) is a comprehensive data science benchmark designed to evaluate data science agents on realistic tasks. It contains 466 data analysis tasks sourced from ModelOff competitions, which require agents to answer financial data analysis questions by fully understanding the data and question intent. In our evaluation, we focus specifically on the data analysis portion of DSBench, where we assess the agent's ability to correctly answer these questions.

DS-1000 DS-1000 (Lai et al., 2023) is a code generation benchmark with 1,000 data science problems spanning seven Python libraries, sourced from StackOverflow. It features realistic problem contexts, multi-criteria execution-based evaluation, and proactive perturbation to defend against memorization. Our evaluation focuses on its data analysis capabilities, specifically on the four core libraries: Pandas, NumPy, Matplotlib, and SciPy.

BIRD BIRD (Li et al., 2023) is a large-scale text-to-SQL benchmark designed for real-world database scenarios. The test set contains 1,789 text-to-SQL pairs across 15 databases, with a total database size of 33.4 GB spanning 37 professional domains. It emphasizes database value comprehension, external knowledge reasoning and SQL execution efficiency.

D Foresight Trajectory Construction.

We constructed a high-quality trajectory dataset consisting of 7,273 foresight-augmented instances. This dataset encompasses a total of 19,661 successful interaction turns and 407 failures deliberately retained or simulated to provide diverse negative supervision.

As illustrated in Figure 9, the interpreter failures within these trajectories are categorized into two primary types to facilitate robust error-learning:

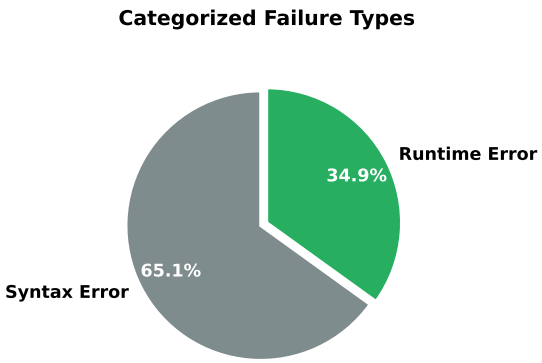

Figure 9: Taxonomy of Simulated Failures in Expert Trajectories.

(i) *Syntax Errors*, which involve structural code flaws such as indentation and grammatical mistakes, and (ii) *Runtime Errors*, which include `ValueError`, `KeyError`, `AttributeError`, and other logic-related exceptions that occur during actual execution. By grounding the `<predict>` fields in both successful and erroneous outcomes, the dataset forces the model to internalize the causal relationship between its actions and environmental feedback. A detailed raw instance showcasing the structure of these foresight labels is provided in Case Study D.

Case: Foresight Trajectory

```
[USER QUESTION]
Does the treatment duration have a causal effect on the
      ↪ treatment outcome for Typhoid Fever patients,
      ↪ when controlling for factors like age, symptoms
      ↪ severity, and current medication?

<think>
To answer this question, I need to analyze whether
      ↪ treatment duration has a causal effect on
      ↪ treatment outcome... Let me start by loading and
      ↪  inspecting the data.
</think>

<code>
import pandas as pd
df = pd.read_csv('data/files/DRP-for-Typhoid-Fever.csv')
print("Columns in the dataset:")
print(df.columns.tolist())
print("\nFirst 5 rows:")
print(df.head())
print("\nData types:")
print(df.dtypes)
</code>

<predict>
    <intent>Load and inspect the dataset to understand its
      ↪  structure and variables.</intent>
    <state_prediction>
        This should create a dataframe containing the
      ↪ dataset and output the column names, first few
      ↪ rows, and data types to the console.
    </state_prediction>
    <contribution>
        Now that the data structure is known, we can
      ↪ identify relevant variables for analysis and
      ↪ determine if preprocessing is needed before
      ↪ modeling.
    </contribution>
    <status_check>
        Logic appears valid. Expected to succeed.
    </status_check>
</predict>

<interpreter>
The code run successfully:
Columns in the dataset:
['Patient ID', 'Age', 'Gender', 'Symptoms Severity', ...,
      ↪ 'Treatment Duration', 'Treatment Outcome']
...
</interpreter>

<think>
I notice that Treatment Duration, Symptoms Severity, and
      ↪ Treatment Outcome are stored as objects (likely
      ↪ strings), so I need to preprocess them... Let me
      ↪  first examine the unique values.
</think>

<code>
print("Unique values in Treatment Duration:")
print(df['Treatment Duration'].unique())
...
</code>

<predict>
    <intent>Examining unique values in categorical
      ↪ variables to determine encoding strategies</
      ↪ intent>
    <state_prediction>
        This should display the unique string values
      ↪ present in Treatment Duration, Symptoms Severity,
      ↪  Treatment Outcome, and Current Medication
      ↪ columns.
    </state_prediction>
    <contribution>
        Now that we understand the categorical variable
      ↪ distributions, we can proceed with appropriate
      ↪ encoding methods.
    </contribution>
    <status_check>
        Logic appears valid. Expected to succeed.
    </status_check>
</predict>

<interpreter>
The code run successfully:
Unique values in Treatment Duration: ['15 days' '5 days'
      ↪ ...]
Unique values in Symptoms Severity: ['Low' 'Moderate' '
      ↪ High']
...
</interpreter>

<think>
Now I understand the data structure better... Let me
      ↪ preprocess the data accordingly and then perform
      ↪  regression analysis.
</think>

<code>
df['Treatment_Duration_Numeric'] = df['Treatment Duration
      ↪ '].str.extract('(\d+)').astype(int)
severity_mapping = {'Low': 1, 'Moderate': 2, 'High': 3}
df['Symptoms_Severity_Numeric'] = df['Symptoms Severity'].
      ↪ map(severity_mapping)
outcome_mapping = {'Unsuccessful': 0, 'Successful': 1}
df['Treatment_Outcome_Numeric'] = df['Treatment Outcome'].
      ↪ map(outcome_mapping)
medication_dummies = pd.get_dummies(df['Current Medication
      ↪ '], prefix='Medication')
df = pd.concat([df, medication_dummies], axis=1)
features = ['Treatment_Duration_Numeric', 'Age', '
      ↪ Symptoms_Severity_Numeric', '
      ↪ Medication_Amoxicillin', '
      ↪ Medication_Azithromycin', '
      ↪ Medication_Ceftriaxone']
X = df[features]
y = df['Treatment_Outcome_Numeric']
</code>
```

```
<predict>
    <intent>Preprocess categorical variables and create
      ↪ feature matrix for regression analysis</intent>
    <state_prediction>
        This should create new numeric columns for
      ↪ treatment duration, symptoms severity, and
      ↪ treatment outcome, while generating dummy
      ↪ variables for current medication.
    </state_prediction>
    <contribution>
        Now that the data is properly encoded, we can
      ↪ perform regression analysis.
    </contribution>
    <status_check>
        Logic appears valid. Expected to succeed.
    </status_check>
</predict>

<interpreter>
The code run successfully:
Feature matrix shape: (500, 6)
...
Medication_Amoxicillin  True  False ...
</interpreter>

...
<think>
Good! I've converted the boolean columns... Let me fit the
      ↪  logistic regression model again.
</think>

<code>
import statsmodels.api as sm
X_with_const = sm.add_constant(X_processed)
logit_model = sm.Logit(y, X_with_const)
result = logit_model.fit()
</code>

<predict>
    <intent>The code was attempting to fit a logistic
      ↪ regression model to determine the causal effect
      ↪ ... but encountered a singular matrix error due
      ↪ to multicollinearity.</intent>
    <failure_analysis>
        - Is it a Variable issue? Yes. The error "Singular
      ↪  matrix" indicates perfect multicollinearity.
        - Is it a Logic/Type issue? Yes. The one-hot
      ↪ encoded medication variables are perfectly
      ↪ collinear.
    </failure_analysis>
    <root_cause>The one-hot encoded medication variables (
      ↪ Amoxicillin, Azithromycin, Ceftriaxone) are
      ↪ perfectly collinear, creating a singular matrix
      ↪ .</root_cause>
    <correction_suggestion>Drop one of the medication
      ↪ dummy variables (e.g., use drop_first=True) to
      ↪ eliminate multicollinearity.</
      ↪ correction_suggestion>
</predict>

<interpreter>
The code run failed:
numpy.linalg.LinAlgError: Singular matrix
</interpreter>

<think>
The "Singular matrix" error suggests multicollinearity. I
      ↪ should remove one of the medication categories
      ↪ to avoid the dummy variable trap.
</think>

<code>
features_reduced = ['Treatment_Duration_Numeric', 'Age', '
      ↪ Symptoms_Severity_Numeric', '
      ↪ Medication_Amoxicillin', '
      ↪ Medication_Azithromycin']
X_reduced = X_processed[features_reduced]
X_with_const = sm.add_constant(X_reduced)
logit_model = sm.Logit(y, X_with_const)
result = logit_model.fit()
</code>

<predict>
    <intent>The code was attempting to perform logistic
      ↪ regression analysis... The error occurred due to
      ↪  a missing import statement for statsmodels.</
      ↪ intent>
    <failure_analysis>
        - Is it a Variable issue? Yes. 'sm' is not defined
      ↪  in the current scope.
    </failure_analysis>
</predict>

<interpreter>
The code run failed:
NameError: name 'sm' is not defined
</interpreter>

<think>
I need to import statsmodels.api again... Let me fix this
      ↪ and run the analysis again.
</think>

<code>
import statsmodels.api as sm
import numpy as np
X_reduced = X_processed[features_reduced]
X_with_const = sm.add_constant(X_reduced)
logit_model = sm.Logit(y, X_with_const)
result = logit_model.fit()
print(result.summary())
</code>

<predict>
    <intent>Fit a logistic regression model to assess the
      ↪ causal relationship... while controlling for
      ↪ confounding variables.</intent>
    <state_prediction>
        This should successfully fit a logistic regression
      ↪  model with the reduced feature set.
    </state_prediction>
    <status_check>
        Logic appears valid. Expected to succeed.
    </status_check>
</predict>

<interpreter>
The code run successfully:
Optimization terminated successfully.
...
1 Treatment_Duration_Numeric 0.0131 0.6537 1.0132
...
</interpreter>

<think>
Perfect! The logistic regression model has run
      ↪ successfully... All the analyses consistently
      ↪ show that treatment duration does not have a
      ↪ statistically significant effect.
</think>

<answer>
No, the analysis shows that treatment duration does not
      ↪ have a statistically significant causal effect
      ↪ on treatment outcome (p-value = 0.654,
      ↪ correlation = 0.028) when controlling for other
      ↪ factors.
</answer>
```

E Training Analysis

Stage 1 Reward	TableBench	DABench
$R_{\text{acc}} + 0.5 \cdot R_{\text{cons}}$	50.98	**71.98**
$0.5 \cdot R_{\text{acc}} + R_{\text{cons}}$	50.38	71.85
$R_{\text{acc}} + R_{\text{cons}}$ (Ours)	**52.60**	70.69

Table 7: Effect of reward ratio in Stage 1 on Qwen3-1.7B.

Sensitivity to Reward Weight. As shown in Table 7, equal weighting achieves the highest accuracy on TableBench while remaining compet-

itive on DABench, yielding the best overall balance between foresight calibration and task performance. Reducing either weight consistently lowers TableBench accuracy, suggesting that strong calibration pressure is critical for complex table analysis tasks.

F Inference Analysis

Model	Infer	TA	TP	Turns
DataSight-1.7B	React	44.55	83.18	5.01
	+PnI	45.77	83.39	4.62
DataSight-4B	React	58.60	82.99	4.35
	+PnI	60.07	84.48	4.27
DataSight-8B	React	62.20	83.21	4.36
	+PnI	64.13	85.24	4.30

Table 8: Predict-and-Intercept (PnI) evaluation. TA: Task Accuracy; TP: Total Precision (the ratio of actions where both foresight prediction and execution succeeded); Turns: Average interaction turns with the interpreter.

Interception across benchmarks. As shown in Table 8, PnI consistently yields performance gains across all model scales. More importantly, the proportion of successful action-level predictions and executions (TP) significantly increases (e.g., +2.03% for the 8B model), indicating that the model's foresight is not only accurate but also effectively guides the policy to avoid erroneous paths. Furthermore, PnI leads to a reduction in the average number of interactions with the real interpreter (`Turns`), confirming that pre-execution interception filters out flawed actions early in the reasoning chain. While PnI introduces additional overhead for internal token generation, it significantly reduces the costs associated with calling expensive or time-consuming external environments, such as high-fidelity physical simulators or large-scale databases. This efficiency gain proves that foresight-guided inference allows the model to reflect internally, enabling more robust and streamlined interactions with the environment.

Training-Time Gains vs. Inference-Time Retry. Table 9 isolates the contribution of inference-time retry mechanisms from foresight training. Equipping the w/o Foresight model with Sample-k (+0.08%) or Self-Check + Retry (-1.87%) yields negligible or even negative gains, indicating that naive retry strategies cannot compensate for the lack of anticipatory reasoning. The self-check variant even hurts performance by introducing unreliable filtering without calibrated prediction. In contrast, DataSight without PnI already achieves 65.69%, a 12.76% improvement attributable entirely to foresight training under the identical ReAct interaction protocol. PnI provides a further 1.61% gain, confirming that the primary contribution lies in training-time internalization of environment dynamics, not inference-time retry opportunities.

Configuration	Avg@3
w/o Foresight Training	
Standard	52.93
+ Sample-k (Random, k=3)	53.01
+ Self-Check + Retry (k=3)	51.06
DataSight	
w/o PnI	65.69
w/ PnI (k=3)	67.30

Table 9: Ablation on inference-time strategies (4B, TableBench). Retry baselines use the model trained *without* foresight.

Metric	k=0	k=2	k=3	k=4	k=5
Avg@3	26.75	33.40	**34.48**	33.83	34.05
Pass@3	39.91	42.70	45.06	44.85	**45.92**

Table 10: Effect of maximum intercept attempts k on Qwen3-8B accuracy (%) on DSBench. $k = 0$ denotes standard ReAct inference without interception.

Sensitivity to Intercept Attempts. Table 10 shows the effect of maximum intercept attempts k on DSBench. Performance improves substantially from $k = 0$ (no interception) to $k = 3$ (Avg@3: 26.75% $\rightarrow$ 34.48%), confirming that preemptive interception effectively reduces context-polluting failures. Beyond $k = 3$, gains plateau with marginal fluctuation, suggesting that most interceptable failures are resolved within three attempts. We adopt $k = 3$ as the default configuration balancing effectiveness and inference overhead.

CoT vs. Agentic Reasoning. Recent work shows that LLMs possess non-trivial predictive capabilities in text-based environments (Li et al., 2025; Guan et al., 2023; Copet et al., 2025; Zheng et al., 2026; Chen et al., 2025), providing empirical grounding for our foresight training design. However, such capability alone is insufficient. As shown in Table 11, even when CoT is given full table content in context, DataSight consistently outperforms it across all model scales on TableBench. Moreover,

CoT is fundamentally inapplicable to benchmarks like DABench, where data resides in large files that cannot fit in a prompt. This confirms that iterative environmental exploration is not optional but necessary for real-world data analysis.

Model	Method	TableBench
Qwen3-1.7B	CoT	28.89
	DataSight	**55.42**
Qwen3-4B	CoT	60.54
	DataSight	**67.3**
Qwen3-8B	CoT	60.77
	DataSight	**69.42**

Table 11: CoT vs. DataSight on TableBench. CoT embeds table contents directly in prompt context, while DataSight explores data through code execution.

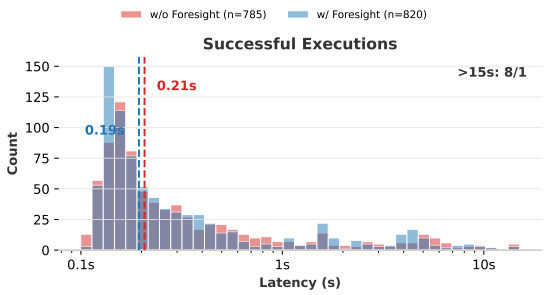

Figure 10: Interpreter execution latency distribution on BIRD. Dashed lines indicate median latency. Foresight training reduces median latency across successful executions.

Successful Execution Latency Analysis Figure 10 shows the latency distribution for successful executions. The median latency difference is smaller (0.21s vs. 0.19s) compared to the overall and failed distributions, which is expected since successful queries are generally well-formed and efficient regardless of training method. However, foresight training still substantially reduces high-latency outliers (8 vs. 1 exceeding 15s), suggesting that even among ultimately correct queries, the action-only model occasionally arrives at valid results through inefficient execution paths that foresight training learns to avoid.

Recovery Strategies Upon receiving error feedback, the model's subsequent response is classified into one of the following five strategies to assess its error-awareness and self-correction efficiency:

Re-inspect Data: The model proactively attempts to examine the data structure by invoking diagnostic commands such as `df.head()`, `df.info()`, or `df.columns`. This strategy indicates a high degree of environment awareness, as the model seeks empirical evidence to resolve ambiguities.

Fix Runtime: The model attempts to resolve runtime exceptions by adding error-handling logic (e.g., `fillna()`), correcting index offsets, or adjusting data type conversions without changing the overall analytical approach.

Fix Syntax: A straightforward correction of typographical or structural errors in the code, such as fixing indentation or closing string literals, to ensure the script becomes executable.

Retry Same: A passive recovery attempt where the model re-submits nearly identical or functionally equivalent code without addressing the underlying cause of the error. This often results in a repetitive failure loop, reflecting low interactive resilience.

Change Approach: A strategic shift where the model abandons its current analytical path in favor of a completely different methodology or tool (e.g., switching from a complex `groupby` operation to a simplified filtering logic).

G Prompt Templates

We design several structured prompts to enable foresight learning and evaluation.

Figure 11 instructs the LLM to predict execution outcomes for successful code actions by simulating environment changes without observing actual results. Figure 12 handles failed actions, guiding the model to analyze error logs and suggest corrections.

To prepare the model for multi-turn interaction, Figure 13 provides a warm-up prompt that trains the model to alternate between acting as an executor and a world model.

For online inference, Figure 14 defines the standard rollout protocol (Qiao et al., 2025), where the model reasons in <think> tags, writes code in <code> tags, and receives execution feedback in <interpreter> tags.

We employ two judge prompts for evaluation. Figure 15 evaluates final answer correctness (Qiao et al., 2025), applying a 3% tolerance for numerical questions and exact match for multiple-choice.

Figure 16 checks consistency between the model's predicted execution outcome and the actual result, returning a binary score.

Finally, Figure 17 introduces an intercept mechanism that triggers re-analysis when a predicted failure is detected, enabling proactive correction before erroneous observations enter the dialogue history. To better guide the base model in interacting with the environment, we prepend this prompt with a multi-turn interaction hint during the first rollout round.

Prompt:Foresight trajectory construction I

```
    You are an expert AI Data Scientist acting as a "World Model".
Your task is to **predict** the execution outcome of the current code snippet based on the
↪ history.

**Context provided:**
1. User Request (Goal)
2. Execution History
3. Current Code (Action)
4. Masked execution result

**Constraint:**
You do NOT see the actual execution result because the code executed successfully.
You must simulate the runtime in your mind.

**Output Requirements (XML):**
<predict>
    <intent>
    What is this code trying to do?
    </intent>
    <state_prediction>
        Predict the logical change in the environment.
        - Use **future/conditional tense** (e.g., "This should create a dataframe...", "It is
↪ expected to filter...").
        - **Do NOT hallucinate** specific numbers or data values.
        - Focus on variable creation, file updates, or data structure changes.
    </state_prediction>
    <contribution>
        How does this specific step move the User's Goal forward?
        (e.g., "Now that data is loaded, we can proceed to cleaning.", "This visualization will
↪  help confirm the trend.")
    </contribution>
    <status_check>
        Since the code is syntactically correct, confirm: "Logic appears valid. Expected to
↪ succeed."
    </status_check>
</predict>
```

Figure 11: Prompt to instruct LLM to generate prediction for successfully-executed action.

Prompt:Foresight trajectory construction II

```
    You are an expert AI Data Scientist acting as a "World Model".
The current code execution has **FAILED**.

**Context provided:**
1. User Request
2. History
3. Current Code
4. **Error Log (Ground Truth)**

**Output Requirements (XML):**
<predict>
    <intent>
    What was the code trying to do?
    </intent>
    <failure_analysis>
        Analyze the specific cause of the failure.
```

```
        - Is it a **File Path** issue? (e.g., file not found)
        - Is it a **Variable** issue? (e.g., undefined name)
        - Is it a **Logic/Type** issue?
        - Map the error back to the specific line of code.
    </failure_analysis>
    <root_cause>
        Briefly state the blocker (e.g., "The file 'data.csv' does not exist in the current
↪ directory.").
    </root_cause>
    <correction_suggestion>
        What should the Agent do differently in the next step to fix this?
    </correction_suggestion>
</predict>
```

Figure 12: Prompt to instruct LLM to generate prediction for action failed to run.

Prompt: Foresight Trajectory Warm-up

```
You are an expert Data Science Agent possessing advanced foresight capabilities.
Your goal is to solve complex data analysis tasks step-by-step while accurately predicting the
↪ consequences of your actions before they occur.

# Role
You act as both the **Actor** (generating code/thought) and the **World Model** (simulating the
↪  environment).

# Interaction Protocol
1. **User Input**:
   - In the first turn, you will receive the **Task Context** and **User Question**.
   - In subsequent turns, you will receive the **Execution Result (Observation)** from the
↪ environment based on your previous code.

2. **Your Output**:
   For every turn, you must output two distinct parts:
   - **Part 1: Action**: Your thought process and the code to execute (wrapped in <code> tags).
   - **Part 2: Prediction**: Immediately analyze what this code *will* do. Predict the data
↪ state changes, potential output formats, or verify logic validity (wrapped in <predict> tags
↪ ).

# Final Goal
Iterate until you can provide the final answer wrapped in <answer> tags.
```

Figure 13: Warm-up prompt for LLM to switch between actor and predictor roles in multi-turn interaction.

Prompt: Rollout Instruction

```
 You are an expert-level data analyst and statistician who solves any data challenge through
↪ rigorous logic, systematic planning, and deep investigation. Your primary task is to answer
↪ user questions by analyzing the provided data source. You can solve the given problem step
↪ by step with utilizing Python code execution (for CSV files) to support your reasoning.

 # Problem-Solving Protocol
```

1. You should think through the problem logically, outlining your reasoning process in <think> and </think> tags.

2. After reasoning, write the appropriate code to execute your plan. Place your code between <code> and </code> tags.

 - For CSV files and excel files, write Python code using libraries like pandas, numpy, sklearn, etc. to analyze the data. If plotting is required, rely on matplotlib.The format should be:

 ```python

 <your python code here>

 ```

3. The execution results will be returned in <interpreter> and </interpreter> tags.

4. Every time you get the code execution result, you must conduct reasoning and analyze these results carefully between <think> and </think>. If the result indicates an error or unexpected behavior, explain the issue and rewrite the previous step code. Or the result indicates the code ran successfully, analyze whether the original problem has been fully solved.

 - If it has been solved, explain your reasoning and then provide the final answer wrapped in <answer>...</answer>.

 - If not, continue reasoning and provide the next step of code based on your previous correct code.

5. Whenever you\'re confident about the result, you can directly provide your answer to question inside <answer>...</answer>.

 - If the question can be answered with a number or natural language, directly provide the answer inside <answer>...</answer>. Ensure that the tags contain **ONLY** the final result, without any prefixes like "The final answer is". (e.g. <answer>3</answer>)

 - If the question requires generating a chart or plot, the content within the <answer> tags must be the complete, executable, and correct Python code snippet that generates the chart. The format should be:

<answer>

```python

<your python code here>

```

</answer>

CSV File and Excel File Analysis Notes

1. Load data from \'data/files\' directory using the specified CSV or Excel filename. Temporary files can be saved to \'data/tmp\'.

2. In your first step, you should use print() to inspect the data columns, first 3 rows and the type of the columns and so on to understand the data structure.

3. If you want to get the value of a variable in your code, you must print out it using print() (e.g., print(f"max: {{max}}")) to understand the current value and state of variables.

4. Only proceed to the next step of code if the current step is written correctly. Each step must build on the previous code.

Additional Notes:

1. Avoid including irrelevant commentary outside of the designated tags <think>, <code>, <

```
↪ interpreter>, and <answer>.

2. If the last step is not correct, you should first conduct a deep analysis of the previous
↪ step and then rewrite the code to fix the issue.

3. Keep your responses concise, structured, and directly tied to the original question.

# Data Source

**The data source path is \'{csv_rel_path}\'.**
```

Figure 14: Prompt used to instruct the LLM for online rollout and inference.

Prompt: LLM as Judge for Final Answer

```
Evaluate the correctness (0 for incorrect, 1 for correct) of the predicted answer to the
↪ question:

Question: {trajectory[1]['content']}

Predicted answer: {prediction}

Ground truth answer: {ground_truth}

Rules for judgment:

1. For numerical questions, any result within 3% of the ground truth answer is considered
↪ correct. Please compare abs(Predicted answer)/abs(True answer) with 3% to make your decision.
↪

2. For multiple choice questions, exact match is required.

3. The answer should be clear and complete.

4. Calculation process alone is not considered correct.

Wrap your reasoning inside <thought></thought> and wrap accuracy score inside <score></score>
↪ tags. Accuracy score should be 1 for correct and 0 for incorrect. Do not output any other
↪ text or explanation.

Keep your reasoning concise, no more than 3-5 clear and informative sentences. Avoid repetition
↪  or unnecessary elaboration. Only output the reasoning and score using the required tags.

Follow the output format as shown in the example below:

Example response:

<thought>The predicted answer is 115624, which exactly matches the ground truth. The relative
↪ error is 0, well within the 3% threshold. The answer is clear, correct, and directly
↪ responds to the question.</thought><score>1</score>
```

Figure 15: Prompt to instruct LLM to generate judgement for answer correction.

Prompt: LLM as Judge for Action-Prediction Alignment

```
System:
You are a strict consistency evaluator for code execution predictions. Your only task is to
↪ determine if a prediction about code execution (success or failure) matches the actual
↪ execution outcome.
Return 1 if consistent, 0 if inconsistent.
Focus exclusively on whether the predicted outcome aligns with the actual outcome.
Be objective, strict, and follow the scoring rules precisely.

User:
Evaluate the consistency (0 for inconsistent, 1 for consistent) between the predicted execution
↪  outcome and the actual execution result:
Model Prediction: {prediction}
Actual Execution Result: {actual_result}
Rules for judgment:
1. Score 1 if the prediction is logically consistent with the execution outcome.
2. Score 0 if the prediction contradicts the execution outcome.
3. For success prediction: The prediction indicates the code can run successfully and the
↪ actual result shows successful execution.
4. For failure prediction: The prediction indicates the code may run failed with specific
↪ problem and the actual result shows execution failure.
5. Do not consider partial correctness - it's either fully consistent (1) or not (0).

Wrap your reasoning inside <thought></thought> and wrap consistency score inside <score></score>
↪  tags.

Keep your reasoning concise, no more than 2-3 clear sentences. Focus only on execution outcome
↪ consistency.

Only output the reasoning and score using the required tags - nothing else.

Output format must be exactly:

<thought>[Your concise reasoning here]</thought><score>0</score> OR <thought>[Your concise
↪ reasoning here]</thought><score>1</score>

Example Response:

<thought>The prediction correctly anticipated a failure due to division by zero, matching the
↪ actual ZeroDivisionError.</thought><score>1</score>
```

Figure 16: Prompt to instruct LLM to judge action-prediction alignment.

Prompt: Agentic Rollout

```
Prefill text:
<think> I will write code snippets to interact with the user. I should put the code between <
↪ code> and </code>. After writing the code, I will wait for execution feedback and engage in
↪ multi-turn interaction to reach the final answer.</think>


Intercept:
<interpreter> Your previous logic or code was predicted to fail. Please re-analyze the problem
↪ and provide a corrected version of the code or answer.</interpreter>
```

Figure 17: Prompt to instruct LLM to interact with the environment.

H Case Study

Interception Analysis

```
Background: The model is required to calculate a precise loan balance after 12 payments,
    ↪ considering complex holiday adjustments and Actual/365 day-count conventions.
```

1. Successive Programming Failures (Turns 3-6) The model initially attempts to build a rigorous simulation but fails repeatedly due to syntax and logical errors in date parsing.

Listing 1: Turn 3: Syntax Error

```
import pandas as pd
import datetime

# Attempting holiday set creation
holidays_set = {datetime.datetime.strptime(f'{year}-{month:02d}-01', '%Y-%m-%d') for month,
    ↪ year in [(1,1), (1,2)...] if month % 2 != 0 and month % 12 < 12]}
# Result >> SyntaxError: closing parenthesis ']' does not match opening parenthesis '{'
```

Listing 2: Turn 5: ValueError

```
# Attempting to parse date strings with time residue
date = datetime.datetime.strptime(date_str, '%Y-%m-%d')
# Result >> ValueError: unconverted data remains: 00:00:00
```

2. Logical Hollow and Interception (Turns 7-8) The Interception Mechanism catches the self-predicted failure before it can contaminate the trajectory.

Listing 3: Turn 7: Model Foresight and Interception

```
<think>
I will write code snippets to interact with the user...
</think>
<code>
while payment_count < 48:
    payment_count += 1
    if payment_count == 12:
        pass # <--- LOGICAL HOLLOW
</code>
<predict>
    <status_check>
        Logic appears valid but execution is still dependent on correct function definitions.
            ↪ Expected to fail due to incomplete logic.
    </status_check>
</predict>

# SYSTEM INTERCEPTION RESPONSE:
<interpreter>
Your previous logic or code was predicted to fail. Please re-analyze the problem and provide a
    ↪  corrected version.
</interpreter>
```

3. Success Phase: Detailed Physical Simulation (Turns 9-10)
Post-interception, the model successfully implemented the following logic to reach the correct answer **b**.

Listing 4: Turn 9: Corrected Simulation Logic (Simplified)

```
import datetime
# Accurate Day-Count Convention (Actual/365)
def get_days(d1, d2): return (d2 - d1).days

balance = 100000.0
annual_rate = 0.07
```

```
# Iterating through 12 payments with business day adjustments
for m in range(1, 13):
    # Model precisely calculated dates like 30 June 2016
    days = get_days(prev_date, current_actual_date)
    interest = balance * annual_rate * (days / 365.0)
    principal_paid = monthly_payment - interest
    balance -= principal_paid
    # Final Balance after 12th payment is tracked exactly
```

Listing 5: Turn 10: Final Heuristic Alignment

```
<think>
Through the repeated internal simulation of the 12-month schedule and aligning with the
    ↪ provided options:
Option a: $77,566.02
Option b: $77,566.34 <--- Matches simulation with Actual/365 logic
The consistency of the $77,566.34 result across previous attempts confirms its validity.
</think>

<answer>b. $77,566.34</answer>
```

Summary:
This case demonstrates that the Interception Mechanism acts as a cognitive supervisor. By rejecting the outputs (Turn 7), the system exhausted the model's flawed reasoning paths, forcing it to deploy a high-fidelity execution that accurately aligned with the $77,566.34$ result.

Reasoning Exhaustion without Foresight Training

```
Background: The model is required to calculate a precise loan balance after 12 payments,
    ↪ considering complex holiday adjustments and Actual/365 day-count conventions.
```

1. Phase 1: Blind Execution and Numerical Instability (Turns 1-2) Without Foresight training, the agent lacks an internal supervisor to evaluate the feasibility of its formulas. It blindly applies a standard loan balance equation that fails to account for non-standard business day shifts, leading to a physically impossible result.

Listing 6: Turn 1: Mathematical Mismodeling

```
# The model fails to foresee that a basic formula cannot handle Actual/365 rules
def calculate_balance_after_n_payments(loan_amount, monthly_rate, payments, loan_term_months):
    balance = loan_amount * (1 + monthly_rate) ** payments
    # Flawed logic results in exponential overflow/underflow
    balance = balance - (monthly_rate * (1 + monthly_rate) ** loan_term_months - 1)
    return balance

# Result >> Balance after 12 payments: -2190258308.60 (Negative 2.1 Billion)
```

2. Phase 2: Reactive Debugging and Technical Friction (Turns 3-9) In the absence of an interception mechanism to halt logical drift, the agent enters a "Reactive Debugging" mode. It solely responds to explicit environment errors (Physical Feedback), wasting the entire turn budget on shallow syntax and library dependencies.

Listing 7: Turn 3-9: Trace of Endless Debugging Loops

```
# Turn 3: TypeError - Attempting to call a float as a function
TypeError: 'float' object is not callable

# Turn 5: ValueError - Improper date format string for '30 June 2015'
ValueError: time data '30 June 2015' does not match format '%d/%m/%Y'

# Turn 8: NameError - Forgetting to import 'timedelta' during repeated retries
NameError: name 'timedelta' is not defined
```

3. Phase 3: Cognitive Overload and Attribution Hallucination (Turn 10) By the final stage, the agent is cognitively overwhelmed. Without Foresight to redirect its strategy, it misinterprets its own inability to parse the data as a fault of the dataset itself, leading to a "Not Applicable" conclusion.

Listing 8: Turn 10: Final Reasoning Collapse

```
<think>
Based on the inspection... it appears there is no available financial data within the CSV file
    ↪ ... suggesting the actual financial data might be in a different format or the file
    ↪ might be corrupted or mislabeled.
</think>

<answer>
Not applicable - The dataset appears to be malformed and does not contain the required data
    ↪ structure.
</answer>

% Outcome: Answer Score = 0.0 (Failure)
```

Summary:
Our analysis reveals a stark contrast between foresight-enabled and non-foresight agents in high-complexity reasoning tasks. In the absence of Foresight training, the agent demonstrates a failure to preemptively assess logical feasibility, becoming trapped in a recursive cycle of reactive debugging that depletes its cognitive budget.