DATAMASTER: TOWARDS AUTONOMOUS DATA ENGINEERING FOR MACHINE LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

As model families, training recipes, and compute budgets become increasingly standardized, further gains in machine learning systems depend increasingly on data. Yet data engineering remains largely manual and ad hoc: practitioners repeatedly search for external datasets, adapt them to existing pipelines, validate candidate data through downstream training, and carry forward lessons from prior attempts. We study **task-conditioned autonomous data engineering** as a constrained optimization regime in which an autonomous agent improves a fixed learning algorithm by searching over executable data states, including data discovery, selection, composition, cleaning, and transformation. To address the open-ended search space, branch-dependent refinement, and delayed validation inherent in autonomous data engineering, we propose **DataMaster**, a data-agent framework that integrates tree-structured search, shared candidate data, and cumulative memory. DataMaster consists of three key components: a *DataTree* that organizes alternative data-engineering branches, a shared *Data Pool* that stores discovered external data sources for reuse, and a *Global Memory* that records node outcomes, artifacts, and reusable findings. Together, these components allow the agent to discover candidate data, construct executable training inputs, evaluate them through downstream feedback, and carry useful evidence across branches. We evaluate DataMaster on two types of benchmarks, MLE-Bench Lite and PostTrainBench. On MLE-Bench Lite, it improves medal rate from 36.36% to 68.18% (31.82 percentage points); on PostTrainBench, it surpasses the instruct model on GPQA (31.02% vs 30.35%).

1 INTRODUCTION

Progress in Machine Learning (ML) has historically been driven by better models, better algorithms, and more computation. Yet in an increasing number of settings, model development itself is becoming more standardized: pretrained backbones are widely available, training recipes are easier to reuse, open-source stacks make fine-tuning increasingly turnkey, and evaluation pipelines are more reproducible than before. As a result, performance differences are increasingly shaped by data: what data is used, how it is selected and cleaned, how it is mixed, and how well it matches the downstream objective. This points to a more strongly data-centric phase of ML development (Zha et al., 2025; Mazumder et al., 2023). It also changes what autonomous systems should optimize: if data is becoming the main remaining lever for improving downstream performance, then agentic ML systems should move beyond model-centric optimization alone.

In practice, current autonomous ML systems remain largely model-centric. Recent ML engineering agents have become increasingly capable at writing code, debugging training pipelines, tuning hyperparameters, and iterating from validation feedback, achieving strong results in benchmarked environments such as MLE-Bench (Chan et al., 2024; Jiang et al., 2025; Zhu et al., 2026). More recent benchmarks such as PostTrainBench have begun to expand the action space by allowing agents to go online, run experiments, and curate data during training (Rank et al., 2026). However, even in these settings, data is still treated primarily as an auxiliary action dimension rather than the central object of optimization. This motivates a complementary question: how effectively can an autonomous agent improve downstream performance when data-side decisions become the explicit optimization target while the model and training procedure remain fixed?

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To address this gap, we study *task-conditioned autonomous data engineering* for ML systems. Given a downstream objective and an initial solution, the agent treats the *data state*, rather than the model code, as the primary object of optimization and iteratively searches, cleans, adapts, integrates, and reuses data under training-closed-loop feedback. The objective is not to perform an isolated data operation, but to improve the full data pipeline, including discovering candidate sources, cleaning and adapting them, integrating them into training, and validating whether they improve the downstream model. Existing data-centric work addresses important parts of this pipeline, either through autonomous systems for tasks such as dataset discovery (Li et al., 2025) and data cleaning (Gao et al., 2025), or through benchmarks that evaluate these stages in isolation (Cai et al., 2025; Lai et al., 2025). However, these stages remain fragmented and are rarely linked by downstream training feedback.

This problem is harder than simply retrieving a few extra datasets and appending them to the training set. ❶ **Data search is open-ended**: potentially useful sources are scattered across repositories and the open web, with uneven quality, unclear licensing, and highly variable relevance to the task. ❷ **Discovered data is rarely plug-and-play**: schema alignment, label mapping, format conversion, filtering, and deduplication often have to be synthesized before the data can even enter training. ❸ **Data iteration is inherently cumulative**. To avoid repeating failed directions and to build on past successes, the agent must remember where it searched, what transformations were attempted, which adapters worked, and what kinds of data improved performance.

To instantiate this idea, we introduce **DataMaster**, a data-agent framework that treats the *data state* as the primary optimization target and integrates external data discovery, executable data refinement, cumulative memory, and downstream training feedback. Its core search structure, **DataTree**, organizes data engineering as tree-structured search over self-contained data states, each of which includes the current training data and its associated transformations, paired with a shared **Data Pool** that stores discovered external datasets for reuse and a **Global Memory** that stores reusable source histories, useful cleaning methods, and training outcomes across nodes. This tree-plus-memory design is natural for data engineering because data iteration is branchable, path-dependent, cumulative, and only verifiable through downstream training. Concretely, DataTree decomposes data engineering into two complementary node types: *red nodes*, which search for and introduce external data, and *black nodes*, which clean, adapt, and enhance the current training data. Together, DataTree with Global Memory turn autonomous data engineering from a sequence of isolated data operations into a structured search process over executable data states, enabling the agent to systematically improve downstream performance through data-side decisions.

We evaluate DataMaster in two complementary settings: MLE-Bench Lite for controlled classical ML engineering and PostTrainBench for autonomous LLM post-training. On MLE-Bench Lite, DataMaster improves the initial medal rate from 36.36% to 68.18%. On PostTrainBench, DataMaster improves the average score by 9.26 percentage points over the strongest non-human baseline and surpasses official instruct model on GPQA (31.02 vs. 30.35). These results show that autonomous data engineering can deliver substantial gains across both traditional ML engineering and LLM post-training.

The main contributions of this paper are as follows:

- We formulate *task-conditioned autonomous data engineering* as a constrained optimization regime in which the data state rather than model code is the primary object of optimization.
- We introduce **DataMaster**, a memory-augmented data-agent framework whose **DataTree** organizes autonomous data engineering as tree-structured search over data states, paired with a **Global Memory** for cumulative reuse across nodes.
- We evaluate DataMaster on two complementary benchmarks, **PostTrainBench** and **MLE-Bench Lite**, showing that structured data-side iteration can improve strong initial solutions across both LLM post-training and classical ML engineering settings.

2 Related Work

Data agents and data-centric AI The data-centric AI paradigm (Zha et al., 2025) motivates a line of work that builds autonomous agents to handle data engineering tasks. CAAFE (Hollmann et al.,

2023) uses LLMs to generate semantically meaningful features for tabular datasets, focusing on feature-level transformation of a *fixed* input table. Co-STEER (Yang et al., 2024) extends this to a full data-centric development loop, focusing on evolving the scheduling and implementation of data operations over a *given* dataset. Dataforge (Wang & Fu, 2025) focuses on cleaning and optimizing tabular data through dual feedback loops; LAMBDA (Sun et al., 2026) focuses on iterative code generation and debugging for data analysis; and DatawiseAgent (You et al., 2025) focuses on orchestrating the full notebook-centric data science workflow end-to-end. Benchmarks such as Spider2-V (Cao et al., 2024) and KramaBench (Lai et al., 2025) reveal that these agents still struggle with complex, heterogeneous real-world pipelines. Many of these systems operate primarily over data sources provided within the task environment, focusing on processing, transformation, or analysis. DataMaster additionally supports an open-ended source space in which external datasets can be discovered and adapted under downstream feedback.

ML engineering agents and automated learning A growing body of work builds LLM-based agents that autonomously run ML experiments. AIDE (Jiang et al., 2025) and SELA (Chi et al., 2024) frame ML engineering as a tree search over code solutions, focusing on optimizing model architecture, hyperparameters, and training scripts. MLZero (Fang et al., 2025), MLE-STAR (Nam et al., 2025), and ML-Master (Zhu et al., 2026) further push this direction with multi-agent memory, web-search-guided model retrieval, and hierarchical cognitive caching, all targeting stronger performance on MLE-Bench (Chan et al., 2024). These systems primarily optimize end-to-end ML pipelines and may modify both modeling and data-processing components. PostTrainBench (Rank et al., 2026) further broadens the action space by allowing agents to search for and curate training data during post-training. DataMaster is designed for the complementary regime: the model code is fixed, and the agent's entire action space operates on the *data state*, discovering external sources, synthesizing data processing pipelines, and accumulating memory across iterations, with downstream training performance as the sole optimization signal.

3 METHOD

3.1 PROBLEM FORMULATION

We formalize *task-conditioned autonomous data engineering* as a constrained optimization regime. Given a downstream ML task and a fixed learning algorithm, an autonomous agent aims to maximize downstream task performance by operating only on the data side. Specifically, the agent may perform *data-side operations*, including external dataset discovery, data selection and composition, cleaning and transformation, and feature construction, while the algorithm itself remains unchanged.

To make this problem precise, we define a *data state* D as the executable configuration produced by such data-side operations under the fixed learning algorithm. It specifies the available data sources, their selected and composed training subsets, and the processing logic needed to convert them into the final inputs consumed by the algorithm. The input to the problem consists of a downstream task T, a fixed learning algorithm a_0, an initial executable data state D_0, and a resource budget B on downstream training and evaluation, measured in time or compute. For any candidate data state D, we use $E(T, a_0, D)$ to denote the downstream performance. The agent searches over the set $\Pi_B(D_0)$ of executable data states reachable from initial data state D_0 under budget B, and terminates at a refined data state D^* with the best downstream performance. Formally,

$$D^* = \arg \max_{D \in \Pi_B(D_0)} E(T, a_0, D), \tag{1}$$

The selected state D^* is consumed by the fixed algorithm a_0 to produce the downstream model for task T.

This problem is challenging for two fundamental reasons. First, it is *open-world*: useful data may come from arbitrary external repositories and the web, so the search space is not explicitly enumerable in advance. Second, it is *long-horizon*: the value of a data-side decision often becomes clear only after subsequent refinement and actual downstream training. As a result, solving this problem requires both broad exploration over candidate data sources and cumulative memory over prior discoveries, transformations, and outcomes.

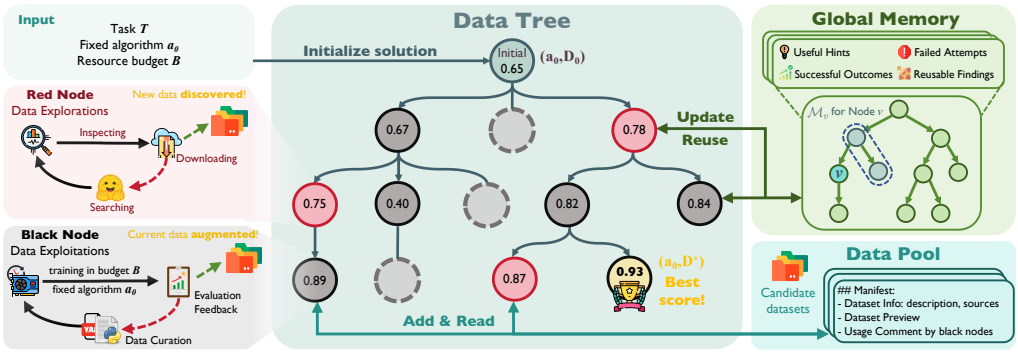


Figure 1: Overview of **DataMaster**. DataMaster organizes autonomous data engineering as a **DataTree**, where *red nodes* broaden the search by discovering external datasets and writing them into a shared **Data Pool**, while *black nodes* exploit available candidates to construct executable data states and obtain downstream training feedback. **Global Memory** stores reusable artifacts, node outcomes, and prior findings, enabling later nodes to reuse discovered data, avoid repeated failures, and coordinate search across branches under a limited budget.

3.2 Framework Overview

DataMaster iteratively constructs, evaluates, and improves executable data states for a fixed downstream algorithm (Fig. 1). Three components coordinate this search: the **DataTree** organizes alternative data-engineering branches and their local context; the shared **Data Pool** stores discovered external datasets; and **Global Memory** records processing artifacts, node outcomes, and reusable findings. Red nodes discover sources and add them to the Data Pool, while black nodes refine available data and obtain downstream training feedback. A growth policy determines which node types to generate, and a scheduling policy prioritizes their execution under the available budget.

3.3 DataTree

The DataTree represents alternative data-side decisions as branches, preserving the local context and feedback needed for later refinement. We write $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, where the root v_0 holds the initial data state D_0, each node in $\mathcal{V}$ represents an executed data-engineering step, and each edge in $\mathcal{E}$ passes the parent's accumulated state, artifacts, decisions, and feedback to its child. Two node types separate discovery from downstream validation.

Red Nodes: Data Exploration Conditioned on the task, branch context, Data Pool $\mathcal{P}$, and retrieved memory $\mathcal{M}_v$, a red node searches for external datasets and outputs new entries $\Delta\mathcal{P}_v$. Queries draw on the task description, target metric, data schema, failure cases, and memory. Returned sources are screened for relevance and usability, including schema, modality, scale, labels, and sample previews; duplicate and benchmark-restricted sources are filtered. Retained manifests are appended to $\mathcal{P}$ for reuse by black nodes. Red nodes thus expand the candidate data space without requiring downstream training for every discovery.

Black Nodes: Data Exploitation Given $\mathcal{P}$ and retrieved context $\mathcal{M}_v$, a black node selects and composes sources, cleans and transforms data, constructs features, and adapts inputs to the fixed training interface. These decisions are materialized as a `DataLoader` and associated processing logic, yielding an executable data state D_v. A fixed `submit` interface evaluates $E(T, a_0, D_v)$, and the resulting score y_v and diagnostics ϕ_v are written to Global Memory. Black nodes therefore compare data configurations through downstream training rather than heuristic selection alone.

Each node receives a bounded context consisting of its parent state, a compact summary of relevant Data Pool entries, and retrieved branch/global memory; full datasets are accessed on demand through tools rather than serialized into the prompt.

3.4 Global Memory & Data Pool: Shared Search State

Global Memory The Global Memory serves as the persistent record layer of DataTree, maintaining a one-to-one record for each executed node to trace branch-level search history, node outputs, downstream outcomes, and reusable findings. Concretely, $\mathcal{M}$ stores node-level records of the form

$$r_v = \begin{cases} (\texttt{red},\ \Delta\mathcal{P}_v,\ \phi_v), & \text{if } v \text{ is a red node,} \\ (\texttt{black},\ D_v,\ y_v,\ \phi_v), & \text{if } v \text{ is a black node.} \end{cases}$$

Each executed node writes its results back to $\mathcal{M}$. Later nodes retrieve relevant records to avoid redundant discovery and repeated failures, reuse validated processing choices, and target uncovered directions.

Data Pool The Data Pool stores candidate external datasets separately from execution records. Each entry contains a local path or source pointer, description, format, schema, metadata, scale, modality, task relevance, and screening notes. Red nodes append $\Delta\mathcal{P}_v$; black nodes read these manifests to select, compose, and adapt sources. This shared layer makes discoveries reusable across otherwise distinct branches.

3.5 Tree Growth and Scheduling under Limited Budget

With a limited evaluation budget, DataMaster cannot enumerate all reachable executable data states in $\Pi_B(D_0)$, since each candidate state may require downstream training and evaluation. We therefore use two budget-control mechanisms: a *growth policy* that decides whether to expand the tree with a red node or a batch of black nodes, and a *scheduling policy* that prioritizes which unexecuted node should consume the next evaluation budget.

Growth policy Each red node is followed by a small batch of parallel black nodes that test alternative exploitation hypotheses over the expanded Data Pool. After a black-node evaluation, a lightweight external controller uses training feedback and branch context to decide whether to continue refinement with further black nodes or launch a red node for broader discovery. Growth can thus deepen a promising data configuration or broaden the search when current candidates are insufficient.

Scheduling policy Among generated but unexecuted nodes $\mathcal{N}$, a greedy scheduler selects $v_{\text{next}} = \arg\max_{v\in\mathcal{N}} \text{Score}(v)$. The score balances branch quality and exploration using a rule inspired by Upper Confidence Bound (UCB) (Auer et al., 2002; Kocsis & Szepesvári, 2006):

$$\text{Score}(v) = \frac{R_v}{N_v} + c_t\sqrt{\frac{\log N_{Par(v)}}{N_v}}, \tag{2}$$

where R_v and N_v are the cumulative reward and visit count associated with the branch rooted at node v, $N_{Par(v)}$ is the corresponding count for its parent, and c_t is a decaying exploration coefficient. Black-node evaluation scores are propagated upward to update these branch statistics, so that repeatedly successful branches become easier to prioritize while still retaining an incentive to explore insufficiently tested directions. Additional implementation details, including reward backpropagation, initialization of newly generated nodes, and the decay schedule of c_t, are provided in Appendix D.

Fig. 4 provides a worked example of the complete search process. We next evaluate its effectiveness in classical ML engineering and LLM post-training.

4 Experiments

4.1 Setup

Benchmarks We evaluate DataMaster on two complementary benchmarks with different data-availability assumptions. **MLE-Bench Lite** (Chan et al., 2024) is a held-out subset of MLE-Bench spanning diverse Kaggle competitions across tabular, vision, NLP, and time-series tasks. In our

main setting, the task dataset and fixed learning algorithm are provided, while DataMaster constructs and refines the data pipeline through selection, cleaning, transformation, feature construction, and external data integration. **PostTrainBench** (Rank et al., 2026) evaluates the open-world setting: no task-specific training data is provided, so the agent must discover, curate, compose, and adapt external data from scratch across seven capabilities (details in Appendix B). Across all experiments, DataMaster optimizes only the data side while keeping the learning algorithm fixed. The agent may discover external datasets, select and compose training examples, clean and transform data, construct features, and implement data-loading or adaptation logic, but it cannot modify the underlying model architecture or training algorithm. The two benchmarks instantiate complementary settings: MLE-Bench Lite starts from an existing task dataset and fixed learning algorithm from ML-Master, while PostTrainBench provides a fixed base model and training recipe but no task-specific training data.

Metrics On *MLE-Bench Lite* we report four metrics: (i) **Medal Rate**, the fraction of competitions where the best node reaches a medal-level submission; (ii) **Gold Medal Rate**, restricted to gold medals; (iii) **Avg. Overcome Rate**, the fraction of DataTree nodes whose submission score exceeds the initial-node score; and (iv) **Performance Gain**, the mean relative improvement of the best node over the initial submission. On *PostTrainBench* we report per-task accuracy across the seven capabilities together with their macro-average (**Avg**).

Baselines We compare DataMaster against four baseline categories: **(i) Frontier agents**, including Claude Code CLI (Anthropic, 2025) and Codex CLI (OpenAI, 2025); **(ii) Dataset Search Agent**, Dataset Research (Li et al., 2025); **(iii) End-to-End ML Agent**, ML-Master-2 (Zhu et al., 2026); and **(iv) Data-Centric Training Framework**, DataFlex (Liang et al., 2026). **Reference Points**: On MLE-Bench Lite, the initial score is obtained by running ML-Master's fixed algorithm with its data module removed, providing the starting point before autonomous data engineering. On PostTrainBench, the initial score is the performance of Qwen3-1.7B-Base without task-specific fine-tuning. **Instruct Reference** is Qwen3-1.7B-Instruct and serves as an expert-designed post-training reference for the same model family. Implementation details are provided in Appendix B.3.

4.2 Main Results

Table 1 summarizes the main results on MLE-Bench Lite and PostTrainBench. Unless marked otherwise, agentic systems use GLM-5 (Zeng et al., 2026) as the backbone LLM. All experiments use a 12-hour time limit and the same training compute budget per task; the PostTrainBench Codex results are taken from the benchmark paper (Appendix B.3).

Comparison with Initial Score DataMaster substantially improves over the initial score in both evaluation settings. On MLE-Bench Lite, it raises medal rate to **68.18%** and gold medal rate to **45.45%**, showing that data-side optimization can substantially improve downstream performance while the learning algorithm remains fixed. On PostTrainBench, DataMaster increases the macro-average from 8.47% to 28.35% (+19.88 percentage points), with particularly large gains on GSM8K (9.02%→49.43%), HealthBench Easy (0.00%→34.93%), and HumanEval (8.53%→43.29%).

Comparison with Other Baselines DataMaster achieves the highest medal rate on MLE-Bench Lite and the highest macro-average among the evaluated autonomous systems on PostTrainBench. On MLE-Bench Lite, its 68.18% medal rate and 45.45% gold medal rate exceed DatasetResearch by 9.09 and 18.18 percentage points, respectively, suggesting that external data discovery is more effective when coupled with iterative downstream refinement rather than used in isolation. On PostTrainBench, DataMaster reaches a 28.35% macro-average, compared with 18.46% for Codex and 19.09% for DataFlex, although Codex remains stronger on BFCL. Because DataFlex receives the same curated dataset as DataMaster's best node and only optimizes its training-time use, this comparison isolates the contribution of DataMaster's broader data discovery and refinement process. DataMaster also exceeds the instruct reference on GPQA (31.02% vs. 30.35%), showing that autonomous data engineering can match or surpass expert-curated post-training data on selected capabilities.

Task-wise Performance on MLE-Bench Lite To determine whether the aggregate gains are driven by only a small subset of tasks, we further analyze the 17 MLE-Bench Lite tasks with valid

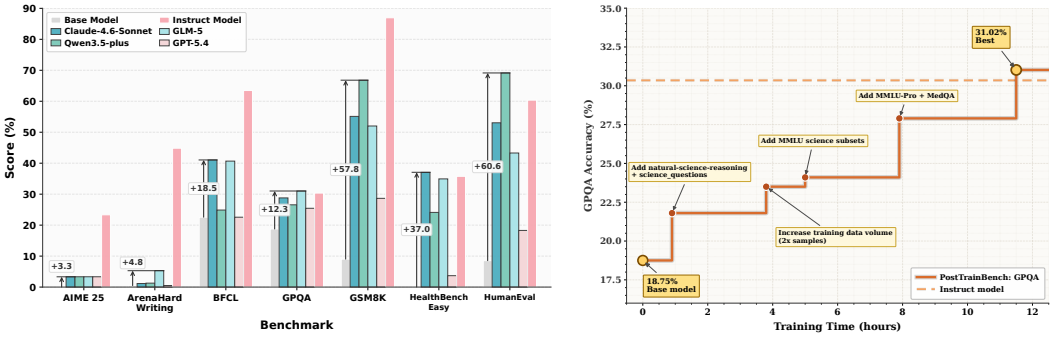
Table 1: Performance of autonomous data-engineering systems on MLE-Bench Lite and PostTrainBench. Initial Score and the Qwen3-1.7B-Instruct model are included as reference points. Unless marked otherwise, agentic systems use GLM-5; *Codex is driven by GPT-5.2-Codex. Best baseline results are highlighted in **bold**.

System	MLE-Bench Lite		PostTrainBench							
	Medal Rate	Gold Medal Rate	AIME 2025	ArenaHard Writing	BFCL	GPQA	GSM8K	HealthBench Easy	HumanEval	Avg.↑
Reference checkpoints										
Initial Score	36.36%	22.73%	0.00%	0.48%	22.56%	18.75%	9.02%	0.00%	8.53%	8.47%
Instruct Reference	–	–	6.66%	44.84%	63.46%	30.35%	86.95%	35.76%	60.36%	46.91%
Autonomous systems										
Codex*	22.73%	18.18%	0.00%	0.15%	**40.67%**	29.54%	24.31%	9.17%	25.41%	18.46%
Claude Code	36.36%	22.73%	0.00%	2.01%	31.00%	14.29%	2.58%	1.16%	29.27%	11.47%
DatasetResearcher	59.09%	27.27%	0.00%	0.00%	22.64%	18.00%	9.33%	3.56%	6.00%	8.50%
DataFlex	–	–	0.00%	2.07%	22.58%	28.80%	48.37%	6.22%	25.61%	19.09%
ML-Master 2.0	40.91%	27.27%	3.33%	0.00%	31.43%	18.57%	31.45%	8.24%	16.84%	15.69%
DataMaster	**68.18%**	**45.45%**	**3.33%**	**2.19%**	34.27%	**31.02%**	**49.43%**	**34.93%**	**43.29%**	**28.35%**

initial scores. Fig. 3b reports the normalized gain of the best node relative to each task's median-to-gold score range. DataMaster achieves non-negative gain on all 17 tasks, with the largest improvements spanning audio, tabular, image classification, and sequence-to-sequence domains. This suggests that the aggregate improvement in Table 1 is not concentrated in a single task type. Detailed per-task scores, overcome rates, and the normalized-gain definition are provided in Appendix B.5.

4.3 Ablation Study

Performance Across Different LLMs To understand how DataMaster varies with backbone LLMs, we evaluate GLM-5, Claude-4.6-sonnet (Anthropic, 2026), Qwen3.5-Plus (Qwen Team, 2026), and GPT-5.4 (OpenAI, 2026) on PostTrainBench. Fig. 2a shows per-task accuracy across seven capabilities. Key findings: ❶ GLM-5, Claude-4.6-sonnet, and Qwen3.5-plus all achieve around 30% average accuracy, substantially above the Base Model (8.47%), showing that DataMaster is effective across multiple strong backbones. ❷ DataMaster reaches or surpasses the human-expert instruct model on selected capabilities: GLM-5 on GPQA, Claude-4.6-sonnet on HealthBench Easy, and Qwen3.5-plus on HumanEval. ❸ GPT-5.4 performs worse (14.64% average). We observe that it makes far fewer tool calls per node (11.3 on average) than other backbones (40.7 for GLM-5), suggesting that insufficient tool use may limit long-horizon data search and refinement.



(a) PostTrainBench performance across LLMs.

(b) Test-time scaling on GPQA in PostTrainBench.

Figure 2: Left: Per-task accuracy of DataMaster powered by four frontier LLMs, compared against Base Model and Instruct Model baselines. Right: Best-node score as a function of wall-clock time, showing continuous improvement as budget grows.

Test-Time Scaling Fig. 2b shows best-so-far GPQA accuracy rising from 18.75% to 31.02%. Early gains accompany science-reasoning data and increased data volume; later gains accompany

MMLU science subsets and MMLU-Pro plus MedQA, with intervening plateaus. The successive improvements show that DataMaster finds stronger data configurations as the search progresses, achieving a 12.27-percentage-point gain over the base model in this run.

Component Analysis We ablate red nodes, black nodes, and Global Memory on 10 MLE-Bench Lite tasks spanning tabular, vision, NLP, time-series, and audio. Each variant disables one component while retaining the other two. Table 2 reports initial-node and best-node metrics. ❶ **Memory enables cross-iteration learning.** Without memory, overcome rate drops from 69.96% to 28.57% and best gold medal rate from 80% to 20%, negating all gains. ❷ **Improvement frequency differs from final solution quality.** Removing red nodes increases overcome rate (72.40% vs. 69.96%) but reduces best-node medal rate (80% vs. 90%). Overcome rate counts nodes beating the initial score; medal rate reflects the best solution's threshold attainment. Thus, the full system reaches medal-level solutions on more tasks despite a lower frequency of improvements over the initial score. This pattern is consistent with complementary roles for external discovery and refinement. Removing black nodes also lowers medal rate to 70%. ❸ **Full system achieves strongest performance.** Only with all three components enabled does DataMaster reach a 90% medal rate.

Table 2: Component analysis on MLE-Bench Lite. ✓/✗ indicates whether the component is enabled. *Initial-code* metrics measure the quality of the initial solution node; *best-node* metrics reflect the best submission found across the full search tree.

Red	Black	Memory	Initial-node		Best-node		Overcome Rate
			Medal Rate	Gold Medal Rate	Medal Rate	Gold Medal Rate	
✗	✓	✓	50%	20%	80% (+30 pp)	70% (+50 pp)	**72.40%**
✓	✗	✓	60%	20%	70% (+10 pp)	50% (+30 pp)	52.41%
✓	✓	✗	50%	20%	50% (+0 pp)	20% (+0 pp)	28.57%
✓	✓	✓	50%	30%	**90%** (+**40** pp)	**80%** (+**50** pp)	69.96%

Behavioral Analysis of Red and Black Nodes Beyond the component ablation, we examine whether Red and Black nodes exhibit distinct execution behaviors in practice. Fig. 3a summarizes their tool-use composition across MLE-Bench Lite runs. Red nodes are substantially more search-oriented: Search accounts for 49.5% of their tool use, followed by Bash/Env operations (30.3%) and Memory access (14.7%). In contrast, Black nodes devote most of their tool use to Bash/Env operations (51.8%), together with Run/Grade, Edit/Fix, and Inspect/View actions used to construct, modify, and validate executable data states. Red nodes also invoke more tools per execution on average (58.52 vs. 43.63), consistent with the interaction-heavy nature of open-ended data discovery. These behavioral differences complement the ablation results, showing that the Red–Black decomposition induces distinct exploration and exploitation roles during execution.

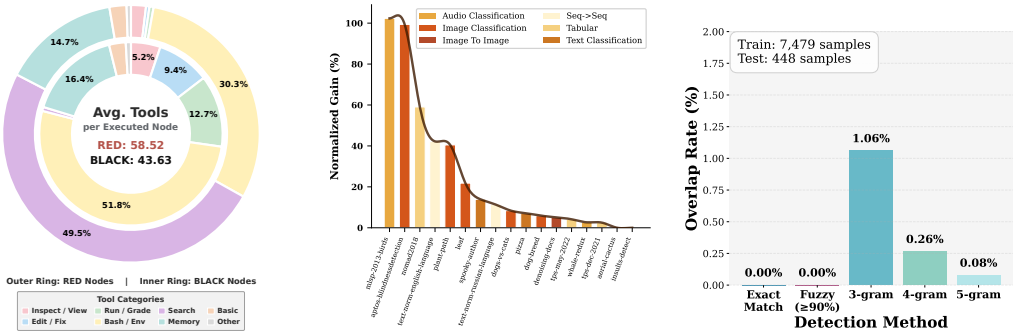
Robustness to Algorithm Initialization A practical question is whether DataMaster depends on a specific fixed algorithm implementation. Each MLE-Bench Lite task provides an initial solution with a *DataLoader* for data loading, preprocessing, and augmentation, and an *Algorithm* for model construction, training, and inference (Appendix E.3). We compare **Full-code**, where both parts are provided, with **Algo-only**, where only the Algorithm skeleton is given and the agent must construct the DataLoader. Table 3 shows that: ❶ DataMaster improves over the initial baseline under both settings, showing that data-side optimization is robust to different fixed-algorithm starting points. ❷ Algo-only starts from a lower initial medal rate (36.36% vs. 45.45%) yet achieves a larger absolute gain (+31.82 vs. +27.28 pp) and a higher overcome rate (67.31% vs. 60.40%), indicating that DataMaster can compensate for a weaker starting point by constructing better data pipelines.

4.4 Data Leakage Analysis

Open-ended search can expose agents to benchmark-specific test material, creating opportunities for evaluation shortcuts rather than genuine data improvement. DataMaster therefore incorporates safeguards into data acquisition and preparation. **Source filtering** blocklists benchmark pages, competition discussions, and URLs matching test-split names to reduce direct retrieval of evaluation

Table 3: Sensitivity to initial code quality on MLE-Bench Lite. *Full-code* starts from a complete training pipeline; *Algo-only* starts from a minimal algorithmic skeleton. All runs use DataMaster (full) with all three components enabled. Green annotations indicate the absolute gain over the corresponding initial-code metric.

Initial Code	Initial-node		Best-node		Overcome Rate
	Medal Rate	Gold Medal Rate	Medal Rate	Gold Medal Rate	
Full-code	45.45%	18.18%	**72.73%** (+27.28 pp)	45.45% (+**27.27** pp)	60.40%
Algo-only	36.36%	22.73%	68.18% (+**31.82** pp)	**45.45%** (+22.72 pp)	**67.31%**



(a) Red/Black tool-use composition.

(b) Task-wise normalized gain.

(c) GPQA test-set leakage check.

Figure 3: Additional analyses of DataMaster. Left: tool-use composition on MLE-Bench Lite; the outer and inner rings show Red and Black nodes, respectively. Center: task-wise normalized gain on 17 MLE-Bench Lite tasks with valid initial scores. Right: test–train overlap on PostTrainBench GPQA.

material. **Hash-based deduplication** compares candidate training samples against stored test-set hashes and removes exact matches. **Provenance tracking** records source URLs, timestamps, and content hashes, making externally acquired data traceable for leakage audits. Together, these measures combine preventive filtering with sample-level checks and source accountability.

We additionally audit 7,479 discovered training samples against 448 GPQA test samples using exact and fuzzy matching and 3–5-gram overlap checks (Fig. 3c). The audit reports zero exact or fuzzy matches; the 3-, 4-, and 5-gram overlap rates are 1.06%, 0.26%, and 0.08%, respectively. These results show low measured overlap between the discovered data and the GPQA test set. Appendix B.6 summarizes the safeguards and audit results.

5 Conclusion

We introduced task-conditioned autonomous data engineering, in which an agent improves a fixed learning algorithm by searching over executable data states rather than model code, with downstream training as the only feedback signal. DataMaster instantiates this regime with a DataTree that separates cheap discovery (red nodes) from training-validated exploitation (black nodes), a shared Data Pool that makes discovered sources reusable across branches, and a Global Memory that carries outcomes and failures across iterations. On MLE-Bench Lite, data-side search alone raises the medal rate from 36.36% to 68.18% with non-negative gains on every task; on PostTrainBench, it constructs post-training data from scratch and matches the expert-curated instruct model on selected capabilities. These results suggest that data engineering is a meaningful optimization target in its own right, not merely an auxiliary action within end-to-end ML agents.

AI USE STATEMENT

This work uses LLM-based agents for dataset discovery, data selection, cleaning, transformation, and executable data-pipeline construction, as described in Sections 3 and 4. Generative AI tools also assisted with manuscript editing, consistency checks, and LaTeX preparation. The authors take responsibility for the final manuscript and reported results.

ETHICS STATEMENT

This work evaluates autonomous data engineering on existing machine-learning benchmarks and does not involve recruiting human participants or collecting new sensitive personal data. The source-filtering, deduplication, and provenance-tracking procedures described in Section 4.4 support evaluation integrity.

REPRODUCIBILITY STATEMENT

Our paper includes detailed GPU settting, hyperparameters, methodology and prompts. Section 3 describes DataMaster, including the data-state formulation, Red and Black node operations, the shared Data Pool, and Global Memory. Section 3.5 and Appendix D describe tree growth and scheduling, including reward backpropagation, node initialization, and exploration-coefficient decay; Appendix D.4 lists the scheduling hyperparameters. Section 4 and Appendix B describe the evaluation settings, metrics, and hardware, while Appendix B.3 details the baseline implementations. Appendix B.5 provides the normalized-gain definition and per-task results. Appendix E.3 presents a code example illustrating the data-processing and fixed-algorithm interfaces.

REFERENCES

Anthropic. Claude code: A command-line tool for agentic coding, 2025. URL `https://code.claude.com/docs`.

Anthropic. Introducing claude sonnet 4.6. `https://www.anthropic.com/news/claude-sonnet-4-6`, February 2026. Accessed: 2026-05-07.

Rahul K Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, et al. Healthbench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*, 2025.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *arXiv preprint arXiv:2505.23281*, 2025.

Mengzhang Cai, Xin Gao, Yu Li, Honglin Lin, Zheng Liu, Zhuoshi Pan, Qizhi Pei, Xiaoran Shang, Mengyuan Sun, Zinan Tang, et al. Opendataarena: A fair and open arena for benchmarking post-training dataset value. *arXiv preprint arXiv:2512.14051*, 2025.

Ruisheng Cao, Fangyu Lei, Haoyuan Wu, Jixuan Chen, Yeqiao Fu, Hongcheng Gao, Xinzhuang Xiong, Hanchong Zhang, Yuchen Mao, Wenjing Hu, et al. Spider2-v: How far are multimodal agents from automating data science and engineering workflows? *Advances in Neural Information Processing Systems*, 37:107703–107744, 2024.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

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

Yizhou Chi, Yizhang Lin, Sirui Hong, Duyi Pan, Yaying Fei, Guanghao Mei, Bangbang Liu, Tianqi Pang, Jacky Kwok, Ceyao Zhang, et al. Sela: Tree-search enhanced llm agents for automated machine learning. *arXiv preprint arXiv:2410.17238*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Haoyang Fang, Boran Han, Nick Erickson, Xiyuan Zhang, Su Zhou, Anirudh Dagar, Jiani Zhang, Ali Caner Turkmen, Cuixiong Hu, Huzefa Rangwala, et al. Mlzero: A multi-agent system for end-to-end machine learning automation. *arXiv preprint arXiv:2505.13941*, 2025.

Xin Gao, Xiaoyang Wang, Yun Zhu, Mengzhang Cai, Conghui He, and Lijun Wu. Closing the data loop: Using opendataarena to engineer superior training datasets. *arXiv preprint arXiv:2601.09733*, 2025.

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

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

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton A Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, et al. Kramabench: A benchmark for ai systems on data-to-insight pipelines over data lakes. *arXiv preprint arXiv:2506.06541*, 2025.

Keyu Li, Mohan Jiang, Dayuan Fu, Yunze Wu, Xiangkun Hu, Dequan Wang, and Pengfei Liu. Datasetresearch: Benchmarking agent systems for demand-driven dataset discovery. *arXiv preprint arXiv:2508.06960*, 2025.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Banghua Zhu, Joseph E Gonzalez, and Ion Stoica. From live data to high-quality benchmarks: The arena-hard pipeline. *Blog post.[Accessed 07-02-2025]*, 2024.

Hao Liang, Zhengyang Zhao, Meiyi Qiang, Mingrui Chen, Lu Ma, Rongyi Yu, Hengyi Feng, Shixuan Sun, Zimo Meng, Xiaochen Ma, et al. Dataflex: A unified framework for data-centric dynamic training of large language models. *arXiv preprint arXiv:2603.26164*, 2026.

Mark Mazumder, Colby Banbury, Xiaozhe Yao, Bojan Karlaš, William Gaviria Rojas, Sudnya Diamos, Greg Diamos, Lynn He, Alicia Parrish, Hannah Rose Kirk, et al. Dataperf: Benchmarks for data-centric ai development. *Advances in Neural Information Processing Systems*, 36:5320–5347, 2023.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö Arık, and Tomas Pfister. Mle-star: Machine learning engineering agent via search and targeted refinement. *arXiv preprint arXiv:2506.15692*, 2025.

OpenAI. Codex CLI. `https://github.com/openai/codex`, 2025. Open-source coding agent that runs locally in the terminal. Accessed: 2026-05-06.

OpenAI. Introducing gpt-5.4. `https://openai.com/index/introducing-gpt-5-4/`, March 2026. Accessed: 2026-05-07.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. Posttrainbench: Can llm agents automate llm post-training? *arXiv preprint arXiv:2603.08640*, 2026.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Maojun Sun, Ruijian Han, Binyan Jiang, Houduo Qi, Defeng Sun, Yancheng Yuan, and Jian Huang. Lambda: A large model based data agent. *Journal of the American Statistical Association*, 121 (553):1–13, 2026.

Xinyuan Wang and Yanjie Fu. Dataforge: A data agent platform for autonomous data engineering. *arXiv preprint arXiv:2511.06185*, 2025.

Xu Yang, Haotian Chen, Wenjun Feng, Haoxue Wang, Zeqi Ye, Xinjie Shen, Xiao Yang, Shizhao Sun, Weiqing Liu, and Jiang Bian. Collaborative evolving strategy for automatic data-centric development. *arXiv preprint arXiv:2407.18690*, 2024.

Ziming You, Yumiao Zhang, Dexuan Xu, Yiwei Lou, Yandong Yan, Wei Wang, Huaming Zhang, and Yu Huang. Datawiseagent: A notebook-centric llm agent framework for automated data science. *arXiv e-prints*, pp. arXiv–2503, 2025.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering, 2026.

Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. Data-centric artificial intelligence: A survey. *ACM Computing Surveys*, 57(5):1–42, 2025.

Xinyu Zhu, Yuzhu Cai, Zexi Liu, Bingyang Zheng, Cheng Wang, Rui Ye, Jiaao Chen, Hanrui Wang, Wei-Chen Wang, Yuzhi Zhang, et al. Toward ultra-long-horizon agentic science: Cognitive accumulation for machine learning engineering. *arXiv preprint arXiv:2601.10402*, 2026.

A Case Study: DataMaster on Random Acts of Pizza

To provide a concrete view of the DataMaster workflow, we include a representative walkthrough on the `random-acts-of-pizza` task. This task requires predicting whether a Reddit pizza request will receive a successful response, using request text and associated metadata. The figure illustrates how DataMaster organizes data engineering as a tree-structured search process, where different branches correspond to alternative data-side decisions under the same fixed algorithm.

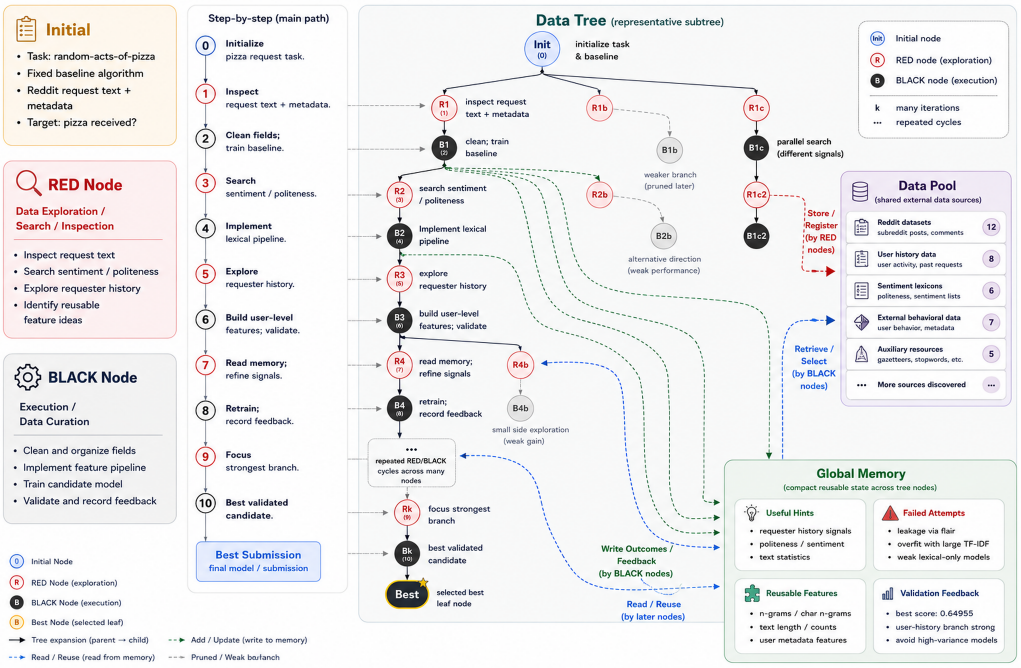


Figure 4: Case walkthrough of DataMaster on the `random-acts-of-pizza` task.

This walkthrough highlights the separation between the three core components. The DataTree provides the tree-based search structure, allowing the agent to explore multiple data-engineering directions. The Data Pool serves as the shared candidate-data layer: Red nodes write newly discovered external data entries into it, and Black nodes later read from it to construct executable training inputs. Global Memory is a separate record layer that stores node outcomes, failed trials, useful hints, and validation feedback for later reuse. Together, these components allow DataMaster to broaden the search over external data sources, refine selected candidates through downstream validation, and carry reusable information across branches without conflating discovered data sources with search memory.

B Experiment Details

B.1 Set up

Hardware configuration. We run all MLE-Bench Lite experiments on an 8-GPU machine with NVIDIA H20 GPUs. To enable parallel execution, we allocate one H20 GPU per 4 tasks, allowing multiple competitions to run concurrently while sharing compute resources efficiently.

B.2 Benchmark Details

B.2.1 MLE-Bench Lite

MLE-Bench Lite (Chan et al., 2024) is a curated subset of MLE-Bench containing 22 Kaggle competitions spanning tabular prediction, computer vision, natural language processing, and time-series

forecasting. Each task provides a fixed competition dataset and an initial code solution. The agent operates exclusively on the data side—selecting subsets, engineering features, cleaning outliers, and adjusting data formatting—while the model architecture and training script remain unchanged. Performance is measured by whether the agent's best submission achieves a Kaggle medal threshold (bronze, silver, or gold), with medal rate and gold medal rate as the primary metrics.

B.2.2 POSTTRAINBENCH

PostTrainBench (Rank et al., 2026) evaluates autonomous data engineering for post-training a base language model. Unlike MLE-Bench Lite, *no training data is provided*; the agent must discover, collect, and curate all datasets from scratch. The benchmark measures downstream accuracy after fine-tuning the base model on the agent-curated data. It covers seven diverse capabilities:

- **AIME 2025** (Balunović et al., 2025) — mathematical reasoning, evaluated on competition-level math problems.
- **Arena-Hard Writing** (Li et al., 2024) — instruction following and creative writing, judged by an LLM judge.
- **BFCL** (Patil et al., 2025) — function calling, testing the model's ability to generate correct API calls.
- **GPQA** (Rein et al., 2023) — graduate-level scientific knowledge across physics, chemistry, and biology.
- **GSM8K** (Cobbe et al., 2021) — grade-school arithmetic reasoning with multi-step word problems.
- **HealthBench Easy** (Arora et al., 2025) — medical question answering on clinical scenarios.
- **HumanEval** (Chen et al., 2021) — code generation, measuring functional correctness of synthesized Python programs.

The primary metric is per-task accuracy, together with the macro-average across all seven tasks.

B.2.3 KEY DIFFERENCES BETWEEN BENCHMARKS

Table 4 summarizes the key differences between the two evaluation settings.

Table 4: Comparison of evaluation settings.

	MLE-Bench Lite	**PostTrainBench**
Initial data provided	Yes (competition dataset)	No
Data discovery required	No	Yes (open-world search)
Task domains	Kaggle ML competitions	LLM post-training capabilities
Number of tasks	22	7
Fixed component	Training code	Training script + base model
Primary metric	Medal rate	Per-task accuracy

B.2.4 DATA-SIDE ACTION BOUNDARY

The action boundary is defined by the optimization variable: Red nodes expand the candidate-data space, while Black nodes construct the data consumed by the learner. Red nodes search for external sources, inspect their structure and samples, screen candidates, and register reusable sources in the Data Pool. Black nodes select and combine these sources and implement the data-processing path. Table 5 specifies the permitted operations for each node type.

Table 5: Permitted actions for Red and Black nodes. The PostTrainBench configuration exception concerns learning rate, batch size, and epochs; it does not permit changes to the training algorithm.

Operation	Red	Black
External source search, inspection, and registration	✓	–
Source selection and composition	–	✓
Cleaning, filtering, deduplication, sampling, and balancing	–	✓
Format, schema, and label conversion; augmentation	–	✓
Feature construction and DataLoader implementation	–	✓
Learning rate, batch size, and epochs (PostTrainBench only)	–	✓
Model architecture, loss, optimizer logic, training algorithm, and model selection	–	–

This separation allows the agent to change which examples and representations enter training while preserving the learner's modeling components. On PostTrainBench, learning rate, batch size, and epochs are exposed as bounded configuration choices to accommodate curated-dataset size and memory requirements. The model architecture, objective, optimizer logic, and training algorithm remain fixed. This exception is specific to PostTrainBench; the MLE-Bench Lite setting retains the fixed training configuration described above.

B.3 Baseline Implementation Details

B.3.1 Baseline

DataFlex DataFlex (Liang et al., 2026) is a data-centric training framework built on LLaMA-Factory that dynamically controls data selection, mixture ratios, and sample reweighting during training. Unlike DataMaster, it does not perform autonomous dataset discovery; it assumes candidate datasets are already available and optimizes their usage during training.

Because DataFlex is built on LLaMA-Factory, it is only applicable to PostTrainBench (language model post-training) and cannot be evaluated on MLE-Bench Lite (heterogeneous ML tasks). For PostTrainBench evaluation, we provide DataFlex with the same curated dataset produced by DataMaster's best node, isolating DataFlex's training-time optimization from the data discovery process. In principle, DataMaster (data discovery and curation) and DataFlex (training-time optimization) are orthogonal and could be combined.

Claude Code CLI. Claude Code CLI (Anthropic, 2025) is Anthropic's command-line agent for agentic coding tasks. It has access to file operations, bash execution, and web search, and can iteratively refine code and data pipelines. We evaluate it on both benchmarks by providing the same initial setup as DataMaster.

Codex CLI. For MLE-Bench Lite, we use the Codex CLI implementation provided by the benchmark authors. For PostTrainBench, we report results using GPT-5.2-codex from the original PostTrainBench paper (Rank et al., 2026). We do not run our own Codex CLI evaluation on PostTrainBench because: (1) the compact API does not support external model endpoints, and (2) when integrated with GLM-5.1, Codex CLI frequently exceeds the maximum token limit and terminates prematurely, preventing long-horizon iterative refinement.

DatasetResearch. DatasetResearch (Li et al., 2025) is a pure data-search agent that discovers external datasets but does not perform code adaptation or training. For our evaluation, we run DatasetResearch to discover datasets, then use Claude Code CLI to adapt the discovered data into the training pipeline. To ensure fair comparison, we control the total training data volume to match DataMaster's best node.

ML-Master 2.0. ML-Master 2.0 (Zhu et al., 2026) is an end-to-end machine learning agent that can modify both data and code. We evaluate it on MLE-Bench Lite using the authors' released implementation and report its PostTrainBench results under the same benchmark protocol used for the other autonomous baselines.

B.4 Reference Points

Initial Score. The initial score represents the baseline performance before any autonomous data engineering.

- On **MLE-Bench Lite**, the initial score is obtained by running ML-Master 2.0's code with its data module removed. This isolates the contribution of the fixed training pipeline (model architecture, hyperparameters, training script) from any data-side improvements. The resulting medal rate (36.36%) and gold medal rate (22.73%) serve as the starting point for measuring data engineering gains.
- On **PostTrainBench**, the initial score corresponds to the base model **Qwen3-1.7B-Base** evaluated directly on the seven downstream tasks without any fine-tuning. This represents the zero-shot or few-shot performance of the pretrained model before any task-specific data curation or training. The average score of 8.47% reflects the base model's limited capability on specialized tasks without post-training.

Instruct Reference. On **PostTrainBench**, we report **Qwen3-1.7B-Instruct** as the instruction-tuned reference corresponding to **Qwen3-1.7B-Base**. Its average score of 46.91% provides a reference point for autonomous data engineering and is not treated as an upper bound or performance ceiling.

B.5 Per-Task Performance Analysis

We analyze DataMaster's per-task performance on 17 MLE-Bench Lite tasks (5 tasks excluded due to invalid initial submissions). We report *normalized gain*, which measures the best-node improvement relative to the median-to-gold score range, and *overcome rate*, the fraction of DataTree nodes that surpass the initial-node score. Fig. 3b visualizes both metrics across all tasks.

Key findings: ❶ DataMaster achieves non-negative gain on all 17 tasks, never degrading performance. Top performers show both high gain and high overcome rate (e.g., 102.33% gain with 85.71% overcome on an audio classification task). ❷ Gain and overcome rate are not always aligned: some tasks exhibit high gain with low overcome rate, where few nodes discover exceptional configurations; others show low gain with high overcome rate, where the initial score is already near-optimal yet DataMaster still improves steadily. ❸ Performance varies by domain: audio and sequence-to-sequence tasks achieve the highest gains, followed by tabular tasks, while image/text classification tasks show more variable gains—particularly when external data is scarce or train/test distributions are near-identical.

Normalized gain computation. For each competition, we compute a *normalized gain* that measures the best-node improvement relative to the score range between the median and gold thresholds:

$$\text{Gain}\% = \begin{cases} \dfrac{\text{best} - \text{initial}}{|\text{gold} - \text{median}|} \times 100 & \text{higher-is-better,} \\ \dfrac{\text{initial} - \text{best}}{|\text{gold} - \text{median}|} \times 100 & \text{lower-is-better.} \end{cases} \tag{3}$$

This normalization accounts for the varying difficulty across competitions: a 0.01 improvement on a tightly clustered metric may represent a larger relative achievement than a 0.05 improvement on a widely spread one.

Table 6 reports the full per-task results, including the initial-node score, best-node score, relative improvement, normalized gain, and overcome rate.

B.6 Data Leakage Safeguards and Audit

The safeguards in Section 4.4 address complementary stages of external-data use. Source filtering excludes benchmark pages, competition discussions, and URLs matching test-split names during acquisition. Hash-based deduplication filters training samples that exactly match stored test-set hashes. Provenance records retain the source URL, timestamp, and content hash for each external data point, supporting source-level inspection alongside sample-level checks.

Table 6: Per-task performance on 17 MLE-Bench Lite tasks with valid initial scores.

Competition	Dir.	Initial	Best	Rel. Impr.	Gain%	Overcome
mlsp-2013-birds	Higher	0.86753	0.93870	+0.07117	102.33%	85.71%
aptos2019-blindness	Higher	0.92106	0.96239	+0.04133	99.36%	20.00%
nomad2018	Lower	0.06033	0.05207	+0.00826	59.04%	88.89%
text-norm-english	Higher	0.99285	0.99575	+0.00290	42.21%	86.70%
plant-pathology	Higher	0.98142	0.99350	+0.01208	40.48%	80.00%
leaf-classification	Lower	0.02367	0.00004	+0.02363	21.81%	92.90%
spooky-author	Lower	0.23456	0.19931	+0.03525	13.89%	60.00%
text-norm-russian	Higher	0.97232	0.97393	+0.00161	11.33%	60.00%
dogs-vs-cats	Lower	0.01028	0.00331	+0.00697	8.36%	87.50%
random-acts-of-pizza	Higher	0.62265	0.64955	+0.02690	7.09%	76.92%
dog-breed	Lower	0.46888	0.44098	+0.02790	5.92%	57.89%
denoising-documents	Lower	0.03622	0.03341	+0.00281	5.08%	14.30%
tps-may-2022	Higher	0.99477	0.99583	+0.00106	4.15%	81.25%
whale-redux	Higher	0.98777	0.99109	+0.00332	2.67%	88.00%
tps-dec-2021	Higher	0.96330	0.96338	+0.00008	2.52%	58.80%
aerial-cactus	Higher	0.99999	1.00000	+0.00001	0.00%	40.00%
insults-detect	Higher	0.95637	0.95637	+0.00000	0.00%	0.00%

The GPQA audit compares 7,479 discovered training samples with 448 test samples. Figure 3c reports 0.00% exact matches and 0.00% fuzzy matches, with the fuzzy check labeled at a similarity threshold of 90%. The reported overlap rates for 3-, 4-, and 5-grams are 1.06%, 0.26%, and 0.08%, respectively. These complementary checks examine exact duplication, approximate matching, and shared short text sequences.

MLE-Bench Lite input-overlap audit. We extend the audit to all 22 MLE-Bench Lite tasks by comparing training inputs in each best-node data state against the corresponding test inputs. The audit uses label-free identifiers appropriate to each data format: MD5 hashes for images, shared predictive-feature vectors excluding identifier and target columns for tabular data, and task-specific input text for text/JSON data. Table 7 aggregates the counts by modality. The total rate is computed from the pooled counts of overlapping and test inputs, rather than by averaging task-level percentages.

Table 7: Input overlap between best-node training data and test data on MLE-Bench Lite. Rates use the test-input count in each row as the denominator.

Modality group	Tasks	Overlap / Test inputs	Rate
Image	7	5 / 80,596	0.0062%
Tabular	10	128 / 672,531	0.0190%
Text / JSON / shared-image	5	0 / 2,055,702	0.0000%
Total	**22**	**133 / 2,808,829**	**0.0047%**

The audit identifies 133 matching inputs among 2,808,829 test examples, yielding an aggregate overlap rate of 0.0047%. Most matches are tabular feature-vector matches: distinct records can share the same categorical, discretized, or missing-value features. We therefore count these conservatively as input overlaps. The audit compares inputs without using test labels and complements the source filtering, hash-based deduplication, and provenance records described above.

B.7 Cost Analysis

To assess the practical cost of DataMaster, we measure wall-clock time and API token consumption (input and output separately) across all MLE-Bench Lite runs. Table 8 compares DataMaster against baselines in terms of compute cost and resulting performance.

Table 8: Efficiency comparison on MLE-Bench Lite. Tokens are reported as average per-task input/output tokens; time is average wall-clock minutes per task. Performance columns reuse the main-table results.

Method	Input Tokens (K)	Output Tokens (K)	API Cost ($)	Time (min)	Medal Rate	Gold Rate
Claude Code	522.5	13.3	0.09	127.6	36.36%	22.73%
Codex	97.3	19.0	0.05	95.4	22.73%	18.18%
ML-Master 2.0	349.0	113.9	0.29	669.6	40.91%	27.27%
DatasetResearch	38.5	64.6	0.15	71.7	59.09%	27.27%
DataMaster	2690.3	596.3	1.61	718.7	68.18%	45.45%

Table 9: Task-wise MLE-Lite test scores and overcome rates. Direction indicates whether higher or lower scores are better for each competition metric.

Competition	**Direction**	**Test Initial Score**	**Test Best Score**	**Overcome Rate**
aerial-cactus-identification	Higher	0.99999	1.00000	40%
aptos2019-blindness-detection	Higher	0.92106	0.96239	20%
mlsp-2013-birds	Higher	0.86753	0.93870	85.71%
new-york-city-taxi-fare-prediction	Lower	None	4.73358	100%
plant-pathology-2020-fgvc7	Higher	0.98142	0.99350	80%
ranzcr-clip-catheter-line-classification	Higher	None	0.89733	None
spooky-author-identification	Lower	0.23456	0.19931	60%
detecting-insults-in-social-commentary	Higher	0.95637	0.95637	0.00%
dog-breed-identification	Lower	0.46888	0.44098	57.89%
dogs-vs-cats-redux-kernels-edition	Lower	0.01028	0.00331	87.50%
histopathologic-cancer-detection	Higher	None	0.99788	100%
jigsaw-toxic-comment-classification-challenge	Higher	None	0.98663	None
nomad2018-predict-transparent-conductors	Lower	0.06033	0.05207	88.89%
random-acts-of-pizza	Higher	0.62265	0.64955	76.92%
text-normalization-challenge-english-language	Higher	0.99285	0.99575	86.70%
denoising-dirty-documents	Lower	0.03622	0.03341	14.30%
leaf-classification	Lower	0.02367	0.00004	92.90%
siim-isic-melanoma-classification	Higher	0.90303	0.91516	92.86%
tabular-playground-series-dec-2021	Higher	0.96330	0.96338	58.80%
tabular-playground-series-may-2022	Higher	0.99477	0.99583	81.25%
text-normalization-challenge-russian-language	Higher	0.97232	0.97393	60%
the-icml-2013-whale-challenge-right-whale-redux	Higher	0.98777	0.99109	88%

Table 10: Average MLE-Lite search-tree and trajectory-step statistics by competition category.

Metric	**Image Cls.**	**Tabular**	**Text Cls.**	**Audio Cls.**	**Image Reg.**	**Seq-to-Seq**	**Image-to-Image**
node_count	29.6250	34.5000	89.5000	83.5000	22.0000	33.0000	16.0000
red_node_count	16.0000	17.7500	66.5000	61.5000	14.0000	15.5000	8.0000
black_node_count	12.6250	15.7500	22.0000	21.0000	7.0000	16.5000	7.0000
initial_node_count	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
avg_steps_per_node	30.7981	35.1650	22.5211	20.9376	36.6818	39.0038	40.7500
red_avg_steps_per_node	28.7049	31.9904	22.1431	18.0587	37.8571	39.8364	39.3750
black_avg_steps_per_node	35.5963	40.5089	25.3585	28.5417	36.7143	39.5919	46.2857

C TOOL-USE AND SOURCE-DISCOVERY DIAGNOSTICS

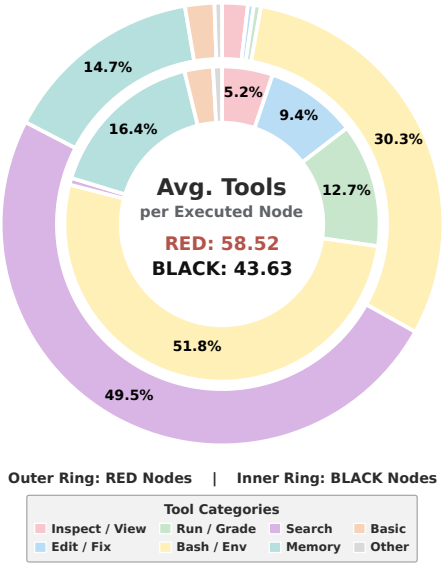


Figure 5: Tool-use composition of Red/Black nodes on MLE-Bench Lite.

Statistic Analysis We analyze the tool-use composition of red and black nodes on MLE-Bench Lite in Fig. 5. Key findings: ❶ red nodes use more tools per execution than black nodes on average (58.52 vs. 43.63), reflecting the interaction-heavy nature of external data discovery. ❷ Red-node actions are dominated by Search (49.5%), followed by Bash/Env (30.3%) and Memory (14.7%), showing that red nodes primarily explore candidate data sources, inspect artifacts, and reuse prior search context. ❸ Black-node actions are dominated by Bash/Env operations (51.8%), with non-trivial Run/Grade, Edit/Fix, and Inspect/View usage, indicating that black nodes focus on constructing executable data states and validating them through downstream execution rather than broad source discovery.

C.1 RED-NODE DISCOVERY AND SOURCE UTILIZATION

Red-node discovery is conditioned on the task, the current Data Pool, and Global Memory. The agent plans queries, discovers candidate sources, inspects their metadata and samples, screens compatibility, and registers reusable sources. These are functional stages rather than a fixed execution sequence: inspection results can trigger further queries or a revised search direction. We summarize source movement through these stages on the seven PostTrainBench tasks in Table 11.

Table 11: Discovery and utilization of external sources on PostTrainBench. Each rate names its denominator because queries, candidate events, inspection pairs, registrations, and unique datasets are different counting units.

Stage	Count	Rate and denominator
Search queries	1,344	–
Returned candidates	6,535	–
Inspected source pairs	887	13.6% of returned candidates
Successfully probed	702	79.1% of inspected pairs
Registration events	464	66.1% of successful probes
Unique canonical datasets	233	–
Used by Black nodes	130	55.8% of unique datasets
Retained in best data states	20	15.4% of used datasets

The discovery trace contains 887 inspected source pairs and 464 registration events, with 47.7% fewer registrations than inspections. Canonicalization yields 233 distinct datasets, of which 130 are used by at least one Black node and 20 are retained in the best-performing data states. The final set represents 15.4% of used datasets and 8.6% of all registered unique datasets. This progressive selection connects broad source discovery to downstream data construction: registration makes a source available for reuse, while execution and evaluation determine which sources enter the best data states.

D SCHEDULING POLICY

This appendix provides the full specification of the UCB1-based scheduling policy introduced in Section 3.5.

D.1 REWARD DESIGN AND BACKPROPAGATION

The DataTree contains two types of executable nodes with fundamentally different outputs: red nodes produce candidate datasets (no training score), while black nodes produce downstream evaluation metrics. To unify them under a single scheduling framework, we define the immediate reward as

$$r_v = \begin{cases} \epsilon, & \text{if } v \text{ is a red node and completes without error,} \\ y_v, & \text{if } v \text{ is a black node with evaluation score } y_v, \\ 0, & \text{otherwise (buggy or failed nodes),} \end{cases}$$

where ϵ is a small positive constant (we use $\epsilon = 0.01$ in all experiments). This ensures that successful data-discovery steps receive non-zero credit without dominating the exploitation signal from actual training outcomes.

Upon completion of node v with reward r_v, the statistics are updated along the entire root path. Let $\mathrm{Path}(v) = (v, p(v), p(p(v)), \ldots, \mathrm{root})$ denote the sequence of nodes from v to the root. For every $u \in \mathrm{Path}(v)$:

$$N_u \leftarrow N_u + 1, \tag{4}$$

$$R_u \leftarrow R_u + r_v. \tag{5}$$

That is, both v itself and all of its ancestors receive the same reward increment and visit increment. This backpropagation mechanism allows parent and grandparent nodes to accumulate evidence about the quality of their subtrees, enabling the scheduler to compare branches at different depths on a common scale.

D.2 NODE INITIALIZATION

When a new frontier node v is created by the growth policy, its statistics are initialized as

$$N_v = 0, \quad R_v = 0.$$

Since $N_v = 0$ would make the UCB1 score in Eq. (2) undefined (division by zero), we handle unvisited nodes as a special case:

$$\mathrm{Score}(v) = \begin{cases} +\infty, & \text{if } N_v = 0, \\ \dfrac{R_v}{N_v} + c_t \sqrt{\dfrac{\log N_{p(v)}}{N_v}}, & \text{if } N_v \geq 1. \end{cases}$$

This optimistic initialization guarantees that every newly generated frontier node is selected for execution before any previously visited node is revisited, ensuring that no branch is left completely unexplored. Once a node has been visited at least once ($N_v \geq 1$), both the exploitation term R_v/N_v and the exploration term are well-defined, and the standard UCB1 formula applies.

D.3 EXPLORATION COEFFICIENT DECAY

The exploration coefficient c_t controls the trade-off between exploiting high-reward branches and exploring under-visited ones. Let t denote the number of node executions completed so far (i.e.,

the global step counter). We adopt a piecewise linear decay schedule that transitions through three phases over the total budget of T rounds:

$$c_t = \begin{cases} c_0, & t < t_1, \\ \max\big(c_0 - \alpha\,(t - t_1),\ c_{\min}\big), & t_1 \le t \le t_2, \\ c_{\min}, & t > t_2, \end{cases}$$

where:

- c_0 is the initial exploration constant,
- α is the linear decay rate per step during the transition phase,
- $c_{\min}$ is the lower bound preventing purely greedy behavior,
- $t_1 = \lfloor p_1 \cdot T \rfloor$ and $t_2 = \lfloor p_2 \cdot T \rfloor$ are the phase boundaries, defined as fractions of the total budget.

The three phases serve distinct purposes:

1. **Exploration phase** ($t < t_1$): c_t remains at its maximum value c_0, encouraging broad coverage of the frontier when little information is available.
2. **Transition phase** ($t_1 \le t \le t_2$): c_t decays linearly, gradually shifting priority toward branches that have already demonstrated strong performance.
3. **Exploitation phase** ($t > t_2$): c_t is clamped at $c_{\min}$, concentrating the remaining budget on the most promising branches while retaining a minimal exploration incentive to avoid premature convergence.

D.4 Hyperparameter Settings

Table 12 lists the default scheduling configuration. For PostTrainBench, we set `num_black`$= 3$ instead of 5 because downstream training and evaluation are more expensive. These parameters specify candidate-generation limits, rather than the numbers of nodes that must be executed. The scheduler selects candidates from the frontier within the available budget, so some generated nodes remain unexecuted. Consequently, the configured Red–Black expansion ratio need not match the observed node-count ratio.

Table 12: Default scheduling hyperparameters.

Symbol	**Description**	**Value**
c_0	Initial exploration constant	$1.414\ (\approx\sqrt{2})$
$c_{\min}$	Lower bound on c_t	0.5
α	Linear decay rate	0.01
p_1	Exploration phase ratio	0.3
p_2	Transition end ratio	0.7
ϵ	Red-node reward	0.01
T	Maximum rounds (budget)	40
Tree structure parameters		
`num_red`	Red nodes per expansion	1
`num_black`	Initial black-node limit per red	5
`max_black_per_red`	Maximum black nodes per red	5

The choice of $c_0 = \sqrt{2}$ follows the theoretical recommendation for UCB1 when rewards lie in $[0, 1]$. The phase ratios $(p_1, p_2) = (0.3, 0.7)$ allocate roughly the first 30% of the budget to pure exploration, the middle 40% to transition, and the final 30% to exploitation. The lower bound $c_{\min} = 0.5$ ensures that even in the late exploitation phase, a branch with significantly fewer visits than its siblings retains a meaningful chance of being selected.

D.5 Alternative Decay Schedules

While we use the piecewise schedule by default, our implementation also supports two alternative decay strategies for ablation purposes:

Linear decay.

$$c_t = \max\big(c_0 - \alpha \cdot t,\ c_{\min}\big).$$

This provides a steady, monotonic decrease from the first step onward, without a dedicated exploration phase.

Exponential decay.

$$c_t = \max\big(c_0 \cdot \gamma^t,\ c_{\min}\big),$$

where $\gamma \in (0, 1)$ is the decay factor (default $\gamma = 0.99$). Exponential decay reduces exploration aggressively in early steps but converges more slowly to $c_{\min}$ compared to linear decay.

D.6 Priority Queue Implementation

The scheduler maintains a max-priority queue over the frontier $\mathcal{N}$. Each time a node completes and new children are generated by the growth policy, the new nodes are inserted with $\mathrm{Score}(v) = +\infty$. After node completion triggers backpropagation (Eqs. 4–5), existing frontier nodes are not re-scored eagerly; instead, scores are computed lazily at selection time. The next node v_{next} is popped from the queue, and execution proceeds. This design keeps scheduling overhead constant per step regardless of frontier size.

E Case Study

E.1 Branch Concentration Bias in Leaf Classification

We further analyze tree-growth asymmetry using the `leaf-classification` task as a case study. We reconstruct the search tree from the UCT node logs using each node's identifier and parent identifier. After excluding the artificial root and initial node, we treat the children of the initial node as root-level branches.

For each search step t, let $n_i(t)$ be the number of expanded nodes assigned to branch i, and let K be the total number of root-level branches. We first convert branch sizes into proportions:

$$p_i(t) = \frac{n_i(t)}{\sum_{j=1}^{K} n_j(t)}.$$

We then measure branch concentration using the normalized Herfindahl-style index:

$$\mathrm{Bias}(t) = \frac{\sum_{i=1}^{K} p_i(t)^2 - \frac{1}{K}}{1 - \frac{1}{K}}.$$

This value ranges from 0 to 1, where 0 indicates uniform expansion across branches and 1 indicates that all expanded nodes are concentrated in a single branch.

The `leaf-classification` tree contains 38 extracted nodes and 5 root-level branches after removing the artificial root and initial node. The final branch sizes are 23, 6, 3, 2, and 2 nodes, showing a clear imbalance in how expansion is allocated. As shown in Fig. 6, the bias decreases during the early search steps, suggesting initial diversification across branches. However, the later increase indicates that the agent increasingly concentrates expansion on a dominant branch. This supports the observation that the search process is not uniform; instead, it exhibits branch-level concentration as the tree develops.

E.2 Red–Black Exploration Ratio Analysis

To further examine the structural behavior of the search process, we compare Red-node and Black-node executions across competition categories. For each task, we compute two ratios. The first is

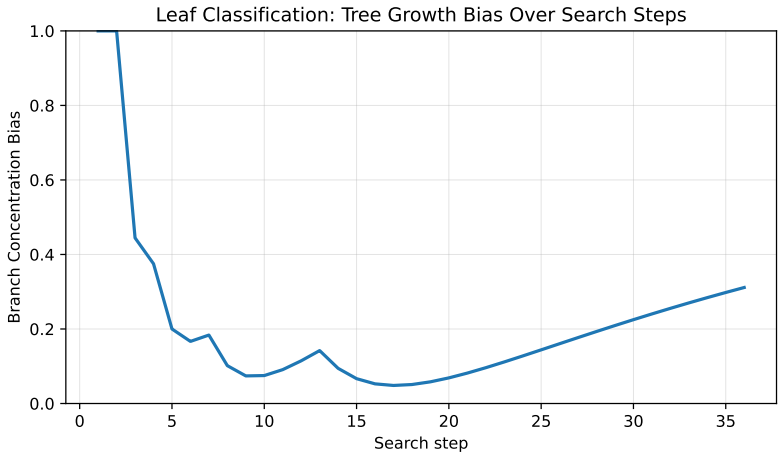


Figure 6: Branch Concentration Bias over search steps for the `leaf-classification` task. Higher values indicate more asymmetric tree growth. The curve initially decreases as the search expands across multiple branches, but later rises as one branch receives a larger share of subsequent expansions.

the executed-node ratio:

$$R_{\text{node}} = \frac{N_{\text{Red}}}{N_{\text{Black}}},$$

where N_{Red} and N_{Black} denote the numbers of executed Red and Black trajectory nodes. The second is the tool-intensity ratio:

$$R_{\text{tool}} = \frac{T_{\text{Red}}/N_{\text{Red}}}{T_{\text{Black}}/N_{\text{Black}}},$$

where T_{Red} and T_{Black} denote the total numbers of tool calls made by executed Red and Black nodes. Ratios are first computed at the task level and then averaged within each competition category.

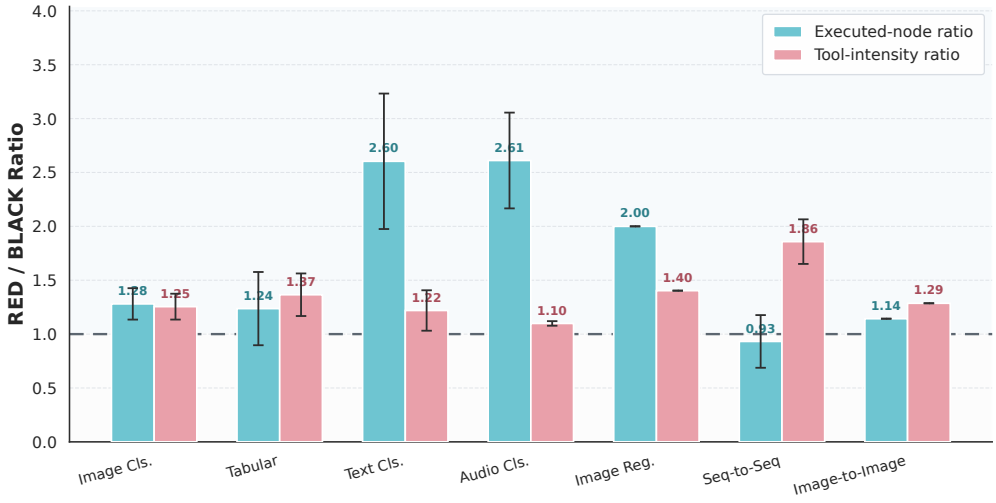


Figure 7: Red–Black exploration ratios by competition category. The blue bars show the executed-node ratio, while the pink bars show the tool-intensity ratio. The dashed horizontal line at $y = 1$ indicates parity between Red and Black nodes. Values above 1 indicate Red-node dominance, whereas values below 1 indicate Black-node dominance. Error bars denote the standard error of the mean across tasks within each category.

Overall, most competition categories show executed-node ratios above 1, suggesting that the agent tends to allocate more executed nodes to Red-node exploration than to Black-node implementation.

The tool-intensity ratio is also generally near or above 1, but it is usually less pronounced than the executed-node ratio. This indicates that the Red–Black imbalance is primarily structural, reflected in the number of executed nodes, while per-node tool-use intensity provides a secondary and more task-dependent source of imbalance.

E.3 Initial Code Examples

We present the initial code used in the dog-breed-identification task as a representative example of the two code settings in Section 4.3. Listing 1 shows the *DataLoader* component, which handles data loading, label encoding, train/validation splitting, and data augmentation and *Algorithm* component, which implements model construction, the training loop and validation. In the **Full-code** setting, both components are provided; in the **Algo-only** setting, only the Algorithm component is included and the agent must construct the DataLoader from scratch.

Listing 1: Initial code structure for dog-breed-identification (both DataLoader and Algorithm components).

```
# ==================== DataLoader Component ====================
class DogDataset(Dataset):
    def __init__(self, ids, labels, img_dir, transform, is_train):
        ...
    def __getitem__(self, idx):
        image = Image.open(...).convert("RGB")
        if self.transform: image = self.transform(image)
        return (image, self.labels[idx]) if self.is_train else image
class MyDataLoader(BaseDataLoader):
    def __init__(self, batch_size=32, num_workers=8,
                 input_dir="./input/", **kwargs): ...
    def setup(self):
        labels_df = pd.read_csv(os.path.join(self.input_dir, "labels.
            csv"))
        le = LabelEncoder()
        labels_df["breed_encoded"] = le.fit_transform(labels_df["breed
            "])
        # Train/val split (fixed val.csv or stratified 80/20)
        ...
        # Augmentation: Resize(384), RandomCrop(320), Flip,
            ColorJitter
        self.train_transform = transforms.Compose([...])
        self.val_transform = transforms.Compose([...])
        # Create DogDataset + DataLoader for train/val/test
        ...
# ==================== Algorithm Component ====================
class MyAlgorithm(BaseAlgorithm):
    def __init__(self, model_name='efficientnet_b3',
                 lr=1e-4, epochs=10, **kwargs): ...
    def fit(self, train_data):
        # Build pretrained EfficientNet-B3 via timm
        self.model = timm.create_model(self.model_name,
            pretrained=True, num_classes=num_classes)
        # AdamW optimizer + CosineAnnealingLR scheduler
        ...
        # Training loop with validation; save best checkpoint
        for epoch in range(self.epochs):
            # forward → loss → backward → step
            ...
            # validate and save best model
            ...
    def predict(self, test_data):
        self.model.eval()
        with torch.no_grad():
            for images in test_loader:
                # TTA: average(original, horizontal flip)
                outputs = self.model(images)
                outputs_flip = self.model(torch.flip(images, [3]))
                probs = (softmax(outputs) + softmax(outputs_flip)) / 2
                ...
        return pd.DataFrame({'id': test_ids, 'breed': predicted})
```

F ADDITIONAL EXPERIMENTS

This section supplements the main evaluation with additional ML-engineering baselines and a controlled comparison of scheduling policies. Both comparisons use the same 10-task MLE-Bench Lite subset as the component analysis in Section 4.3.

F.1 ADDITIONAL ML-ENGINEERING BASELINES

We compare DataMaster with AIDE (Jiang et al., 2025), MLE-STAR (Nam et al., 2025), CAAFE (Hollmann et al., 2023), and MLZero (Fang et al., 2025) under a 12-hour limit per task. Medal and gold medal rates use the official human-leaderboard thresholds. Table 13 distinguishes the optimization targets and whether external dataset discovery is enabled in the evaluated configuration. This comparison complements the main benchmark results by including systems that optimize code, model choices, or features over the provided training data.

Table 13: Additional baseline comparison on the 10-task MLE-Bench Lite subset. External data discovery refers to the evaluated configuration. Each task contributes 10 percentage points to a medal rate.

Method	**Optimization target**	**External discovery**	**Medal rate**	**Gold rate**
AIDE	Algorithm / code	–	30%	20%
MLE-STAR	Model refinement	–	20%	10%
CAAFE	Feature engineering	–	50%	20%
MLZero	Algorithm / code	–	70%	50%
DataMaster	Data discovery and adaptation	✓	**90%**	**80%**

DataMaster reaches medal thresholds on nine of ten tasks and gold thresholds on eight. Relative to MLZero, the strongest additional baseline in this comparison, the improvements are 20 percentage points in medal rate and 30 points in gold medal rate. The comparison shows strong performance for optimizing the data state, including external discovery and downstream adaptation, alongside approaches that primarily optimize the learner or the supplied features.

F.2 SCHEDULING POLICY ABLATION

To evaluate the contribution of frontier selection, we replace the UCB scheduler with round-robin expansion or uniform random selection. The growth policy, search budget, and remaining components are held fixed across the three configurations. Round-robin selects frontier nodes in order, random selection samples a frontier node uniformly, and UCB combines the accumulated reward with an exploration bonus as described in Appendix D. Table 14 reports the resulting medal rates on the 10-task subset.

Table 14: Scheduling policy ablation on the 10-task MLE-Bench Lite subset, holding the growth policy, budget, and other components fixed.

Scheduler	**Frontier selection**	**Medal rate**	**Gold rate**
Round-robin	Select nodes in order	50%	40%
Random	Uniform random selection	50%	50%
UCB (ours)	Reward-guided exploration and exploitation	**90%**	**80%**

UCB reaches medal thresholds on nine tasks, compared with five for each alternative. Its gold medal rate of 80% exceeds round-robin and random selection by 40 and 30 percentage points, respectively. These results support reward-guided frontier selection under the same growth policy and budget.