DSFlow: Evolutionary Workflow Optimization for Generalizable LLM-Based Data Science Automation

Anonymous ACL submission

Abstract

Large language model (LLM) agents create new opportunities for end-to-end data science automation, but existing systems often overfit to dataset-specific schemas, preprocessing routines, and code patterns, making them brittle under task shifts and costly to reuse. We propose `DSFlow`, a workflow-optimization framework that represents data-science solutions as compositions of reusable operators rather than monolithic, dataset-coupled programs. `DSFlow` evolves operator-based workflows to discover transferable structures, expands its operator library through synthesis and composition, and uses a two-stage evaluation strategy that filters candidates with inexpensive plan-level scoring before selectively running full executions. This design keeps optimization practical while improving transfer to unseen tasks. Experiments on heterogeneous benchmarks spanning tabular, image, text, and time-series modalities show that `DSFlow` achieves strong zero-shot generalization, reaching an average rank of 2.62. In addition, `DSFlow` grows its operator library from 10 manually designed operators to more than 70 discovered ones, improving accuracy from 0.8070 to 0.8448 and demonstrating scalable knowledge accumulation. Our code is available at https://anonymous.4open.science/r/data_science_dsflow-0E54.

1 Introduction

Recent advances in LLM agents have created new opportunities for automating data science (Sun et al., 2025; Liu et al., 2025a). Among the many tasks involved, end-to-end data science automation plays the central role: the LLM agent must translate task requirements into data processing, model development, and deployment for prediction services (Liang et al., 2025; Trirat et al., 2024a). The ability to automate this pipeline is critical, as it directly determines whether LLM agents can reduce dependence on expert knowledge and truly democratize data insights. However, building such agents remains challenging; current systems struggle with generalization, leaving fundamental questions about how to design robust LLM agents capable of reliable end-to-end data science execution.

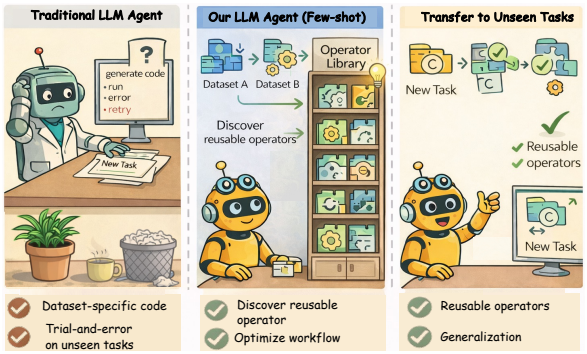

Figure 1: From dataset-specific code to transferable workflows via `DSFlow`. **Left**: Dataset-specific code generation causes traditional LLM agents to overfit seen tasks and struggle on new ones. **Middle**: `DSFlow` abstracts workflows into reusable operators and optimizes their composition on a few tasks. **Right**: Reusing learned operators enables efficient transfer to unseen tasks and improves generalization.

Due to its importance, automating data science has attracted growing interest, leading to a variety of LLM-based agents. Approaches such as DS-Agent (Guo et al., 2024) rely on adapting past solutions within fixed pipeline structures, while adaptive code-generation agents (e.g., AutoMind (Ou et al., 2025), AIDE (Jiang et al., 2025)) iteratively refine code solutions through execution feedback. Despite these advances, robust transfer across tasks and modalities remains limited because these systems optimize dataset-specific code rather than learning reusable workflow abstractions. Fixed pipelines are inherently rigid and degrade quickly under task or distribution shifts (Guo et al., 2024). Execution-driven code-generation methods offer more flexibility (Huang et al., 2022) but

their search procedures produce solutions tightly bound to dataset-specific schemas and preprocessing logic, requiring costly re-optimization whenever the task changes, as shown in Figure 1. As a result, little knowledge is retained across tasks. These limitations highlight the need for agents that learn transferable data science workflows, enabling few-shot optimization on seen tasks and principled generalization to unseen ones.

To this end, we represent data-science workflows as compositions of reusable operators rather than dataset-specific code. Operators act as transferable functional units (e.g., feature extraction, model training, evaluation), abstracting away dataset schemas and preprocessing heuristics and enabling workflow structures that generalize across tasks and modalities. However, optimizing such workflows is challenging because performance can only be assessed after full executions (often requiring hours of model training), which makes naïve exploration infeasible and further exacerbates difficulties posed by heterogeneous objectives and data types. Although genetic search is naturally suited for heterogeneous spaces, a direct application remains too costly and prone to producing task-specific solutions. We therefore propose `DSFlow`, which discovers transferable data-science workflows by evolving operator-based programs through mutation-only variation, hierarchical fitness evaluation, and elite-guided selection under restricted full executions. Specifically, `DSFlow` expands the operator library via operator synthesis and composition during variation, and applies hierarchical fitness evaluation that filters candidates with cheap plan-level scoring before selectively triggering full executions, enabling few-shot workflow optimization and principled generalization to unseen tasks.

In summary, our work makes the following contributions:

❶ **Problem Formulation.** We formulate end-to-end data-science automation as a workflow optimization problem over a structured space of operator-based programs. This formulation explicitly separates workflow structure, control flow, and LLM configurations, and targets cross-task generalization rather than dataset-specific code optimization.

❷ **Framework.** We introduce `DSFlow`, an operator-centric workflow optimization framework for LLM agents. `DSFlow` evolves reusable workflow programs via mutation-only evolutionary search, dynamically expands the operator library through operator synthesis and composition, and employs hierarchical fitness evaluation to balance exploration efficiency and execution cost.

❸ **Comprehensive Evaluation and Insights.** We conduct extensive experiments on a diverse suite of data-science tasks spanning multiple objectives (regression, classification, exploratory analysis) and modalities (tabular, image, text, and time-series). `DSFlow` consistently achieves superior cross-dataset and cross-task transferability compared to representative LLM-agent baselines, as shown in Table 1. Our analysis reveals three key insights: (i) operator-level workflow optimization substantially improves generalization over dataset-specific, code-centric pipelines; (ii) dynamic workflow optimization enables rapid adaptation to unseen tasks with minimal re-optimization; and (iii) synthesizing new operators improves performance by expanding reusable workflow primitives.

2 Related Work

LLM Agent Workflow Optimization. Recent work increasingly frames agentic workflow design as an optimization problem rather than a hand-crafted process (Yuksekgonul et al., 2024; Gao et al., 2025). AFlow (Zhang et al., 2024a) formulates workflow construction as program search via Monte Carlo Tree Search, while ADAS (Hu et al., 2024a) enables a meta-agent to autonomously write and refine workflow programs. Learning-oriented approaches such as AVATAR (Wu et al., 2024) further introduce contrastive reasoning and pseudo-gradient feedback to iteratively improve multi-step tool use and prompting. These developments collectively reflect a shift toward self-improving agents, where workflows are generated, evaluated, and optimized automatically instead of relying on static prompt chains (Wang et al., 2025). However, to the best of our knowledge, no prior work applies workflow optimization to data-science settings, which differ fundamentally from general reasoning tasks because workflow quality can only be assessed through expensive full executions (e.g., model training), making naive trial-and-error exploration infeasible.

LLM Data Science Agents. LLM-based data-

science agents aim to automate pipelines from EDA to model development, with recent advances emphasizing adaptive planning (Huang et al., 2022; Zhang et al., 2024b; Chan et al., 2024; Trirat et al., 2024b). InsightPilot demonstrates structured analysis actions for interpretable EDA (Ding et al., 2023), while AutoKaggle (Li et al., 2024) assigns specialized roles that iteratively generate and test code for end-to-end competition-style tasks. Case-based methods like DS-Agent (Guo et al., 2024) retrieve and adapt human-authored notebooks to stabilize planning, and learning-based systems such as ML-Agent (Liu et al., 2025b) use reinforcement learning to teach smaller LLMs ML engineering behaviors. However, these agents typically operate on dataset-specific code or fixed action schemas, limiting their ability to generalize across tasks. In contrast, `DSFlow` treats automation as workflow optimization over reusable operators, enabling transferable execution strategies that avoid costly re-optimization for each new dataset.

3 Preliminary

In this section, we first introduce the data science task and the LLM-based agent in Section 3.1. We then present the problem formulation for workflow optimization in Section 3.2. Representative example workflows are illustrated in Figures 10 and 11.

3.1 Data Science Task and Agent

Data Science Task. A data science task is defined as $x = (q, f, d)$, where q is a natural-language task description, f denotes raw data stored in heterogeneous formats (e.g., `.csv`, `.jpg`, `.dif`) spanning multiple modalities (e.g., tabular, image, and text), and d provides auxiliary context such as logs or retrieved documents. An LLM-based agent solves x by interacting with an execution environment through iterative code generation and execution, yielding a solution artifact (e.g., a trained model, an analytical report, or another task-specific outputs).

LLM Data-Science Agent. An LLM-based data-science agent solves a task by instantiating a sequence of data-science operators through programmatic code generation and interaction with an execution environment. Its behavior is represented as an operator program,

$$W = (\mathcal{P}, E, \lambda) \in \mathcal{S}, \tag{1}$$

where $\mathcal{P} = (\mathrm{Op}_1, \ldots, \mathrm{Op}_K)$ is an ordered operator sequence drawn from a predefined pool $\mathcal{O}$, E specifies control flow (e.g., sequencing, branching, retry). The configuration set $\lambda = \{\lambda_t\}_{t=1}^{K}$ provides operator-level LLM parameters, where each $\lambda_t = (M, \tau, \rho, T)$ controls the model selection, temperature, prompting strategy, and output formatting. $\mathcal{S}$ denotes a restricted space of operator programs.

Given a workflow state s_t, the LLM instantiates the operator-level policy and generates an action

$$a_t \sim \pi_\theta(\mathrm{Op}_t(s_t) \mid \lambda_t), \tag{2}$$

where $s_t = (x, o_0, \ldots, o_{t-1})$ accumulates the task description x and observation o_{t-1} (e.g., intermediate outputs, execution logs, and error signals) from previous steps. The action a_t typically corresponds to executable code, textual edits, analytical decisions, or other operator-specific instructions. The execution environment then returns an observation,

$$o_t = \mathrm{Exec}(\mathrm{Op}_t, a_t, s_t), \tag{3}$$

and iterative application produces a trajectory $y = (a_0, o_0, \ldots, a_T, o_T, a)$, where a denotes the final solution artifact (e.g., a trained model, an analytical report, or task-dependent outputs). As one illustrative case, a forward workflow may sequentially perform data acquisition, exploratory analysis, feature engineering, model training, and evaluation, with E enforcing the execution order and λ shaping the LLM's behavior at each operator.

3.2 Problem Formulation

Automated Workflow Optimization. Given a data-science task $x = (q, f, d) \in \mathcal{X}$ and a task-level utility function $G(W, x) \in \mathbb{R}$, the objective is to identify a workflow that achieves high performance on observed tasks while generalizing effectively to unseen tasks. We formulate this as a search problem over a workflow space $\mathcal{S}$. A search procedure $\mathcal{F}$ operates on $\mathcal{S}$ and returns a workflow candidate by exploring possible configurations of operator sequences, control-flow structures, and LLM configurations, i.e., $\mathcal{F} : \mathcal{S} \to W$. Formally, workflow optimization seeks,

$$W^* = \arg\max_{W \in \mathcal{S}} \mathbb{E}_{x \sim \mathcal{X}}[G(W, x)], \tag{4}$$

where $\mathcal{X}$ represents the task distribution. In practice, $\mathcal{F}$ may correspond to expert-defined rules,

gradient-free black-box optimization (e.g., evolutionary strategies or MCTS), or a meta-level LLM policy that incrementally synthesizes workflows conditioned on execution feedback. Thus, workflow optimization jointly concerns structural design ($\mathcal{P}$ and E), behavioral configuration (λ), and transferable generalization across tasks. In this work, we focus on the meta-level LLM policy setting to solve the automated workflow optimization problem.

4 DSFlow: Workflow Discovery and Optimization

4.1 Overview.

Formulation. Given a workflow $W = (\mathcal{P}, E, \lambda)$, we define a workflow optimization mapping $\mathcal{F}_{\text{DSFlow}} : \mathcal{S} \to W$ that aims to optimize W under the objective $\max_{W \in \mathcal{S}} \mathbb{E}_{x \sim \mathcal{X}}[G(W, x)]$. The objective is estimated by executing W on a task instance $x \sim \mathcal{X}$ and producing an execution trajectory $y = (a_0, o_0, \ldots, a_T, o_T, a)$. However, evaluating $G(W, x)$ for many workflow candidates requires repeatedly instantiating the operator–level policy $a_t \sim \pi_\theta(\text{Op}_t(s_t) \mid \lambda_t)$ and performing full executions, which is computationally prohibitive. To balance exploration efficiency and evaluation cost, `DSFlow` $\mathcal{F}_{\text{DSFlow}}(\cdot)$ formulates workflow discovery as a genetic optimization problem and performs evolutionary search over $W \in \mathcal{S}$ with restricted full executions. Each generation proceeds in four stages: Population Initialization, Variation, Hierarchical Fitness Evaluation, and Selection.

Task Scope and Workflow Generation. This work focuses on data modeling tasks (e.g., machine learning tasks, scientific discovery, etc.) across diverse task types and data modalities, aiming to improve the generalizability of workflow execution in realistic data science scenarios. To support cross–task transfer, a predefined operator pool provides initial building blocks, but `DSFlow` does not restrict search to these operators alone. Instead, $\mathcal{F}_{\text{DSFlow}}$ autonomously discovers and composes new operators in a continual manner, progressively expanding and refining the operator pool. Through this lifelong update process, the system searches for improved workflows W^* that generalize even to previously unseen data science tasks.

4.2 Evolutionary Search Algorithm

Recent workflow-optimization methods (Hu et al., 2024b; Zhang et al., 2024a) are mostly developed for generic reasoning tasks (e.g., symbolic reasoning or code synthesis), where feedback is immediate and trial-and-error is cheap. In data-science workflows, performance can only be assessed through expensive external execution (e.g., model training), which makes large-scale exploration impractical. The problem is further complicated by cross-task and cross-modality heterogeneity, spanning diverse objectives (regression, classification, clustering, and exploratory analysis) and data types (tabular, image, text, and time-series) (Luo et al., 2024). Although genetic search is a natural fit for such heterogeneous spaces (Fu et al., 2022; Hassan et al., 2023), a direct application still suffers from poor workflow generalization and excessive full-execution cost. `DSFlow` addresses these issues by (i) dynamically expanding the operator space via synthesis and composition of reusable operators and (ii) using a two-stage evaluator that filters candidates with cheap plan-level scoring before selectively running full executions, enabling few-shot optimization and cross-task transfer.

Population Initialization. We initialize the population by cloning a baseline workflow W_0, which specifies a valid triple $W_0 = (\mathcal{P}_0, E_0, \lambda_0)$ corresponding to a standard data-science pipeline (e.g., acquisition, EDA, feature engineering, training, and evaluation). To establish an initial fitness reference, a coarse proxy of the utility of W_0 is computed. In practice, an LLM serves as a judge and directly assigns a plausibility score (e.g., on a 1–10 scale), yielding a lightweight assessment of workflow reasonableness.

Variation. Since $W = (\mathcal{P}, E, \lambda)$ only permits shallow reordering of operator logic under recombination (e.g., swapping the order of EDA and planning), crossover offers limited contribution to operator capability or cross-task, cross-modality transfer. The evolutionary search is therefore implemented using mutation-only updates. For each non-elite position in the next population, a parent workflow $W = (\mathcal{P}, E, \lambda)$ is sampled from the current population and locally perturbed through programmatic edits:

- Introducing new operators into $\mathcal{P}$ by synthesizing an operator Op that does not previously exist in the operator pool $\mathcal{O}$ (e.g., creating data-cleaning operator that wraps regex-based text normalization into a single callable unit);

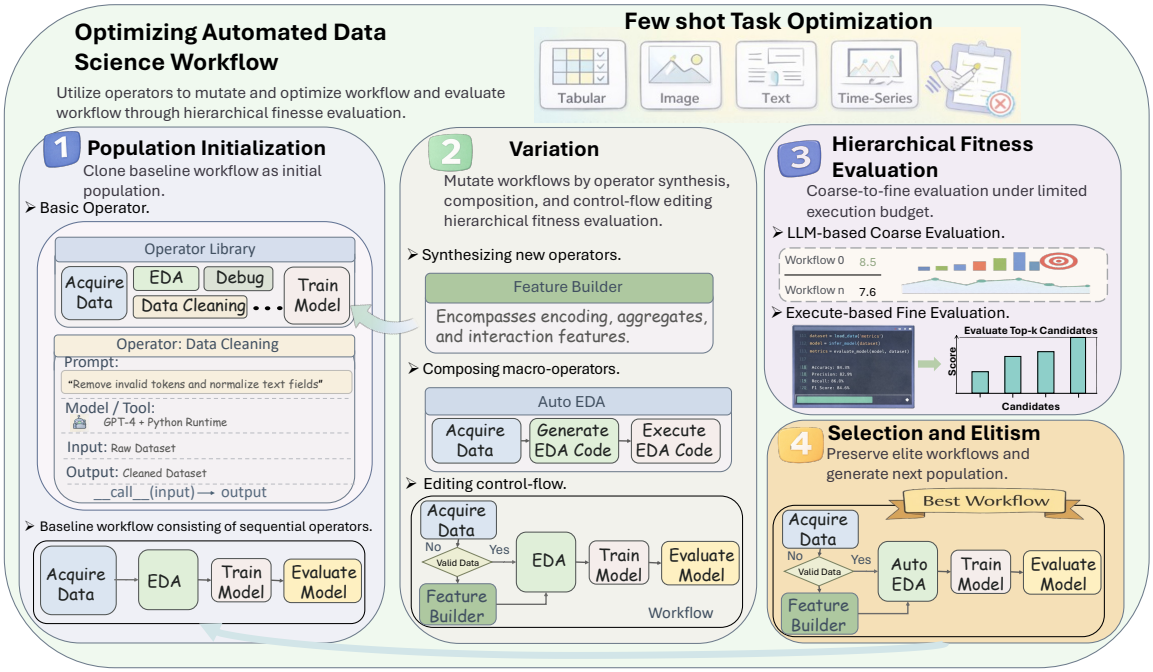

Figure 2: `DSFlow` is an evolutionary workflow optimization framework for few-shot, end-to-end data science automation across heterogeneous modalities. Given a few-shot task (tabular, image, text, or time series), it searches for a transferable workflow composed of reusable executable operators under a limited evaluation budget. In **Step (1) Population Initialization**, `DSFlow` clones a baseline workflow to initialize a population from an extensible *operator library*. In **Step (2) Variation**, workflows evolve through operator synthesis, operator composition, and control-flow editing, enabling exploration beyond fixed pipelines. In **Step (3) Hierarchical Fitness Evaluation**, an LLM-based judge first ranks workflows to prune weak candidates, then shortlisted workflows are executed for reliable metrics at lower cost. In **Step (4) Selection and Elitism**, top-performing workflows are retained to generate the next population until convergence. `DSFlow` is evaluated on *unseen* benchmark datasets in a zero-shot setting, demonstrating robust transfer through operator-level abstractions.

- Composing multiple low-level operators into a higher-level macro-operator to enable more complex functionality (e.g., chaining data acquisition, code generation, and execution into an automatic EDA operator);

- Altering control-flow instructions in E (e.g., inserting conditional retry or skip logic) to enhance execution robustness through constructs such as `if–else` and loop-based self-debugging.

These mutations generate $M-1$ distinct workflow variants in $\mathcal{S}$ around the elite parent, expanding the search over the operator-program space while preserving syntactic validity. These mutations fill all non-elite slots in the next population and keep the population size M fixed. Throughout evolution, W_0 (or its most recent elite successor) is always retained as the guaranteed elite genome, serving as a stable reference point for the search. Under this setup, we represent the population at generation g as $\mathcal{W}_g = \{W_g^{(1)}, \ldots, W_g^{(M)}\}$, which contains M alternative realizations of $(\mathcal{P}, E, \lambda)$ within the search space $\mathcal{S}$.

Hierarchical Fitness Evaluation. Directly computing the utility function $G(W, x)$ for all candidates requires full execution of the trajectory y on a set of tasks $x \sim \mathcal{X}$, including model training and inference, which are prohibitively expensive. Therefore, we adopt a two-stage fitness mechanism aligned with the task utility:

- *Coarse evaluation.* All candidates are partially executed to score the feasibility of instantiating $W = (\mathcal{P}, E, \lambda)$ on task inputs $x = (q, f, d)$. Concretely, the workflow is run only through the planning and code-generation phases, stopping before expensive training and large-scale execution. The resulting proxy score reflects structural soundness, code completeness, and executability, providing a low-cost approximation of $G(W, x)$ serves as the primary optimization signal for the population evolution.

- *Fine evaluation.* Only an elite subset, typically the baseline and the top-k candidates under coarse scoring, is fully executed to obtain the true task utility $G(W, x)$ (i.e., the benchmark score induced by the terminal action a in the trajectory y.). Fine evaluation may be deferred to the end of the search (default mode, where only the final elite set is evaluated) or applied once per generation in a high-fidelity mode. In both cases, it provides high-precision estimates of $G(W, x)$ while keeping the number of full executions bounded.

Selection and Elitism. At the coarse level, candidates in $\mathcal{W}_g$ are ranked by their coarse fitness. The highest-scoring candidate is deterministically selected as the elite genome $W_g^\star$ and propagated unchanged to the next generation, where $W_g^\star = (\mathcal{P}_g^\star, E_g^\star, \lambda_g^\star)$. The remaining members of $\mathcal{P}_{g+1}$ are generated by mutating $W_g^\star$ to exploit improvements in $\mathcal{S}$ while preserving exploration through stochastic variation. When fine evaluation is used during the search, fine scores can optionally be combined with coarse scores (e.g., via a weighted sum) to stabilize selection under noisy coarse signals.

The genetic search terminates after a fixed number of generations or when the fine-evaluated elite score shows no improvement across several rounds, indicating convergence under the given computation budget. The final workflow W^* is then selected from the fine-evaluated elite set (e.g., the baseline and top-k candidates), providing an approximate maximizer of Eq. (4) over $\mathcal{S}$. The discovered workflow is subsequently added to the operator library as a reusable high-level template, enriching the space of candidate programs for future tasks. To avoid redundant library growth, each newly added operator is registered only if its unique `(name, code_hash)` signature, computed from a normalized implementation body, does not already exist in the library. In practice, optimization is performed on only a few-shot set of data-science tasks $\mathcal{X}_{\text{val}}$ by solving $\max_{W \in \mathcal{S}} \mathbb{E}_{x \sim \mathcal{X}_{\text{val}}}[G(W, x)]$. The resulting workflow W^* is then evaluated on a disjoint set of unseen tasks $\mathcal{X}_{\text{test}}$, enabling a direct assessment of cross-task generalization. We provide a complexity analysis of `DSFlow` in Appendix A. Appendix C.4 reports real token usage across agents; Figure 8 shows that `DSFlow` consistently reduces input tokens by operating on compact, reusable workflow representations rather than repeatedly encoding dataset-specific details into prompts.

5 Experiments

We evaluate whether `DSFlow` discovers transferable workflows and isolate the effects of workflow optimization, hierarchical evaluation, and operator expansion through three research questions:

RQ1: Does workflow evolution improve generalization?

RQ2: How much do optimization and hierarchical evaluation contribute?

RQ3: Does operator expansion improve capability?

RQ4: Can DSFlow bootstrap dataset-specific search?

5.1 Experiments Setup

Datasets and Task Suite To evaluate cross-task generalization, we construct a task suite spanning diverse learning objectives and data modalities. In total, we evaluate *13* tasks across three task categories and *7* data modalities. The suite includes classical machine-learning tasks, Kaggle-style competition tasks, and more challenging open-ended scientific discovery tasks, covering regression, classification, and exploratory analysis over tabular, image, text, and time-series datasets. This setting reflects realistic data-science workloads, where workflow policies must generalize across heterogeneous tasks and pipelines. Detailed task descriptions are provided in Appendix C.1.

LLM Agents We benchmark `DSFlow` against several LLM-based agent baselines, including manual-style pipelines agents: AutoKaggle (Li et al., 2024), Data Interpreter (Hong et al., 2025), DSAgent (Guo et al., 2024), and DeepAnalyze (Zhang et al., 2025), and search-based workflow agent: AIDE (Jiang et al., 2025), AutoMind (Ou et al., 2025), and Aflow (Zhang et al., 2024a). Each agent is configured with the same backbone LLM and access to identical tool APIs to ensure fair comparison. The details of baseline agents can be seen in Appendix C.2

Implementation Details. Each agent is run on a single GPU with 40 GB memory, and each sandbox execution is capped at a maximum wall-clock time of six hours per dataset. It is important to note that only `DSFlow` and AFlow perform workflow optimization on a small set of seen tasks and are subsequently evaluated on disjoint unseen tasks, providing a direct measure of cross-task transfer-

Table 1: Performance comparison across agents. It is noted that `DSFlow` workflows are executed on these unseen datasets, i.e., the workflow optimizer is evaluated in a zero-shot setting without using any of the benchmark datasets during workflow optimization. Colorful icons indicate data modality (Tx: text, Tb: tabular, TS: time series, ST: spatiotemporal, Im: image, Mi: microscopy image, 3D: 3D shape/pharmacophore-derived features) and problem type (BC: binary classification, MC: multiclass classification, RG: regression, FC: forecasting). ↑(↓) indicates higher(lower) is better. **Bold** numbers indicate the best performance among agents for each dataset. × denotes missing results (not supported / not applicable / not successful submission). **-Opt** disables operator expansion (i.e., no discovered operators are added to the operator library), while **-Hier** removes hierarchical fitness evaluation (i.e., candidates are scored without the coarse-to-fine evaluation pipeline). **Avg. Rank**↓ is the average rank of each agent across all datasets when ranking agents by their performance on each dataset (smaller is better). **Success Rate**↑ is the percentage of datasets for which an agent produced a valid result (larger is better).

Dataset	AutoKaggle	Data Interpreter	DSAgent	DeepAnalyze	AIDE	AutoMind	AFlow	-Opt	-Hier	DSFlow (ours)
bike-sharing-demand Tb TS FC↓	0.330	0.347	0.303	0.338	0.295	**0.293**	0.303	0.550	0.560	0.431
handwriting TS MC↑	×	0.222	0.179	×	0.139	**0.542**	0.059	0.058	0.106	0.113
histopathologic-cancer-detection Im BC↑	0.500	0.361	0.378	×	0.946	**0.993**	×	0.888	0.600	0.935
ili TS FC↓	×	5.502	5.512	10000000000.000	5.939	5.173	5.310	5.316	5.211	**4.983**
liverpool-ion-switching TS MC↑	×	0.036	×	×	×	×	×	0.699	0.227	**0.828**
ventilator-pressure-prediction TS RG↓	×	5.646	×	4.452	×	×	**1.708**	10.297	3.995	1.927
demand-forecasting-kernels-only TS FC↓	×	12.285	11.292	×	11.987	11.533	×	12.245	×	**11.218**
instant-gratification Tb BC↑	×	0.503	**0.575**	×	0.562	0.546	×	×	×	0.558
see-click-predict-fix ST RG↓	0.838	0.374	**0.368**	×	0.417	0.370	×	×	×	0.473
playground-series-s3e1 Tb RG↓	×	0.602	0.629	×	0.600	0.605	0.598	×	×	**0.593**
playground-series-s3e11 Tb RG↓	×	0.306	×	0.304	0.303	×	0.304	0.303	0.302	**0.302**
dkpes-model-development-1 Tb 3D RG↓	0.906	0.938	1.000	0.938	×	0.969	**0.844**	×	**1.000**	**0.844**
bbbc002-cell-count Mi RG↑	**78.000**	47.847	13.993	×	×	60.370	×	70.980	33.605	**78.000**
Avg. Rank↓	4.30	5.12	4.45	5.90	3.56	2.90	3.50	5.11	5.06	**2.62**
Success Rate↑	38%	**100%**	77%	38%	69%	77%	54%	69%	69%	**100%**

ability. Other agents rely on code-specific optimization strategies that do not generalize across datasets. For example, AIDE employs a code-centric search procedure with three hand-designed actions (improve, review, and debug), each producing an end-to-end code block tailored to the current dataset, which limits transfer to new tasks and modalities. For `DSFlow`, workflow optimization is performed on only three datasets, and the optimized workflow is then directly evaluated on all remaining unseen tasks. Full implementation details and model configurations are provided in Appendix C.3.

5.2 Evaluation and Findings

5.2.1 (RQ1): Main Experiments

We evaluate multiple LLM-based agents on the data science task, covering diverse unseen datasets across tabular, time-series, spatiotemporal, natural images, microscopy images, and tabular+3D-derived molecular features, as shown in Table 1. It is noted that `DSFlow` is evaluated in a strict zero-shot setting: its workflow optimizer does not use any of the benchmark datasets during workflow optimization. Due to the page limit, we provide additional analyses on other base models and token consumption in Appendix C. Our observations (Obs.) are as follows:

Obs.❶ `DSFlow` delivers strong zero-shot performance and robustness. `DSFlow` achieves the best **Avg. Rank** (2.38) and joint-best **Success Rate** (100%), with the best score on 7/13 datasets and a tie on *dkpes-model-development-1*, showing reliable transfer across heterogeneous tasks. **Obs.❷ `DSFlow` balances accuracy and efficiency.** While AFlow is faster (0.64h), it has weaker rank (3.56) and lower success (62%). `DSFlow` maintains perfect success with moderate runtime (1.67h), far below heavier baselines such as AutoKaggle (10.01h) and AIDE (12.67h), indicating efficient allocation of compute to high-value candidates. **Obs.❸ Remaining failures reveal room for specialized operators.** `DSFlow` still lags on *bike-sharing-demand* (0.431 vs. 0.293), *handwriting*, and *see-click-predict-fix*, suggesting the need for more modality-specific preprocessing or modeling while preserving zero-shot transfer.

5.2.2 (RQ2): Ablation Study

We conduct an ablation study to answer **RQ2** by evaluating two variants of `DSFlow`: (i) **-Opt**, which disables dynamic workflow optimization and uses a fixed baseline workflow; and (ii) **-Hier**, which removes the proposed coarse-to-fine evaluation and relies only on coarse-grained scoring. Results are reported in Table 1. ***Obs.❹ Dynamic workflow optimization and hierarchical evaluation are both essential and complementary for robust zero-shot performance.** Removing either component consistently degrades performance. Disabling optimization (**-Opt**) causes the largest drop (Avg. Rank: 5.11 vs. 2.62; Success Rate: 69% vs. 100%), with notable failures on challenging time-series and scientific datasets (e.g., *ili* and *bbbc002-cell-*

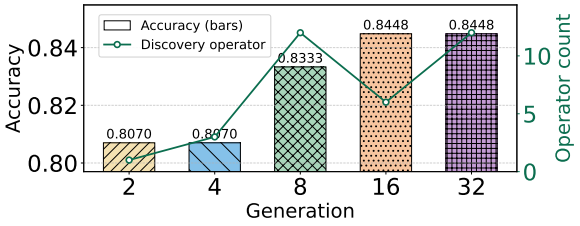

Figure 3: Impact of operator expansion generations on performance and operator discovery.

count). Removing hierarchical evaluation (**-Hier**) also hurts performance (Avg. Rank: 5.06), showing that coarse scoring alone cannot reliably identify high-quality workflows under limited budgets. Overall, the full `DSFlow` consistently outperforms both ablations, indicating that adaptive workflow evolution and hierarchical evaluation provide complementary gains for stable transfer to unseen tasks.

5.2.3 (RQ3): Dynamic Operator Expansion

In this subsection, we study whether enabling `DSFlow` to *synthesize new operators* improves generalization compared to a fixed operator set. We vary the operator-expansion budget by changing the number of expansion generations (2/4/8/16/32). In each generation, `DSFlow` performs workflow variation via operator synthesis and composition, and evaluates candidates using hierarchical fitness evaluation. For a direct and interpretable quantification, we conduct this study on a data-analysis task where accuracy can be reported straightforwardly. We report (i) the best achieved accuracy and (ii) the number of newly discovered operators.

Obs.❺ Operator expansion improves performance and then saturates. As shown in Figure 3, small budgets (2–4 generations) remain near 0.807 accuracy, indicating limited transfer with a nearly fixed operator set. Expanding to 8 generations raises accuracy to 0.8333 as new operators add reusable capabilities, and 16 generations reaches the best result (0.8448). Further expansion to 32 generations brings no additional gain, suggesting diminishing returns once the library covers the main transferable patterns.

6 (RQ4): Bootstrapped Search Refinement

Although `DSFlow` improves cross-task generalization, it is not designed to exhaustively search each target dataset at test time. To recover peak dataset-specific performance when additional compute is available, `DSFlow` can serve as a cold-start initializer for search-based agents such as AIDE. The downstream search begins from a `DSFlow`-derived workflow that already captures robust preprocessing, operator ordering, validation, and failure-handling patterns, allowing the search budget to focus on high-impact refinements rather than basic pipeline repair. Empirically, this hybrid `DSFlow`-AIDE setup consistently improves over `DSFlow` alone across nearly all evaluated datasets, with notable gains on *handwriting* (0.1129→0.5765), *bike-share* (0.4308→0.3085), *histo-cancer* (0.9352→0.9615), and *ventilator-pp* (1.9273→1.7753). We provide the full procedure, analysis, and additional results in Appendix B.

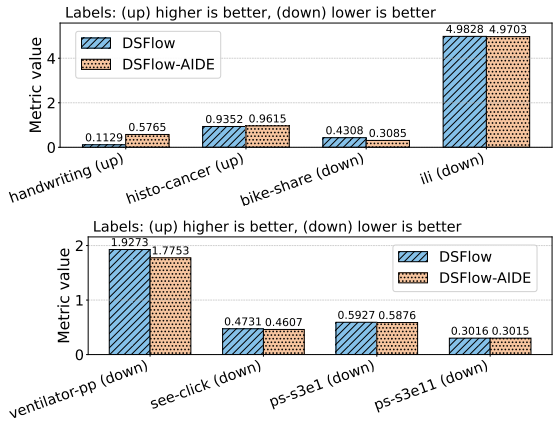

Figure 4: `DSFlow`-initialized AIDE improves downstream search refinement across unseen tasks.

7 Conclusion

We study how to make LLM-based data-science agents generalize beyond dataset-specific code and brittle preprocessing. We propose `DSFlow`, which casts end-to-end automation as workflow optimization over reusable operators rather than monolithic programs. `DSFlow` evolves transferable workflow structures, expands its operator library through synthesis and composition, and uses hierarchical evaluation to filter candidates with cheap plan-level scoring before selective full execution. This design keeps search practical under expensive executions while improving transfer to unseen tasks. Experiments across diverse tasks show that workflows learned from only a few examples transfer reliably, improving robustness and reducing re-optimization versus prior agents.

Limitations

Despite the significant improvements in generalization achieved by our approach, it does not consistently reach state-of-the-art performance across all tasks. A key limitation is that our method does not explicitly perform large-scale or exhaustive search, nor does it substantially increase test-time computation. Prior work has demonstrated that, for certain tasks, performance can be further improved through explicit search or increased computational budgets at inference time. As a result, our framework may be less effective in settings where such search-intensive strategies are critical for achieving peak performance. Exploring tighter integration between our workflow and search-based optimization methods remains an important direction for future work.

References

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

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, et al. 2024. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. *arXiv preprint arXiv:2410.05080*.

R Ding, S Han, and D Zhang. 2023. Insightpilot: An llm-empowered automated data exploration system. In *EMNLP 2023. ACL special interest group on linguistic data (SIGDAT)*.

Tianfan Fu, Wenhao Gao, Connor Coley, and Jimeng Sun. 2022. Reinforced genetic algorithm for structure-based drug design. *Advances in Neural Information Processing Systems*, 35:12325–12338.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. 2025. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*.

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*.

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

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

Shengran Hu, Cong Lu, and Jeff Clune. 2024a. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024b. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

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

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, and Nan Duan. 2022. Execution-based evaluation for data science code generation models. In *Proceedings of the Fourth Workshop on Data Science with Human-in-the-Loop (Language Advances)*, pages 28–36.

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

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

Zujie Liang, Feng Wei, Wujiang Xu, Lin Chen, Yuxi Qian, and Xinhui Wu. 2025. I-mcts: Enhancing agentic automl via introspective monte carlo tree search. *arXiv preprint arXiv:2502.14693*.

Fan Liu, Zherui Yang, Cancheng Liu, Tianrui Song, Xiaofeng Gao, and Hao Liu. 2025a. Mm-agent: Llm as agents for real-world mathematical modeling problem. *arXiv preprint arXiv:2505.14148*.

Zexi Liu, Jingyi Chai, Xinyu Zhu, Shuo Tang, Rui Ye, Bo Zhang, Lei Bai, and Siheng Chen. 2025b. Ml-agent: Reinforcing llm agents for autonomous machine learning engineering. *arXiv preprint arXiv:2505.23723*.

Daqin Luo, Chengjian Feng, Yuxuan Nong, and Yiqing Shen. 2024. Autom3l: An automated multimodal machine learning framework with large language models. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 8586–8594.

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

Maojun Sun, Ruijian Han, Binyan Jiang, Houduo Qi, Defeng Sun, Yancheng Yuan, and Jian Huang. 2025. A survey on large language model-based agents for statistics and data science. *The American Statistician*, pages 1–14.

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

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

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. 2025. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*.

Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis Ioannidis, Karthik Subbian, Jure Leskovec, and James Y Zou. 2024. Avatar: Optimizing llm agents for tool usage via contrastive reasoning. *Advances in Neural Information Processing Systems*, 37:25981–26010.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2024a. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

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

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

A Complexity Analysis

Let $F(T)$ denote the cost of a full workflow evaluation over T tasks, and C the cost of a cheap planning-only evaluation without code execution. Let G be the number of generations, M the population size, and k the number of candidates selected for fine evaluation. A standard genetic algorithm (GA) evaluates all candidates with full execution in every generation, incurring a total cost of $O(G \cdot M \cdot F(T))$, which scales directly with T. In contrast, `DSFlow` adopts a two-stage evaluation strategy: all candidates undergo cheap screening at cost $O(G \cdot M \cdot C)$, while only the baseline and top-k candidates are fully evaluated once, yielding an additional cost of $O((k+1) \cdot F(T))$. Since $F(T) \gg C$ in practice, the dominant term becomes $(k+1) \cdot F(T)$, reducing the number of expensive evaluations by a factor of approximately $(G \cdot M)/(k+1)$ compared to GA. Even when fine evaluation is performed at each generation, the cost reduces to $O(G \cdot k \cdot F(T))$, achieving a linear reduction factor of M/k. Crucially, as $F(T)$ grows with the number of tasks, `DSFlow` confines this growth to only $k+1$ evaluations rather than $G \cdot M$, making it substantially more scalable in multi-task settings.

B Bootstrapping Workflow Search with DSFlow Initialization

Although `DSFlow` substantially improves generalization, it does not attain state-of-the-art performance on every task. A key reason is that `DSFlow` optimizes workflows for transfer across datasets, rather than performing intensive test-time search directly on each target dataset. In contrast, search-based methods can often boost peak performance by spending more test-time computation to repeatedly propose, execute, and refine dataset-specific pipelines, albeit at a higher runtime cost.

To bridge this gap, `DSFlow` can serve as a strong *cold-start initializer* for downstream search-based agents. Instead of starting from scratch, the search procedure begins from a `DSFlow`-derived workflow that already encodes effective operator choices, ordering, and failure-robust execution patterns. This improves the starting point of search, reduces early trial-and-error, and allows the search budget to focus on high-impact refinements.

Procedure. Given a target dataset, we first run `DSFlow` to produce an optimized workflow skeleton (e.g., preprocessing operators, model family, validation protocol, and training/evaluation structure). We then use this workflow as the initial solution for a search-based agent (e.g., AIDE), which performs dataset-specific refinement through its iterative improvement loop. In this hybrid setup, `DSFlow` provides transferable structure and robustness, while the search-based agent contributes additional dataset-level exploration and tuning.

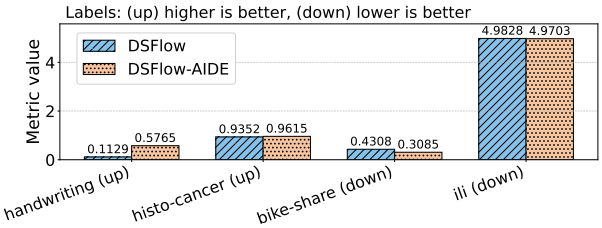

Figure 5: `DSFlow` initialization improves downstream search-based refinement (AIDE) on diverse unseen tasks. We compare `DSFlow` alone with `DSFlow`-AIDE (`DSFlow`-initialized AIDE) on four benchmarks. Values above bars are the final test metrics; (↑) means higher is better and (↓) means lower is better. `DSFlow`-AIDE consistently improves performance over `DSFlow`(e.g., handwriting: 0.1129→0.5765; histo-cancer: 0.9352→0.9615; bike-share: 0.4308→0.3085; ili: 4.9828→4.9703).

Why it helps. `DSFlow` initialization raises the initial quality of candidate solutions and reduces wasted search steps caused by brittle preprocessing or mis-specified evaluation. Consequently, the downstream search agent can devote more compute to meaningful modeling improvements (e.g., hyperparameter tuning, metric alignment, and ensembling) rather than repeatedly fixing basic pipeline issues. This hybrid strategy is especially beneficial on tasks where peak performance depends on extensive test-time exploration.

Results. As shown in Figure 5 and 6, bootstrapping AIDE with a `DSFlow`-initialized workflow yields consistent gains over `DSFlow` across nearly all datasets. In particular, `DSFlow`-AIDE achieves large improvements on challenging tasks such as *handwriting* (0.1129→0.5765, (↑)) and *bike-share* (0.4308→0.3085, (↓), 28.4% reduction), while also improving *histo-cancer* (0.9352→0.9615, (↑)) and *ventilator-pp* (1.9273→1.7753, (↓), 7.9% reduction). We refer to this `DSFlow`-initialized refinement stage as *fine-tuning*, since it specializes a transferable workflow into a target-dataset solution under additional test-time compute.

C Experiments

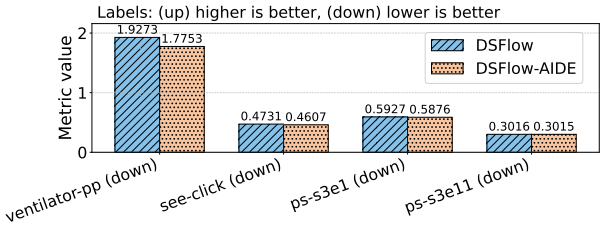

Figure 6: Additional results of `DSFlow`-initialized refinement across regression-style tasks. `DSFlow`-AIDE improves or matches `DSFlow` on all four datasets with (↓) metrics (ventilator-pp: 1.9273→1.7753; see-click: 0.4731→0.4607; ps-s3e1: 0.5927→0.5876; ps-s3e11: 0.3016→0.3015), showing that `DSFlow` provides a strong and stable starting workflow for search-based specialization.

C.1 Dataset Details

In this section, we further introduce the datasets used to evaluate the transferability of our framework. Although recent advances have made significant progress in automating data science workflows with large language models, most existing studies primarily focus on improving performance on narrowly defined downstream benchmarks, such as MLE-Bench (Chan et al., 2024). Moreover, these evaluations typically involve simplified data science tasks with limited data modality diversity. To address these limitations, we conduct a comprehensive experimental study across a diverse suite of heterogeneous tasks spanning tabular, time-series, image, spatiotemporal, microscopy image, and 3D shape– and pharmacophore-derived modalities. Importantly, our evaluation includes more open-ended tasks that emphasize data-driven scientific discovery, thereby providing a more realistic and holistic assessment of LLMs for real-world data science practice (Hu et al., 2024c; Chen et al., 2024). We conduct experiments on a diverse suite of heterogeneous tasks spanning tabular, time-series, image, spatiotemporal, microscopy image, and 3D shape/pharmacophore-derived modalities. Table 3 summarizes the detailed specifications of these datasets. Specifically, the benchmark encompasses a wide range of task types, including regression, binary and multiclass classification, multi-label prediction, forecasting, and counting, with evaluation metrics selected to align with task characteristics (e.g., RMSE, RMSLE, MAE, SMAPE, Accuracy, Macro-F1, and AUC-ROC).

C.2 Baselines

We compare our method with a diverse set of representative baselines spanning both manual-style pipelines and search-based workflow optimization. Specifically, AutoKaggle is a classical Kaggle-solving system that relies on templated feature engineering and model-training pipelines to produce submission-ready solutions quickly. Data Interpreter is analysis-oriented: it emphasizes dataset understanding and preprocessing by generating summaries, cleaning routines, and visualization driven modeling suggestions. DSAgent is a general-purpose data science agent that solves end-to-end tasks through an iterative analyze, model, evaluate and refine loop. DeepAnalyze further strengthens the analysis stage with deeper diagnostics and error analysis to guide subsequent modeling decisions in an end-to-end agentic manner.

On the search-based side, AIDE is code-centric: it iteratively generates and revises dataset-specific end-to-end code via predefined improve, review, and debug actions to optimize performance on the current task. AutoMind adopts a multi-round planning and execution strategy with self-reflection and error correction to progressively refine solutions. Finally, AFlow is originally designed for workflow-level optimization in general reasoning tasks; we adapt it to data science automation by instantiating its search-and-evaluate framework over data science workflows, and use it as a workflow-level baseline that iteratively explores candidate workflows and scores them based on execution feedback.

C.3 Implementation Details

For `DSFlow`, workflow optimization follows a two-stage screening strategy consisting of a coarse evaluation stage and a fine evaluation stage. In the coarse stage, all candidate workflows are assessed based on planning quality without executing generated code, while in the fine stage, only shortlisted candidates are fully executed and evaluated. In our setup, the population size is set to $M = 3$, the number of generations to $G = 4$, and the number of candidates selected for fine evaluation to $K = 2$. Workflow optimization is performed on only three datasets, after which the optimized workflow is directly evaluated on all remaining unseen tasks.

The three datasets used for workflow optimization are: (i) *tabular-playground-series-dec-2021*, a tabular multi-class classification task evaluated using Accuracy, where synthetic tabular data (derived from the Forest Cover Type Prediction dataset) are used to predict the target column *Cover_Type*; (ii) *detecting-insults-in-social-commentary*, a binary

Table 2: Performance comparison across agents on `gpt-5-mini`. It is noted that all agent workflows are executed on these unseen datasets in a zero-shot setting, without using any of the benchmark datasets during workflow optimization. Colorful icons indicate data modality (Tx: text, Tb: tabular, TS: time series, ST: spatiotemporal, Im: image, Mi: microscopy image) and problem type (BC: binary classification, MC: multiclass classification, RG: regression, FC: forecasting). ↑(↓) indicates higher(lower) is better. **Bold** numbers indicate the best performance among agents for each dataset. × denotes missing results (not supported / not applicable / not successful submission). **Avg. Rank**↓ is the average rank of each agent across all datasets when ranking agents by their performance on each dataset (smaller is better). **Success Rate**↑ is the percentage of datasets for which an agent produced a valid result (larger is better).

Dataset	AutoKaggle	Data Interpreter	DSAgent	DeepAnalyze	AIDE	AutoMind	AFlow	DSFlow
bike-sharing-demand Tb TS FC↓	0.62456	×	**0.28730**	0.39388	0.29845	0.30464	0.31068	0.30178
handwriting TS MC↑	0.09882	×	×	**0.29294**	×	0.24118	0.21882	0.21529
histopathologic-cancer-detection Im BC↑	×	0.50000	×	×	0.77846	0.77843	0.74525	**0.88980**
ili TS FC↓	5.09391	5.95074	×	5.03484	**2.16372**	4.49336	4.00419	3.04394
liverpool-ion-switching TS MC↑	0.03612	×	0.47311	0.53738	0.54403	**0.92039**	0.88253	0.47193
ventilator-pressure-prediction TS RG↓	11.22098	×	×	**1.92137**	×	×	2.78218	9.50939
demand-forecasting-kernels-only TS FC↓	11.97930	11.97930	43.61269	×	**10.95133**	10.95235	99.11387	**10.95133**
instant-gratification Tb BC↑	×	×	×	×	×	×	×	×
see-click-predict-fix ST RG↓	0.83782	1.12259	0.39020	0.62293	**0.36763**	0.37859	0.48133	0.52237
playground-series-s3e1 Tb RG↓	0.61518	×	**0.59081**	0.59301	0.59401	0.61049	0.61952	0.60115
playground-series-s3e11 Tb RG↓	0.32096	×	0.29864	×	**0.29764**	0.29903	0.30301	0.29990
sciencebench-005-dkpes-model-development-1 Tb RG↓	1.00000	×	1.00000	1.00000	1.00000	×	0.93750	**0.92188**
sciencebench-011-bbbc002-cell-count Mi RG↑	**78.00000**	63.34634	63.31750	×	16.51945	77.69474	13.89729	54.62807
Avg. Rank↓	5.27	5.50	3.31	3.69	**2.50**	2.90	4.25	3.29
Success Rate↑	85%	38%	62%	62%	77%	77%	**92%**	**92%**

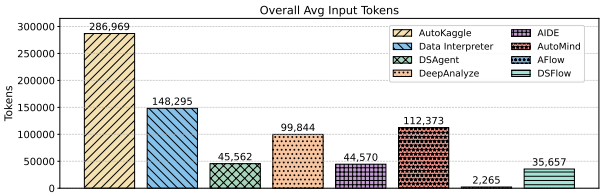

Figure 7: Overall average input token consumption across different LLM-based data science agents.

text classification task evaluated with AUC-ROC, where the goal is to predict the probability of a comment being insulting based on its textual content; and (iii) *plant-pathology-2020-fgvc7*, a multi-label image classification task evaluated using column-wise ROC AUC, which requires predicting class probabilities (*combinations*, *healthy*, *rust*, *scab*) for apple leaf images.

Importantly, workflows for other data modalities, including time-series and spatiotemporal data, microscopy images, and 3D data (e.g., 3D shape- or pharmacophore-derived features), are never included during the workflow optimization phase. During the testing stage, the optimized workflow is directly applied to these remaining datasets without any further modification or tuning. As a result, performance on these tasks purely reflects cross-task and cross-modality transfer, rather than any form of modality-specific workflow adaptation.

We also guard against uncontrolled operator-library growth as workflows evolve. At the generation stage, the meta-optimization prompt explicitly asks the model to avoid operators with overlapping functionality or duplicate semantics, reducing redundant synthesis before an operator is proposed. At the storage stage, the library enforces deterministic deduplication: each operator is registered as a unique `(name, code_hash)` pair, where `code_hash` is the SHA-256 hash of the normalized implementation body. When a candidate operator is generated, DSFlow first canonicalizes its code by stripping superficial formatting differences and then computes the hash. The operator is inserted only if no existing entry has the same name and hash; otherwise, it is treated as an existing operator and reused rather than added again. This concrete identifier-and-hash check prevents identical implementations from being duplicated while still allowing meaningful revisions when either the declared functionality or implementation genuinely changes.

C.4 Token Consumption Analysis

In this section, we analyze the token consumption of different LLM-based data science agents, as shown in Figures 7 and 8. Methods that rely on long prompt contexts or repeated code-level interactions incur substantially higher input token costs. In particular, AutoKaggle and Data Interpreter consume significantly more input tokens due to dataset-specific instructions and repeated analytical context injection. In contrast, `DSFlow` maintains consistently low input token usage by operating over compact, reusable workflow representations rather than repeatedly encoding dataset details into prompts. Figure 8 shows a similar

Table 3: Dataset specifications.

Dataset	Modality	Task Type	Metric	Data Summary	Target
bike-sharing-demand	Tabular + time series	Forecasting (regression)	RMSLE	Hourly bike demand with weather/time features (e.g., `datetime`, `season`, `holiday`, `workingday`, `weather`, `temp`, `humidity`, `windspeed`).	`count` (hourly total rentals)
handwriting	Time series	Multiclass classification	Accuracy	Handwriting trajectory sequences (length 152, 3D features), 26 classes.	Character class (0–25)
histopathologic-cancer-detection	Image	Binary classification	AUC-ROC	PCam pathology image patches.	Tumor presence in the center region (0/1)
ili	Time series	Forecasting	MSE/MAE	CDC ILI multivariate sequences (7 features).	Next 24 steps for all 7 features
liverpool-ion-switching	Time series	Multiclass classification	Macro-F1	Electrical signal `signal` over `time`.	`open_channels` (discrete channel count)
ventilator-pressure-prediction	Time series	Regression	MAE	Ventilator control inputs and lung attributes (`R`, `C`, `u_in`, `u_out`, `time_step`).	Airway pressure `pressure`
demand-forecasting-kernels-only	Time series	Forecasting	SMAPE	Historical sales by date/store/item (`date`, `store`, `item`, `sales`).	Future `sales`
instant-gratification	Tabular	Binary classification	AUC-ROC	Highly anonymized numeric features (hundreds of columns).	`target` (0/1)
see-click-predict-fix	Spatiotemporal	Regression (multi-target)	RMSLE	Civic issue reports with location/time/text/source fields.	`num_views`, `num_votes`, `num_comments`
playground-series-s3e1	Tabular	Regression	RMSE	Housing features (`MedInc`, `HouseAge`, `AveRooms`, `Latitude`, `Longitude`, etc.).	`MedHouseVal`
playground-series-s3e11	Tabular	Regression	RMSLE	Store/product attributes and sales features (`store_sales`, `unit_sales`, `units_per_case`, `store_sqft`, etc.).	`cost`
dkpes-model-development-1	Tabular + 3D shape/pharmacoph	Regression	RMSE	Pharmacophore/3D shape features (e.g., `TanimotoCombo`, `ShapeTanimoto`, `ColorScore`) and chemical group indicators (`3-Keto`, `12-Hydroxy`, etc.).	`Signal-inhibition`
bbbc002-cell-count	Microscopy image	Regression (counting)	Accuracy	BBBC002 cell microscopy images with human count references.	Cell count per image

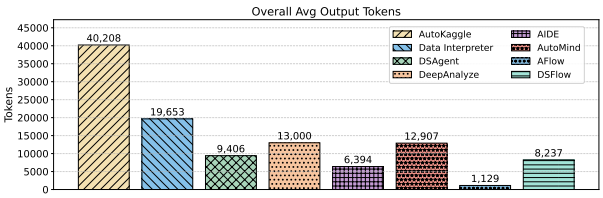

Figure 8: Overall average output token consumption across different LLM-based data science agents.

trend for output tokens. Code-centric, trial-and-error–driven agents (e.g., AutoMind and DeepAnalyze) generate longer outputs because of iterative code generation, debugging, and verbose intermediate reasoning. By contrast, **DSFlow** produces concise outputs by executing pre-validated workflows and operators, avoiding unnecessary regeneration of large code blocks or textual explanations. Overall, **DSFlow** achieves one of the lowest token footprints among all compared methods, demonstrating that workflow-level abstraction not only improves performance and transferability but also substantially reduces token consumption, making it more cost-effective and scalable under resource constraints.

C.5 Experiments on Other Base Model

We further conduct experiments using `gpt-5-mini` as the base model to evaluate the effectiveness, robustness, and generalization ability of different agent-based systems. As shown in Table 2, **DSFlow** consistently demonstrates strong and competitive performance compared to existing agents. In terms of overall ranking, DSFlow achieves an average rank of 3.29, outperforming several widely used baselines such as AutoKaggle, Data Interpreter, DeepAnalyze, and AFlow, while remaining highly competitive with the best-performing methods across a wide range of tasks. Notably, **DSFlow** attains a 92% success rate when instantiated with `gpt-5-mini` , the highest

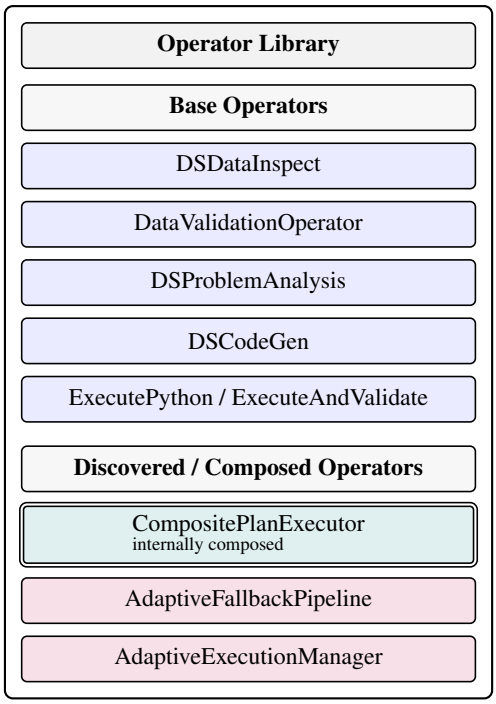

Figure 9: **Operator library.** Base operators provide atomic capabilities, while DSFlow discovers higher-level operators and composite executors that encapsulate reusable patterns.

among all compared agents, indicating its strong generalizability in producing valid solutions across heterogeneous datasets

C.6 Optimized Workflow

The workflows presented in this section are not manually engineered. Instead, they are automatically discovered and optimized by DSFlow through workflow-level search and operator composition. Starting from a minimal seed workflow (Figure 10), DSFlow iteratively restructures control flow, introduces validation-aware routing, and synthesizes higher-level composite operators, resulting in more robust and transferable execution patterns, as illustrated in Figure 11.

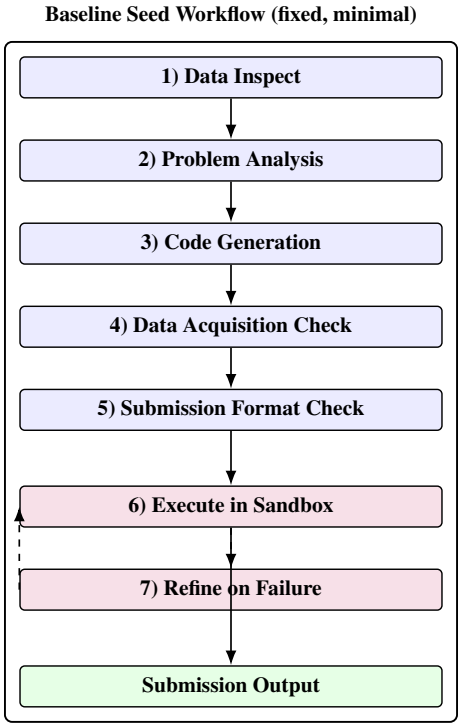

Figure 10: **Baseline seed workflow.** A fixed plan–code–execute pipeline with deterministic validation and a refine–retry loop on execution failure.

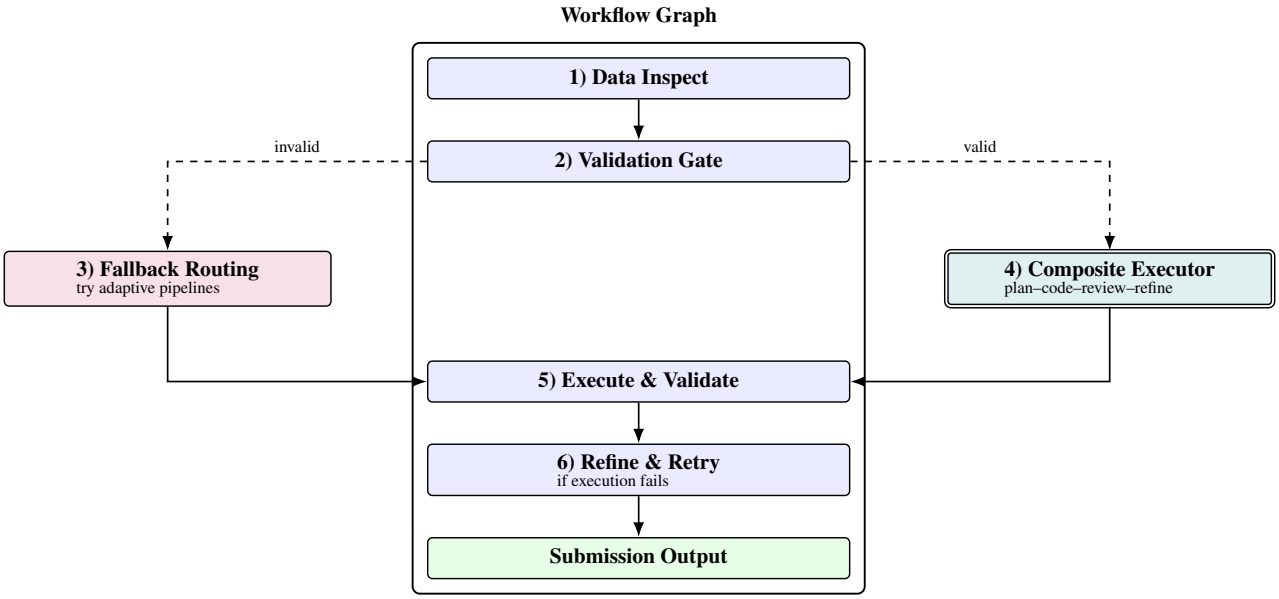

Figure 11: **Workflow execution graph.** DSFlow executes workflows with a validation gate and adaptive routing: invalid data triggers fallback pipelines, while valid data uses a composite executor for plan–code–review–refine. Both paths merge into execute–validate and optional refine–retry before producing the submission.