DS-Lighting: Making Agent Harnesses Explicit for Data-Science Automation

Anonymous Author(s)

Abstract

Large Language Model (LLM) agents have shown promise for automating data-science workflows, yet their end-to-end performance depends critically on the agent harness that represents tasks, manages execution state, constrains output artifacts, and provides evaluation feedback. Existing data-science agents often leave this harness implicit, making results difficult to reproduce, compare, and attribute across heterogeneous tasks. We introduce `DS-Lighting`, a unified harness toolkit that makes harness design explicit for data-science automation. `DS-Lighting` decomposes the harness into four reusable layers: data, workflow, execution, and evaluation, and represents diverse agents as executable operator programs that support both predefined pipelines and adaptive search. We further integrate multiple open-source data-science benchmarks into an MLE-Bench-style task format, enabling controlled comparison under a shared task interface, sandboxed runtime, and metric protocol. Experiments across agents, harnesses, models, and ablations show that explicit harness design improves reproducibility, comparability, and reliability, while reducing avoidable system-level failures in end-to-end data-science workflows.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; • **Information systems** → *Data management systems.*

Keywords

Data science agents, agent harnesses, workflow automation, evaluation, large language models

ACM Reference Format:

Anonymous Author(s). 2018. DS-Lighting: Making Agent Harnesses Explicit for Data-Science Automation. In *Proceedings of (Conference acronym 'XX).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large Language Model (LLM) agents have recently enabled a new paradigm for automating data-science workflows [23]. Given a natural-language objective and associated datasets, they can generate code, interact with execution environments, and iteratively refine analytical solutions. However, their success is not determined by model capability alone [14]. End-to-end performance also depends critically on the *agent harness*: the system layer that manages data context, workflow execution, feedback, and output evaluation. This harness is especially important for data-science tasks, where heterogeneous inputs, long-horizon execution, and structured output requirements can cause failures even when the model produces plausible reasoning or executable code.

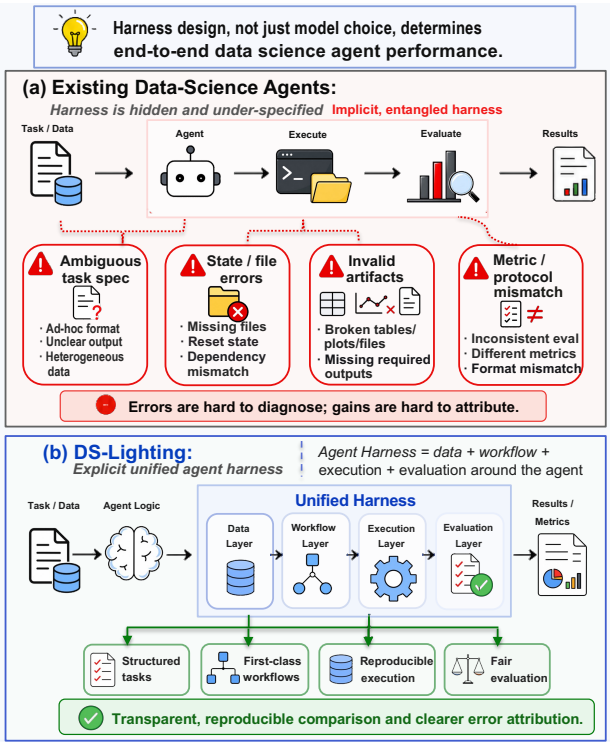

Figure 1: Existing data-science agents vs. `DS-Lighting` (ours). The contrast motivates a harness-centered design: existing agents entangle task data, execution, and evaluation, while `DS-Lighting` decouples them to ground data, govern execution, and align outputs with metrics.

Despite recent progress, existing data-science agents largely treat the harness as an implementation detail rather than a first-class research object. Prior systems propose diverse mechanisms for automating data-science workflows. For example, DS-Agent [4] decomposes tasks into a manually crafted pipeline covering knowledge retrieval, problem analysis, code generation, and execution, whereas AIDE [9] frames automation as iterative code optimization over candidate solutions. Other efforts explore planning [5], self-refinement [21], and multi-step tool use [28]. However, these systems implicitly encode different harness assumptions about task representation, execution state, intermediate validation, and output evaluation. As a result, it remains difficult to determine whether performance differences arise from stronger models, better workflows, or system-specific harness choices.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

Building a data-science harness is a non-trivial systems problem. Unlike answer-only or single-step tool-use tasks, data-science agents must execute long-horizon workflows over real datasets and produce artifacts that can be validated by task-specific metrics. This creates two coupled challenges. First, the harness must unify diverse agent workflows, from fixed pipelines to adaptive search and multi-agent collaboration, so that heterogeneous agents can be instantiated, orchestrated, and compared through a common interface. Second, it must manage reliable end-to-end execution by grounding heterogeneous inputs, maintaining consistent code and file states, and enforcing task-specific output constraints. Without these mechanisms, agents may follow plausible reasoning paths but execute incompatible workflows, solve the wrong problem, break runtime state, or generate artifacts that cannot be evaluated.

To address these challenges, we introduce `DS-Lighting`, a unified harness toolkit that makes harness design explicit for LLM-based data-science automation. `DS-Lighting` decomposes data-science harnessing into four reusable layers: a data layer that turns raw tasks and assets into structured task contracts, a workflow layer that represents agents as executable operator programs, an execution layer that provides sandboxed stateful runtime control, and an evaluation layer that validates artifacts and returns standardized feedback. This design supports both predefined pipelines and adaptive search-based workflows under a common interface. To enable controlled evaluation, `DS-Lighting` further integrates multiple open-source data-science benchmarks into an MLE-Bench-style task format, allowing agents and harnesses to be compared under the same task interface, execution environment, and metric protocol.

Our contributions are threefold. First, we formulate the agent harness as a first-class object in LLM-based data-science automation, showing that task representation, execution state, artifact constraints, and evaluation feedback are central to reliable and comparable end-to-end performance. Second, we introduce `DS-Lighting`, a unified harness toolkit that decomposes data-science automation into four reusable layers: data, workflow, execution, and evaluation. Through an operator-program abstraction, `DS-Lighting` represents diverse agents as executable workflows, supporting both predefined pipelines and adaptive search under a shared execution interface. Third, we build a unified evaluation suite by converting multiple open-source data-science benchmarks into an MLE-Bench-style task format, allowing agents and harness frameworks to be compared with the same task interface, sandboxed runtime, and metric protocol. Experiments across agents, harnesses, ablations, models, and failure modes demonstrate that explicit harness design improves reproducibility, comparability, and end-to-end reliability.

2 Related Work

Data Science Benchmarks. Recent benchmarks evaluate LLM agents on data-science tasks ranging from exploratory analysis to end-to-end modeling. Data-analysis benchmarks test table understanding, cleaning, transformation, visualization, and iterative analysis [6]; for example, DA-Code [8] emphasizes fine-grained code-generation operations, while DSEval [26] targets interactive refinement. Data-modeling benchmarks instead require complete machine-learning pipelines, from dataset inspection and feature engineering to training and evaluation [1, 7].

LLM Data Science Agents. LLM-based agents have attracted increasing attention for automating end-to-end data-science workflows [23]. Existing systems generally follow two paradigms: fixed workflows and adaptive search. Fixed-workflow agents decompose tasks into predefined stages such as problem analysis, code generation, execution, and validation [17]; examples include DS-Agent [4], which uses a procedural pipeline, and Data Interpreter [5], which organizes analysis through hierarchical workflows with incremental validation. Adaptive agents, including AIDE [9] and AutoKaggle [11], iteratively propose, execute, evaluate, and refine candidate solutions [15, 24].

Agent Harnesses. Beyond data science, recent work has highlighted evaluation harnesses, executable environments, and runtime infrastructure for LLM agents. Benchmarks such as WebArena [27], AppWorld [16], and OSWorld [20] move beyond static question answering by requiring tool use, stateful interaction, and programmatic validation. Frameworks such as HAL [10] standardize orchestration, logging, cost tracking, trajectory inspection, and reproducible comparison, while AutoHarness [14] and SafeHarness [12] study harness constraints for reliability and safety. However, these efforts are not designed for data-science workflows, where data context, code execution, output artifacts, and metric-based evaluation must be managed jointly. `DS-Lighting` fills this gap with a data-science-specific harness that separates data, workflow, execution, and evaluation concerns while supporting existing agents and flexible operator-program workflows.

3 Preliminaries

Data Science Task. A data-science task is represented as,

$$x = (q, D, A), \tag{1}$$

where q is the natural-language objective (e.g., "predict house prices from historical sales records"), D specifies task data assets (e.g., tables, images, time series, graphs, or multimodal inputs), and A provides auxiliary requirements such as documentation, external knowledge, submission constraints, or evaluation metrics. The task output y is the required artifact, such as an analytical report, prediction file, or model.

Agent Harness. A data-science agent is governed by an *agent harness* $\mathcal{H}$, the system layer that manages how the model uses task context, executes workflows, and incorporates feedback. In `DS-Lighting`, the harness is organized around four responsibilities: constructing data context, instantiating workflows, executing actions, and evaluating artifacts. Given a task x, the agent produces an output according to,

$$y \sim \pi_\theta(\cdot \mid x, \mathcal{H}), \tag{2}$$

where π_θ denotes the underlying LLM.

Agent Workflow. Within the harness, an agent is represented as a workflow W that specifies how data-science capabilities are organized and executed across multiple steps. A workflow is,

$$W = (O, C_W, \lambda), \tag{3}$$

where O denotes the operator library of reusable atomic capabilities (e.g., data inspection, code generation, model training, validation,

and result submission), C_W denotes the workflow controller that composes and invokes operators during execution (e.g., fixed ordering, conditional branching, iterative refinement, or search-based selection), and λ denotes workflow configurations (e.g., prompts, model settings, temperature, maximum iterations, time budget, and validation thresholds).

At step t, the selected operator $\mathrm{Op}_t \in \mathcal{O}$ invokes an atomic capability, possibly calling the LLM, interacting with the execution environment, and returning an observation. A simple workflow may follow a fixed sequence (e.g., InspectData, GenerateCode, TrainModel, Validate, and Submit), whereas an adaptive workflow selects operators from intermediate observations, such as retrying code generation after an execution error or switching models after weak validation scores.

Execution Loop. The workflow and the harness interact through an iterative loop. At step t, the state s_t summarizes the current execution context (e.g., task requirements, available data, code context, generated files, and previous feedback). The controller selects an operator $\mathrm{Op}_t = C_W(s_t)$, and the selected operator conditions the model to generate an action,

$$a_t \sim \pi_\theta(\cdot \mid s_t, \mathrm{Op}_t). \tag{4}$$

The execution layer runs the action and returns an observation $o_t = \mathrm{Exec}(a_t, s_t)$ (e.g., outputs, errors, artifacts, validation signals, or metric feedback). The harness then updates the state,

$$s_{t+1} = \mathrm{Update}_{\mathcal{H}}(s_t, a_t, o_t). \tag{5}$$

Executing W yields a trajectory $\tau_W = \{(s_t, \mathrm{Op}_t, a_t, o_t)\}_{t=1}^{T}$, from which the final output y is produced and evaluated.

4 DS-Lighting: A Layered Agent Harness

`DS-Lighting` realizes the agent harness $\mathcal{H}$ as four coordinated layers that transform raw task inputs into executable workflows and evaluated outputs. The *Data Layer* 4.1 builds structured task contracts from task descriptions and data files; the *Workflow Layer* 4.2 turns each contract into an operator program; the *Execution Layer* 4.3 runs actions in a controlled environment and returns observations; and the *Evaluation Layer* 4.4 validates artifacts, step-level feedback, and task-level metrics. Figure 2 illustrates the framework, with compact code-style examples available in the supplementary material for loading tasks, running predefined or custom agents, and launching benchmark suites.

4.1 Data Layer

The data layer builds the task context used by downstream workflows. Raw data-science inputs often mix underspecified instructions with heterogeneous assets, making it hard for agents to infer the goal, locate relevant files, understand schemas, and satisfy output requirements. To reduce this ambiguity, the data layer maps the raw task description q_{raw} and data assets D_{raw} to a structured task contract $x = (q, D, A)$ through two components: *Data Analyzer* and *Task Handler*. The Data Analyzer summarizes the assets into a compact data report, and the Task Handler merges it with the task description to produce an agent-ready contract.

Data Analyzer. Data Analyzer provides a data perception stage before workflow execution. Given the raw data assets D_{raw}, it produces an agent-readable report $R_D = \mathrm{DataAnalyzer}(D_{\mathrm{raw}})$ by extracting two types of information: a compact data profile $\phi(D_{\mathrm{raw}})$ and a size-aware sample $S_D \subset D_{\mathrm{raw}}$. The profile summarizes data organization (e.g., directory layout, file formats, schemas, and column-level statistics) and output-format requirements (e.g., column ordering, schema constraints, required file names, and submission fields). The sample provides representative records or file snippets, allowing the agent to inspect actual data content without loading the full dataset.

Task Handler. Task Handler converts the natural-language task description and the data report into a standardized task contract, written as $x = \mathrm{TaskHandler}(q_{\mathrm{raw}}, R_D)$. The resulting contract instantiates $x = (q, D, A)$ by formalizing the task objective q, constructing the standardized data specification D, and collecting auxiliary requirements A (e.g., input/output paths, submission schema, evaluation constraints, and expected artifacts). This task contract provides a consistent interface for workflow construction in the next layer. In practice, the data layer converts diverse open-source benchmarks (e.g., DABench, DACode, etc) into a unified MLE-Bench-style task format. Each task includes a description, public inputs, hidden references, sample submissions, and output constraints, enabling consistent execution, artifact validation, and evaluation across heterogeneous benchmarks. Further task-grounding details are provided in the supplementary material.

4.2 Workflow Layer

The workflow layer represents each agent as an executable operator program conditioned on the task contract. Rather than hard-coding monolithic pipelines, `DS-Lighting` separates reusable capabilities from control logic: operators define atomic actions, while a controller schedules them during execution. Fixed workflows use stable operator sequences, whereas search-based workflows select operators dynamically from the current state for exploration, refinement, and recovery. This shared abstraction lets diverse agents be built from extensible components and compared under the same execution and evaluation interface.

In a *manual workflow*, the controller follows a predefined operator sequence, selecting operators by workflow step rather than by the current state,

$$C_W(t) = \mathrm{Op}_t, \quad \mathrm{Op}_t \in (\mathrm{Op}_1, \ldots, \mathrm{Op}_K). \tag{6}$$

This captures existing agents or user-defined pipelines with fixed procedures (e.g., problem analysis, code generation, execution, and validation) without modifying their internal logic.

In a *search-based workflow*, the controller selects operators according to the current state:

$$\mathrm{Op}_t = C_W(s_t), \quad \mathrm{Op}_t \in \mathcal{O}. \tag{7}$$

Here, C_W may be implemented as a search algorithm or an LLM-based policy, enabling adaptive exploration, refinement, and recovery based on intermediate observations.

At each step, the selected operator Op_t conditions the model to generate an action $a_t \sim \pi_\theta(\cdot \mid s_t, \mathrm{Op}_t)$, providing the workflow with its next operation. The workflow layer is the development interface for building data-science agents: existing agents can use built-in

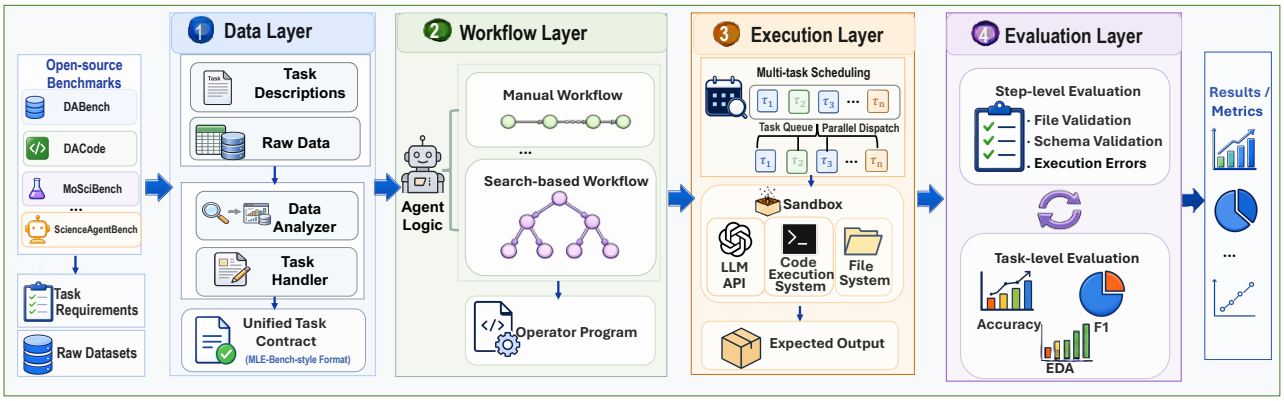

Figure 2: Overview of `DS-Lighting` as a layered agent harness. The data layer constructs task contracts, the workflow layer organizes agent capabilities into operator programs, the execution layer controls sandboxed interaction and artifact production, and the evaluation layer validates outputs and returns feedback.

templates, while new ones extend the operator library, compose workflow programs, or define custom controllers. By separating capabilities, workflow structure, and control policy, `DS-Lighting` unifies reusable templates and flexible operator-program designs, allowing agents to consume grounded task contracts, interact with the runtime, and receive evaluation-aligned feedback through one orchestration interface. Further workflow-layer details are provided in the supplementary material.

4.3 Execution Layer

The execution layer grounds workflow actions into concrete interactions with the environment. Given an action a_t produced by an operator, it executes the action in a sandboxed runtime and returns an observation: $o_t = \text{Exec}(a_t, s_t)$, where s_t provides the current execution context (e.g., available data files, code context, intermediate variables, generated artifacts, and runtime configuration). The observation o_t may include program outputs, execution errors, generated files, validation signals, or resource usage. The harness then updates the state $s_{t+1} = \text{Update}_{\mathcal{H}}(s_t, a_t, o_t)$, incorporating the effects of execution such as generated files, variables, logs, and feedback. Over the workflow, this produces a trajectory $\tau = \{(s_t, \text{Op}_t, a_t, o_t)\}_{t=1}^{T}$ from which evaluation and feedback can be derived.

This structure lets each operator run with full context while preserving intermediate results and feedback across steps. For concurrent workflows $\{x_1, x_2, \dots\}$, the execution layer uses a scheduler to coordinate actions under shared resource constraints: cross-task admission selects executable workflows, and within-task dispatch runs ready actions in parallel. Global limits on CPU/GPU allocation and LLM concurrency are enforced throughout, allowing agents to focus on generating actions and consuming observations. In practice, this execution governance tracks code context, dependencies, generated files, variables, logs, and resource usage, keeping long-horizon workflows state-consistent and reducing avoidable failures from missing files, stale states, or invalid runtime assumptions.

4.4 Evaluation Layer

The evaluation layer validates outputs and provides structured feedback. It operates at two levels: step-level evaluation, which checks intermediate artifacts during execution, and task-level evaluation, which computes the final score after completion.

Step-Level Evaluation Module. Step-level evaluation invokes operators during execution. Given state s_t and evaluation action a_t, the evaluator returns an observation,

$$o_t = \text{EvalStep}(a_t, s_t), \tag{8}$$

which may include file-structure checks, schema validation, execution diagnostics, missing-output detection, or semantic feedback. These observations are returned in structured, agent-readable form and incorporated into subsequent workflow decisions through the execution loop.

Task-Level Evaluation Module. Task-level evaluation scores a completed workflow. Given task x, trajectory τ_W, and output y, the evaluator applies

$$m = G(x, \tau_W, y), \tag{9}$$

where m is the task-level score. For data-analysis tasks, G may assess the completeness and correctness of reports, plots, or analytical conclusions. For modeling tasks, G typically computes prediction metrics such as accuracy, AUC, F1, or RMSE. Together, step-level and task-level evaluation implement evaluation alignment: evaluator-defined requirements are exposed not only as final scores, but also as structured feedback during execution. This allows agents to repair missing artifacts, schema mismatches, execution errors, or invalid submissions before final evaluation.

5 Experiments

We evaluate `DS-Lighting` through four harness-level research questions:

- **RQ1: Can `DS-Lighting` unify diverse agents through the workflow layer?**
- **RQ2: How does `DS-Lighting` compare with existing agent-harness frameworks?**

Table 1: Benchmark comparison of data-science agents reproduced with the DS-Lighting all-in-one toolkit. The table includes both manual-workflow agents and adaptive/search-workflow agents, demonstrating that DS-Lighting can instantiate heterogeneous agent workflows under the same task interface, execution environment, and evaluator. All metrics are accuracy-style scores, where higher values are better (↑); Overall Avg reports the mean performance across tasks.

Task	AutoKaggle	Data Interpreter	DSAgent	DeepAnalyze	AIDE	AutoMind	ReAct
Workflow	Manual	Manual	Manual	Manual	Adaptive	Adaptive	Adaptive
DeepSeek-V3.1-Terminus							
Comprehensive Preprocessing↑	0.60470	0.71110	0.75560	0.54550	0.62220	0.70450	0.62222
Correlation Analysis↑	0.88410	0.91550	0.88890	0.64290	0.80560	0.86110	0.93056
Distribution Analysis↑	0.85480	0.89060	0.87500	0.70970	0.81250	0.76560	0.89062
Feature Engineering↑	0.77550	0.86000	0.86000	0.69390	0.80000	0.78000	0.86000
Machine Learning↑	0.78950	0.89470	0.89470	0.61110	0.94737	0.78950	0.89474
Outlier Detection↑	0.73530	0.80000	0.77140	0.51430	0.68570	0.65710	0.71429
Summary Statistics↑	0.85060	0.88640	0.91110	0.67820	0.82220	0.78890	0.85556
Overall Avg	0.78493	0.85119	0.85096	0.62794	0.78508	0.76381	0.82400
gpt-5-mini							
Comprehensive Preprocessing↑	0.71111	0.77778	0.82222	0.64444	0.75556	0.75556	0.68889
Correlation Analysis↑	0.86111	0.87500	0.88889	0.75000	0.88889	0.93056	0.84722
Distribution Analysis↑	0.89062	0.85938	0.85938	0.73438	0.90625	0.89062	0.85938
Feature Engineering↑	0.78000	0.84000	0.84000	0.68000	0.86000	0.86000	0.78000
Machine Learning↑	0.78947	0.78947	0.73684	0.73684	0.78947	0.84211	0.78947
Outlier Detection↑	0.77143	0.77143	0.77143	0.60000	0.77143	0.80000	0.71429
Summary Statistics↑	0.90000	0.93333	0.93333	0.62222	0.91111	0.91111	0.82222
Overall Avg	0.81482	0.83520	0.83601	0.68113	0.84039	0.85571	0.78592

- **RQ3: Are the key components of DS-Lighting useful for reliable evaluation and execution?**
- **RQ4: What further insights can be drawn from additional analyses?**

5.1 Experimental Setup

Data-Science Tasks. We evaluate **DS-Lighting** with a unified, all-in-one toolkit that integrates representative benchmarks, including DABench [6], DACode [8], MosciBench [13], and ScienceAgentBench [3]. This setup allows different agents and harness configurations to be evaluated through the same task interface, with each benchmark runnable by a single line of code for reproducible large-scale evaluation. The suite spans closed-form data analysis, executable code generation, multimodal scientific discovery, and scientific program generation; detailed task statistics are provided in the supplementary material.

LLM Agents and Harnesses. *LLM Agents.* **DS-Lighting** natively integrates and reproduces seven state-of-the-art data-science agents in an all-in-one toolkit, covering both manual-workflow agents (e.g., AutoKaggle [11], Data Interpreter [5], DS-Agent [4], DeepAnalyze [25]) that follow fixed operator sequences and adaptive search-workflow agents (e.g., AIDE [9], AutoMind [15], ReAct [22]) that iteratively draft, debug, and refine code. All agents are instantiated through the same data, workflow, execution, and evaluation layers and run in a unified sandbox environment with controlled dependencies, ensuring reproducible and directly comparable evaluations. *Agent Harness Frameworks.* To the best of our knowledge, there is currently no dedicated agent-harness framework tailored to data-science agents, whose workflows require heterogeneous data access, code execution, artifact generation, and benchmark-specific validation. We therefore adapt representative general-purpose agent harnesses to the data-science setting as baselines, including `VanillaHarness` [22], `LangChain` [2], `AutoGen` [19], and `OpenHands` [18]. All harnesses use the same task interface, sandboxed execution environment, and official evaluation metrics, making the comparison focus on harness design rather than differences in task setup.

Evaluation Metrics. We evaluate task performance using the official metric specified by each benchmark. Concretely, these metrics fall into two categories: for benchmark-level tasks, we report an overall accuracy that summarizes task success under the unified evaluation protocol; for Kaggle-style competition tasks, where each task defines its own evaluation objective, we report the corresponding official score (e.g., classification accuracy, F1, RMSE, etc.). To further characterize system efficiency, we additionally report token consumption and wall-clock time. A complete list of metrics is provided in the supplementary material.

Implementation Details. For the main agent-comparison experiments, we instantiate each agent following its official implementation and recommended configuration whenever available. Unless otherwise specified, all agents use `DeepSeek-V3.1-Terminus` as the shared base model, which controls for model choice and makes performance differences more attributable to agent workflows and harness design. We additionally conduct a base-model study by fixing the agent framework and varying the underlying LLM, including `gpt-5.5`, `claude-opus-4.7`, `gpt-5.4-mini`, `gemini-3.1-pro`, and `qwen3.5-plus`. For further analyses where the agent framework is not the variable of interest, we use the general-purpose ReAct framework as the default agent framework.

5.2 Unified Evaluation of Data-Science Agents

To answer RQ1, Table 1 provides a unified benchmark comparison of seven data-science agents that are uniformly reproduced by **DS-Lighting**'s all-in-one toolkit, covering both manual-workflow agents and adaptive/search-workflow agents across seven DABench

Table 2: Comparison with existing agent-harness frameworks. Higher values are better.

Framework	DABench	DACode	MosciBench	ScienceAgentBench
VanillaHarness	**0.8521**	0.3073	0.4659	0.1863
LangChain	0.8210	0.3013	0.5000	0.1373
AutoGen	0.7704	0.1974	0.1932	0.0980
OpenHands	0.7626	0.2361	0.3182	0.0882
DS-Lighting (ours)	**0.8521**	**0.3709**	**0.6477**	**0.1961**

task types and two backbone LLMs. We summarize the main findings as observations (abbreviated as Obs.). **Obs.❶ Workflow style matters across task types.** Fixed-workflow agents are particularly reliable on standardized analyses: under `DeepSeek-V3.1-Terminus`, Data Interpreter reaches 0.91550 on Correlation Analysis and 0.88640 on Summary Statistics, while DSAgent reaches 0.88890 and 0.91110 on the same tasks. These results suggest that predefined workflows remain effective when tasks can be decomposed into stable data-analysis steps. **Obs.❷ Robustness and peak performance are distinct agent properties.** Manual-workflow agents like Data Interpreter and DSAgent achieve strong averages, showing stable performance on standardized tasks. In contrast, search-based agents like AIDE, AutoMind, and ReAct often top selected task types, but less uniformly. Thus, unified evaluation should report both aggregate performance and task-level behavior. Additional data-modeling and efficiency results appear in Section 5.5.

5.3 Comparison with Agent-Harness Frameworks

To answer RQ2, we compare **DS-Lighting** with representative agent-harness frameworks under the same task, model, and execution settings. Each framework receives the same public task contract, input files, backbone model, step budget, execution feedback, and final official evaluator. **DS-Lighting** never uses hidden labels, private answers, or intermediate metric scores; its extra checks only enforce public output requirements, including artifact existence, filenames, directory structure, and schema consistency. This setup isolates harness-level execution and validation from differences in task information or evaluator access. Table 2 shows that **DS-Lighting** is best or tied-best across all four benchmarks, highlighting the value of explicit task context, execution-state management, artifact constraints, and evaluation feedback. **Obs.❶ General-purpose harnesses are competitive on simple settings but transfer less reliably.** On DABench, **DS-Lighting** matches `VanillaHarness` (0.8521), indicating that a lightweight harness can be sufficient when tasks are relatively standardized. However, the gap becomes clear on more heterogeneous benchmarks: **DS-Lighting** improves over the strongest baseline on DACode (0.3709 vs. 0.3073), MosciBench (0.6477 vs. 0.5000), and ScienceAgentBench (0.1961 vs. 0.1863). This pattern suggests that data-science harness design becomes more important as tasks require richer data access, code execution, and artifact validation. **Obs.❷ Harness specialization improves robustness across benchmark types.** General-purpose frameworks show uneven transfer: `LangChain` is competitive on MosciBench but weaker on ScienceAgentBench, while `AutoGen` and `OpenHands` trail across benchmarks. Under the same task interface and metrics, **DS-Lighting**'s consistent gains point to the value of structured task contracts, controlled execution, and benchmark-aware evaluation.

5.4 Ablation Study of System-Level Mechanisms

To answer RQ3, we ablate the three mechanisms introduced in Section 4: Data Grounding (DG), Execution Governance (EG), and Evaluation Alignment (EA). Each variant removes one mechanism while keeping the benchmark inputs, base agent, execution environment, and final evaluator unchanged. Specifically, w/o DG removes the structured task contract and provides agents with raw task descriptions and files; w/o EG disables stateful sandbox governance and artifact tracking while retaining basic code execution; and w/o EA removes intermediate evaluator-aligned feedback while preserving the final official metric. These controlled variants isolate the effects of each mechanism in Figure 3. **(1) Data grounding preserves task fidelity.** Removing DG lowers macro-average accuracy from 0.5167 to 0.3721, with large drops on DACode (0.3709 to 0.2376), MosciBench (0.6477 to 0.3750), and ScienceAgentBench (0.1961 to 0.0588). This confirms that grounding is not merely extra prompt context: it makes files, schemas, fields, and output contracts legible to the agent. **(2) Execution governance is the main reliability bottleneck.** Removing EG causes the largest decline, reducing macro-average accuracy to 0.2714 and collapsing DABench from 0.8521 to 0.2724. Thus, task understanding must be paired with runtime observation, artifact validation, and recovery to keep code execution, file state, and submissions synchronized. **(3) Evaluation alignment enables task-relevant repair.** Removing EA lowers macro-average accuracy to 0.3971, especially on MosciBench (0.6477 to 0.3409) and ScienceAgentBench (0.1961 to 0.0588), showing that evaluator-aligned feedback turns grading failures into actionable repair signals.

5.5 Additional Experimental Results

Token Consumption across Agents. Figure 4 compares average total token consumption on DABench, normalized by counted task for a fairer comparison. **Obs.❶ Token efficiency differs sharply across agents.** React is the most token-efficient agent, using only 5.1K tokens per task on average. By contrast, AutoKaggle consumes 123.1K tokens per task, or 24.21× more than React. Even relative to the most efficient traditional agent, DeepAnalyze, React reduces token consumption by 51.2%. This gap reflects workflow design: AutoKaggle maintains larger contexts and repeats reasoning, code generation, and execution cycles, whereas React uses shorter action-observation loops with less context accumulation.

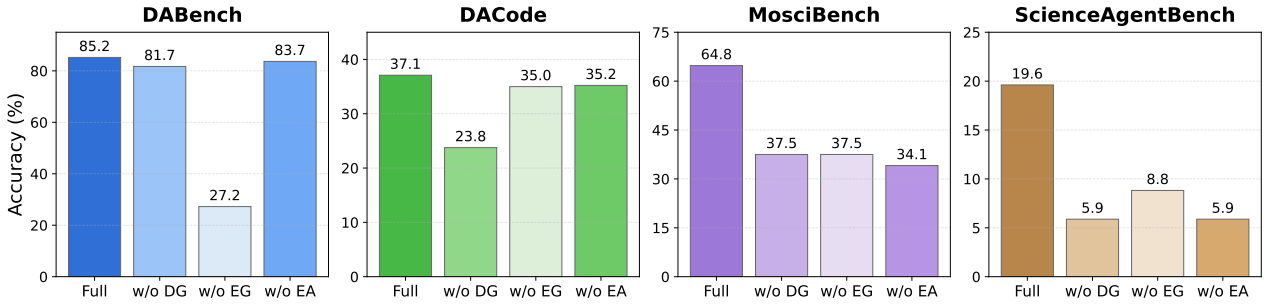

Figure 3: Ablation of DS-Lighting's system-level mechanisms.

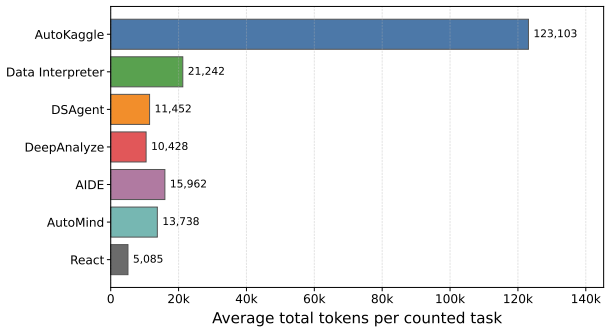

Figure 4: Average token consumption of different agents.

Obs.❷ Input context dominates token cost. Figure 5 decomposes total usage into input and output tokens. Across agents, input tokens account for 82.1%–92.5% of the total budget; React follows the same trend, with 86.0% input tokens and 14.0% output tokens. Thus, token cost is driven mainly by prompt-side context, including task descriptions, intermediate observations, execution histories, and tool feedback. Reducing redundant context and controlling history growth are therefore key to improving token efficiency in data-analysis agents.

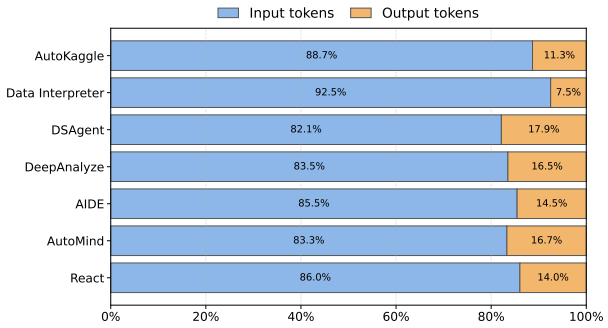

Figure 5: Input/output token composition of different agents.

Runtime Comparison across Agents. Figure 6 compares average runtime across agent frameworks reproduced under DS-Lighting on DABench, normalized by counted task for a fairer comparison. **Obs.❸ Runtime efficiency varies widely across frameworks.** React is the fastest, requiring only 0.69 minutes per task, while AutoKaggle is the slowest, requiring 22.62 minutes per task. This yields a 32.93× gap between the fastest and slowest frameworks. Among non-React agents, Data Interpreter and DeepAnalyze are the fastest (2.44 and 2.51 minutes per task), DSAgent is moderate (6.22 minutes), and AIDE and AutoMind are slower (9.99 and 9.66 minutes). AutoKaggle is a clear outlier, reaching 141.39 cumulative hours in total. **Obs.❹ Workflow complexity drives runtime cost.** Lightweight interpreter-style agents such as Data Interpreter and DeepAnalyze use shorter analysis and code-execution loops, leading to lower overhead. By contrast, search- or competition-oriented frameworks such as AutoKaggle, AIDE, and AutoMind introduce more planning, candidate generation, evaluation, and refinement, increasing both model-call latency and tool-execution cost. Overall, the reproduced agents show a clear trade-off: compact workflows are faster, while multi-stage search and refinement workflows incur much higher cumulative runtime.

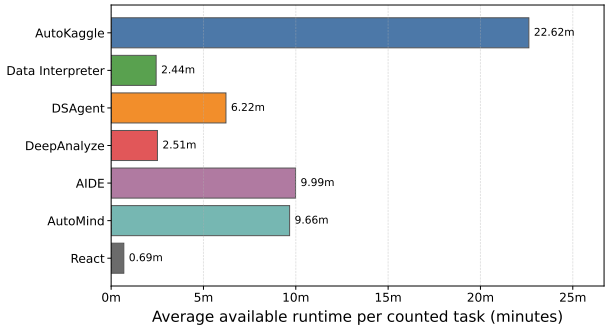

Figure 6: Average running time of different agents.

Runtime Comparison across Frameworks. Figure 7 compares LangChain ReAct and DS-Lighting ReAct across foundation models on DABench. **Obs.❺ The harness implementation strongly affects runtime.** Across five complete model runs, LangChain ReAct takes 642.7 minutes in total, whereas DS-Lighting ReAct takes 236.9 minutes, giving a 2.71× overall speedup. The speedup appears on four of five models, with the largest gain on GPT-5.5 (5.49×), followed by Qwen3.5 Plus (3.38×), Claude Opus 4.7 (3.27×), and Gemini 3.1 Pro (2.04×). **Obs.❻ Runtime must be interpreted with task completion.** Qwen3 Next 80B is the only case where LangChain ReAct appears faster, but this is mainly due to frequent execution errors and earlier termination rather than more efficient problem solving. Overall, DS-Lighting ReAct reduces framework-level overhead through a more compact execution loop, showing that runtime is shaped by both the backbone model and the surrounding harness.

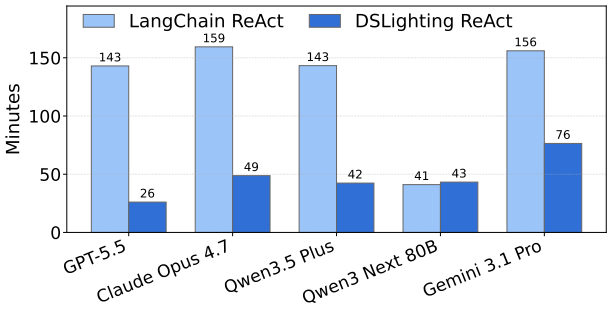

Figure 7: Runtime comparison between LangChain ReAct and `DS-Lighting` ReAct across models on DABench.

Kaggle-Style Competition Tasks. Figure 8 compares Data Interpreter and AIDE on the nine Kaggle-style datasets where both agents produce valid results. **Obs.❼ Performance is task-dependent across competition settings.** AIDE performs better on five datasets, while Data Interpreter performs better on four. AIDE is stronger on classification-oriented tasks such as *histopathologic-cancer-detection* and *instant-gratification*, whereas Data Interpreter remains competitive on several forecasting and regression tasks. This suggests that AIDE's iterative search-and-refinement workflow helps on some tasks, but the simpler code-interpreter workflow can be more stable on others; no single framework dominates across all competition-style settings.

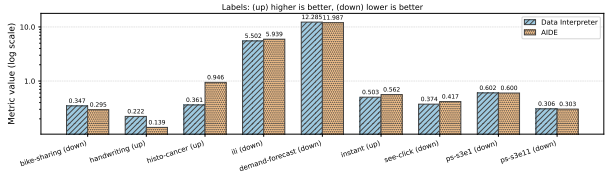

Figure 8: Comparison between Data Interpreter and AIDE on nine Kaggle-style competition datasets with valid results from both agents.

5.6 Further Analysis

To answer RQ4, we analyze model sensitivity and failure modes on DABench. **Obs.❶ Harness design can outweigh backbone scale.** Figure 9 shows that **`DS-Lighting`** paired with Qwen surpasses LangChain paired with GPT-5.5. This suggests that end-to-end performance is not determined by model strength alone: a well-structured harness can better expose task context, regulate execution, and convert model outputs into valid artifacts.

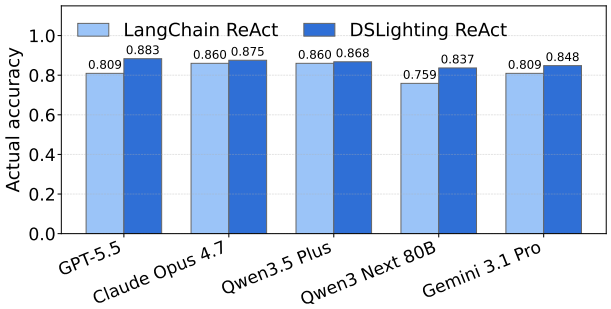

Figure 9: Performance across different models.

Obs.❷ The harness mainly removes avoidable system-level failures. To better understand where agents fail, we use an LLM-as-judge protocol to inspect failed trajectories and categorize each error by its dominant cause, such as task-grounding mistakes, execution-control failures, evaluation-alignment issues, or intrinsic model limitations. Figure 10 shows that **`DS-Lighting`** reduces non-capability failures, especially those tied to grounding, runtime control, and evaluator-aligned repair. As these avoidable failures decrease, the remaining errors concentrate more on intrinsic model limitations, suggesting that the harness improves reliability by removing system-level failure sources rather than merely masking reasoning errors.

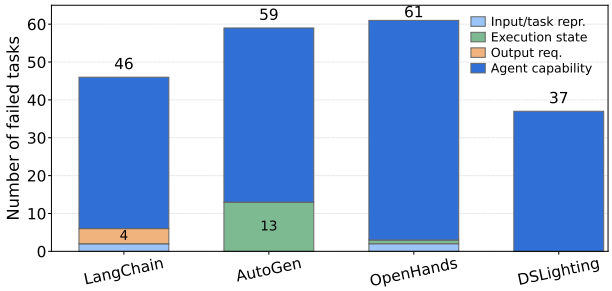

Figure 10: Failure-mode analysis.

6 Conclusion

We presented **`DS-Lighting`**, a unified harness toolkit for developing and evaluating data-science agents. By making task context, workflow execution, and evaluation explicit through reusable data, workflow, execution, and evaluation layers, **`DS-Lighting`** supports heterogeneous agents under a common interface, enables controlled and reproducible comparison, and reveals failures in long-horizon workflows. Experiments show that harness design is central to agent performance: **`DS-Lighting`** improves robustness across agents and benchmarks, reduces avoidable execution and evaluation failures, and helps smaller backbone models approach stronger ones.

References

[1] Jun Shern Chan et al. 2025. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *ICLR*. Association for Computing Machinery, New York, NY, USA, 1–1.
[2] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain.
[3] Ziru Chen et al. 2025. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *International Conference on Learning Representations*, Vol. 2025. Association for Computing Machinery, New York, NY, USA, 96934–96990.
[4] Siyuan Guo et al. 2024. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453* 1, 1 (2024), 1–1.
[5] Sirui Hong et al. 2024. Data Interpreter: An LLM Agent for Data Science. *arXiv preprint arXiv:2402.18679* 1, 1 (2024), 1–1.
[6] Xueyu Hu et al. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507* 1, 1 (2024), 1–1.
[7] Qian Huang et al. 2023. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302* 1, 1 (2023), 1–1.
[8] Yiming Huang et al. 2024. Da-code: Agent data science code generation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computing Machinery, New York, NY, USA, 13487–13521.
[9] Zhengyao Jiang et al. 2025. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138* 1, 1 (2025), 1–1.
[10] Sayash Kapoor et al. 2025. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation. *arXiv preprint arXiv:2510.11977* 1, 1 (2025), 1–1.

[11] Ziming Li et al. 2024. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424* 1, 1 (2024), 1–1.
[12] Xixun Lin et al. 2026. SafeHarness: Lifecycle-Integrated Security Architecture for LLM-based Agent Deployment. *arXiv preprint arXiv:2604.13630* 1, 1 (2026), 1–1.
[13] Fan Liu et al. 2026. Towards multimodal data-driven scientific discovery powered by LLM agents. In *The Fourteenth International Conference on Learning Representations*. Association for Computing Machinery, New York, NY, USA, 1–1.
[14] Xinghua Lou et al. 2026. AutoHarness: improving LLM agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329* 1, 1 (2026), 1–1.
[15] Yixin Ou et al. 2025. Automind: Adaptive knowledgeable agent for automated data science. *arXiv preprint arXiv:2506.10974* 1, 1 (2025), 1–1.
[16] Harsh Trivedi et al. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computing Machinery, New York, NY, USA, 16022–16076.
[17] He Wang et al. 2025. DSMentor: Enhancing Data Science Agents with Curriculum Learning and Online Knowledge Accumulation. *arXiv preprint arXiv:2505.14163* 1, 1 (2025), 1–1.
[18] Xingyao Wang et al. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, Vol. 2025. Association for Computing Machinery, New York, NY, USA, 65882–65919.
[19] Qingyun Wu et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First conference on language modeling*. Association for Computing Machinery, New York, NY, USA, 1–1.
[20] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems* 37 (2024), 52040–52094.
[21] Xu Yang et al. 2025. R&D-Agent: An LLM-Agent Framework Towards Autonomous Data Science. *arXiv preprint arXiv:2505.14738* 1, 1 (2025), 1–1.
[22] Shunyu Yao et al. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* 1, 1 (2022), 1–1.
[23] Ziming You et al. 2025. DatawiseAgent: A Notebook-Centric LLM Agent Framework for Adaptive and Robust Data Science Automation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computing Machinery, New York, NY, USA, 1099–1123.
[24] Jiayi Zhang et al. 2024. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762* 1, 1 (2024), 1–1.
[25] Shaolei Zhang et al. 2025. Deepanalyze: Agentic large language models for autonomous data science. *arXiv preprint arXiv:2510.16872* 1, 1 (2025), 1–1.
[26] Yuge Zhang et al. 2024. Benchmarking data science agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computing Machinery, New York, NY, USA, 5677–5700.
[27] Shuyan Zhou et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* 1, 1 (2023), 1–1.
[28] Xinyu Zhu et al. 2026. Toward Ultra-Long-Horizon Agentic Science: Cognitive Accumulation for Machine Learning Engineering. *arXiv preprint arXiv:2601.10402* 1, 1 (2026), 1–1.

A Usage Examples

DS-Lighting exposes a compact user-facing API for moving from a single task to full benchmark evaluation under the same task contract. At the task level, users can call `load_data` to construct the public data context and then invoke `run_agent` to execute a selected workflow. At the agent level, predefined workflows can be swapped by changing only the `workflow` argument, while custom workflows can be defined by composing asynchronous operators inside a `BaseWorkflow`. The same configuration can then be passed to `DSBenchmark` for suite-level evaluation. Figures 11–14 illustrate this progression from single-task execution, to predefined and custom agents, to benchmark runs.

```python
from dslighting import load_data, run_agent

task = load_data("bike-sharing-demand")
result = run_agent(data=task, workflow="aide", model="
    gpt-4o")

# Or launch the same task directly by id.
result = run_agent(task_id="bike-sharing-demand",
    workflow="aide", model="gpt-4o")
```

Figure 11: Running a DS-Lighting agent on a single task.

Figure 11 shows the shortest path for running one data-science task: the loader builds the task contract, and the agent runner handles workflow execution and artifact collection. This keeps the task interface identical whether the input is an already loaded task object or a benchmark task id.

```python
from dslighting import Agent, run_agent

agent = Agent(workflow="aide", model="gpt-4o")
result = agent.run(task_id="bike-sharing-demand")

# Switch workflows without changing the task interface
    .
result = run_agent(task_id="bike-sharing-demand",
    workflow="react", model="gpt-4o")
```

Figure 12: Using and switching predefined DS-Lighting agent workflows. Supported workflows include `aide`, `react`, `dsagent`, `automind`, `autokaggle`, `data_interpreter`, `deepanalyze`, and `aflow`.

Figure 12 illustrates that predefined agents share the same public interface. In practice, users can compare workflows such as `aide`, `react`, `dsagent`, or `autokaggle` by changing one argument while keeping the model, task id, and benchmark runner fixed.

```python
from dslighting.arch.operators import Operator
from dslighting.arch.workflows import BaseWorkflow
class MyOperator(Operator):
    async def __call__(self, description, data_dir):
        return {"answer": f"Solved task from {data_dir
    }"}
class MyAgent(BaseWorkflow):
    async def solve(self, description, io_instructions
    , data_dir, output_path):

        result = await self.operators["solve"](
    description, data_dir)
        output_path.write_text(result["answer"])
agent = MyAgent(operators={"solve": MyOperator()},
    services={}, agent_config={})
result = agent.run(data="data/bike-sharing-demand",
    task="Analyze the dataset")
```

Figure 13: Defining and running a minimal custom DS-Lighting agent.

Custom agents follow the same contract but expose the workflow internals. As shown in Figure 13, each workflow implements `solve(...)`, calls operators, and writes the required artifact to `output_path`, making custom logic compatible with the same evaluator.

```python
from dslighting.api import DSBenchmark
from dslighting.core import ConfigBuilder

config = ConfigBuilder().build_config(workflow="aide",
     model="gpt-4o")
result = DSBenchmark("dabench", data_dir="data/dabench
    ").run(config=config)

# benchmark_id can be "dabench", "dacode", "moscibench
    ", or "sciencebench".
result = DSBenchmark(benchmark_id, data_dir="data/
    benchmark-data").run(config=config)
```

Figure 14: Running benchmark suites with DS-Lighting. Internally, ScienceAgentBench is selected with the benchmark id `sciencebench`.

Finally, Figure 14 shows how the single-task setup scales to benchmark-level evaluation. The benchmark runner reuses the same workflow configuration across all tasks in a suite and applies the corresponding official evaluation protocol.

B Task Grounding and Benchmark Normalization

To evaluate heterogeneous data-science benchmarks through a common interface, **DS-Lighting** normalizes every task into an MLE-Bench-style competition directory. Each task contains a natural-language description, preparation and grading scripts, public agent-visible artifacts, and private references reserved for evaluation:

```
task-id/
  config.yaml
  description.md
  prepare.py
```

```
  grade.py
  raw/
  prepared/
    public/       # visible to the agent
    private/      # hidden references for grading
```

Listing 1: Normalized task layout.

The conversion assigns each asset a clear role. `raw/` stores original benchmark files; `prepare.py` materializes `prepared/public/` and `prepared/private/`; and `grade.py` implements the task-local evaluator. The accompanying metadata specifies the task type, description, preparer, grader, and expected submission contract:

```
competition_type: code
description: description.md
grader:
  name: compare_csv
  grade_fn: file:grade.py:grade
preparer: file:prepare.py:prepare
```

Listing 2: Example normalized task metadata.

At runtime, the Task Handler reads this metadata and constructs the task contract $x = (q, D, A)$. Here, q contains the objective and metric description; D exposes only `prepared/public/` plus Data Analyzer summaries such as file types, schemas, missing-value statistics, and sampled rows; and A defines the required output artifact, including filename, format, file/directory kind, sample submission, and grading mode. Agents are sandboxed with access only to public artifacts and must write outputs according to this contract. After execution, **DS-Lighting** validates the artifact path and format, then routes evaluation through the unified service: artifact-submission tasks call the local `grade.py`, while open-ended or judge-based tasks use the judge evaluator. This normalization prevents private-answer leakage, removes benchmark-specific I/O assumptions from agents, and enables all workflows to be compared through the same task contract and evaluation path.

C Workflow Layer and Operator Programs

DS-Lighting represents each agent as an executable *operator program* conditioned on the task contract $x = (q, D, A)$. The contract specifies the task objective, public data artifacts, output constraints, and evaluation interface, while the workflow defines how an agent acts on this contract over time. This separation keeps the task interface fixed and makes agents differ only in their control logic.

Formally, **DS-Lighting** maintains an operator library

$$\mathcal{O} = \{\mathrm{Op}_1, \ldots, \mathrm{Op}_K\}. \tag{10}$$

Each operator is an atomic executable capability, such as task analysis, planning, code generation, code execution, artifact validation, review, or memory summarization. At step t, the workflow state s_t stores the task contract, messages, generated code, execution outputs, validation results, artifacts, and workflow-specific memory. A controller C_W selects an operator and applies it to the current state:

$$\mathrm{Op}_t = C_W(s_t, x), \qquad \mathrm{Op}_t \in \mathcal{O}. \tag{11}$$

The selected operator conditions the model or tool executor to produce the next action, and the returned observation updates the workflow state:

$$a_t \sim \pi_\theta(\cdot \mid x, s_t, \mathrm{Op}_t), \tag{12}$$

$$s_{t+1} = U_W(s_t, \mathrm{Op}_t, a_t, y_t), \tag{13}$$

where y_t is the observation from the environment or evaluator, such as generated code, sandbox output, validation feedback, or a score estimate.

Manual workflows. A manual workflow follows a predefined operator schedule, selecting operators by step index rather than adaptive feedback:

$$\mathrm{Op}_t = C_W(t) = \mathrm{Op}_{\sigma_W(t)}, \qquad \mathrm{Op}_{\sigma_W(t)} \in \mathcal{O}. \tag{14}$$

Here, σ_W is the fixed sequence specified by the workflow, such as `analyze` → `plan` → `generate-code` → `execute` → `validate`. This captures existing agents and user-defined pipelines by wrapping each step as an operator with a common input/output interface.

Search-based workflows. A search-based workflow chooses operators from the current state. The controller may be a heuristic, tree-search procedure, or LLM-based policy, enabling the workflow to branch, refine attempts, recover from execution errors, revisit earlier decisions, or terminate once the artifact satisfies the output contract. For example, failed execution can trigger debugging, a valid but weak result can trigger refinement, and successful validation can trigger submission.

Execution interface. Manual and search-based workflows share the same harness interface: each operator receives the grounded task contract and returns structured outputs such as plans, code, logs, reviews, or artifact paths. Side effects are isolated in the sandbox workspace, and final artifacts are checked against the output contract before grading. Thus, **DS-Lighting** can orchestrate heterogeneous agents while comparing them under the same grounding, runtime, validation, and benchmark evaluation pipeline.

D Dataset Details

We evaluate agents on data science benchmarks covering general data analysis, code generation, multimodal scientific discovery, and scientific program generation. The full benchmark suite contains 947 tasks, including 257 from DABench, 500 from DACode, 88 from MoSciBench, and 102 from ScienceAgentBench.

DABench. DABench [6] is a data analysis benchmark designed to evaluate whether language agents can solve closed-form analytical questions over tabular datasets. The benchmark covers common data analysis skills, including summary statistics, correlation analysis, distribution analysis, feature engineering, data preprocessing, outlier detection, and basic machine learning. In our evaluation, DABench contains 257 normalized tasks. Each task requires the agent to inspect the provided data, perform the necessary analysis, and submit an answer in a structured CSV format. Tasks are evaluated automatically using exact-match or tolerance-based grading against ground-truth answers.

DACode. DACode is a data science code generation benchmark for evaluating agents on practical data manipulation, modeling, and visualization tasks. Unlike closed-form question answering benchmarks, DACode emphasizes executable code generation and artifact production. The benchmark includes 500 tasks spanning

data wrangling, data interpretation, text and CSV processing, plotting, database manipulation, and machine learning tasks such as classification, regression, clustering, and competition-style prediction. Outputs include CSV files, JSON files, database files, plots, and machine learning prediction files, which are evaluated using task-specific automatic graders.

MoSciBench. MoSciBench is a multimodal scientific discovery benchmark that evaluates agents on end-to-end scientific analysis tasks. It contains 88 tasks across six scientific domains: Earth science, biomedical engineering, cheminformatics, health psychology, population genomics, and Earth system science. The benchmark covers multiple data modalities, including time series, tabular data, images, mass spectra, molecular structures, genotype matrices, and text. Tasks are organized around scientific discovery question types such as descriptive analysis, correlational reasoning, causal inference, predictive modeling, and pattern discovery. Each task requires the agent to analyze raw scientific data and produce a structured answer that can be automatically evaluated.

ScienceAgentBench. ScienceAgentBench is a benchmark for assessing language agents on data-driven scientific discovery tasks derived from peer-reviewed research. It contains 102 tasks extracted from 44 papers across four scientific disciplines: bioinformatics, chemoinformatics, geoscience, and cognitive neuroscience. Each task is formulated as a self-contained scientific program generation problem, where the agent must write and execute code to produce the required output artifact. Outputs include figures, tables, arrays, text files, JSON files, and other scientific artifacts. Evaluation is performed using task-specific metrics such as image similarity, accuracy, RMSE, MAE, F1, ROC-AUC, exact match, and correlation-based scores.

MLE-bench. additional also test kaggle style competiiton from mle-bench [1]. Table 3 summarizes the datasets used for data modeling, which are curated to represent heterogeneous predictive tasks across multiple modalities. The benchmark includes tabular and time-series forecasting problems (e.g., *bike-sharing-demand*, *ili*, *demand-forecasting-kernels-only*), classification tasks over sequential signals (e.g., *handwriting*, *liverpool-ion-switching*), and an image-based medical classification task (i.e., *histopathologic-cancer-detection*). It also includes multimodal or weakly structured settings such as *see-click-predict-fix*, which combines spatiotemporal and textual signals for multi-target regression. Each dataset is evaluated using its official competition metric (e.g., RMSLE, RMSE, AUC-ROC, Macro-F1), enabling consistent comparisons of end-to-end modeling quality across agents.

Table 3: Kaggle-style competition dataset summary.

Dataset	Modality	Task	Metric
bike-sharing-demand	Tabular + time	Forecasting (regression)	RMSLE
handwriting	Time series	Multiclass classification	Accuracy
histopathologic-cancer-detection	Image	Binary classification	AUC-ROC
ili	Time series	Forecasting (multivariate)	MSE/MAE
demand-forecasting-kernels-only	Time series	Forecasting (regression)	SMAPE
instant-gratification	Tabular	Binary classification	AUC-ROC
see-click-predict-fix	Spatiotemporal + text	Multi-target regression	RMSLE
playground-series-s3e1	Tabular	Regression	RMSE
playground-series-s3e11	Tabular	Regression	RMSE

E Implementation Details

Unified reproduction of agent-harness frameworks. We reproduce VanillaHarness, LangChain, AutoGen, OpenHands, and `DS-Lighting` under the same data-science benchmark adapter to ensure a controlled and easy-to-follow comparison. For each task, the adapter builds a unified task contract containing the original task description, public input files, sample artifacts when available, required output filename or directory structure, and the official evaluation interface, while never exposing private answers, hidden labels, or official scores during solving. All frameworks receive this same contract in the prompt, run inside an isolated per-task workspace with the same public files, model, step budget, and execution feedback, and are graded only after termination using the same official benchmark metric. The general-purpose baselines are adapted to this contract through their native execution mechanisms: VanillaHarness uses a direct ReAct-style code-execution loop, LangChain uses its ReAct message-history interface, AutoGen uses AssistantAgent with PythonCodeExecutionTool, and OpenHands uses its default CLI agent. `DS-Lighting` follows the same public task interface and final grader, but additionally applies its built-in harness mechanisms for protocol checking, context management, execution governance, and public output-contract inspection. These checks only enforce syntactic public requirements such as artifact existence, filename, and location; they do not reveal correctness, labels, or metric values. Therefore, the comparison isolates framework-level execution behavior under identical task information and evaluation metrics, with `DS-Lighting`'s gains attributable to its runtime design rather than access to extra benchmark supervision.

Operator implementation and contract validation. `DS-Lighting` does not treat workflow steps as free-form prompt fragments. Instead, each step is implemented as an operator with an explicit asynchronous API, declared inputs and outputs, and controlled side effects. Operators can be composed sequentially, in parallel, conditionally, or through a dynamic DAG runtime, where each edge explicitly binds an upstream output key to a downstream input key. A node is dispatched only after all dependencies have succeeded and all required inputs are available; Figure 15 shows a representative declarative operator node. Interface contracts are enforced at three levels: structured LLM outputs are validated against schemas before downstream use; execution operators check runtime arguments and run generated code inside an isolated sandbox; and benchmark-facing artifacts are checked against the required submission contract, including output name, artifact type, required files, suffix constraints, and format-level requirements such as CSV columns or sample-submission consistency. Figure 16 gives an example of a structured operator output validated before use. Thus, workflow state is passed through explicit typed objects and node results rather than implicit prompt history alone.

F Prompt and Task-Contract Examples

Figure 15 and Figure 17 illustrate how `DS-Lighting` represents each benchmark instance as a structured task contract and renders it into an agent-facing prompt. The contract follows $x = (q, D, A)$,

```
{
  "task_id": "string",
  "benchmark": "string",
  "q": {"task_description": "string", "objective": "string", "metric": "string"},
  "D": {
    "working_directory": "./",

    "public_artifacts": [
      {"path": "string", "role": "train | test | sample_submission | metadata",
       "modality": "tabular | text | image | time_series", "format": "csv | json | parquet | txt",
       "profile": {"status": "ok | degraded | error", "rows_sampled": "integer",
                   "columns": [{"name": "string", "dtype": "string", "missing_pct": "number"}]}}
    ]
  },

  "A": {"submission": {"root_kind": "file | directory", "output_name": "string"},
        "evaluation": {"mode": "artifact_submission | judge", "official_metric": "string"}}
}
```

Figure 15: Schema of the structured task contract used by DS-Lighting.

```
{
  "operator_name": "LLMBasedReviewOperator",
  "output_type": "ReviewResult",
  "schema": {
    "is_buggy": "bool",
    "summary": "string",

    "metric_value": "float | null",
    "lower_is_better": "bool | null"
  },
  "example_output": {
    "is_buggy": false,
    "summary": "The code executed successfully and produced a valid submission.",
    "metric_value": 0.5876,

    "lower_is_better": true
  }
}
```

Figure 16: Example structured operator output validated before downstream use.

where q specifies the task objective, D describes public data artifacts and analyzer summaries, and A specifies the required answer artifact and official evaluation interface. All compared harnesses receive the same rendered task contract and public data-analysis feedback.

```
Task Description:
Compute the requested statistic from the Titanic table.
Write the final answer to the required submission file.

Data Analyzer Report:
- train.csv: tabular csv, 891 sampled rows, 12 columns
  PassengerId: int64, missing=0.0%, examples=[1, 2, 3]
  Fare: float64, missing=0.0%, examples=[7.25, 71.2833, 8.05]
- sample_submission.csv: sample submission
  answer: float64

I/O Requirements:
Save the final submission in the current working directory.
The required filename is submission.csv.
Use sample_submission.csv as the schema reference.
Do not rename the output file.

Evaluation:
The artifact is evaluated by the benchmark's official grader.
```

Figure 17: Prompt-style rendering of a DABench-style task contract.