FAIL LOUDLY: AN AUDITABLE RUNTIME FOR AGENTIC DATA ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have enabled data-science agents to automate multi-step analyses over heterogeneous files. However, incorrect choices regarding data sources, scope, or statistical definitions often lead to silent errors: computations execute successfully but produce plausible yet incorrect outputs that fail to answer the intended question. To mitigate this, we present RADAR[1], an auditable runtime that makes an agent's analytical choices inspectable and supports their revision through execution feedback. RADAR operates through three core mechanisms. First, an evidence-preserving exploration module retrieves task-relevant content while retaining source locations and observation coverage. Next, the runtime uses typed operators to record the agent's declared inputs, operation arguments, and resulting observations. Finally, runtime validation checks proposed operations against these observations. When a conflict is detected, the runtime rejects the operation or provides diagnostic feedback, allowing the agent to revise its choices before errors propagate. This design enables agents to fail loudly while leaving semantic interpretation to the LLM. On KramaBench, RADAR achieves overall scores of 0.723 with full source retrieval and 0.747 with gold sources supplied, corresponding to relative gains of 35.9% and 28.8% over the strongest baselines. Beyond KramaBench, RADAR achieves relative performance gains of 14.0% on DA-Code and 59.3% on DABStep, demonstrating its applicability across diverse agentic data-analysis workflows.

1 INTRODUCTION

LLM-based data-science agents can automate analyses over heterogeneous files, but successful code execution does not establish that an analysis answers the intended question. These agents interpret natural-language tasks, locate and combine data, and execute code to produce answers and artifacts (Yao et al., 2023; Wu et al., 2024; Zhang et al., 2024a; Guo et al., 2024; Hong et al., 2025; Nam et al., 2026). Translating a natural-language question into code requires commitments about the source, scope, field semantics, and statistical definition (De Bie et al., 2022). Incorrect choices can produce silent errors: plausible but incorrect outputs delivered without an explicit indication of failure (Vir et al., 2026). Unlike exceptions or missing files, these errors need not interrupt execution, allowing later computations to proceed on an incorrect basis (Stoisser et al., 2026).

Our audit shows that silent errors persist even in an agent equipped with iterative verification. Among 580 zero-scored DS-STAR (Nam et al., 2026) runs across KramaBench, DA-Code, and DABStep (Lai et al., 2026; Huang et al., 2024; Egg et al., 2025), 405 (69.8%) delivered normal-looking but incorrect results without reporting failure (Figure 1A). For example, a baseline asked to place two Top-5 rankings side by side merges them without a pairing condition (Figure 1B). The operation succeeds and produces a well-formed 25-row table with incorrect row correspondence. Such failures call for feedback that identifies suspect analytical choices despite successful execution.

Existing approaches largely rely on models to interpret source information and assess an analysis, leaving room for explicit runtime checks on individual operations. File descriptions and specialized file agents help navigate heterogeneous collections within a bounded context (Nam et al., 2026; Salemi et al., 2026). Recent work iteratively verifies plans using code and execution results or probes

[1]Code is available at `https://anonymous.4open.science/r/Data-Analysis-RADAR/`.

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

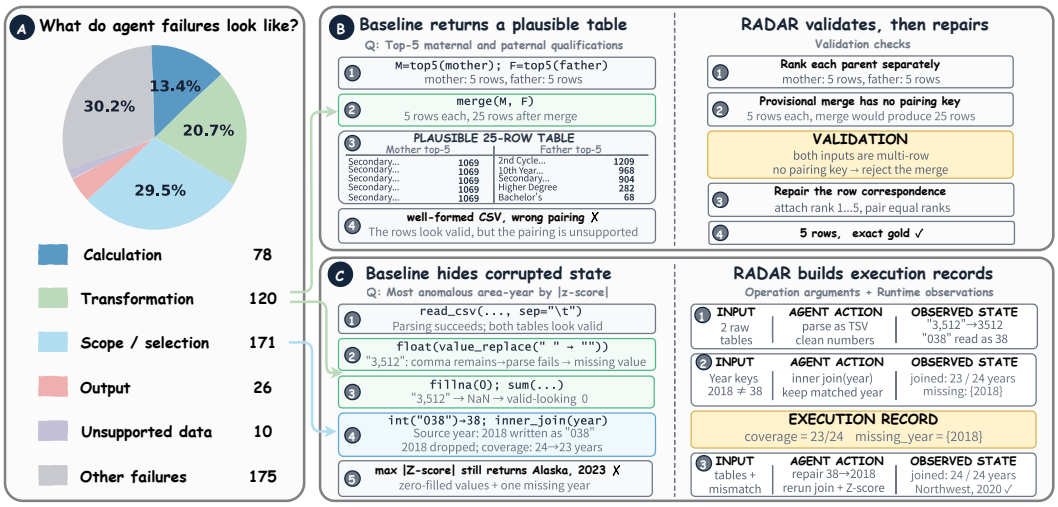

Figure 1: Silent errors and runtime interventions by RADAR. (A) Among 580 zero-scored DS-STAR runs across three benchmarks, 405 (69.8%) return normal-looking but incorrect outputs; we categorize their primary error types alongside other failures. (B) A baseline combines two Top-5 rankings without a pairing key, producing a well-formed 25-row table with incorrect row correspondence. On the same task, RADAR rejects an unkeyed merge before execution, prompting the agent to pair entries by rank. (C) A baseline silently loses one year after a successful inner join, while RADAR records the agent's inspection output that reveals the missing year and guides repair.

intermediate states to assess silent and grounding errors (Nam et al., 2026; Qiu et al., 2026). Checking an individual choice requires linking it to the relevant source observations and their coverage. When summaries omit critical constraints or these links are lost between analysis steps, unsupported interpretations can remain unchallenged as later computations depend on them (Laban et al., 2026). The challenge is to preserve this context across exploration and execution, and use it to determine when an operation warrants intervention.

Our key insight is that selected analytical choices become checkable when linked to source evidence and runtime observations with explicit scope and coverage. For example, a complete enumeration of a column's values can establish that a proposed filter value is absent, whereas its omission from a partial preview cannot. In the ranking example (Figure 1B), the missing pairing condition and recorded input sizes provide grounds to reject an implicit join and request an explicit correspondence. Such checks identify local conflicts, while the agent remains responsible for interpreting the task and choosing an appropriate revision.

To operationalize this, we present RADAR, an auditable runtime that connects evidence-preserving exploration, typed execution, and runtime validation. Evidence-preserving exploration combines retrieval and profiling with LLM semantic selection, returning selected content with source locations, observation scope and coverage. Typed operators expose operation arguments and declared dependencies, and their execution records link each invocation to observations of its inputs and results. Runtime validation uses these records to check operation arguments, execution results, and output requirements against the available evidence. On conflict, it responds through rejection, targeted diagnostic feedback, or bounded internal repair. This design separates semantic interpretation from runtime validation, enabling agents to fail loudly and revise their analysis.

Across KramaBench, DA-Code, and DABStep, RADAR improves task performance while reducing both observed silent errors and non-delivery. Across 1,054 tasks, silent errors decrease from 405 for DS-STAR to 289 for RADAR, a reduction of 28.6%, while non-deliveries decrease from 84 to 24. On KramaBench, it achieves Overall scores of 0.723 under Full Input and 0.747 under Oracle Input, compared with 0.532 and 0.580 for the strongest baseline in each regime. The advantage with gold sources supplied indicates that the gains extend beyond source retrieval. On DA-Code and DABStep, it improves over DS-STAR, the strongest baseline on both benchmarks, by 6.5 and 19.5 points, respectively. Ablations show that both evidence-preserving exploration and runtime validation contribute to higher task scores and fewer silent errors.

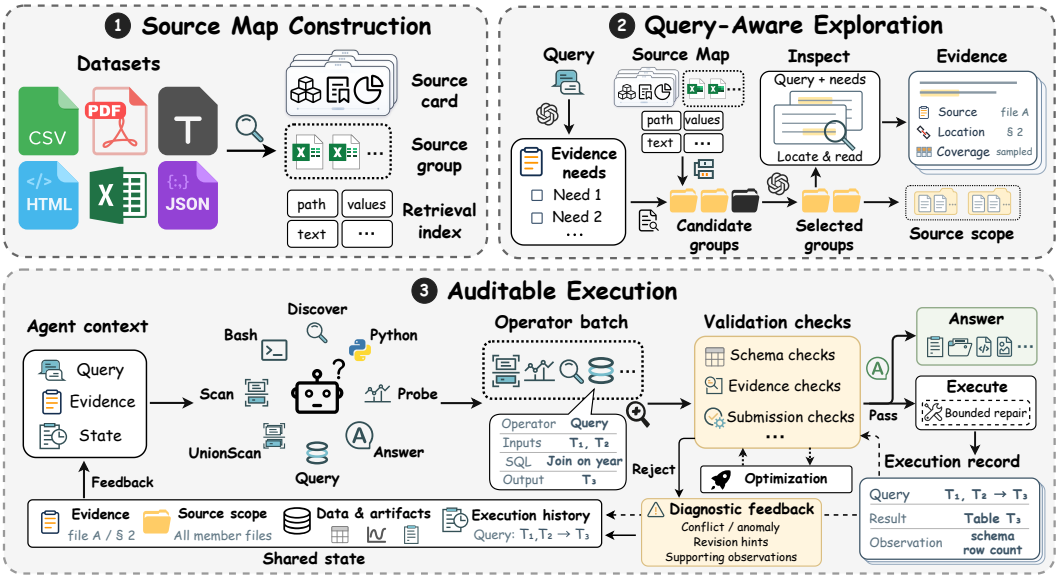

Figure 2: Overview of RADAR. Source Map construction supports query-aware source selection and inspection. The agent submits typed operator batches to the runtime, which applies validation and optimization, records runtime observations in shared state, and returns feedback for revision.

2 Evidence-Preserving Exploration

RADAR retrieves evidence for analysis while preserving its source locations and observation coverage. The input is a natural-language query q over a data lake of heterogeneous files. Source Map Construction in Figure 2 ❶ builds a query-independent representation of these files for retrieval. Query-Aware Exploration in Figure 2 ❷ then selects relevant sources and inspects their contents using q. The outputs are the retrieved evidence E_q and the selected source scope $\mathcal{X}_q$, which identifies the files retained for execution. Both initialize the analysis in Section 3.

2.1 Source Map Construction

Source Map construction records what each file contains and where its contents can be found, so these observations can be reused across queries. Because a file may contain several addressable parts, such as workbook sheets, RADAR profiles these parts separately as source units. Each unit retains its file path and position, such as a sheet name, table index, or JSON pointer. A format-specific profiler records each unit's observed structure and representative content, including schemas and value characteristics, document sections, or array shapes and types. The profiler stores these descriptions in a per-file record called a Source Card. Each card also records observation coverage, distinguishing complete inspection from observations limited by the profiling budget.

Related files or units are grouped so that retrieval and selection can consider them together. RADAR forms these source groups from units in the same workbook, files with shared path patterns, and compatible tables in the same directory. For example, files partitioned by year can be retrieved as a group so that source selection can consider multiple years together. These groups form the set $\mathcal{G}$, with each remaining source assigned to its own group. To make the recorded information searchable, RADAR builds a retrieval index over paths and names, observed columns and values, and descriptive content. Appendix A specifies the grouping rules and indexed fields.

2.2 Query-Aware Exploration

Query-aware exploration uses the information needed to answer q to guide source retrieval, selection, and inspection. An LLM first decomposes q into evidence needs, each describing one piece of source information required by the question. Each retrieval view v contains either q alone or q paired with one evidence need. For each view, BM25 (Robertson & Zaragoza, 2009) ranks source

groups separately for each indexed field $f \in \mathcal{F}$. A group at position r in a field ranking receives a contribution of $1/(\kappa + r)$, where $\kappa > 0$ is an offset. We sum these contributions across fields to obtain

$$s_v(g) = \sum_{f \in \mathcal{F}} \frac{1}{\kappa + \mathrm{rank}_{v,f}(g)},$$

where $\mathrm{rank}_{v,f}(g)$ is group g's position in field f's ranking for view v. Combining ranks prioritizes groups that rank highly in multiple fields without requiring BM25 scores from different fields to be directly comparable.

Candidate construction retains groups retrieved for different evidence needs. RADAR interleaves the rankings from the retrieval views to form the candidate set $\mathcal{K}_q \subseteq \mathcal{G}$, keeping the first occurrence of each group. Retrieval over individual source units also highlights relevant members of each candidate group. For each candidate, the LLM selector receives member paths, observed fields and values, content previews from Source Cards, and matched document sections. Using q and its evidence needs, the selector chooses groups likely to contain the required evidence, forming $\mathcal{G}_q \subseteq \mathcal{K}_q$.

RADAR inspects the selected groups using q and its evidence needs to retrieve relevant records and passages. For tables, RADAR combines structural information from Source Cards with original rows selected by lexical and semantic matching. For document text, it ranks sections and passages by relevance and retains adjacent context for lists and tables. The resulting evidence E_q identifies file paths and positions within each file, allowing the agent to return to the underlying content. Accompanying reports identify which sources were inspected, which portions are presented, and content omitted because of inspection limits.

The handoff to execution preserves all files in the selected groups, including files that contributed no displayed evidence. The selected source scope is

$$\mathcal{X}_q = \bigcup_{g \in \mathcal{G}_q} \mathrm{Files}(g),$$

where $\mathrm{Files}(g)$ denotes the files belonging to group g. The Evidence and Source scope outputs in Figure 2 ❷ therefore serve different roles, providing observed content and retaining files for further analysis. The agent receives $(E_q, \mathcal{X}_q)$ as its initial context, and the runtime retains both in the shared state shown in Figure 2 ❸. Subsequent operations can load the selected sources and inspect additional content, adding observations beyond the initial evidence.

3 Auditable Execution

Auditable execution uses the evidence and source scope from exploration to check operations, record invocations and runtime observations, and support revision (Figure 2 ❸). Using the initial context, the agent submits operator batches whose structured arguments and declared inputs expose its analytical choices to the runtime. Validation checks use the available evidence to determine whether to reject an invocation or allow it to proceed. Execution observations and failures can trigger diagnostic feedback or bounded internal repair. Execution records and diagnostic feedback update shared state and inform the agent's next batch.

3.1 Typed Operators and Execution Records

Typed operators expose the arguments and input references needed to connect each operation to relevant observations. Each type specifies structured arguments, an execution procedure, and the observations it returns. Table 1 summarizes the eight operator types, which cover source discovery, data access, inspection, computation, and answer submission. For example, Query executes SQL through DuckDB (Raasveldt & Mühleisen, 2019), with `inputs` naming its input tables and `output` naming the result. The Query illustrated in Figure 2 joins T_1 and T_2 on year and names its result T_3. The declared inputs let the runtime check the SQL against observations of the referenced tables, while the output name allows subsequent operations to reference the result.

After execution, the runtime creates an execution record containing the invocation's arguments, declared inputs, and resulting observations. Shared state retains these records together with exploration

Table 1: Typed operators, their main arguments, and runtime observations.

Operator type	Main arguments	Function and runtime observations
Discover	`query`, `excluded`	Retrieve sources by semantic query or path pattern; return paths.
Bash	`command`	Inspect files with shell commands; record output and exit status.
Scan	`file`	Load a table; record provenance, schema, row count, and preview.
UnionScan	`files`, `source_column`	Union by column name; record source labels and table metadata.
Probe	`table`, `columns`	Inspect columns; report value frequencies, ranges, and null rates.
Query	`sql`, `inputs`, `output`	Run SQL; record result schema, row count, and visible records.
Python	`code`, `inputs`, `output`	Run code; record values, program output, and artifact metadata.
Answer	`result_type`, `result`	Submit an explicit result; check its type and artifact requirements.

evidence, the selected source scope, loaded data, and artifact metadata. When an invocation names T_3 as an input, the runtime can retrieve its recorded schema and row count, along with any Probe observations for its columns. The agent may submit several invocations in one response, forming a batch that the runtime processes in order. Records and diagnostic feedback update shared state after each invocation, so an earlier Probe can supply evidence for a later Query in the same batch.

3.2 Validation Checks

The Validation checks stage in Figure 2 compares choices expressed in operation arguments with observations of the referenced inputs. Algorithm 1 shows how the runtime checks a Query invocation. Each check requires a supported SQL structure and the observations specified by its rule, and is skipped when either requirement is unmet. If none rejects the invocation, it follows the Pass branch and proceeds to execution, while the agent remains responsible for deciding whether the analysis answers the question.

Observation coverage determines whether an unobserved filter value provides grounds for rejection. An exploration preview identifies inspected content, while the literal coverage check requires a complete value enumeration from Probe. For a supported equality predicate $T.c = v$, let $L_{T,c}$ be Probe's returned list of non-null values for table T and text column c. With $d_{T,c}$ denoting the distinct non-null count and τ the enumeration bound, the required completeness condition is

$$\mathrm{Complete}(T, c) \iff 0 < d_{T,c} \leq \tau \ \wedge \ |L_{T,c}| = |\operatorname{set}(L_{T,c})| = d_{T,c}.$$

The check rejects a string literal v absent from $\operatorname{set}(L_{T,c})$ only when this condition holds. For example, if the complete list contains only `USA` and `Canada`, the check rejects `country = 'United States'` and returns the observed values. The same omission from a partial preview leaves the check without sufficient evidence to reject.

Schema, evidence, and submission checks cover different sources of conflict. Within its supported cases, the field binding check rejects a column name absent from all known input schemas and query-defined aliases. For an implicit join between multirow inputs, the runtime requests an explicit pairing condition or an explicitly intended cross product. In the ranking example (Figure 1B), combining two five-row inputs without a pairing condition triggers this rule before the operation produces 25 combinations. Answer checks the declared result type and applicable artifact requirements before accepting the result and ending the analysis. Appendix B.2 specifies the full set of checks and their supported cases.

Algorithm 1: Pre-execution checks for a Query invocation

Input: Query invocation o and current shared state S
for each *Query check* r **do**
  if r *does not support the SQL structure of* o **then continue**
  $e \leftarrow$ retrieve the observations required by r for the inputs of o from S
  if e *does not meet the evidence requirements of* r **then continue**
  if *the rejection condition of* r *holds for* (o, e) **then**
    return *Reject with the conflict, supporting observations, and revision hints*
return *Proceed to execution*

3.3 Diagnostic Feedback and Bounded Repair

Diagnostic feedback supports revision after a rejected invocation or a suspicious execution result. The Reject branch in Figure 2 returns the conflict, supporting observations, and revision hints. For invocations that proceed to execution, the resulting observations may also warrant diagnostic feedback. For example, when a filtered or joined Query succeeds but returns no rows, the runtime inspects the referenced values and join keys and returns diagnostics alongside the result. Since an empty result can be valid, the agent decides whether to accept it or revise the operation. Advisory rules use shared state to guide the next batch through the Feedback path without rejecting an invocation. Revised invocations undergo the same checks.

The Execute stage may attempt bounded internal repair for SQL execution failures and failed or suspicious data loads, using an LLM with the relevant error and source observations. For SQL repair, the prompt requires preserving analytical intent. The runtime rejects a candidate if its underlying data sources differ from those of the submitted SQL or either source set cannot be extracted. Execution records retain both SQL versions; failed repair returns diagnostics to the main agent.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate on KramaBench (Lai et al., 2026), DA-Code (Huang et al., 2024), and DABStep (Egg et al., 2025). KramaBench contains 104 tasks over heterogeneous data lakes spanning six domains. We consider Full Input, which requires source retrieval, and Oracle Input, which supplies the gold sources. DA-Code contains 500 tasks covering Data Wrangling, Machine Learning (ML), and exploratory data analysis (EDA). DABStep contains 450 tasks over structured and unstructured financial data.

Baselines. We compare six agent-based solutions: DS-STAR's file analysis and iterative verification (Nam et al., 2026); DA-Agent's reasoning–action–observation loop (Huang et al., 2024); Data Interpreter's task graphs and progressive verification (Hong et al., 2025); AutoGen's multi-agent coordination (Wu et al., 2024); smolagents Deep Research (Hugging Face, 2025); and DS-Guru, KramaBench's provided baseline (Lai et al., 2026). The KramaBench human expert study is included for reference.

Implementation. All systems receive the same raw files within each input setting and use their own exploration pipelines. The main experiments use DeepSeek-V4-Flash (Preview) for all systems with temperature 0. All LLM components in RADAR use this model. We allow all systems up to 25 main-agent decision rounds per task. We report total query-time API cost over each evaluated task suite and mean per-task latency, which includes offline preprocessing time amortized across queries.

4.2 Main Results

RADAR achieves the highest Overall scores across all three benchmarks with DeepSeek-V4-Flash (Tables 2, 3, and 4). On KramaBench, it leads all six domains under both Full Input and Oracle Input. With gold sources supplied, its Overall score reaches 0.747, compared with 0.580 for DS-STAR, the strongest baseline in this setting. This result indicates that the gains extend beyond source retrieval. On DA-Code, RADAR improves across Data Wrangling, ML, and EDA, with its largest margin in Data Wrangling. This pattern is consistent with its emphasis on inspecting source data and checking transformations during execution. On DABStep, the Overall improvement is concentrated in Hard tasks, where RADAR achieves 47.1% versus 22.8% for DS-STAR, an increase of 24.3 percentage points. These tasks account for 84% of the benchmark, so their gains outweigh a 5.5-point decrease on Easy tasks.

4.3 Effect on Silent Errors

To assess silent errors beyond benchmark scores, we audit zero-scored runs across the three benchmarks and group silent errors into five categories: wrong scope or selection, wrong transformation, wrong calculation, wrong output, and fabricated or unsupported data. Each silent error is assigned one primary category; remaining failed runs are classified as other failures.

Table 2: End-to-end scores and resource use on KramaBench across backbone models. Overall is the task-count-weighted mean. The Human row is from the KramaBench expert study. Bold marks the best automated system within each input setting and backbone. GPT-5.6 Luna uses high reasoning effort; GPT-4.1-mini uses the 2025-04-14 snapshot.

		Domains							Resource use	
System	Models	Archaeology	Astronomy	Biomedical	Environment	Legal	Wildfire	Overall	Cost ($)	Latency (s)
Full Input (source retrieval from the complete data lake)										
Human		0.583	0.700	1.000	0.769	0.867	0.662	0.768	–	–
DS-Guru	DeepSeek V4-Flash	0.000	0.083	0.000	0.150	0.200	0.305	0.158	1.27	49
smolagents DR		0.417	0.250	0.556	0.533	0.533	0.746	0.532	6.38	636
AutoGen		0.417	0.250	0.111	0.333	0.500	0.380	0.372	3.16	183
Data Interpreter		0.250	0.167	0.000	0.300	0.200	0.681	0.301	0.80	170
DA-Agent		0.417	0.208	0.111	0.469	0.567	0.651	0.467	1.32	105
DS-STAR		0.333	0.167	0.500	0.250	0.667	0.708	0.484	3.47	307
RADAR		**0.500**	**0.625**	**0.667**	**0.733**	**0.800**	**0.809**	**0.723**	1.16	89
DS-Guru	GPT-5.6 Luna	0.333	0.083	0.111	0.150	0.250	0.309	0.221	2.08	301
smolagents DR		0.333	0.500	0.444	0.700	0.396	0.599	0.504	7.44	476
AutoGen		**0.417**	0.167	0.444	0.460	0.433	0.475	0.415	4.22	49
Data Interpreter		**0.417**	0.250	0.444	0.200	0.233	0.309	0.283	1.69	70
DA-Agent		**0.417**	0.417	0.556	0.683	0.667	0.703	0.610	1.32	80
DS-STAR		0.333	0.250	0.444	0.500	0.733	0.578	0.530	7.87	188
RADAR		**0.417**	**0.708**	**0.667**	**0.783**	**0.853**	**0.761**	**0.738**	1.24	75
DS-Guru	GPT-4.1 mini	0.083	0.000	0.000	0.000	0.067	0.179	0.065	1.29	24
smolagents DR		0.167	0.125	0.222	0.350	0.250	0.319	0.257	6.22	77
AutoGen		0.250	0.271	0.167	0.307	0.331	0.388	0.307	4.76	45
Data Interpreter		0.167	0.167	0.000	0.350	0.220	0.429	0.256	4.28	249
DA-Agent		0.250	0.208	0.111	0.380	0.430	0.386	0.338	1.91	56
DS-STAR		0.167	0.250	0.222	0.413	0.420	**0.453**	0.359	6.44	76
RADAR		**0.417**	**0.333**	**0.278**	**0.493**	**0.600**	0.406	**0.460**	2.29	47
Oracle Input (gold sources supplied)										
DS-Guru	DeepSeek V4-Flash	0.167	0.083	0.111	0.150	0.200	0.286	0.183	0.72	43
smolagents DR		0.417	0.333	0.667	0.402	0.600	0.746	0.545	4.27	346
AutoGen		0.417	0.417	0.556	0.452	0.633	0.535	0.522	1.55	62
Data Interpreter		0.333	0.250	0.222	0.319	0.333	0.681	0.382	0.52	101
DA-Agent		0.333	0.375	0.333	0.652	0.600	0.503	0.511	0.94	95
DS-STAR		0.333	0.417	0.556	0.508	0.700	0.719	0.580	0.93	245
RADAR		**0.583**	**0.667**	**0.778**	**0.783**	**0.800**	**0.764**	**0.747**	0.91	67

Table 3: DA-Code mean scores (%) on 500 tasks. EDA aggregates 79 Data Insights, 73 Data Manipulation, 70 Statistical Analysis, and 78 Visualization tasks, weighted by task count. Completion represents the proportion of tasks for which the model produces results.

System	Data Wrangling	ML	EDA	Easy	Medium	Hard	Overall	Completion
DS-Guru	16.5	34.6	16.5	27.5	18.5	17.3	20.1	70.8
smolagents DR	29.2	44.8	39.6	53.8	36.6	28.7	38.5	91.6
AutoGen	31.4	41.8	35.2	46.8	33.7	30.7	35.8	86.6
Data Interpreter	21.0	24.5	32.6	36.8	26.9	25.3	28.6	84.8
DA-Agent	33.9	51.5	44.3	55.4	42.1	36.3	43.6	90.0
DS-STAR	33.6	58.8	46.6	61.4	42.6	42.2	46.4	96.8
RADAR	**42.4**	**62.6**	**53.2**	**69.0**	**50.4**	**43.9**	**52.9**	**98.4**

Figure 3 shows that RADAR has the lowest combined rate of non-delivery and silent errors on each benchmark. Compared with DS-STAR, it reduces both outcomes across all three benchmarks. Across 1,054 tasks, silent errors decrease from 405 to 289, a reduction of 28.6%, while non-deliveries decrease from 84 to 24. The combined rate is 29.7%, compared with 46.4%–83.0% for the baselines.

Among substantive deliveries, the silent-error rate decreases from 41.8% for DS-STAR to 28.1% for RADAR (Figure 4). Transformation errors show the largest absolute decrease, from 12.4% to 3.7%, consistent with the runtime's checks on input relationships and execution results. Output

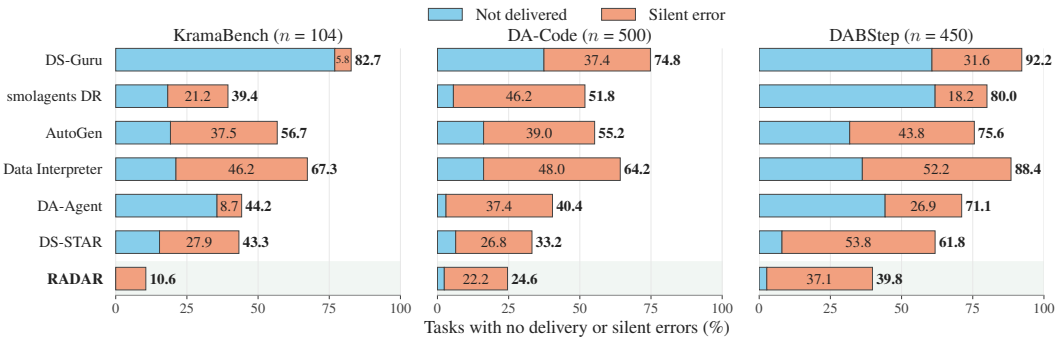

Figure 3: Non-delivery and silent errors across systems on each benchmark.

Table 4: DABStep accuracy (%). Easy, Hard, and Overall cover 72, 378, and 450 tasks, respectively; **bold** marks the best result per column.

System	Easy	Hard	Overall
DS-Guru	45.8	0.5	7.8
smolagents DR	73.6	9.8	20.0
AutoGen	69.4	15.9	24.4
Data Interpreter	66.7	1.1	11.6
DA-Agent	75.0	20.1	28.9
DS-STAR	**86.1**	22.8	32.9
RADAR	80.6	**47.1**	**52.4**

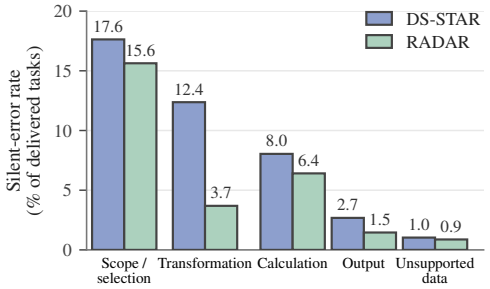

Figure 4: Silent-error rates by category among delivered tasks for DS-STAR and RADAR.

errors decrease by 45.7%, aligning with output-contract validation. Scope or selection remains the most frequent category, decreasing from 17.6% to 15.6%.

4.4 Component Analysis

We ablate evidence-preserving exploration and runtime validation across all three benchmarks. The exploration ablation uses only file metadata and provides no initial table or document evidence. Without runtime validation, checks, feedback, and internal repair are disabled.

Table 5 shows that both ablations reduce performance across all three benchmarks. Removing evidence-preserving exploration has the largest effect on KramaBench, consistent with its need to locate and interpret sources across heterogeneous data lakes. DABStep shows the largest decrease from removing runtime validation, consistent with the need to check intermediate decisions across multi-step analyses.

Figure 5 shows that the score reductions are accompanied by more silent errors on every benchmark. Across all 1,054 tasks, removing evidence-preserving exploration increases the count from 289 to 409, while removing runtime validation increases it to 377.

4.5 Efficiency and Backbone Robustness

Efficiency. Table 2 shows that under KramaBench Full Input with DeepSeek-V4-Flash, RADAR costs $1.16 and averages 89 seconds per task, versus $6.38 and 636 seconds for smolagents DR, the strongest baseline by Overall score. Under Oracle Input, it costs $0.91 and averages 67 seconds, versus $0.93 and 245 seconds for DS-STAR, the strongest baseline in this setting.

Backbone robustness. To assess performance across backbone models, we additionally evaluate all systems with GPT-5.6 Luna and GPT-4.1-mini under KramaBench Full Input. The inclusion of the earlier GPT-4.1-mini model is motivated by concerns about benchmark contamination. As shown

Table 5: Ablation results across the three benchmarks. Parentheses show decreases in percentage points from Full.

Setting	KramaBench	DA-Code	DABStep
Full	**72.3**	**52.9**	**52.4**
w/o evidence-preserving exploration	61.6 (−10.7)	48.9 (−4.0)	44.2 (−8.2)
w/o runtime validation	64.6 (−7.7)	50.6 (−2.3)	42.2 (−10.2)

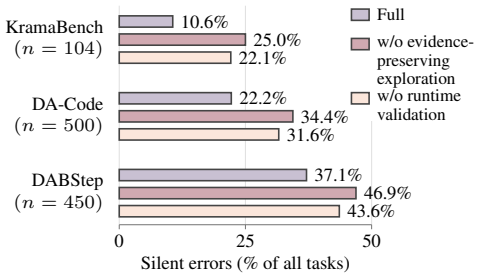

Figure 5: Silent-error rates under component ablations.

in Table 2, RADAR achieves Overall scores of 0.738 and 0.460, respectively, outperforming all evaluated baselines with both backbones.

5 Related Work

Data-science agents. Data-science agents aim to automate end-to-end analysis. Benchmarks cover scientific discovery, multi-step reasoning, deep data research, and table question answering (Chen et al., 2025; Egg et al., 2025; Hu et al., 2024; Jing et al., 2025; Liu et al., 2026b; Nie et al., 2026; Wu et al., 2025), while broader studies survey the area and examine the limitations of open models (Zhu et al., 2026a;b). Existing systems support visualization (Yang et al., 2024), insight discovery and report generation (Abaskohi et al., 2025; Liu et al., 2026a; Ma et al., 2023; Xu et al., 2025; Zhang et al., 2024b), heterogeneous data analysis (Nam et al., 2026; Qi et al., 2026; Sun et al., 2025; Zhang et al., 2024a), and general data-science workflows (Hong et al., 2025; You et al., 2025; Guo et al., 2024). Recent work trains open models for general data analysis or process verification (Qiao et al., 2026; Zhang et al., 2026; Qiu et al., 2026), while EvoDS develops reusable skills and long-horizon context (Yang et al., 2026). Adapting fine-tuned methods to new domains or model backbones may require additional data and retraining, and they may not apply to proprietary API models, motivating RADAR's training-free design. Closest to our setting, DS-STAR and Blackboard use file analyzers or partition agents to access heterogeneous collections (Nam et al., 2026; Salemi et al., 2026). Their downstream outputs may omit the source location, scope, or coverage needed to audit later choices; RADAR preserves this context in evidence.

Runtimes for AI-driven analytics. Work on AI-driven analytics covers LLM-backed maps, filters, joins, and aggregations over unstructured data, as well as execution optimization for cost and output quality (Liu et al., 2025; Kumarasinghe et al., 2026; Patel et al., 2025; Russo et al., 2025; Shankar et al., 2025; Song et al., 2026). AOP predefines semantic operators and uses LLMs to compose and interactively execute pipelines for complex data-lake queries (Wang & Li, 2025). Russo & Kraska (2026) connect optimized semantic-operator execution with Deep Research agents through agent-backed search and compute operators and reusable materialized contexts. Across these systems, explicit operators support pipeline construction and optimized execution. In RADAR, typed operators make agent-selected data operations, their dependencies, and intermediate states observable as execution records for runtime validation.

6 Conclusion

We present RADAR, an auditable runtime for agentic data analysis. It preserves source locations, observation scope, and coverage, and links typed operations to runtime observations so that conflicts can prompt diagnostic feedback and revision. Across KramaBench, DA-Code, and DABStep, RADAR achieves the highest Overall scores among the evaluated agents, with 28.6% fewer observed silent-error cases than DS-STAR. On KramaBench, it also costs less than the strongest baseline in each input regime. Runtime validation remains limited by available evidence; broader validation coverage and cost-based optimization are directions for future work.

AI USE STATEMENT

Generative AI assisted with coding, figures and tables, language editing, experimental design discussions, and review. The authors reviewed the methods, experimental configurations, data checks, statistical calculations, code, and claims, and take responsibility for all content and artifacts.

REFERENCES

Amirhossein Abaskohi, Amrutha Varshini Ramesh, Shailesh Nanisetty, Chirag Goel, David Vazquez, Christopher Pal, Spandana Gella, Giuseppe Carenini, and Issam H. Laradji. Agentada: Skill-adaptive data analytics for tailored insight discovery, 2025. URL https://arxiv.org/abs/2504.07421.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. ScienceAgentBench: Toward rigorous assessment of language agents for data-driven scientific discovery. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 96934–96990, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f12b4df26344f3be803c06b555252efe-Paper-Conference.pdf.

Tijl De Bie, Luc De Raedt, José Hernández-Orallo, Holger H. Hoos, Padhraic Smyth, and Christopher K. I. Williams. Automating data science. *Communications of the ACM*, 65(3):76–87, 2022. doi: 10.1145/3495256.

Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. Dabstep: Data agent benchmark for multi-step reasoning, 2025. URL https://arxiv.org/abs/2506.23719.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. DS-agent: Automated data science by empowering large language models with case-based reasoning. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=LfJgeBNCFI.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Robert Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Yongxin Ni, Zhibin Gou, Zongze Xu, Yuyu Luo, and Chenglin Wu. Data interpreter: An LLM agent for data science. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19796–19821. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1016. URL https://aclanthology.org/2025.findings-acl.1016/.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. InfiAgent-DABench: Evaluating agents on data analysis tasks. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 19544–19572. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/hu24s.html.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. 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*, pp. 13487–13521. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.748. URL https://aclanthology.org/2024.emnlp-main.748/.

Hugging Face. smolagents: Open deep research. https://github.com/huggingface/smolagents/tree/main/examples/open_deep_research, 2025. Software repository.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. DSBench: How far are data science agents from becoming data science experts? In *International Conference on Learning Representations (ICLR)*, 2025.

Udesh Kumarasinghe, Tyler Liu, Ahmed R. Mahmood, Chunwei Liu, and Walid G. Aref. ipdb – optimizing semantic sql queries, 2026. URL https://arxiv.org/abs/2601.16432.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=VKGTGGcwl6.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton A. Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, Jun Chen, Markos Markakis, Matthew Russo, Weiyang Wang, Ziniu Wu, Mike Cafarella, Lei Cao, Samuel Madden, and Tim Kraska. KRAMABENCH: A benchmark for AI systems on data-to-insight pipelines over data lakes. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=fZfUdeCC5X.

Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. Palimpzest: Optimizing AI-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, The Netherlands, January 2025. URL https://www.vldb.org/cidrdb/2025/palimpzest-optimizing-ai-powered-analytics-with-declarative-query-processing.html.

Shicheng Liu, Yucheng Jiang, Sajid Farook, Camila Nicollier Sanchez, David Fernando Castro Pena, and Monica S. Lam. DataSTORM: Deep research on large-scale databases using exploratory data analysis and data storytelling. *arXiv preprint arXiv:2604.06474*, 2026a.

Wei Liu, Peijie Yu, Michele Orini, Yali Du, and Yulan He. Hunt instead of wait: Evaluating deep data research on large language models. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=NooBkSo7oz.

Pingchuan Ma, Rui Ding, Shuai Wang, Shi Han, and Dongmei Zhang. InsightPilot: An LLM-empowered automated data exploration system. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2023. doi: 10.18653/V1/2023.EMNLP-DEMO.31.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Raj Sinha, Jinwoo Shin, and Tomas Pfister. DS-STAR: Data science agent for solving diverse tasks across heterogeneous formats and open-ended queries, 2026. URL https://arxiv.org/abs/2509.21825.

Fan Nie, Junlin Wang, Harper Hua, Federico Bianchi, Yongchan Kwon, Zhenting Qi, Owen Queen, Shang Zhu, and James Zou. DSGym: A standardized and holistic framework for evaluating and training data science agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=2WrNPRW9Km.

Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators and their optimization: Enabling llm-based data processing with accuracy guarantees in lotus. *Proceedings of the VLDB Endowment*, 18(11):4171–4184, 2025.

Ruyi Qi, Zhou Liu, and Wentao Zhang. DataCross: A unified benchmark and agent framework for cross-modal heterogeneous data analysis. *arXiv preprint arXiv:2601.21403*, 2026.

Shuofei Qiao, Yanqiu Zhao, Zhisong Qiu, Xiaobin Wang, Jintian Zhang, Zhao Bin, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Scaling generalist data-analytic agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=5PxFqpIYWC.

Zhisong Qiu, Shuofei Qiao, Kewei Xu, Yuqi Zhu, Lun Du, Ningyu Zhang, and Huajun Chen. Rewarding the scientific process: Process-level reward modeling for agentic data analysis. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 11866–11877. ACM, August 2026. doi: 10.1145/3770855.3819049. URL https://doi.org/10.1145/3770855.3819049.

Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pp. 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3320212. URL https://doi.org/10.1145/3299869.3320212.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009. doi: 10.1561/1500000019.

Matthew Russo and Tim Kraska. Deep research is the new analytics system: Towards building the runtime for ai-driven analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2026.

Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. Abacus: A cost-based optimizer for semantic operator systems. *arXiv preprint arXiv:2505.14661*, 2025.

Alireza Salemi, Mihir Parmar, Palash Goyal, Yiwen Song, Jinsung Yoon, Hamed Zamani, Tomas Pfister, and Hamid Palangi. Llm-based multi-agent blackboard system for information discovery in data science, 2026. URL https://arxiv.org/abs/2510.01285.

Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *Proc. VLDB Endow.*, 18(9):3035–3048, May 2025. ISSN 2150-8097. doi: 10.14778/3746405.3746426. URL https://doi.org/10.14778/3746405.3746426.

Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. QUITE: A query rewrite system beyond rules with LLM agents, 2026. URL https://arxiv.org/abs/2506.07675.

Josefa Lia Stoisser, Marc Boubnovski Martell, Sidsel Boldsen, Kaspar Märtens, and Robert Kitchen. Ambig-DS: A benchmark for task-framing ambiguity in data-science agents, 2026. URL https://arxiv.org/abs/2605.09698.

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

Reya Vir, Lydia Chilton, Zhuo Zhang, and Eugene Wu. Flowcheck: Helping end-users specify and verify intent in vibe-coded web apps, 2026. URL https://arxiv.org/abs/2608.28880.

Jiayi Wang and Guoliang Li. AOP: Automated and interactive LLM pipeline orchestration for answering complex queries. In *Conference on Innovative Data Systems Research (CIDR)*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, Tongliang Li, Zhoujun Li, and Guanglin Niu. Tablebench: a comprehensive and complex benchmark for table question answering. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. ISBN 978-1-57735-897-8. doi: 10.1609/aaai.v39i24.34739. URL https://doi.org/10.1609/aaai.v39i24.34739.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-pack: Packed resources for general chinese embeddings, 2024. URL https://arxiv.org/abs/2309.07597.

Wenyi Xu, Yuren Mao, Xiaolu Zhang, Chao Zhang, Xuemei Dong, Mengfei Zhang, and Yunjun Gao. DAgent: A relational database-driven data analysis report generation agent. *arXiv preprint arXiv:2503.13269*, 2025.

Zherui Yang, Fan Liu, Yansong Ning, and Hao Liu. EvoDS: Self-evolving autonomous data science agent with skill learning and context management. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 6128–6139. ACM, August 2026. doi: 10.1145/3770855.3818002. URL https://doi.org/10.1145/3770855.3818002.

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, Zhiyuan Liu, Xiaodong Shi, and Maosong Sun. MatPlotAgent: Method and evaluation for LLM-based agentic scientific data visualization. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11789–11804, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.701. URL https://aclanthology.org/2024.findings-acl.701/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Ziming You, Yumiao Zhang, Dexuan Xu, Yiwei Lou, Yandong Yan, Wei Wang, Huamin Zhang, and Yu Huang. 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*, pp. 1099–1123. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.58. URL https://aclanthology.org/2025.emnlp-main.58/.

Shaolei Zhang, Ju Fan, Meihao Fan, Yizhe Liu, Yuxin Zhang, and Xiaoyong Du. DeepAnalyze: Agentic large language models for autonomous data science. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=tcPZvRXQhX.

Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yueting Zhuang. Data-copilot: Bridging billions of data and humans with autonomous workflow. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024a. URL https://openreview.net/forum?id=iUEBnQyfWE.

Xilin Zhang, Zhixin Mao, Ziwen Chen, and Shen Gao. Effective tool augmented multi-agent framework for data analysis. *Data Intelligence*, 6(4):923–945, 2024b. doi: 10.3724/2096-7004.DI.2024.0013.

Yizhang Zhu, Liangwei Wang, Chenyu Yang, Xiaotian Lin, Boyan Li, Wei Zhou, Xinyu Liu, Zhangyang Peng, Tianqi Luo, Yu Li, Chengliang Chai, Chong Chen, Shimin Di, Ju Fan, Ji Sun, Nan Tang, Fugee Tsung, Jiannan Wang, Chenglin Wu, Yanwei Xu, Shaolei Zhang, Yong Zhang, Xuanhe Zhou, Guoliang Li, and Yuyu Luo. A survey of data agents: Emerging paradigm or overstated hype?, 2026a. URL https://arxiv.org/abs/2510.23587.

Yuqi Zhu, Yi Zhong, Jintian Zhang, Ziheng Zhang, Shuofei Qiao, Yujie Luo, Lun Du, Da Zheng, Ningyu Zhang, and Huajun Chen. Why do open-source llms struggle with data analysis? a systematic empirical study. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence and Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence and Sixteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'26/IAAI'26/EAAI'26. AAAI Press, 2026b. ISBN 978-1-57735-906-7. doi: 10.1609/aaai.v40i41.40831. URL https://doi.org/10.1609/aaai.v40i41.40831.

A Evidence-Preserving Exploration

Source groups and retrieval. The system forms groups in the following order: units from the same workbook, files with a shared path pattern, and tables in the same directory that share a format and column structure. To compare paths, it replaces each sequence of two or more digits with a placeholder. To compare column names, it ignores case and order. It also ignores a numeric suffix when the corresponding name without that suffix is present. Groups formed from a workbook or a shared path pattern require at least two members; groups based on column structure require at least three. Each remaining source forms its own group, so every source belongs to one group.

The indexed fields are paths and names, observed columns and values, and descriptive content. After combining field rankings for each retrieval view (Section 2.2), the system interleaves the lists by rank, visiting individual evidence needs before the full query and keeping the first occurrence of each group. Candidate and selection limits appear in Table 8. If all groups fit within the selection limit, the system retains them all.

Evidence inspection and initial context. For tables, the system selects rows using lexical matching and semantic scores from `BAAI/bge-small-en-v1.5` (Xiao et al., 2024). It scores the eligible text cells in each row and uses the highest cell score as the row's semantic score. Reciprocal-rank fusion then combines the lexical and semantic rankings. If semantic retrieval is unavailable or there are more than 3,000 distinct eligible text cells, the system uses lexical matching alone. For documents, it selects relevant sections and passages, keeping their headings and the surrounding text needed to interpret lists or tables. In both cases, the evidence retains its source locations. Inspection reports identify content omitted because of the limits in Table 8.

Appendix D provides the prompts for identifying evidence needs and selecting source groups.

B Auditable Execution

B.1 Execution Records

Table 1 defines the eight operator interfaces. Within a batch, a rejected invocation returns its diagnostic, and a failed execution retains any available observations; the remaining invocations can proceed. A successful Answer ends the batch.

Python receives declared input tables as DataFrames and registers any returned DataFrame or Series as a queryable table. The execution record retains returned values, program output, and output-file observations. Records also identify reused results.

B.2 Validation Checks and Feedback

Table 6 specifies validation checks and their required observations. Checks run in registration order and stop at the first rejection. An operation rewritten by optimization is checked again, with at most three iterations.

Answer validates the declared result type. For file results, it checks existence, nonempty content, and a modification time within the current run. CSV outputs also undergo format and applicable template checks.

Additional feedback. Six advisory rules provide guidance on handling failures, revisiting source selection, inspecting data after repeated failures, addressing Probe anomalies, producing boolean answers, and writing SQL. Execution diagnostics inspect empty filter or join results and suspicious LAG/LEAD null patterns (Section 3.3).

B.3 Bounded Internal Repair

Bounded internal repair uses an LLM to handle SQL execution failures and failed or suspicious data loads. The SQL specialist receives the query, error, schemas, data samples, and the agent's rationale; its prompt requires preserving the analysis while fixing syntax, casting, or encoding issues. The

Table 6: Validation checks applied before operator execution and the observations required to trigger rejection.

Check	Condition and required observations
Field binding	For direct table references with resolved input schemas, a referenced column is absent from every known input schema and is not a query-defined alias. Hidden rowid and field access through a known struct column are exempt.
Literal coverage	In a directly bound single-table query, a string literal in a supported positive WHERE equality or IN predicate is absent from Probe's complete enumeration for that table and text column. Probe must return all $d \leq 30$ distinct, non-null values. CTEs, subqueries, joins, and set operations are excluded; predicates under OR/NOT/CASE/IF are skipped.
Implicit join	A join has no ON/USING, the enclosing SELECT has no WHERE, and both inputs are classified as potentially containing multiple rows from recorded row counts or query structure. Explicit CROSS, NATURAL, and POSITIONAL joins retain their specified semantics.
Window ordering	An order-sensitive window function (LAG, LEAD, ROW_NUMBER, ranking, NTILE, CUME_DIST, or first/last/nth value) lacks ORDER BY in its window.
Answer type	The result type is unrecognized, or the submitted value does not match the structure required by that type. Accepted types are number, string, boolean, list, JSON object, table, and artifact.
Premature default	The proposed Answer is empty or uses a default value. No successful Discover or Probe call, or qualifying non-default result from Query or Python, supports it. The runtime requests additional evidence.
Structured fallback	The answer is a nested empty placeholder, or an all-zero numeric list without an observed computation supporting it. The runtime requests a concrete result.
Zero after empty joins	At least two successful inner joins over nonempty inputs returned no rows. The proposed Answer is zero, but no subsequent Query/Python has returned a one-row result containing zero.
Discovery progress	A fifth consecutive Discover is proposed without another substantive operator intervening; Probe neither increments nor resets this count. The runtime requests source inspection.

runtime rejects a repair candidate if its underlying data sources differ from those of the submitted SQL or either set cannot be extracted. Execution records retain the submitted and repaired SQL. Loading repair reads the supplied local file into a DataFrame; errors from the main Python operator return to the analysis agent. Budgets and prompts appear in Appendices C.1 and D.

B.4 A Recorded Revision: Pairing Two Rankings

DA-Code task `dm-csv-003`, used in Figure 1B, requests the five most frequent maternal and paternal qualifications in descending frequency, in a four-column CSV. The run loads the 4,424-row `student_data.csv` and reads `README.md` for qualification-code definitions. Figure 6 shows the recorded rejection, the agent's revision, and the resulting output.

(a) Proposed SQL and feedback

```
SELECT *
FROM mother_qual, father_qual
ORDER BY mother_qual.frequency DESC
```

Runtime diagnostic

`join_without_condition`

"JOIN on 'father_qual' has no ON/USING — this is a cartesian product (rows explode N × M). Add 'ON a.key = b.key' matching the join key, or use 'CROSS JOIN' if intentional."

→

(b) Agent revision

```
-- Added to both ranking CTEs:
ROW_NUMBER() OVER (
  ORDER BY COUNT(*) DESC) AS rn

-- Final pairing:
FROM mother_qual mq
JOIN father_qual fq
  ON mq.rn = fq.rn
ORDER BY mq.frequency DESC
```

Each ranking receives a row number; the revised join pairs rows with the same rank.

(c) Recorded output `top_qualifications.csv`

Mother's qualification	Freq.	Father's qualification	Freq.
Secondary Education - 12th Year	1069	2nd Cycle of the General High School Course	1209
Technical-Professional Course	1009	10th Year of Schooling	968
Higher Education - Degree	438	Secondary Education - 12th Year	904
Basic Education 3rd Cycle	130	Higher Education - Degree	282
Higher Education - Bachelor's Degree	83	Higher Education - Bachelor's Degree	68

Figure 6: Recorded rejection and revision on DA-Code `dm-csv-003`. The highlighted SQL changes from combining two rankings without a condition to pairing rows by rank. The agent writes and reads back the five-row CSV before submission. SQL excerpts omit qualification mapping, aggregation, and output aliases; comments identify the excerpted changes. Output column headings are shortened for display.

B.5 A Recorded Revision: Restoring Year Coverage

Table 7 connects the three stages of Figure 1C to recorded agent actions on KramaBench `wildfire-hard-7`.

Table 7: Execution records supporting inspection, revision, and verification in Figure 1C. Entries summarize recorded outputs; step numbers start from zero.

Stage in Figure 1C	Agent action	Recorded evidence
1. Prepare inputs (steps 0–8)	Inspect both files through Scan and Bash; print table shapes and year lists through Python.	Both tables contain **24 rows**. The human-caused years span 2001–2024; the lightning source has `038` between 2019 and 2017, appearing as integer 38 in Python.
2. Inspect the join (step 9)	Compute the merged table in Python and print its shape and year list.	The merge returns **23 rows**, with **2018 absent** from the printed years. The input and result observations expose the loss of year coverage.
3. Revise and verify (steps 10–12)	Inspect the year type and affected row, change integer 38 to 2018, and recompute the merge and Z-scores.	The corrected row is printed. The new merge contains **all 24 years**, including 2018; the recomputed result is **Northwest, 2020**.

C Additional Evaluation Details

C.1 Implementation Settings and Construction Cost

Table 8 lists the default settings for profiling, retrieval, inspection, and repair. The main model and temperature are given in Section 4.

Table 8: Default implementation settings. Table and document evidence limits apply per group; other limits have the scope stated below.

Stage	Setting
Offline profiling	1,000 profile rows, 3 preview rows, and 256 columns per unit; 128 units per file; 1,000,000 text bytes and 16,000,000 structured-data bytes.
Group retrieval	BM25 $k_1 = 1.2$, $b = 0.75$; reciprocal-rank fusion offset 60; 50 candidates per field ranking; 20 candidate groups after combining rankings, plus groups containing filenames named in the question; up to 4 selected groups.
Table evidence	Original rows from up to 4 table units; a scan budget of 2,000 rows per unit; up to 12 evidence items and 6,000 characters.
Document evidence	Up to 5 sections, 10 evidence blocks, 16 raw evidence units, and 6,000 raw-evidence characters.
Probe	10 returned values and 12 columns by default; range statistics enabled. The literal coverage check and its required observations are specified in Appendix B.2.
Internal repair	Up to 3 LLM turns per invocation; 8,192 output tokens per turn by default. The repair specialist uses the same model as the main agent.

Interaction budget. All systems, including the baselines, use a limit of 25 main-agent rounds and a 1,200-second per-task timeout. The round limit applies to the main agent's analysis decisions. In RADAR, repair calls within an operator use the separate budget in Table 8. During final submission, the runtime permits one additional Answer call to correct a rejection and one Python call to complete a missing or empty output file. Recorded query-time token usage includes SQL repair and these final submission calls.

Component ablation configurations. Table 9 specifies the components retained and removed in Table 5. Both ablations use DeepSeek-V4-Flash (Preview), temperature 0, and the interaction budget specified above.

Table 9: Component configurations for the ablations. Basic execution and agent-directed tool use remain available in both ablations.

Component	Full	w/o evidence-preserving exploration	w/o runtime validation
Source profiling, content retrieval, and evidence inspection	Enabled	File metadata only	Enabled
Initial context for analysis	Exploration evidence and selected source scope	Evidence needs and selected source scope	Exploration evidence and selected source scope
File access and all eight operators	Retained	Retained	Retained
Execution records and basic Probe observations	Retained	Retained	Retained
Validation checks	Enabled	Enabled	Disabled
Diagnostic feedback and advisory rules	Enabled	Enabled	Disabled
Internal SQL and loading repair	Enabled	Enabled	Disabled

The exploration ablation uses file paths, names, extensions, and sizes for grouping and retrieval. It retains query decomposition into evidence needs and selection limits, while omitting content profiles, schema-based grouping, inspection, and initial table or document evidence. Discover uses

the same metadata index. This ablation measures the joint contribution of acquiring, selecting, and passing source evidence to analysis.

Without runtime validation, the components marked Disabled in Table 9 are removed. Ordinary execution errors and agent-directed revisions remain. This ablation measures the combined contribution of validation checks, feedback, and bounded internal repair.

Offline construction. Table 10 reports the time needed to build the Source Map for each KramaBench domain, using the default budget of 1,000 profiling rows and three preview rows. The measurements cover the full construction process, with files processed sequentially in a single process on an Intel Xeon Platinum 8480+ host.

Table 10: Offline construction costs on KramaBench. Sizes use decimal MB.

Domain	Files	Time (s)	Source Cards (MB)
Archaeology	5	1.53	0.03
Astronomy	1,538	293.79	16.22
Biomedical	8	8.72	0.37
Environment	37	4.01	0.14
Legal	132	0.78	0.28
Wildfire	24	39.98	0.12
Total	1,744	348.81	17.17

C.2 Runtime Intervention Audit

Audit population. We audit complete execution histories from DeepSeek-V4-Flash (Preview) runs on all 104 KramaBench Full Input tasks (Table 2), 500 DA-Code tasks (Table 3), and 450 DABStep tasks (Table 4), including runs with zero, partial, and full scores.

Probe inspections across all three benchmarks. Table 11 summarizes 507 Probe operations on inputs and intermediate tables, returning 505 reports. Each report is counted at most once within each observation category. Format hints include date-parsing advice; anomaly warnings identify possible missing-value markers or combined fields. Figure 7 connects these observations to subsequent computation.

Table 11: Probe inspections across all 1,054 audited tasks. Each entry gives the number of operations, with the number of unique tasks in parentheses. Rows describing report contents count operations that return the indicated information.

Observation	KramaBench ($n = 104$)	DA-Code ($n = 500$)	DABStep ($n = 450$)
Recorded Probe operations	79 (48)	262 (191)	166 (127)
Operations returning a report	79 (48)	262 (191)	164 (126)
Value frequencies	68 (41)	218 (166)	163 (125)
Value or date ranges	47 (31)	195 (143)	119 (106)
Nonzero null rates	23 (18)	101 (84)	74 (73)
Format hints	34 (21)	69 (54)	60 (57)
Explicit anomaly warnings	4 (3)	9 (6)	0 (0)

Figure 7 shows how a Probe observation relates to subsequent Python code on DA-Code `data-sa-013`. The task asks for the 95% Value at Risk (VaR) of an equally weighted stock portfolio over 2005–2006 and 2007–2009.

(a) Probe observation

```
Probe "portfolio" (1511 rows):
  'Date' VARCHAR, distinct=1511:
  date range (VARCHAR,
    parsed %d/%m/%Y):
  min=2004-12-31 00:00:00
  max=2010-12-30 00:00:00

  stored as VARCHAR -- wrap with
  TRY_STRPTIME("Date", '%d/%m/%Y')
  before any date
  filter/sort/EXTRACT(year)
```

The report identifies the column, its date format, and the observed range.

(b) Subsequent Python

```
df = tables['portfolio']
df['Date'] = pd.to_datetime(
    df['Date'], format='%d/%m/%Y')
df = df.sort_values('Date')\
       .reset_index(drop=True)
# ... returns and losses ...
df['Year'] = df['Date'].dt.year
period1 = df['Year'].isin(
    [2005, 2006])
period2 = df['Year'].isin(
    [2007, 2008, 2009])
var1 = np.percentile(
    losses[period1], 95)
var2 = np.percentile(
    losses[period2], 95)
```

(c) Recorded output `result.csv`

VaR_95, 2005–2006	VaR_95, 2007–2009
0.014687184472834538	0.0579057406681419

Figure 7: Probe observations and subsequent computation on DA-Code `data-sa-013`. Matching highlights identify the same column and date format in the report and Python code. The code parses dates before sorting and selecting the requested years to compute 95% VaR. Report and code excerpts are reformatted for display; imports, return calculations, and file-writing code are omitted.

From Probe observations to analysis code. To examine how an observation relates to subsequent computation, we inspect every Probe report that explicitly warns that a date column is stored as VARCHAR. On DA-Code, 25 reports across 21 tasks contain this warning. After 20 of these reports (80%, across 17 tasks), subsequent SQL, Python, or a program written through Bash explicitly converts at least one of the identified date columns from the same data source. On KramaBench, one of the four reports containing this warning is followed by such a conversion. Across DA-Code and KramaBench, 21/29 reports (72.4%) are therefore followed by code that converts an identified date column. The DABStep traces contain no reports with this warning.

Counting protocol. For validation checks, we count each rejected operator invocation or internal SQL repair candidate as one event. This includes output-file checks and checks on the data sources used during SQL repair. Checks that pass or are skipped do not contribute to these counts. Execution-interface requirements and controls on repeated queries and rewrites are reported separately. Repairs copied into reused results and advisory messages copied in conversation history are counted only at their original occurrence. Tables report events and unique tasks separately.

Runtime activity. Table 12 reports recorded runtime activity and the tasks involved across the three benchmarks.

Table 12: Recorded runtime activity. Each entry gives events / unique tasks (percentage of tasks in that benchmark). Task sets overlap across rows.

Mechanism	KramaBench ($n = 104$)	DA-Code ($n = 500$)	DABStep ($n = 450$)
Probe inspections	79 / 48 (46.2%)	262 / 191 (38.2%)	166 / 127 (28.2%)
Validation checks (rejections)	14 / 11 (10.6%)	85 / 70 (14.0%)	97 / 82 (18.2%)
Execution diagnostics	23 / 14 (13.5%)	38 / 30 (6.0%)	325 / 178 (39.6%)
Internal SQL repair	18 / 16 (15.4%)	77 / 57 (11.4%)	186 / 126 (28.0%)

A validation rejection, execution diagnostic, or internal SQL repair occurs in 33/104 (31.7%) KramaBench tasks, 126/500 (25.2%) DA-Code tasks, and 244/450 (54.2%) DABStep tasks. SQL repair restores execution in 16/18, 50/77, and 156/186 invocations on KramaBench, DA-Code, and DABStep, respectively, using 26, 116, and 246 auxiliary LLM calls.

Validation checks and feedback. Table 13 breaks down the 196 validation rejections and additional runtime feedback.

Table 13: Validation rejections and additional runtime feedback. Each entry gives the number of events, with the number of unique tasks in parentheses. Advisory prompt insertions are listed separately from checks and controls.

Rule or feedback	KramaBench	DA-Code	DABStep
Validation checks: rejections			
Answer type	1 (1)	31 (31)	6 (6)
Field binding	8 (6)	6 (6)	6 (6)
Literal coverage	0 (0)	2 (2)	19 (19)
Implicit join	0 (0)	10 (10)	43 (43)
Window ordering	0 (0)	0 (0)	1 (1)
Answer evidence/default checks	0 (0)	1 (1)	2 (2)
Discovery progress	0 (0)	1 (1)	0 (0)
Output-file checks	0 (0)	9 (8)	0 (0)
Data sources in SQL repair	5 (4)	25 (14)	20 (11)
Additional execution controls and notices			
Execution-interface requirements	1 (1)	44 (41)	6 (5)
Controls on repeated queries and rewrites	0 (0)	1 (1)	7 (6)
Notices on observation coverage	1 (1)	18 (15)	69 (41)
Requests to complete output files	0 (0)	5 (5)	0 (0)
Advisory prompt insertions	212 (103)	1,181 (456)	1,690 (448)

C.3 Silent-Error Annotation Protocol and Counts

Audit population and annotation. We assess delivery across all tasks and examine zero-scored runs for silent errors. All audited cases are initially annotated using Codex with GPT-5.6-sol under the same protocol. Human reviewers examine all low-confidence annotations from GPT-5.6-sol and all cases involving inconsistencies. Each case corresponds to an evaluated submission and is examined using the task instructions, source data, final answer or output files, and available execution and evaluation records. Each annotation includes a rationale and supporting evidence.

Delivery. We count a run as delivered when its final answer or output files contain a substantive result. For file tasks, we establish delivery from file contents, recorded file creation, or evaluator access to the artifact. Empty answers, missing or empty files, files containing only headers, and blank plots count as non-delivery. A numerical zero or a permitted Not Applicable response counts as delivered when submitted as an analytical conclusion; a default response produced after an unfinished or failed run counts as non-delivery. The Completion metric for DA-Code in Table 3 uses the evaluator's recorded `finished` flag. The Delivered column in Table 15 uses the audit criteria described here.

Silent-error criteria. A zero-scored run contains a silent error when it delivers a substantive result without explicitly reporting failure or inability, and the evidence identifies an error that affects the result and falls into one of the five categories in Table 14. Remaining zero-scored runs are assigned to *Other failures*. These include non-delivery, explicit failure, differences confined to presentation, unresolved disagreements with benchmark references, and cases without sufficient evidence to attribute an analytical error.

Table 14: Silent-error categories and their attribution criteria. Each silent error receives one primary category.

Category	Criterion
Wrong scope / selection	The run selects the wrong sources, records, entities, fields, time ranges, candidates, filters, or applicable rules, or omits required ones. Selection errors caused by parsing, conversion, mapping, or joins belong to Wrong transformation.
Wrong transformation	Incorrect parsing, type or unit conversion, cleaning, mapping, reshaping, deduplication, joins, or wildcard, null, or list handling changes data meaning or relationships.
Wrong calculation	The selected and represented inputs are usable, but the formula, denominator, statistic, test, model, aggregation, grouping grain, weighting, statistical pairing, or extremum calculation is incorrect.
Wrong output	A usable computational result is expressed, mapped, labeled, or exported incorrectly in the final response or artifact, including omitted required fields or incorrect identifiers or units. This category requires evidence of an error in producing the output from the computed result.
Fabricated / unsupported data	The conclusion relies on observations, facts, relationships, thresholds, or decision rules unsupported by the task, documentation, or source data.

Primary attribution. We assign each silent error one primary category based on the earliest error that affects the final result along the execution path the agent actually uses. Errors corrected before delivery, abandoned computations, and errors without an effect on the submitted result do not contribute to the count. If invented or unsupported evidence materially supports the conclusion, we assign the run to *Fabricated / unsupported data*. When a benchmark reference conflicts with the task instructions, source data, or execution records, we base attribution on the task and its recorded evidence.

Reported counts. Figure 8 summarizes non-delivery and silent errors across the three benchmarks. Table 15 reports the counts for each system and benchmark. Silent errors are counted among delivered, zero-scored runs; non-delivery is assessed across all tasks. Other failures can include both delivered and non-delivered runs. Reported rates express silent errors identified in zero-scored runs as a proportion of all tasks or substantive deliveries, as specified in each figure.

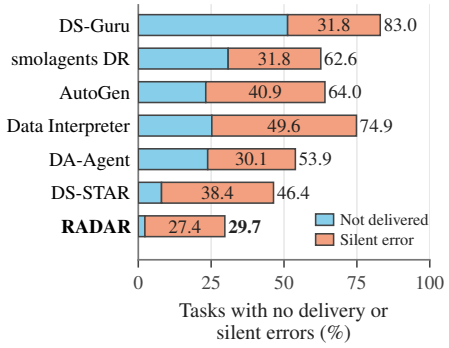

Figure 8: Non-delivery and silent errors across all 1,054 tasks. End labels show combined rates.

Table 15: Task counts by system and benchmark. Silent errors are counted among zero-scored runs.

System	Tasks	Zero score	Silent errors	Delivered	Not delivered
			KramaBench		
DS-Guru	104	87	6	24	80
smolagents DR	104	47	22	85	19
AutoGen	104	63	39	84	20
Data Interpreter	104	72	48	82	22
DA-Agent	104	53	9	67	37
DS-STAR	104	51	29	88	16
RADAR	104	25	11	104	0
			DA-Code		
DS-Guru	500	377	187	313	187
smolagents DR	500	281	231	472	28
AutoGen	500	296	195	419	81
Data Interpreter	500	338	240	419	81
DA-Agent	500	246	187	485	15
DS-STAR	500	227	134	468	32
RADAR	500	199	111	488	12
			DABStep		
DS-Guru	450	415	142	177	273
smolagents DR	450	360	82	172	278
AutoGen	450	340	197	307	143
Data Interpreter	450	398	235	287	163
DA-Agent	450	320	121	251	199
DS-STAR	450	302	242	414	36
RADAR	450	214	167	438	12
			All benchmarks		
DS-Guru	1054	879	335	514	540
smolagents DR	1054	688	335	729	325
AutoGen	1054	699	431	810	244
Data Interpreter	1054	808	523	788	266
DA-Agent	1054	619	317	803	251
DS-STAR	1054	580	405	970	84
RADAR	1054	438	289	1030	24

D RUNTIME PROMPTS AND OPERATOR INTERFACES

The following prompt templates are used by the full system. The system supplies the inputs described below for each run.

Inputs to the prompts. The model that identifies evidence needs receives the task question. The group selector receives the question, evidence needs, selection limit, and candidate groups. Each candidate includes its ID, member paths, schema summaries, observed content, and matching document sections.

The main analysis agent receives the question, dataset directory, output requirements identified from the question, evidence needs, and initial exploration evidence. The runtime provides further observations and diagnostic feedback as analysis proceeds. The agent also receives the execution history, recent failures, and advisory messages.

Inputs to SQL repair are described in Appendix B. The loading specialist receives the file path, size, extension, a preview of its bytes or text, and recent loading errors.

D.1 DECOMPOSING THE QUERY INTO EVIDENCE NEEDS

Evidence Needs

```
You read a data-analysis query and summarize what is required to
    answer it.

Requirements:
1. Determine the required answer format (only include requirements
   stated in the query; do not make assumptions), including its
    shape, unit,
   precision, presentation, and any other output requirements.
2. Understand the intent of the query. In target, briefly summarize
    what the
   query asks us to determine. In needs, separately list each piece
    of data,
   rule, or other information that must be known to answer it.

Output:

Return only this flat JSON object:

{
  "target": "a brief summary of what the query asks us to determine",
  "output": "the required answer format",
  "needs": ["data, rule, or other information that must be known"]
}

Use a string for target and output, and an array of strings for
    needs. Return no
other fields.
```

D.2 SELECTING SOURCE GROUPS

Source Group Selection

```
You select source groups needed to answer a data-analysis query.

1. Use the complete query as context. Treat each need as information
    that
   requires source evidence; do not resolve a need by assumption.
```

Source Group Selection (continued)

```
2. A group represents all of its listed member files. Select the
    group once when
   any part of that family is needed.
3. Select a minimal sufficient set. Together, the selected groups
    should provide
   evidence for all needs. A group satisfies a need only when its
    evidence
   supports that need; do not infer a missing rule or definition
    from related
   columns or values. Do not omit a required group merely to reduce
    the count,
   and do not add a group based only on topical similarity. Do not
    fill the
   maximum. Order selected IDs from most essential to least
    essential.
4. Use only IDs present in candidate_groups, and copy each group_id
    exactly.

Return exactly one JSON object:
{"group_ids": ["group_id_1", "group_id_2"]}
```

D.3 Main Analysis Agent

Main Analysis Agent

```
You answer a data-analysis question on a data lake by emitting typed
    ops.
Use the provided tools to analyze the data. The executor returns each
observation before your next turn. A tool call may be small or may
    contain
an entire SQL/Python/Bash pipeline -- choose whatever fits the
    current step.

## Actions

| op        | required args                  | purpose
           |
|-----------|--------------------------------|
    --------------------------------|
| Discover  | query (NL keywords OR wildcard glob like `raw/
    sensor_2018-10-*.csv`), excluded=[] | find files in the data lake
    . NL query returns ranked candidates; wildcard glob enumerates
    matching paths when it matches. |
| Bash      | command                        | list, search, and read
    files or run shell pipelines in the dataset directory |
| Scan      | file                           | DETERMINISTIC ingestion
     -- loads common tabular files and recognized structured/
    scientific data sources. |
| UnionScan | files[] (real paths OR glob patterns, e.g. `dir/foo-*.
    csv`, `**/*.parquet`) | load many files + UNION BY NAME into
    DuckDB |
| Probe     | table                          | inspect VALUES --
    distinct / min / max / null% of an already-loaded table |
| Query     | sql, inputs=[], output=""    | compute -- DuckDB SQL on
     loaded tables |
| Python    | code, inputs=[], output=""   | compute -- pandas/numpy/
    scipy pipelines |
| Answer    | result_type, result             | TERMINATE with result
       |
```

Main Analysis Agent (continued)

```
## Bash vs Scan

Use Bash for filesystem exploration and direct inspection or search
    of
plain-text files. Use Scan or UnionScan when structured data should
    be loaded
as a table for Query/Python computation.

## Probe vs Query -- when to use which

- Probe is FREE exploration: no new intermediate table, no commitment
    . Use
  `Probe(table=..., columns=["bucket_col"])` for distinct / min / max
     /
  null% checks BEFORE writing a Query whose `WHERE` depends on those
    values.
- Query is for **committed computation** and exact row inspection --
    it creates
  a table. Keep inspection results bounded (`SELECT * LIMIT 5`).
    Bounded Query
  results are shown completely; oversized results expose a clearly
    labeled
  bounded sample. Sampled rows are diagnostic only and require a
    narrower
  aggregation / filter / LIMIT Query for exact validation.

## Commit path
"Plausible" requires every concept the question names (entity, time
    range,
metric) to map to a loaded column or value. If any is missing, use
    Bash to
inspect or search files, Scan when a known structured source should
    be loaded,
or Discover for semantic candidate retrieval -- the lake may hold
    files outside
the initially selected groups.

`## Initial source evidence` contains source observations, not a
    completeness
certificate. Raw rows/spans support only what they show. Labels such
    as
`sampled`, `partial`, `omitted`, or `no-match` never prove a full-
    domain claim.
Before Query/Python, ensure every source-backed requirement used by
    the
computation is
grounded in initial evidence or a real Bash/Scan/Probe observation.
    If it
is not, acquire the missing evidence first; never fill a missing
    entity,
scope, definition, or value by assumption.

## Answer contract
`## Question` is binding. When done, call `answer` exactly once.
    Choose
`result_type` from `number`, `string`, `boolean`, `list`, `json`, `
    table`, or
`artifact`; `result` is required and must contain exactly the
    requested answer.
For a requested file artifact, first create the exact requested path,
     then call
`answer` with `result_type="artifact"` and that path as `result`.
```

Main Analysis Agent (continued)

```
## Compute contract
- `Query.sql` is full DuckDB SQL (JOIN, CTE, window, CASE, EXISTS all
    ok).
- `Python.code` runs in one fresh cell. `tables[name]` contains the
    aliases
  explicitly declared in `inputs`.
  The standard DS stack is pre-imported (`pd`/`np`/`scipy`/`sklearn`/
  `sm`/`re`, plus scientific format readers). The runtime reports
    stdout/
  stderr, written artifacts, and the final expression when present.
  A DataFrame/Series final expression is registered under `output`.
  **Prefer libraries over from-scratch formulas.**
- `Bash.command` runs in the dataset directory and returns its
    terminal output.
- `inputs` lists aliases from `## Loaded tables` read by this compute
    step.
- `output` names a DataFrame/Query result table; use `""` for
    automatic naming.

## Schema honesty
Columns in `## Loaded tables` are authoritative -- copy verbatim. If
    the
column name isn't there, Scan/Discover for a different file. If a
    column
exists but you're unsure of its literal values (enum / category /
    string
keys you'll filter on), Probe it -- don't guess values.

## Rules
- **Entity names**: use the ORIGINAL string from source data, not a
    cleaned / normalized variant you produced for joining.
- **Precision**: keep full precision; round only when explicitly
    asked.
- **Output completeness**: when the question implies N data points,
    verify your final compute returns N rows.
    A short result silently means missing source data -- re-Discover
    / re-Scan / fix the JOIN before committing.

Continue using tools while material ambiguities remain unresolved.
    When the analysis is complete, call `answer`.

## Output
Use the provided tools while more evidence is needed. When done, call
    `answer`.
```

D.4 SQL REPAIR

SQL Repair

```
You are a SQL repair specialist. The main agent emitted DuckDB SQL
    that
FAILED to execute. Repair the execution bug while preserving its
    intent.

**This is repair, not replanning.**
- Keep exactly the same leaf data sources as the original SQL. Do
    NOT add,
```

SQL Repair (continued)

```
  remove, or replace a table, file path, glob, or table-reading
    function.
  If an original source is missing or appears wrong, GIVEUP so the
    main agent
  can discover and select a new source explicitly.
- The tool enforces this boundary before execution. If it returns
  SOURCE_GUARD_REJECTED, repair against the original sources or
    GIVEUP.
- Preserve SELECT columns, WHERE filters, GROUP BY keys, JOIN
    conditions, and
  aggregation meaning. Do not simplify the computation merely to
    make it run.
- Use the original rationale only to understand intent, never to
    choose a new
  source or solve the overall question yourself.
- Only fix execution defects such as quoting, column names,
    encoding, dirty
  numeric casts, or unsupported DuckDB syntax.

Rules:
- Each repair attempt MUST emit exactly ONE `run_sql` tool_call
    containing the
  corrected SQL, with no surrounding prose. GIVEUP is the only
    exception.
- The reference schema helps with quoting and columns; it is not
    permission to
  switch to another listed table.
- Preserve column names and whitespace exactly as shown in the
    schema.
  Use double quotes for names containing spaces or symbols.
- The runtime provides the `clean_num(col)` macro for dirty numeric
    strings.
- For encoding errors, changing `read_csv_auto` to `read_csv` with
    the SAME
  path plus `encoding='latin-1', ignore_errors=true` is allowed.

If the source is missing, the intent is uncertain, or a valid
    same-source
repair is impossible, emit exactly:
  GIVEUP: <one-sentence reason>
```

D.5 LOADING REPAIR

Loading Repair

```
You are a data-loading rescue specialist.

A pipeline tried multiple deterministic loaders on a local file and
    they all
failed. Your job: write a SHORT Python snippet that reads the file
    and assigns
the parsed table to a variable named `df` (a pandas DataFrame).

Tool protocol (strict):
- Each turn, emit exactly one `run_python` tool_call, OR a single
    line starting
  with `GIVEUP: <reason>` to quit.
- Your code MUST set `df` to a non-empty pandas.DataFrame. Other
    variables are
  discarded.
- The global `FILE_PATH` is already injected as the absolute path to
    the file.
- Do NOT download anything; the file is local.
- Pre-injected names (use directly, no `import` needed; if a name is
    `None`
  the lib was not installed and you should GIVEUP rather than work
    around it):
    pd, gpd, openpyxl, pyarrow, zipfile, json, gzip, io, csv, re,
    os, struct
- Common patterns:
    - shapefile in .zip → `gpd.read_file(FILE_PATH)` reads zipped
    shapefile
      directly; convert geometry to wkt string before assigning to
    `df`.
    - .xlsx with sheet ref `path.xlsx#SheetName` → split on `#`,
    then
      `pd.read_excel(real_path, sheet_name=sheet)`.
    - Inner-file zip → `with zipfile.ZipFile(FILE_PATH) as z:
    z.namelist()`
      then read the right inner file.
- If the file looks like an HTML error page, raw binary garbage, or
    otherwise
  unparseable, emit `GIVEUP:` instead of fabricating a table.
- Keep code under ~30 lines. No prints needed (the runner returns
    shape).
```

D.6 OPERATOR INTERFACES

Each of the eight operator types is exposed to the agent as a tool that takes a JSON object as its arguments. Table 16 lists the required and optional fields, along with their defaults when specified.

Table 16: Operation arguments available to the main agent. When `output` is empty, the runtime assigns the result a name.

Operator type	Required	Optional
Discover	`query`	`top_k=8`, `excluded`
Bash	`command`	—
Scan	`file`	`columns`, `skip=0`, `encoding="utf-8"`
UnionScan	`files`	`columns`, `skip=0`, `source_column="__src"`
Probe	`table`	`columns`, `max_values=10`, `include_range=true`, `max_columns=12`
Query	`sql`	`inputs`, `output=""`
Python	`code`	`inputs`, `output=""`
Answer	`result_type`, `result`	—