DAVBENCH: BENCHMARKING AGENTIC VERIFICATION OF REAL-WORLD DATA WORK

Anonymous authors
Paper under double-blind review

ABSTRACT

Data work addresses user-specified data needs, spanning data engineering (DE), which builds and transforms datasets and data pipelines, and data analysis (DA), which derives metrics and analytical results. Reliable verification is critical for preventing faulty transformations and flawed analytical conclusions before candidate data work is delivered. Yet existing benchmarks largely evaluate agents' ability to complete data tasks, while overlooking their ability to verify whether candidate data work satisfies user requirements and is free of defects. We introduce **DAVBench** (Data Agentic Verification Benchmark), comprising **216 repository-level verification tasks of data work** with reproduced real-world defects across data engineering pipelines, analytical workflows, and delivered artifacts. Tasks involve connecting evidence across data, code, and domain documentation, tracing defects through multi-stage workflows, and distinguishing faulty from correct candidates. We evaluate verification along two dimensions: ***(1) verification quality***, which measures the accuracy and completeness of verification reports using location matching and evidence-grounded semantic judging; and ***(2) repair utility***, which measures how effectively these reports help a downstream model correct faulty candidate work under behavioral tests. Our empirical results show that current agents still struggle to verify real-world data work reliably. Even the flagship model GPT-6 Astra achieves a finding F1 of only **49.82%** and complete defect coverage on just **39.61%** of faulty tasks. Further analysis reveals that verification quality is a key bottleneck for correcting data work. When given reference verification reports, a downstream repair model successfully repairs **69.48%** of tasks, compared with **24.03%** without reports and only **37.66%** with reports from the best-performing verifier. Overall, DAVBench provides a testbed for improving the verification of real-world data work.

1 INTRODUCTION

Data work addresses user-specified data needs, spanning data engineering (DE), which builds and transforms datasets and data pipelines, and data analysis (DA), which derives metrics and analytical results (Lin et al., 2023; Yan & He, 2020; Jing et al., 2025). Errors in data workflows and artifacts can propagate through downstream datasets and analyses, leading to faulty transformations and flawed analytical conclusions (Song & He, 2021; Tu et al., 2023; Karlas et al., 2025). Reliable verification of such data work is therefore essential (Song & He, 2021). However, verification often demands substantial time and domain expertise, making specialized human review difficult to scale (Song & He, 2021; Tu et al., 2023). This challenge is becoming more pronounced as advances in LLMs and coding agents increasingly enable AI assistance in data engineering and analysis (DeepSeek-AI, 2026; Yao et al., 2023; Jing et al., 2025; Lei et al., 2025; 2026) and can significantly accelerate the completion of data-related tasks (Merali, 2025). The resulting high volume of candidate data work therefore makes manual verification a bottleneck.

However, verification of real-world data work is complicated by the need to jointly reason over delivered artifacts, workflow logic, data semantics, and project context. First, faulty data workflows can execute successfully while containing silent logical errors that yield incorrect results without obvious failures, making the underlying errors difficult to detect (Qiu et al., 2026). Second, verification evidence is dispersed across large, heterogeneous project workspaces (Lei et al., 2025; 2026). Verifiers may need to cross-check domain requirements and the data semantics against what the

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.

code actually computes. Third, defects can be hidden within multi-stage data workflows. An error in the final artifact may originate several steps earlier, requiring verifiers to trace intermediate computations and data dependencies (Karlas et al., 2025; Hasan et al., 2026). Effective verification often involves planning an investigation, formulating hypotheses, and selecting checks that distinguish competing explanations. For example, in the tourism-analysis task in Figure 1, the candidate workflow executes successfully and produces a seemingly plausible guest-count table. A verifier can compare the delivered counts with the source statistics and documented units to assess whether the candidate satisfies the requirement. This exposes an inconsistency that can then be traced to the transformation code, where values already expressed in guests are multiplied by 1,000 again. Yet current data-agent benchmarks largely position agents as task solvers, evaluating the completion of data engineering and analysis tasks (Lei et al., 2026; Jing et al., 2025) or the debugging of SQL (Li et al., 2025), rather than the independent verification of existing candidate data work. DAVBench targets this gap by evaluating whether agents can reliably verify real-world candidate data work.

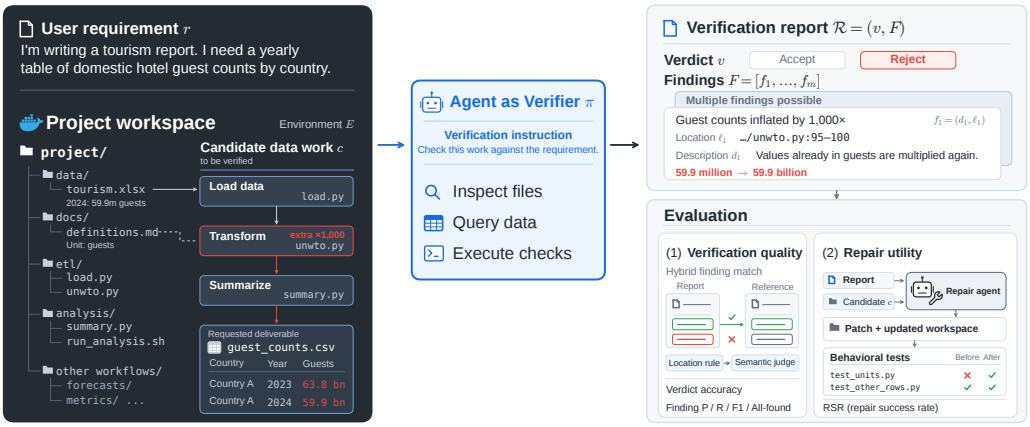


Figure 1: Overview of a DAVBench verification task. The verifier assesses candidate data work against the user requirement by tracing evidence across code, data, and documentation, with evaluation spanning verification quality and downstream repair utility.

To address this gap, we introduce **DAVBench** (Data Agentic Verification Benchmark), a benchmark for evaluating agents' ability to verify real-world data work. DAVBench comprises **216 repository-level verification tasks** spanning data engineering and analysis, including cases with reproduced real-world defects in data workflows and delivered artifacts. Each task pairs a user requirement with candidate data work in a project workspace. Drawing on documented review feedback and subsequent fixes, we reconstruct candidate data work from real projects and establish reference verdicts and localized defect annotations (Figure 2). The reconstructed workspaces retain domain data, workflow code, and documentation of business rules and analytical methods as evidence for assessing whether the candidate satisfies its requirement. These workspaces average **952 code files** and **47.6 data files**, requiring verifiers to identify and connect relevant evidence across the project context. DAVBench also includes correct candidates, allowing us to evaluate whether verifiers can detect real defects while avoiding false alarms.

Our evaluation examines both the quality of verification reports and their practical value for downstream repair. **(1) Verification Quality** measures whether agents make correct acceptance decisions and identify defects accurately and completely. Verdicts are scored against reference accept/reject labels. For findings, we combine rule-based location matching with evidence-grounded LLM judging of the reported diagnosis. **(2) Repair Utility** measures how effectively verification reports help a fixed downstream repair agent correct faulty candidate work. Behavioral tests grounded in project tests and documented fixes assess whether defects are corrected (fail-to-pass, F2P) while previously correct behavior is preserved (pass-to-pass, P2P).

Experiments with eight verifier models show that current agents still struggle to verify real-world data work reliably. Even the flagship model GPT-6 Astra achieves a finding F1 of only **49.82%** and complete defect coverage on just **39.61%** of faulty tasks. Performance is particularly weak on

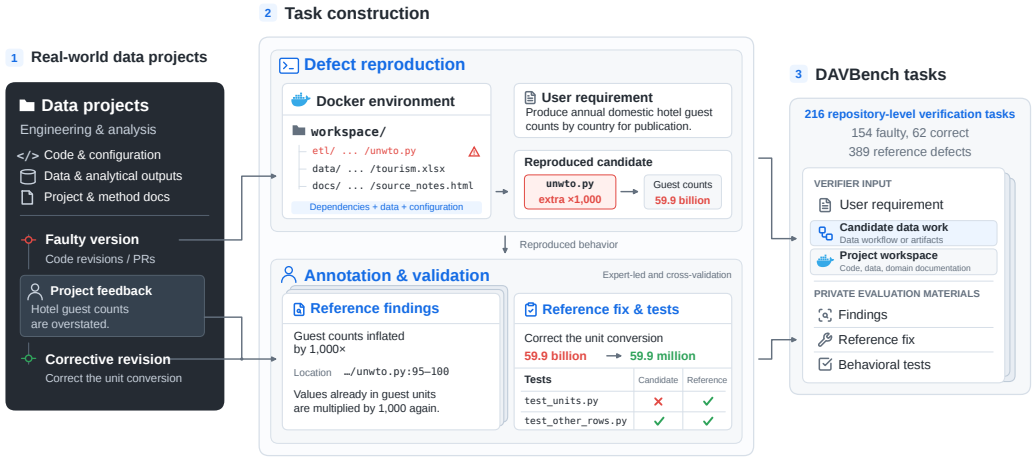


Figure 2: Construction of DAVBench. Defects documented in real-world data projects are reproduced in executable Docker workspaces and paired with user requirements. Expert-led annotation uses project feedback, subsequent fixes, and reproduced behavior to establish reference findings. Behavioral tests check the reproduced errors and their corrections. Validated corrected versions provide correct candidates.

data transformation and analytical modeling tasks, while complete verification becomes substantially harder as the number of defects increases, with the mean All-found Rate across models falling to just **0.4%** on tasks containing four or more reference defects. Repair experiments further indicate that verification quality is a key bottleneck for correcting data work. A downstream repair model succeeds on **69.48%** of tasks with reference verification reports, compared with **24.03%** without reports and only **37.66%** with reports from the best-performing verifier. Assigning GPT-5.6-Sol to verification and Qwen 3.8-27B to repair also outperforms the reverse assignment, further highlighting the importance of strong verification. Our hybrid scorer agrees with human judgments on **97.55%** of finding–reference pairs, while comparisons across four LLM judges yield consistent verifier rankings. Overall, DAVBench provides a testbed for improving the verification of real-world data work and its utility for downstream correction.

2 Benchmark Construction

Table 1 compares DAVBench with data-work completion, debugging, and verification benchmarks.

2.1 Task Formulation

Each task in DAVBench pairs a user requirement r with candidate data work c produced to meet it (e.g., a data pipeline, analysis script, or output artifact). The requirement specifies the intended data-work outcome, and the verifier is asked to determine whether the candidate satisfies it. Candidates may be faulty or correct. Each candidate is provided in an executable environment E containing a repository-level project workspace, where related code, data, and domain documentation provide evidence for verification (Figure 1). Given (r, c, E), the verifier π assesses whether the candidate should be accepted and returns a verification report $\mathcal{R}$. When the verifier returns Reject, the report describes and localizes the identified defects.

$$\mathcal{R} = \pi(r, c, E).$$

The report consists of a verdict and a list of findings.

$$\mathcal{R} = (v, F), \qquad v \in \{\text{Accept}, \text{Reject}\},$$
$$F = [f_1, \ldots, f_m], \qquad f_i = (d_i, \ell_i),$$

where v gives the acceptance decision. Since a candidate may contain multiple defects, F is a list of m findings, each with a defect description d_i and location ℓ_i. The list is empty when no defects are reported ($m = 0$).

Table 1: Comparison with related benchmarks. DE: data engineering; DA: data analysis; DV: data visualization; SE: software engineering; recon.: reconstructed. Task counts are dataset totals; Repo indicates repository-level context, with Partial applying to a subset of tasks.

Benchmark	# Tasks	Domain	Objective	Source	Repo	Input	Output	Evaluation
Data-work solving								
BIRD (Li et al., 2023)	12,751	SQL	Generation	Manual	×	Question + DB	SQL	Execution
Spider 2.0 (Lei et al., 2025)	632	SQL	Completion	Adapted	Partial	Req. + DB	Answers / DBs	Execution
BIRD-CRITIC (Li et al., 2025)	1,100	SQL	Debugging	Real/reproduced	×	Issue + SQL + DB	SQL fixes	Tests
ELT-Bench (Jin et al., 2025)	100	DE	Pipeline completion	Constructed	✓	Spec. + codebase	Pipelines/models	Execution
DAComp (Lei et al., 2026)	210	DE / DA	Completion	Manual	Partial	Req. + workspace	Code/reports	Execution + judge
DV-World (Meng et al., 2026)	260	DV	Completion / Fix	Real + Manual	×	Task + data	Visual artifacts	Hybrid
Review and verification								
AACR-Bench (Zhang et al., 2026)	200	SE	Code review	PRs/augmented	✓	PR + repo	Comments	Review quality
SWR-Bench (Zeng et al., 2025)	1,000	SE	Code review	Real PRs	✓	PR + repo	Comments	Review quality
ClaimDB (Theologitis et al., 2026)	53,368	Data facts	Fact verification	Generated	×	Claim + DB	Labels	Label accuracy
Data-work verification								
DAVBench	**216**	**DE / DA**	**Verification**	**Real + recon.**	✓	**Req. + candidate + workspace**	**Verdict + findings**	**Location + judge Repair tests**

2.2 Evaluation Protocol and Metrics

We evaluate both the quality of verification and the utility of the resulting report $\mathcal{R}$ for a downstream repair agent.

(1) Verification Quality. We assess the correctness of both verdicts and findings against the reference annotations. Verdict accuracy measures agreement with the reference accept/reject decision. For findings, our hybrid protocol accommodates reasonable localization ambiguity while requiring semantic agreement. It first checks whether the reported line falls within an admissible reference range, then uses an evidence-grounded LLM judge to determine whether the reported and reference findings identify the same underlying defect. We report task-averaged finding precision, recall, and F1 over faulty candidates, along with *All-found Rate (%)*, the proportion of faulty tasks for which every reference defect is identified.

(2) Repair Utility. Verification is also valuable for guiding the correction of faulty data work. We therefore compare downstream repair outcomes when reports $\mathcal{R}$ from different verifiers are provided to a fixed repair agent starting from the same candidate. The resulting repairs are evaluated using behavioral tests: fail-to-pass (F2P) tests check whether errors are corrected, while pass-to-pass (P2P) tests check whether previously correct behavior is preserved. We report *Repair Success Rate* (RSR, %), the percentage of tasks whose repairs pass all evaluation tests. Details of both evaluations, including the judging protocol and scoring formulas, are provided in Appendix B.1.

2.3 Task Construction

DAVBench is constructed through an expert-led pipeline that curates real-world defects, reconstructs executable candidate data work, establishes evidence-grounded reference findings, and independently validates each task (Figure 2).

Real-World Defect Curation. To capture defects encountered in real data work, we screen over 5,000 pull requests (PRs) from open-source data engineering and analysis projects. These PRs provide documented evidence of defects, with review feedback identifying problems and subsequent fixes providing additional evidence for their causes and corrections. We apply a multi-stage filtering process covering *data-work relevance*, *defect significance*, *supporting evidence*, and *reconstruction and verification feasibility*. Detailed selection criteria are provided in Appendix E.1. This process yields a curated source pool of over 250 PRs from 47 DE repositories covering data integration and transformation and 103 DA repositories covering business analytics and scientific data analysis.

Reconstruction of Candidate Data Work. To construct realistic verification tasks, we reconstruct candidate data work in three steps. *(1) Candidate and Environment Setup:* We select a project state in which the documented defects are still present and reconstruct its repository-level project workspace in Docker with the required runtimes, data-processing frameworks, and database services. *(2) Defect Reproduction and Validation:* Using project data, downloaded public datasets, or adapted inputs, we reproduce the documented faulty behavior in the reconstructed candidate and retain generated outputs, such as tables or charts, when they form part of the candidate under verifi-

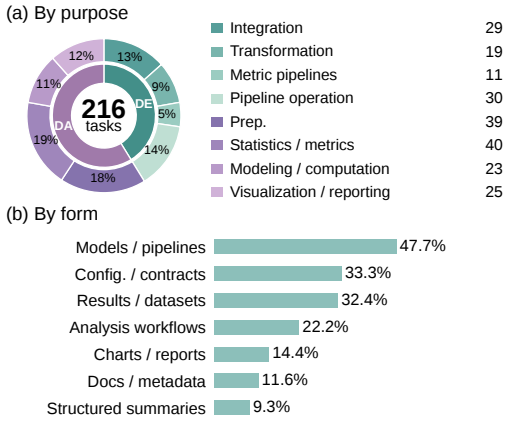


Figure 3: Distribution of DAVBench tasks by primary data-work purpose and candidate form. Each task has one primary purpose; candidate forms may overlap.

Table 2: Key statistics of DAVBench.

Group	Metric	Value
Scope	# Tasks (DE / DA)	216 (89 / 127)
	# Repositories	102
	# Reference defects	389
	# Correct candidates	62 (28.7%)
	# Requirement words	26.2
Workspace	# Code files	952
	# Code lines (mean / med.)	97.1k / 7.1k
	# Data files	47.6
	Data / docs	97.2% / 82.4%
	DB coverage	35.6%
Verification	# Defects / faulty task	2.53
	Multi-defect tasks	66.2%
	# Behavioral tests	4.65
Languages	Python	44.0%
	SQL	32.9%
	R	11.1%

cation. We derive reference corrections from documented fixes and construct behavioral tests (F2P and P2P) from project tests and defect evidence to validate both faulty and corrected behavior. *(3) Verification Task Specification:* Each task pairs a user requirement derived from project materials with neutral instructions asking whether the candidate satisfies that requirement and what defects, if any, it contains. Further reconstruction details, including measures to mitigate contamination and answer leakage, are provided in Appendix E.2.

Evidence-Grounded Defect Annotation. To establish reliable reference findings, we ground each defect in documented project feedback, subsequent fixes, and reproduced behavior. Annotators record each reference defect as $f_j^* = (d_j^*, \mathcal{L}_j^*)$, with a diagnosis d_j^* and one or more admissible location ranges $\mathcal{L}_j^*$. Multiple ranges accommodate reasonable localization ambiguity. For each defect, annotators also maintain an internal evidence graph linking the relevant data, code, documentation, and reproduced behavior that support its independent rediscovery in the reconstructed environment. Validated corrected versions provide correct candidates.

Validation and Iterative Refinement. After annotation, DAVBench tasks undergo independent verification and review. Task authors first perform execution and semantic checks against a quality checklist (Appendix E.4). Two additional annotators then independently complete each task and review it against the same checklist, including red-teaming of the behavioral tests. Disagreements trigger revisions to the task, reference findings, or tests followed by repeated checks, with unresolved cases escalated to the expert team.

2.4 Dataset Statistics

Table 2 and Figure 3 summarize DAVBench's 216 tasks (89 DE and 127 DA) from 102 repositories. The benchmark has four notable characteristics. *(1) Diverse Data Work.* Tasks span nine primary purposes across DE and DA and multiple overlapping candidate forms, including pipelines, datasets, analytical workflows, and reports. *(2) Diverse Real-World Defects.* The 389 reference defects span eight categories, including computation, record selection, data contracts, execution dependencies, and artifact correctness. *(3) Large, Heterogeneous Workspaces.* Task workspaces contain substantial project context, averaging 952 code files, 97.1k lines of code, and 47.6 data files, together with data and domain documentation needed to assess candidate behavior. *(4) Multiple Defects and Correct Candidates.* Among faulty candidates, 66.2% contain multiple reference defects, while 62 correct candidates test whether verifiers can avoid false alarms. To our knowledge, DAVBench is the first repository-level benchmark for verifying real-world data work across DE and DA, providing a challenging testbed with diverse reconstructed real-world defects, heterogeneous workspaces, and both faulty and correct candidates. Appendix F provides additional dataset statistics and category definitions.

Table 3: Main results on DAVBench. Finding P/R/F1 and All-found are computed on 154 faulty candidates; Reject and Accept denote verdict accuracy on 154 faulty and 62 correct candidates, respectively. FP/task counts false findings per correct candidate. Repair utility is evaluated on the 154 faulty candidates using GPT-5.6-Sol as the fixed repair agent; Δ is the RSR change from the no-report condition. Scores are percentages except FP/task; best verifier results are bold.

	Faulty candidates ($n = 154$)					**Correct** ($n = 62$)		**Repair** ($n = 154$)	
Verifier	Reject↑	P↑	R↑	F1↑	All-found↑	Accept↑	FP/task↓	RSR↑	Δ (pp)↑
GPT-6 Astra	**94.16**	**49.94**	**60.60**	**49.82**	**39.61**	80.65	0.26	**37.66**	**+13.64**
GPT-5.6-Sol	**94.16**	44.96	46.11	41.86	24.68	59.68	0.53	28.57	+4.55
Gemini 3.8 Flash	70.78	38.98	38.85	36.08	24.03	**82.26**	0.15	28.57	+4.55
DeepSeek V4 Flash	79.22	30.68	33.46	29.22	20.13	61.29	0.42	25.32	+1.30
GLM 5.3 Flash	67.53	22.81	35.12	24.94	22.73	51.61	0.52	24.68	+0.65
GLM 5.3	79.87	26.95	44.07	29.85	25.97	58.06	0.50	29.22	+5.19
Qwen 3.8 Max	74.68	27.24	36.44	27.68	22.08	**82.26**	0.19	22.08	-1.95
Qwen 3.8-27B	46.75	20.42	21.02	18.07	12.99	64.52	**0.08**	18.83	-5.19
No report	–	–	–	–	–	–	–	24.03	0.00
Reference report	–	–	–	–	–	–	–	69.48	+45.45

3 Experiments

3.1 Experimental Setup

We evaluate eight LLMs, including flagship proprietary and open-weight models: GPT-6 Astra, GPT-5.6-Sol, Gemini 3.8 Flash, DeepSeek V4 Flash, GLM 5.3 Flash, GLM 5.3, Qwen 3.8 Max, and Qwen 3.8-27B. We use mini-SWE-agent (Yang et al., 2024) as a lightweight bash-only agent scaffold following recent agent benchmarks (Yang et al., 2026; Android Bench, 2026), which helps reduce confounding from harness-specific features. Agents operate through a terminal in pre-built task-specific Docker environments containing the reconstructed project workspace, required dependencies, and runtime services. All models receive the same runtime budget for each task to ensure fair comparison. For repair utility, GPT-5.6-Sol serves as the primary fixed repair agent. We also conduct three-run stability checks for verification and repair runs, with standard deviations below 2 percentage points in Finding F1 and RSR, respectively (Appendix A). Full experimental settings are provided in Appendix A.

3.2 Main Results

Table 3 summarizes overall verification quality and downstream repair utility.

Current agents still struggle to verify real-world data work. Across eight models, finding F1 ranges from 18.07% to 49.82%. Even the strongest verifier, GPT-6 Astra, reaches only 49.82% finding F1, leaving substantial room for improvement in accurately and completely identifying defects.

Detecting faulty work does not imply complete verification. GPT-6 Astra and GPT-5.6-Sol both correctly reject 145 of the 154 faulty candidates (94.16%). Yet Astra achieves complete defect coverage on only 61 tasks (39.61%), and Sol on 38 (24.68%). Thus, models can often recognize that a candidate is faulty while still leaving some defects unidentified.

Defect detection can come at the cost of false alarms. Gemini 3.8 Flash and Qwen 3.8 Max achieve the highest accuracy on correct candidates, both at 82.26%, whereas GPT-5.6-Sol reaches 59.68% despite having the second-highest finding F1. Sol also reports more false findings per correct candidate (0.53) than Gemini (0.15) or Qwen Max (0.19). Reliable verification therefore requires both finding real defects and recognizing when candidate data work is correct.

Verification quality is a key bottleneck for downstream repair. The fixed repair agent succeeds on 24.03% of faulty tasks without a report, compared with 69.48% when given reference verification reports. Reports from current verifiers yield repair success rates of 18.83–37.66%, substantially



Figure 4: Performance across tasks and defects. (a) Finding F1 and repair sucess rate (RSR) by DE/DA. (b) Finding F1 by data-work type; (c) Recall by defect type; The "Top 5" rows in (b,c) average the top five verifiers. (d) All-found Rate (solid) and finding recall (dashed) by the number of reference defects per task; n denotes tasks.

below the reference-report condition and, for some verifiers, even below the no-report baseline. This gap indicates substantial headroom from more accurate and complete verification reports.

3.3 Verification Performance Analysis

To understand where current verifiers fall short, we analyze performance across DE and DA, data-work types, defect mechanisms, and the number of defects per task, and further examine common failure patterns behind missed defects. Verification performance does not show a consistent DE–DA ordering across models; detailed domain-level results are provided in Appendix C.1. We therefore focus on finer-grained variation across data-work types and defect characteristics. Category definitions and examples are provided in Appendix F.

Data transformation and analytical modeling are particularly difficult to verify. Figure 4(b) breaks finding F1 down by the primary purpose of the candidate data work. Among the fixed Top 5 verifiers, DE finding F1 averages 28.6% for data transformation and 40.3% for metric pipelines, compared with 52.3% for pipeline operation. In DA, modeling and computation averages 32.2% F1, below statistics and metrics at 40.9%. These gaps suggest that checking computations against data semantics, business definitions, and analytical assumptions remains challenging.

Errors in delivered content and workflow state are frequently missed. Figure 4(c) compares recall across reference-defect categories. Among the Top 5 verifiers, recall averages 31.1% for errors affecting delivered content and 28.2% for errors in workflow state, compared with 49.5% for computation errors. These results suggest that checking deliverables against their requirements and tracking state across workflow stages remain difficult for current verifiers.

Complete defect coverage deteriorates rapidly as defects accumulate. Figure 4(d) shows a widening gap between finding individual defects and identifying every defect in a task. For tasks with four or more reference defects, mean finding recall remains 29.8%, whereas mean All-found Rate falls to only 0.4%. Models can therefore still identify some problems in multi-defect tasks, but rarely produce complete verification reports.

Verification failures extend beyond evidence access. Across 411 reviewed missed defects from four verifiers, related project materials were surfaced without a diagnosis in 50.1% of cases, checks omitted critical conditions in 22.9%, and key evidence was missing in 11.9%. DE misses more often involve unchecked transformation conditions, whereas DA misses more often reflect invalid criteria or misinterpretation, suggesting that check selection and evidence-to-diagnosis synthesis are major bottlenecks beyond evidence access. Further analysis of failure patterns is provided in Appendix D.

3.4 Repair Utility Analysis

To examine whether report utility depends on the repair model, we additionally evaluate Qwen 3.8-27B using the same verifier reports on all 154 faulty tasks.

Verifier report effects transfer across repair agents. Relative to each repair agent's no-report baseline, seven of the eight verifiers change RSR in the same direction, with six improving RSR for both repair agents. Verifier rankings under GPT-5.6-Sol and Qwen 3.8-27B are also strongly correlated (Spearman's $\rho = 0.81$). These results indicate that verifier comparisons are not tied to the primary repair agent. They also support using a locally deployable model such as Qwen 3.8-27B for lower-cost repair-utility evaluation.

Table 4: Repair success rates (%) with two repair agents ($n = 154$).

Verification report	GPT-5.6-Sol	Qwen 3.8-27B
GPT-6 Astra	**37.66**	**33.12**
GPT-5.6-Sol	28.57	27.92
Gemini 3.8 Flash	28.57	28.57
DeepSeek V4 Flash	25.32	24.68
GLM 5.3 Flash	24.68	24.03
GLM 5.3	29.22	24.03
Qwen 3.8 Max	22.08	22.08
Qwen 3.8-27B	18.83	17.53
No report	24.03	18.18
Reference report	69.48	61.69
Rank Corr. (ρ)	1.00	0.81

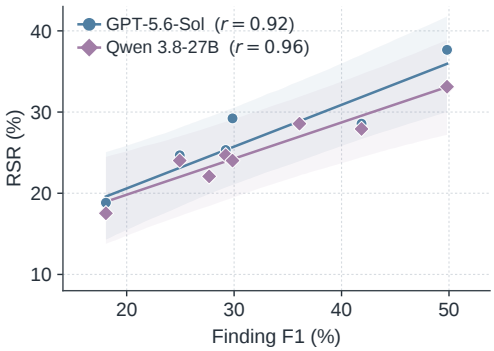


Figure 5: Finding F1 and RSR with 95% task-bootstrap intervals.

Stronger verification can matter more than the choice of repair model. Repair outcomes are often similar when the same verification report is paired with different repair agents. For six of the eight verifier reports, RSR differs by at most 1.30 percentage points between GPT-5.6-Sol and Qwen 3.8-27B, with identical outcomes for Gemini 3.8 Flash and Qwen 3.8 Max. A direct role swap further illustrates this effect: using GPT-5.6-Sol for verification and Qwen 3.8-27B for repair achieves 27.92% RSR, whereas reversing their roles yields only 18.83%. This comparison suggests that improving verification can have greater downstream value than upgrading the repair model alone.

Verification quality provides an efficient signal for downstream repair utility. Figure 5 compares finding F1 with downstream RSR across the eight verifier reports. Finding F1 is strongly correlated with RSR for both GPT-5.6-Sol ($r = 0.92$) and Qwen 3.8-27B ($r = 0.96$) as repair agents. This relationship makes verification quality useful for screening and iterating on verifier variants before running more costly downstream repair experiments. Repair-based evaluation remains complementary, as it directly tests whether verification feedback leads to successful corrections without breaking previously correct behavior. Appendix C.4 provides additional repair metrics and robustness analyses.

3.5 Evaluation Reliability

Hybrid scoring closely matches human judgments. To validate the scoring procedure, three annotators independently assess 245 finding–reference pairs from 42 verification reports, reaching strong agreement on the combined semantic-and-location labels (Fleiss' $\kappa = 0.966$). Against the majority human labels, the full hybrid scorer achieves 97.55% agreement and 93.48% pair-level F1 (Table 5). The two scoring stages contribute complementary checks: location matching retains 44 of 46 unanimously confirmed valid matches (95.65%), while the semantic judge agrees with human labels on 98.00% of the 50 location-qualified pairs and rejects cases that point to compatible locations but diagnose different defects. Together, these results support the use of hybrid matching for evaluating verification findings.

Verifier comparisons remain stable across semantic judges. For this analysis, we rescore the same reports from six verifiers on a randomly sampled 130-task subset using four different semantic judges, while keeping the reference annotations, location matching, and aggregation procedure fixed. Across judges, verifier F1 changes by at most 1.78 percentage points, and rankings under the alternative judges have Spearman correlations of 0.94–1.00 with those under DeepSeek V4 Flash (Table 6). The resulting model comparisons are therefore insensitive to the choice among the evaluated semantic judges. Appendix B.2 provides the human-annotation protocol, stage-level scoring results, and additional judge-specific details.

Table 5: Agreement with human labels (%) on finding–reference pairs.

	Semantic stage ($n = 50$)	Full scorer ($n = 245$)	
Judge	Agree.	Agree.	Pair F1
DeepSeek V4 Flash	98.00	97.55	93.48
GPT-5.6-Sol	100.00	97.96	94.62
GPT-6 Astra	100.00	97.96	94.62
Gemini 3.5 Flash	94.00	96.73	91.11

Table 6: Finding F1 (%) under alternative semantic judges.

	Judge			
Verifier	DeepSeek	GPT-5.6	GPT-6	Gemini
GPT-5.6-Sol	44.69	45.03	45.03	45.03
Gemini 3.8 Flash	37.81	39.21	39.21	39.59
DeepSeek V4 Flash	30.27	30.46	30.68	29.91
GLM 5.3 Flash	26.20	26.46	26.46	26.46
GLM 5.3	32.56	32.72	32.72	31.30
Qwen 3.8 Max	29.30	30.97	30.59	30.29
Rank Corr. (ρ)	1.00	0.94	1.00	0.94

4 Related Work

Data agent evaluation. Recent benchmarks evaluate increasingly realistic data work, including data analysis (Hu et al., 2024; Jing et al., 2025), enterprise SQL workflows (Lei et al., 2025; Huo et al., 2026), and repository-level data engineering and analysis (Lei et al., 2026). DV-World (Meng et al., 2026) studies visualization creation and repair, while SWE-SQL (Li et al., 2025) evaluates debugging of reproduced SQL issues. These benchmarks primarily require agents to produce, complete, or repair data work; DAVBench instead evaluates independent verification of existing candidate work.

Data quality and pipeline debugging. Prior systems validate datasets and transformations using inferred constraints, historical profiles, embedded checks, and pipeline tests (Song & He, 2021; Tu et al., 2023; Sinthong et al., 2021; Yang et al., 2025), while other work diagnoses failures and interventions in data and ML pipelines (Karlaš et al., 2024; Hasan et al., 2026; Grafberger et al., 2025). These approaches primarily assess data-quality properties or failures relative to predefined or inferred validation signals. DAVBench instead evaluates whether complete candidate data work satisfies a user requirement, using execution alongside code, data semantics, documentation, and delivered artifacts as evidence.

Agents as reviewers and verifiers. Recent work studies LLMs as verifiers of outputs and agent behavior. DataPRM (Qiu et al., 2026) verifies intermediate steps in agentic data analysis, LLM-as-a-Verifier (Kwok et al., 2026) scores candidate solutions and trajectories, and other work studies critique, self-correction, and structured-data verification (Gou et al., 2024; Theologitis et al., 2026). SWR-Bench (Zeng et al., 2025) and AACR-Bench (Zhang et al., 2026) evaluate repository-level code review. Data-work verification differs in requiring agents to connect data semantics, domain rules, workflow logic, and delivered artifacts, including cases where successful execution still yields semantically incorrect results. DAVBench evaluates localized defect reports and their utility for downstream repair.

5 Conclusion

We introduced DAVBench, a benchmark for evaluating agents' ability to verify real-world data work. DAVBench contains 216 repository-level tasks with reconstructed real-world defects, executable project environments, reference findings, and behavioral tests, covering both data engineering and analysis. Experiments with eight models show that current agents can often recognize faulty candidate work but still miss many of its defects, with the strongest verifier reaching only 49.82% finding F1 and complete defect coverage on 39.61% of faulty tasks. Downstream repair experiments further show that verification quality remains a key bottleneck: reference reports raise repair success from 24.03% without a report to 69.48%, while reports from current verifiers reach at most 37.66%. Together, these results highlight the need for agents that can more accurately and completely verify data workflows and artifacts before they are delivered or used downstream. DAVBench provides a testbed for developing and evaluating such verification capabilities.

AI USE STATEMENT

Generative AI tools were used to assist with manuscript editing, literature search, and feedback on experimental analyses and interpretation. All AI-assisted content was reviewed by the authors, and all reported experimental results, benchmark annotations, and claims were independently checked by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

DAVBench is constructed from publicly accessible data engineering and analysis projects. We restrict benchmark construction and release to project code and data whose licenses or usage terms permit the intended use, and retain source and version information for external public datasets. Verifier-visible environments exclude private evaluation materials and historical defect-revealing information.

REPRODUCIBILITY STATEMENT

We provide detailed benchmark construction and validation procedures in Section 2.3 and Appendix E, scoring and evaluation details in Section 2.2 and Appendix B.1, and model, agent, runtime, and run-stability settings in Appendix A. Verifier, repair, and semantic-judge prompts are provided in Appendix H. We will release the benchmark tasks, evaluation code, and supporting materials permitted by the licenses and usage terms of the underlying sources.

REFERENCES

Android Bench. Android Bench Technical Report, 2026.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026.

Google DeepMind. Gemini 3.8 Flash Model Card, 2026. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-8-Flash-Model-Card.pdf.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. In *International Conference on Learning Representations*, 2024.

Stefan Grafberger, Paul Groth, and Sebastian Schelter. mlidea: Interactively improving ML data preparation code via shadow pipelines. *Proceedings of the VLDB Endowment*, 18(12):5359–5362, 2025. doi: 10.14778/3750601.3750671.

Jahid Hasan, Stanley Jiang, Tejendra Singh, Sainyam Galhotra, Romila Pradhan, and Divesh Srivastava. PipeLens: Identifying interventions for resolving malfunctioning data science pipelines. *Proceedings of the VLDB Endowment*, 19(11):3188–3201, 2026. doi: 10.14778/3836663.3836682.

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, 2024.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, Hongyu Liu, Chenhao Ma, Fatma Ozcan, Yannis Papakonstantinou, and Reynold Cheng. BIRD-INTERACT: Re-imagining Text-to-SQL Evaluation via Lens of Dynamic Interactions. In *International Conference on Learning Representations*, 2026.

Tengjun Jin, Yuxuan Zhu, and Daniel Kang. ELT-Bench: An end-to-end benchmark for evaluating AI agents on ELT pipelines, 2025.

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*, 2025.

Bojan Karlas, Babak Salimi, and Sebastian Schelter. Navigating data errors in machine learning pipelines: Identify, debug, and learn. In *IEEE International Conference on Data Engineering*, pp. 4523–4529, 2025. doi: 10.1109/ICDE65448.2025.00344.

Bojan Karlaš, David Dao, Matteo Interlandi, Sebastian Schelter, Wentao Wu, and Ce Zhang. Data debugging with shapley importance over machine learning pipelines. In *International Conference on Learning Representations*, 2024.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. Llm-as-a-verifier: A general-purpose verification framework, 2026.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *International Conference on Learning Representations*, 2025.

Fangyu Lei, Jinxiang Meng, Yiming Huang, Junjie Zhao, Yitong Zhang, Jianwen Luo, Xin Zou, Ruiyi Yang, Wenbo Shi, Yan Gao, Shizhu He, Zuo Wang, Qian Liu, Yang Wang, Ke Wang, Jun Zhao, and Kang Liu. DAComp: Benchmarking data agents across the full data intelligence lifecycle. In *International Conference on Learning Representations*, 2026.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, Ziwei Tang, Yuanshuai Li, Florensia Widjaja, Xintong Zhu, Feige Zhou, Yongfeng Huang, Yannis Papakonstantinou, Fatma Ozcan, Chenhao Ma, and Reynold Cheng. SWE-SQL: Illuminating LLM pathways to solve user SQL issues in real-world applications. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3247.

Yiming Lin, Yeye He, and Surajit Chaudhuri. Auto-BI: Automatically build BI-models leveraging local join prediction and global schema graph. *Proceedings of the VLDB Endowment*, 16(10): 2578–2590, 2023. doi: 10.14778/3603581.3603596.

Jinxiang Meng, Shaoping Huang, Fangyu Lei, Jingyu Guo, Haoxiang Liu, Jiahao Su, Sihan Wang, Yao Wang, Enrui Wang, Ye Yang, Hongze Chai, Jinming Lv, Anbang Yu, Huangjing Zhang, Yitong Zhang, Yiming Huang, Zeyao Ma, Shizhu He, Jun Zhao, and Kang Liu. DV-World: Benchmarking Data Visualization Agents in Real-World Scenarios, 2026.

Ali Merali. Scaling laws for economic productivity: Experimental evidence in LLM-assisted consulting, data analyst, and management tasks, 2025.

OpenAI. GPT-6 Astra System Card, 2026a. URL https://deploymentsafety.openai.com/gpt-6-astra/gpt-6-astra.pdf.

OpenAI. GPT-5.6 System Card, 2026b. URL https://deploymentsafety.openai.com/gpt-5-6/gpt-5-6.pdf.

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*, pp. 11866–11877, 2026. doi: 10.1145/3770855.3819049.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Phanwadee Sinthong, Dhaval Patel, Nianjun Zhou, Shrey Shrivastava, Arun Iyengar, and Anuradha Bhamidipaty. DQDF: Data-quality-aware dataframes. *Proceedings of the VLDB Endowment*, 15 (4):949–957, 2021. doi: 10.14778/3503585.3503602.

Jie Song and Yeye He. Auto-validate: Unsupervised data validation using data-domain patterns inferred from data lakes. In *Proceedings of the 2021 International Conference on Management of Data*, pp. 1678–1691, 2021. doi: 10.1145/3448016.3457250.

Michael Theologitis, Preetam Prabhu Srikar Dammu, Chirag Shah, and Dan Suciu. ClaimDB: A Fact Verification Benchmark over Large Structured Data, 2026.

Dezhan Tu, Yeye He, Weiwei Cui, Song Ge, Haidong Zhang, Shi Han, Dongmei Zhang, and Surajit Chaudhuri. Auto-validate by-history: Auto-program data quality constraints to validate recurring data pipelines. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4991–5003, 2023. doi: 10.1145/3580305.3599776.

Cong Yan and Yeye He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 International Conference on Management of Data*, pp. 1539–1554, 2020. doi: 10.1145/3318464.3389738.

Hongtao Yang, Zhichen Xu, Sergey Yudin, and Andrew Davidson. Unlocking the power of CI/CD for data pipelines in distributed data warehouses. *Proceedings of the VLDB Endowment*, 18(12): 4887–4895, 2025. doi: 10.14778/3750601.3750613.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-Bench: Can Language Models Rebuild Programs From Scratch?, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Z.ai. Glm-5.3: Frontier coding with emergent cyber capabilities. `https://z.ai/blog/glm-5.3`, 2026. Accessed 2026.

Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. SWR-Bench: Assessing LLM Performance in Real-World Code Review Comment Generation, 2025.

Lei Zhang, Yongda Yu, Minghui Yu, Xinxin Guo, Zhengqi Zhuang, Guoping Rong, Dong Shao, Haifeng Shen, Hongyu Kuang, Zhengfeng Li, Boge Wang, Guoan Zhang, Bangyu Xiang, and Xiaobin Xu. AACR-Bench: Evaluating automatic code review with holistic repository-level context, 2026.

A Experimental Details

Agent and Execution Environment. All verifier and repair models use mini-SWE-agent (Yang et al., 2024) as a common lightweight bash-only scaffold. This minimal interface places the emphasis on model capability rather than harness-specific tools, and follows the bash-oriented evaluation setting used in prior agent benchmarks. Through the terminal, agents can inspect project files and documentation, query data and databases, and execute project code and tests.

Each task runs in a pre-built Docker image containing the reconstructed candidate workspace together with the runtimes, dependencies, database services, and other tools required by the corresponding project. These environments are prepared and checked during benchmark construction so that evaluation begins from an executable project state rather than requiring agents to reconstruct the software environment themselves. All models receive the same task image and a fixed wall-clock budget for each task. Budgets are 20 or 30 minutes depending on the task's execution requirements.

Model Settings. We use provider-default reasoning/thinking and sampling settings unless otherwise specified by the serving endpoint. Verifier and repair runs use a maximum completion length of 8,192 tokens. Table 7 lists the model identifiers used in our experiments.

Table 7: Model names and identifiers used in our experiments.

Model	**Identifier**
GPT-6 Astra (OpenAI, 2026a)	`gpt-6-astra`
GPT-5.6-Sol (OpenAI, 2026b)	`gpt-5.6-sol`
Gemini 3.8 Flash (Google DeepMind, 2026)	`gemini-3.8-flash`
DeepSeek V4 Flash (DeepSeek-AI, 2026)	`deepseek-v4-flash-0731`
GLM 5.3 Flash (Z.ai, 2026)	`glm-5.3-flash`
GLM 5.3 (Z.ai, 2026)	`glm-5.3`
Qwen 3.8 Max (Qwen Team, 2026)	`qwen3.8-max-0902`
Qwen 3.8-27B (Qwen Team, 2026)	`qwen3.8-27b`

Repair Settings. The main repair-utility evaluation uses GPT-5.6-Sol as a fixed repair agent on all 154 faulty candidates. For each candidate, the repair agent receives either no verification report, a report produced by an evaluated verifier, or the reference report. Reference reports use the same report format and contain reference findings, but do not expose reference patches or hidden behavioral tests. Qwen 3.8-27B is additionally evaluated as a second repair agent in Section 3.4.

Semantic Judge. The evidence-grounded semantic scorer uses DeepSeek V4 Flash as its primary LLM judge. Judge calls use a maximum completion length of 16,384 tokens. Alternative judges in the evaluation-reliability analysis receive the same prompt and evidence inputs.

Run-to-Run Stability. Given the substantial cost of repeated repository-level verification with executable environments, code and data interaction, and frontier-model API calls such as GPT-6 Astra, the main experiments use a single run per model–task pair. To assess run-to-run stability, we randomly sample 60 tasks spanning both DE and DA and repeat three representative models from different families three times. Table 8 shows limited variation in Finding F1, with standard deviations of only 1.85–1.98 percentage points, while the model ordering remains unchanged across all three runs. These results suggest that the single-run full-benchmark comparisons are not primarily driven by stochastic agent execution on this subset. Given the evaluation cost, we prioritize broader coverage across model families in the main experiments to characterize differences in verification performance and behavior.

Table 8: Run-to-run stability of Finding F1 (%) on a randomly sampled 60-task subset.

Model	Run 1	Run 2	Run 3	Mean $\pm$ SD
GPT-5.6-Sol	41.61	37.90	39.77	39.76 ± 1.85
DeepSeek V4 Flash	26.48	25.08	22.58	24.71 ± 1.98
Qwen-3.8-27B	22.04	24.49	20.66	22.40 ± 1.94

Table 9: Run-to-run stability of GPT-5.6-Sol as the repair agent on 60 sampled repair instances.

Run	Successful repairs	RSR (%)
1	24 / 60	40.00
2	22 / 60	36.67
3	24 / 60	40.00

Repair-agent stability. We additionally assess run-to-run variation of the GPT-5.6-Sol repair agent on 60 randomly sampled repair instances, covering four verifier-report conditions with 15 instances per condition. We repeat the repair procedure three times on the same instances. Repair Success Rate is 40.00%, 36.67%, and 40.00% across the three runs (mean 38.89%, SD 1.92 percentage points), indicating similarly small run-to-run variation in downstream repair outcomes.

B Evaluation Details

B.1 Scoring Protocol

Verification quality. A verification report is evaluated along two complementary components: whether its verdict is correct and whether its reported findings accurately and completely identify the reference defects. Scoring findings requires resolving two sources of ambiguity. First, a defect may be reasonably localized to more than one nearby code or configuration span, making exact-line matching unnecessarily brittle. Second, location alone is insufficient: two findings may point to the same region while diagnosing different problems, while superficially similar diagnoses may refer to different underlying defects. We therefore use a two-stage matching procedure that combines location-based candidate matching with evidence-grounded semantic judging.

Admissible defect locations. For each reference defect $f_j^* = (d_j^*, \mathcal{L}_j^*)$, annotators specify a diagnosis d_j^* and a set of admissible location ranges

$$\mathcal{L}_j^* = \{\rho_{j1}^*, \ldots, \rho_{jK_j}^*\}.$$

Each range $\rho = (p, [a, b])$ consists of a file path p and an inclusive line interval $[a, b]$. Multiple ranges are used when the same defect can be validly localized to different spans, such as the faulty expression itself, the surrounding transformation, or another location that directly exposes the same root cause. This avoids penalizing a verifier solely because its localization differs from an arbitrary annotation boundary.

Let the location reported for finding f_i be

$$\ell_i = (p_i, \lambda_i),$$

where p_i is the reported file path and λ_i is the reported line number. A reported finding and reference defect pass the location stage when the reported line falls within at least one admissible reference range in the same file:

$$M_{\text{loc}}(i, j) = \mathbf{1}\left[\exists\, (p, [a, b]) \in \mathcal{L}_j^* \text{ s.t. } p_i = p \ \wedge\ a \le \lambda_i \le b\right].$$

Thus, location matching acts as a permissive grounding constraint rather than requiring an exact annotated line.

Evidence-grounded semantic matching. Location agreement alone does not establish that two findings identify the same defect. Conversely, judging only the diagnosis text can conflate findings that describe a shared symptom but arise from different causes. We therefore apply semantic judging only to reported–reference pairs satisfying $M_{\text{loc}}(i, j) = 1$.

For each eligible pair, LLM as judge (e.g. DeepSeek V4 Flash) receives the task instruction, the two diagnoses, and line-numbered excerpts from the frozen candidate around the reported and reference locations. The judge determines whether the reported finding itself identifies the same underlying defect as the reference finding. It is instructed to use the supplied candidate evidence to disambiguate

the findings, but not to infer or complete a vague diagnosis on the verifier's behalf. We denote this decision by

$$M_{\text{sem}}(i, j) \in \{0, 1\},$$

and define an eligible finding match as

$$M(i, j) = M_{\text{loc}}(i, j)\, M_{\text{sem}}(i, j).$$

Because a verifier may report multiple findings near the same defect, credited findings are matched one-to-one with reference defects, preventing duplicate reports from receiving repeated credit. The frozen judging prompt is provided in Appendix H. The complete hybrid scorer agrees with human labels on 97.55% of sampled finding–reference pairs; further validation is reported in Section 3.5 and Appendix B.2.

Finding metrics. For faulty task q, let n_q be the number of reported findings, g_q the number of reference defects, and t_q the number of one-to-one matched finding–reference pairs. We compute

$$P_q = \frac{t_q}{n_q}, \qquad R_q = \frac{t_q}{g_q}, \qquad \text{F1}_q = \frac{2t_q}{n_q + g_q}.$$

When no findings are reported ($n_q = 0$), we set $P_q = 0$; recall and F1 are consequently zero for faulty tasks. Finding precision, recall, and F1 are macro-averaged over the $N = 154$ faulty tasks.

Finding-level recall does not distinguish between identifying some defects in every task and completely verifying a subset of tasks. We therefore additionally report *All-found Rate (%)*, which measures complete defect coverage:

$$\text{AllFound} = \frac{100}{N} \sum_{q=1}^{N} \mathbf{1}[t_q = g_q].$$

Missing verification reports receive zero finding scores.

Verdicts and correct candidates. Verdict accuracy is computed over both faulty and correct candidates. The 62 correct candidates contain no reference defects and are therefore excluded from finding precision, recall, F1, and All-found Rate. For these candidates, we separately report false findings per task,

$$\text{FP/task} = \frac{1}{62} \sum_{q=1}^{62} n_q,$$

to measure whether a verifier invents defects in work that should be accepted.

Repair utility. Reference-matching metrics measure whether a report identifies known defects, but they do not directly measure whether the report is useful for correcting the candidate. We therefore evaluate report utility behaviorally by providing reports from different verifiers to a fixed downstream repair agent, while holding the initial candidate and repair setting constant.

For faulty task q, let $T_q = T_q^{\text{F2P}} \cup T_q^{\text{P2P}}$ denote its behavioral test suite. Fail-to-pass (F2P) tests check whether the target faulty behavior is corrected, while pass-to-pass (P2P) tests check whether previously correct behavior is preserved. A repair is counted as successful only when it is valid and passes every test:

$$s_q = \mathbf{1}\left[\text{repair is valid } \wedge\ \forall t \in T_q,\ t \text{ passes}\right].$$

For each repair agent and report condition, we report

$$\text{RSR}(\%) = \frac{100}{N} \sum_{q=1}^{N} s_q, \qquad N = 154.$$

B.2 Evaluation Reliability Details

Human Validation Protocol. We sample 42 verification reports from 35 tasks, containing 111 reported findings and 88 reference-defect instances. For each report, we form all finding–reference pairs, yielding 245 pairs in total. Three annotators independently judge whether each pair identifies

the same underlying defect and, for matching pairs, whether the reported location is valid. Model identities and automated scorer decisions are hidden during annotation, and majority vote determines the human reference. Annotators agree on 99.59% of semantic labels (Fleiss' $\kappa = 0.992$) and 98.37% of combined semantic-and-location labels ($\kappa = 0.966$). The resulting human reference contains 49 positive and 196 negative pairs.

Stage-Level Scorer Analysis. Table 10 decomposes the hybrid scorer into its location and semantic stages. Among the 46 positive pairs with unanimous agreement on both diagnosis and location, 44 pass the location filter. The location filter also admits six pairs that refer to different defects, which are rejected by the semantic judge. Pair-level F1 in Table 10 measures agreement with human pair-matching decisions and is distinct from the task-macro finding F1 used to evaluate verifiers.

Alternative Judges. The four judges in Table 5 evaluate the same 50 location-qualified pairs using the same prompt and evidence inputs. For the model-ranking comparison in Table 6, we rescore fixed reports from six verifiers while holding the reference annotations, location matching, and task-macro aggregation unchanged. Spearman correlations are computed from unrounded F1 scores with average ranks for ties. DeepSeek V4 Flash serves as the reference judge for the reported rank correlations.

Table 10: Automated matching against majority human labels. Counts refer to finding–reference pairs; agreement and F1 are percentages.

Scoring stage	Pairs	TP	FP	FN	TN	Agreement	F1
Semantic judge after location filtering	50	43	0	1	6	98.00	98.85
Complete location + semantic scorer	245	43	0	6	196	97.55	93.48

C Additional Experimental Analyses

C.1 Verification Results by Domain

Table 11 reports the complete verification results for DE and DA underlying Figure 4(a). Performance differences between DE and DA vary across models, motivating the finer-grained work-type analysis in the main text.

C.2 Static and Dynamic Verification

We compare four models under two verification workflows on the same 154 faulty candidates. Both support multi-step inspection of the project workspace. *Static* verification allows agents to browse and search files, inspect code, data, documentation, and existing artifacts, but does not execute candidate code. *Dynamic* verification additionally provides a shell for querying data and executing code, tests, or agent-designed checks. The Dynamic results are those used in the main experiments.

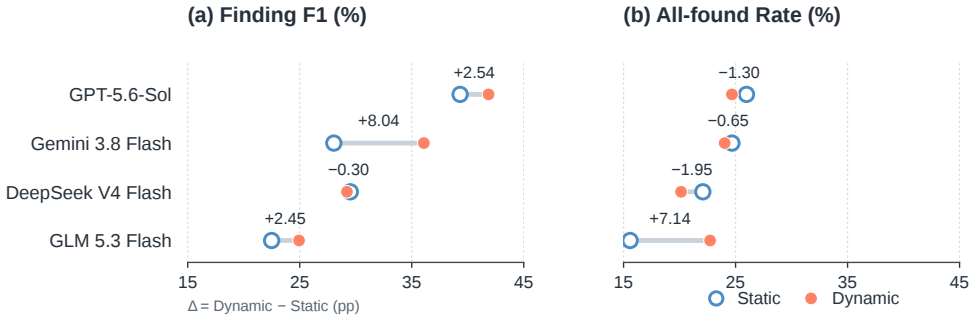


Figure 6: Static and Dynamic verification on the same 154 faulty candidates. Labels show Dynamic minus Static in percentage points.

Table 11: Verification quality in data engineering (DE) and data analysis (DA). Finding P/R/F1 and All-found Rate are computed on faulty candidates; Reject and Accept denote verdict accuracy on faulty and correct candidates, respectively. F/task and FP/task denote reported findings and false findings per task. Scores are percentages except these counts; best results within each domain are bold.

	Reject						Accept	
Model	Acc.	F/task	P	R	F1	All-found	Acc.	FP/task
DE: 65 Reject / 24 Accept								
GPT-6 Astra	**98.46**	4.08	**53.43**	**61.21**	**50.75**	**32.31**	79.17	0.33
GPT-5.6-Sol	93.85	2.57	53.35	44.59	44.40	18.46	79.17	0.21
Gemini 3.8 Flash	72.31	2.31	43.02	43.60	39.70	23.08	**100.00**	**0.00**
DeepSeek V4 Flash	78.46	1.62	36.79	30.58	31.05	16.92	70.83	0.42
GLM 5.3 Flash	76.92	2.31	29.67	31.86	27.90	16.92	41.67	0.58
GLM 5.3	73.85	2.63	31.00	38.36	31.29	18.46	62.50	0.46
Qwen 3.8 Max	66.15	1.77	29.23	26.61	25.51	12.31	91.67	0.08
Qwen 3.8-27B	41.54	0.72	22.31	14.31	15.04	7.69	45.83	0.21
DA: 89 Reject / 38 Accept								
GPT-6 Astra	91.01	2.83	**47.39**	**60.15**	**49.13**	**44.94**	**81.58**	0.21
GPT-5.6-Sol	**94.38**	2.66	38.83	47.23	40.00	29.21	47.37	0.74
Gemini 3.8 Flash	69.66	1.44	36.03	35.37	33.44	24.72	71.05	0.24
DeepSeek V4 Flash	79.78	2.37	26.22	35.56	27.88	22.47	55.26	0.42
GLM 5.3 Flash	60.67	2.88	17.80	37.49	22.78	26.97	57.89	0.47
GLM 5.3	84.27	4.08	23.98	48.24	28.80	31.46	55.26	0.53
Qwen 3.8 Max	80.90	3.10	25.78	43.61	29.26	29.21	76.32	0.26
Qwen 3.8-27B	50.56	1.40	19.04	25.92	20.28	16.85	76.32	**0.00**

Dynamic verification improves finding accuracy more consistently than complete coverage. Figure 6 shows that Finding F1 increases for GPT-5.6-Sol (+2.54 points), Gemini 3.8 Flash (+8.04), and GLM 5.3 Flash (+2.45), while remaining nearly unchanged for DeepSeek V4 Flash (-0.30). Precision increases for all four models, but All-found Rate increases only for GLM 5.3 Flash. For Gemini, the 8.04-point F1 gain is driven mainly by higher precision (+12.53 points), while recall changes by only +1.10 points and complete coverage remains nearly unchanged. These results indicate that executable interaction can help agents produce more accurate findings, but does not by itself ensure more complete verification.

The effects also differ across domains and models (Figure 7). On DA tasks, Gemini gains 15.02 points in precision with nearly unchanged recall and All-found Rate, whereas GLM 5.3 Flash gains 8.22 points in recall and 8.99 points in All-found Rate. Thus, executable interaction can improve either the accuracy or the coverage of verification, but models differ in how they exploit this capability.

C.3 Performance by Source Recency

Because DAVBench is reconstructed from publicly available project histories, we examine whether verifier performance is systematically higher on older source PRs. We group the 154 faulty tasks by source-PR creation time into 28 tasks from 2024 or earlier, 42 tasks from 2025, and 84 tasks from 2026, and recompute Finding F1 for each verifier within each group.

Performance does not exhibit a monotonic advantage on older source PRs. Seven of the eight verifiers achieve higher Finding F1 on the 2025 tasks than on tasks from 2024 or earlier, which is inconsistent with a simple pattern in which older public sources are systematically easier. Verifier rankings are also highly stable across source-recency groups: the rankings on the $\leq$2024 and 2025 groups are identical (Spearman's $\rho = 1.00$), while the 2026 ranking has $\rho = 0.95$ with each of the two earlier groups. The ranking on the 2026 tasks is also identical to that on the full benchmark.

Performance shifts on the most recent tasks further depend on the data-work domain rather than showing a uniform recency effect. Across all eight verifiers, Finding F1 increases on newer DE tasks while decreasing on newer DA tasks; for example, Gemini 3.8 Flash changes by $+2.87$ points

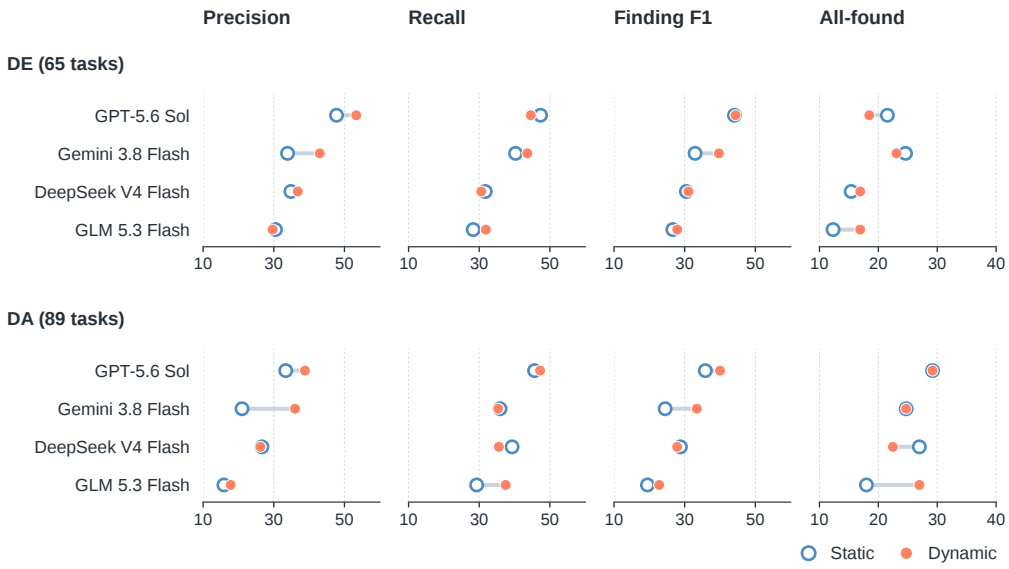


Figure 7: Static and Dynamic verification by domain on the same 65 DE and 89 DA faulty tasks. Precision, recall, and F1 are task-macro finding scores; All-found Rate measures complete defect coverage.

on DE but -19.88 points on DA. Together, these results show no systematic concentration of verifier performance on older source PRs and indicate that the main model comparisons remain stable across source recency.

C.4 Repair Utility Details

All repair analyses use the same 154 faulty candidates and report conditions as the main experiments. Reference reports follow the same format as model-generated reports and contain reference findings, but expose neither reference patches nor hidden behavioral tests.

Verification reports affect both correction and preservation. Across report conditions, F2P varies substantially with the supplied verification report, whereas P2P is generally higher but can still decrease for particular reports and repair agents. For example, with GPT-5.6-Sol as the repair agent, P2P falls from 85.71% without a report to 71.43% with GLM 5.3's report. Even reference reports do not eliminate such regressions, indicating that downstream repair can disrupt previously correct behavior independently of verification quality. These results motivate evaluating both correction and preservation when measuring repair utility.

Unmatched findings are associated with repair regressions. Across available task–report conditions, P2P failure rises from 3.76% to 9.08% for GPT-5.6-Sol and from 2.34% to 7.65% for Qwen 3.8-27B when comparing reports with zero versus at least one unmatched finding. The trend becomes stronger as unmatched findings accumulate: with at least five unmatched findings, P2P failure reaches 27.03% and 22.97% for the two repair agents, respectively. P2P failure also increases with the number of unmatched findings, with Spearman's $\rho = 0.18$ and 0.19 for the two repair agents. These results show that reports containing more unmatched findings are more likely to be followed by regressions in previously correct behavior.

Verification-side metrics consistently track downstream repair utility. Figure 8 extends the main-text F1 analysis to finding precision, recall, and All-found Rate. Across the eight verifier reports, all four verification metrics are positively correlated with repair gains under both repair agents, with correlations ranging from $r = 0.84$ to 0.96. Coverage-oriented metrics also show strong relationships with repair outcomes, indicating that downstream utility depends not only on avoiding incorrect findings but also on identifying the defects needed for repair. The F1–RSR relationship

Table 12: Finding F1 (%) by source-PR creation time on the 154 faulty candidates.

Verifier	$\leq$2024 ($n = 28$)	2025 ($n = 42$)	2026 ($n = 84$)
GPT-6 Astra	49.97	50.97	49.19
GPT-5.6-Sol	36.75	43.00	43.00
Gemini 3.8 Flash	39.73	43.46	31.18
GLM 5.3	27.25	32.47	29.41
DeepSeek V4 Flash	30.70	32.61	27.03
Qwen 3.8 Max	27.18	29.52	26.92
GLM 5.3 Flash	23.47	25.66	25.07
Qwen 3.8-27B	21.35	20.46	15.78

Table 13: Task-level repair outcomes (%) for two repair agents on 154 faulty tasks. F2P and P2P each require all tests in the corresponding group to pass; RSR requires both.

	GPT-5.6-Sol repair agent			Qwen 3.8-27B repair agent		
Verification report	F2P ↑	P2P ↑	RSR ↑	F2P ↑	P2P ↑	RSR ↑
GPT-6 Astra	38.96	85.71	37.66	34.42	81.82	33.12
GPT-5.6-Sol	29.22	83.12	28.57	29.22	85.71	27.92
Gemini 3.8 Flash	29.87	85.06	28.57	28.57	85.06	28.57
DeepSeek V4 Flash	25.32	85.71	25.32	24.68	85.06	24.68
GLM 5.3 Flash	25.32	79.22	24.68	24.68	84.42	24.03
GLM 5.3	30.52	71.43	29.22	25.32	79.22	24.03
Qwen 3.8 Max	22.73	86.36	22.08	22.73	84.42	22.08
Qwen 3.8-27B	18.83	85.06	18.83	17.53	87.66	17.53
No report (A_0)	24.03	85.71	24.03	18.18	87.66	18.18
Reference report (A_{gold})	72.73	81.82	69.48	64.94	83.77	61.69

Table 14: P2P failure rate (%) by the presence of unmatched verification findings. Counts are task–report conditions with available reports and confirmed P2P outcomes.

Repair agent	0 unmatched	$\geq$1 unmatched
GPT-5.6-Sol	3.76	9.08
Qwen 3.8-27B	2.34	7.65

remains positive when any single verifier is removed, ranging from 0.824–0.951 for GPT-5.6-Sol and 0.916–0.973 for Qwen 3.8-27B.

D Verification Failure Analysis

We further analyze why verifiers miss reference defects by examining a reviewed set of 411 missed defect–verifier cases from four verifiers across 89 faulty tasks, comprising 168 DE and 243 DA cases. For each miss, we relate the reference defect to the project materials surfaced during verification, the checks performed, and the final report. Failure patterns are assigned from observable tool outputs and final reports against the corresponding reference defect; multiple patterns may apply to one miss.

Relevant evidence often fails to translate into a diagnosis. Among the 411 reviewed misses, related project materials were surfaced without a corresponding diagnosis in 206 cases (50.1%). Checks omitted critical conditions in 94 cases (22.9%), while key evidence was not obtained in 49 cases (11.9%). These results indicate that many missed defects arise after relevant evidence has already entered the investigation, highlighting the importance of selecting informative checks and connecting available evidence to a defect diagnosis. The benchmark itself frequently requires such cross-evidence reasoning: 301 of the 389 reference defects draw on at least two material types in their annotated evidence graphs, excluding the task requirement.

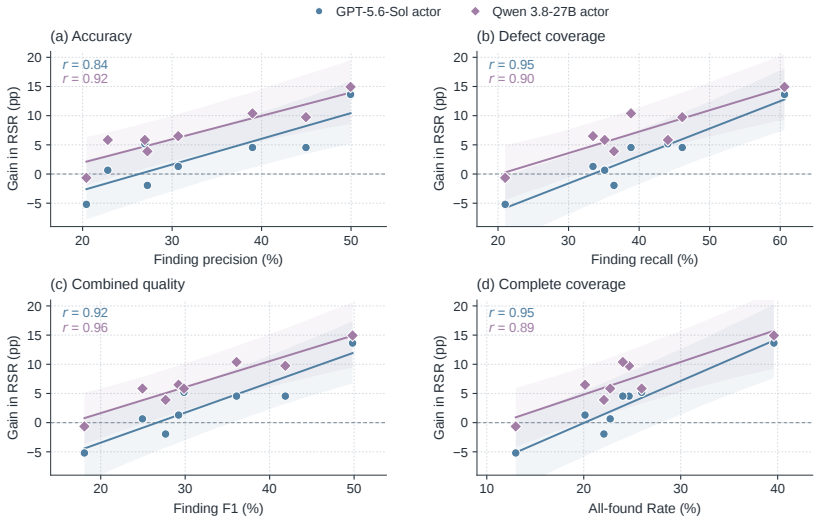


Figure 8: Verification quality and repair gains across eight verifier reports. Repair gains are measured relative to each repair agent's no-report baseline; shaded bands show 95% task-bootstrap intervals.

Failure patterns differ between DE and DA. Checks omit critical conditions more often among DE misses (53/168, 31.5%) than DA misses (41/243, 16.9%), with the same direction across all four verifiers. In contrast, invalid criteria or misinterpretation of code and data are more common among DA misses (34/243, 14.0%) than DE misses (9/168, 5.4%). Among 56 DE misses involving entity identity, granularity, or schema relationships, 16 omit critical checking conditions; among 33 DA misses involving statistical populations or analytical targets, 8 involve invalid criteria or misinterpretation. These patterns point to complementary weaknesses in checking the conditions under which transformations are correct and determining whether computations represent the intended analytical quantity.

More interaction alone does not yield better verification. We additionally examine 109 tasks with aligned interaction records across the four verifiers. Within each verifier, the correlation between changes in interaction rounds and changes in Finding F1 ranges from -0.14 to $+0.14$. Thus, simply increasing the amount of interaction does not consistently improve verification quality, reinforcing the importance of informative checks and evidence-grounded diagnosis.

E Benchmark Construction Details

E.1 Source Selection Criteria

Source Pool. We build the candidate source pool from publicly accessible data engineering and analysis projects spanning diverse real-world settings, including data modeling and transformation, public-data and statistical publishing, biomedical and scientific analysis, energy and geospatial modeling, and metric- or policy-oriented analytics. These projects expose a broad range of data work, such as source ingestion, data transformation and integration, metric computation, analytical modeling, and generation of delivered artifacts. Projects in the source pool serve only as candidates for defect discovery; individual changes are included only after satisfying the screening criteria below. We restrict benchmark construction to project code and data sources whose licenses or usage terms permit the intended benchmark use. For external public datasets used during reconstruction, we retain their source and version information and package local copies only when permitted by the corresponding terms.

Source selection aims to identify documented defects that are both representative of real data work and reconstructable as self-contained verification tasks. Screening therefore considers not only whether a reported issue is a substantive data-work defect, but also whether its cause and correc-

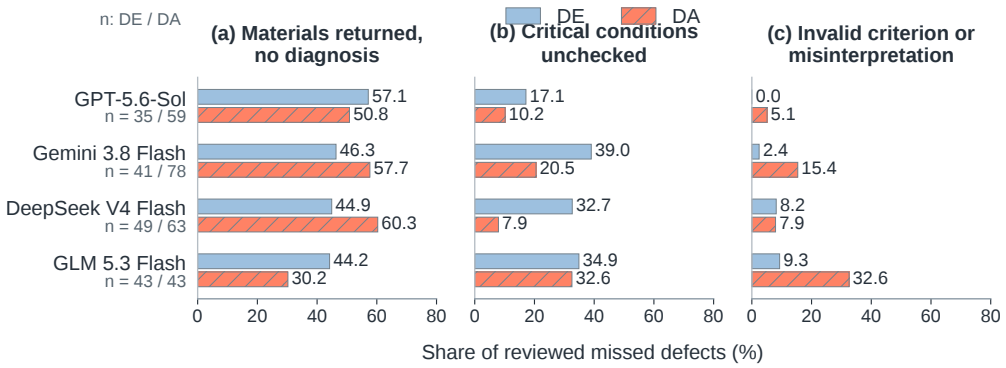


Figure 9: Observed failure patterns among reviewed missed defects, separated by verifier and domain. Shares are computed within each verifier–domain subset of reviewed misses; failure patterns may overlap.

tion are sufficiently supported, whether a faulty candidate can be isolated, and whether correctness can later be judged from the reconstructed project environment. Table 15 summarizes the selection criteria; reproduction and execution checks are performed during subsequent task construction.

Table 15: Criteria used to screen real-world defect sources for DAVBench.

	Selection criterion	What to check
☐	**Data-work relevance**	The changed work concerns data transformations, computations, analytical methods, or outputs. Documentation-only changes, routine dependency updates, and repository maintenance are excluded.
☐	**Defect significance**	The reported problem affects data-processing or analytical correctness, or behavior necessary for the workflow to operate. Stylistic preferences and feature requests without evidence of an existing error are excluded.
☐	**Supporting evidence**	Project feedback identifies a concrete problem and connects it to the affected work. A generic review comment or automated suggestion alone is insufficient evidence of a defect.
☐	**Traceable correction**	Corrective revisions address the same problem described in the feedback. Corrective revisions address the documented problem; merged corrections are preferred when available.
☐	**Faulty candidate and scope**	An identifiable project revision still contains the reported defect. Unrelated changes and problems introduced only in later revisions are kept outside its scope.
☐	**Reconstruction feasibility**	Available code, data, and runtime dependencies support reproducing the problem. Sources are included only when the relevant data conditions can be reproduced faithfully.
☐	**Independent verification**	Task requirements and project materials provide a basis for judging correctness without access to private review feedback or reference fixes.
☐	**Source uniqueness**	Repository and PR identities are checked against previously collected sources. Repeated fixes, backports, and multiple task versions are tracked rather than counted as independent sources.

E.2 Reconstruction of Candidate Data Work

Reconstruction turns a documented data-work defect into candidate data work that can be investigated within its project environment. The central requirement is to preserve both the faulty behavior and the evidence needed to judge its correctness.

Candidate and Environment Setup. Review feedback and corrective revisions identify a project version in which the documented defect is still present. We build a Docker environment containing the candidate project, its dependencies, and the tools needed to execute and investigate it. For dbt projects, this includes related models, macros, source definitions, seed data, and integration-test projects, preserving the dependencies needed to inspect upstream and downstream computations. We prioritize locally runnable backends supported by the project: PostgreSQL is used when natively supported, while DuckDB provides a self-contained alternative when its SQL and adapter behavior preserve the defect. Database profiles, services, and dependencies are configured during construction, and the project is exercised through commands such as `dbt seed`, `dbt run`, and `dbt test`. Python and R workflows similarly retain their project code and required libraries.

Data and Workflow Reproduction. Data reconstruction is designed to make the documented defect observable while preserving the conditions needed to investigate its cause. We reuse project datasets when available and download public data and supporting documentation, retaining local copies and recording their sources and versions. When original inputs are unavailable, constructed or adapted data preserve the schema, relationships, and state conditions relevant to the defect, such as entity identifiers, historical updates, or time partitions. Candidate and corrected computations are then exercised on the same inputs to confirm that the reconstructed environment reproduces the documented faulty behavior and its correction. For example, a revenue-calculation task retains a quantity-only invoice update followed by a month without a new invoice: the candidate completes execution but produces revenue of 2997 instead of 300. For artifact-centered tasks, we execute the candidate's actual generation code and retain its outputs alongside the relevant inputs and code, allowing discrepancies in a delivered artifact to be traced through the workflow.

Verification Task Specification. Two principles guide the public task specification and verifier-visible materials.

(1) Requirement Clarity and Sufficiency. The user requirement states the intended data-work outcome or use of the deliverable, grounded in project materials. Relevant data definitions, method documentation, and scope constraints provide sufficient evidence to assess whether the candidate satisfies that requirement; correctness does not depend on requirements available only in private review records.

(2) Diagnosis Neutrality. Instructions and added materials describe the task and environment without identifying the faulty component, highlighting suspicious records, or prescribing a diagnostic procedure. Normal project evidence remains available, while historical diagnoses, reference fixes, and evaluation tests are kept separate, leaving the verifier to independently identify and investigate potential defects.

Illustrative Reconstruction Example. Table 16 illustrates how a real-world defect is reconstructed into a verification task while preserving both the faulty behavior and the evidence needed to assess it.

Contamination and Leakage Mitigation. We apply safeguards across task construction, workspace preparation, and evaluation access.

(1) Defect-preserving task reconstruction. Public PRs and issue discussions serve as provenance for real-world defects, but DAVBench does not directly replay their original repository snapshots or descriptions. We reconstruct each defect as a task-specific executable instance, adapting code, configuration, data, or runtime infrastructure when needed to make the workflow self-contained. For example, database-backed workflows may be reproduced with PostgreSQL or DuckDB when the relevant behavior is preserved. We validate that these adaptations retain the defect-triggering computation, data relationships, or state conditions and reproduce the observable difference between faulty and corrected behavior. User requirements are constructed from project context and the intended data-work outcome rather than copied from the source PR or issue, and candidate outputs are regenerated by executing the reconstructed workflow.

(2) Prepare the verifier-visible workspace. Defect-revealing review and issue comments, reference fixes, and hidden tests are excluded from the task environment. Normal project code, data, and domain documentation remain available. Where repository history could expose historical corrections, the `.git` directory is removed.

Table 16: Illustrative reconstruction of a verification task, showing how the candidate, project workspace, supporting evidence, and delivered artifact are preserved for independent verification.

Component	Reconstruction choice	Role in verification
Candidate work	Retain the transformation that applies an extra factor of 1,000 to guest counts.	Preserves the faulty computation to be verified.
Project workspace	Reconstruct the repository and executable environment containing the candidate code and its relevant dependencies.	Allows the verifier to inspect and execute the surrounding workflow rather than judging an isolated snippet.
Input data	Package the tourism-data archive used by the workflow.	Provides the source values needed to check the candidate computation.
Domain documentation	Retain unit metadata and official statistical materials.	Provides the semantic evidence needed to interpret the guest counts correctly.
Delivered artifact	Execute the candidate producer to generate the annual guest-count table supplied for verification.	Exposes the observable downstream result of the faulty transformation.
User requirement	Request an annual domestic hotel guest-count table for publication.	Defines the intended outcome against which the candidate is judged.

(3) Restrict answer access. Private evaluation materials remain outside verifier-visible mounts, and task instructions prohibit retrieving upstream answers.

We additionally examine performance by source-PR recency in Appendix C.3; verifier performance shows no monotonic advantage on older sources, and model rankings remain highly stable across recency groups.

E.3 Evidence-Grounded Defect Annotation

Reference annotations are designed to capture both the underlying defect and the evidence by which it can be independently rediscovered in the reconstructed environment.

Reference Findings. For each faulty task, annotators construct a set of reference findings

$$F^* = \{f_1^*, \ldots, f_k^*\}, \qquad f_j^* = (d_j^*, \mathcal{L}_j^*),$$

where d_j^* describes the underlying defect and $\mathcal{L}_j^*$ contains one or more admissible location ranges. Multiple ranges accommodate reasonable localization ambiguity when the same underlying defect can be localized to more than one valid span.

Evidence Graphs. Project feedback and corrective revisions establish the historical basis for each reference defect. Annotators then maintain an internal evidence graph linking the defect diagnosis to relevant data, code, documentation, and reproduced observations in the reconstructed environment. Evidence items are grounded by file paths, line ranges, quotations, or data locators, while directed edges record how intermediate claims depend on this evidence and on preceding reasoning steps. The graph documents how the defect can be independently rediscovered from the project environment; it is neither provided to the evaluated verifier nor required in its output.

Figure 10 illustrates this structure with a monthly-buyer metric.

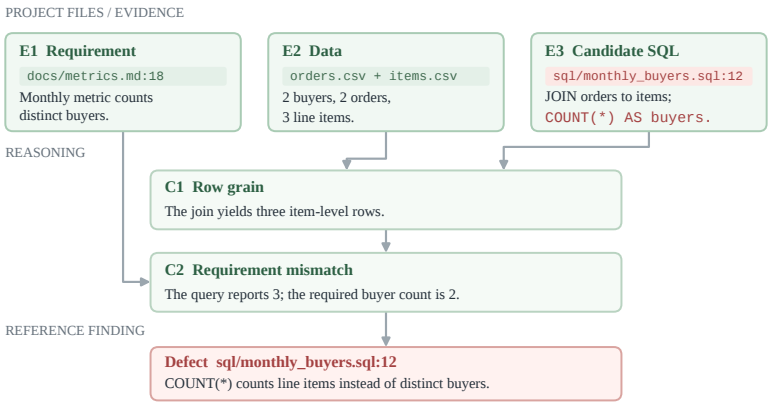


Figure 10: An evidence graph linking requirement documentation, project data, and candidate code to intermediate claims and a localized defect diagnosis. Arrows indicate support dependencies; file paths are illustrative.

Annotation Refinement. Reference findings are initially grounded in documented project feedback, subsequent fixes, and reproduced behavior. During task validation, independently discovered discrepancies are compared against the task requirement and project evidence and are incorporated into the reference findings when supported. Validated corrected versions are checked against the task requirement and used to provide correct candidates.

Table 17: Task-quality checklist used for independent validation and revision.

	Check	Criterion
☐	**Known-defect coverage**	Relevant review and corrective leads are included in reference findings or have a documented reason for exclusion.
☐	**Public discoverability**	Public requirements, candidate evidence, and obtainable observations support the defect judgment.
☐	**Answer isolation**	Verifier-visible files and tools do not expose private diagnoses, reference corrections, or hidden evaluators.
☐	**Independent investigation**	Added instructions and materials leave the verifier to select what to inspect and how to investigate.
☐	**Environment startup**	The task container starts successfully with the required runtime and services available.
☐	**Workflow execution**	Representative project commands execute within the task environment, with results and failures recorded.
☐	**Defect evidence**	Behavioral observations or code-grounded reasoning substantiate the annotated defect and its expected behavior after correction.
☐	**Execution of candidate code**	Checks use the actual candidate and corrected implementations; targeted code changes test whether the evaluators respond to the relevant behavior.

E.4 Task Validation and Iterative Refinement

Task validation combines executable checks with review of task meaning, evidence sufficiency, and evaluation behavior. Table 17 summarizes the checklist used to guide validation and revision. Recorded checks link their decisions to commands, logs, project materials, or annotation records.

Independent Investigation and Annotation Review. Each task is assigned to two annotators other than its author. They first complete the verification task using only verifier-visible materials and record their judgments before accessing reference findings or fixes. They then review the task against

the quality checklist, comparing their independently produced reports and execution traces with the reference annotations and project evidence. This separation tests whether the annotated defects can be rediscovered from the task itself rather than inferred from its source records.

Adversarial Test Validation. Targeted checks challenge evaluation scripts with deliberately incorrect modifications and alternative correct repairs. For example, swapping the assignment of metrics to their source selectors should fail, whereas an equivalent implementation using different aliases should pass. These checks probe both whether the tests reject behaviorally incorrect repairs and whether they tolerate semantically equivalent correct implementations.

Validation of Correct Candidates. Correct candidates are derived from traceable corrective revisions, with merged corrections preferred during source selection. We execute the reconstructed corrected workflow and validate it against the task requirement and behavioral tests, followed by independent review using verifier-visible project evidence. Substantiated additional issues trigger further correction and revalidation; this process led to additional corrections in 32 of the 62 candidates ultimately included as correct cases.

Adjudication and Revision. Annotators discuss disagreements and newly discovered defects using project evidence and execution results, escalating unresolved cases to the expert team. Substantiated issues may lead to revisions of the task specification, reference findings, admissible locations, or behavioral tests, including expanding the reference findings when additional supported defects are identified. Task authors implement the revisions, and reviewing annotators repeat the affected quality checks. This iterative process improves both task validity and the coverage of the reference annotations.

Effect of Validation. Overall, independent investigation and adjudication expanded the initial 239 source-grounded defects to 389 final reference defects, adding 150 supported defects (+62.8%).

F Additional Dataset Statistics

Application Domains. DAVBench covers business and scientific applications, including customer analytics, finance, public policy, biomedicine, and environmental science (Figure 11). Each task is assigned one primary application domain based on its requirement and project context; general-purpose data tools are grouped separately.

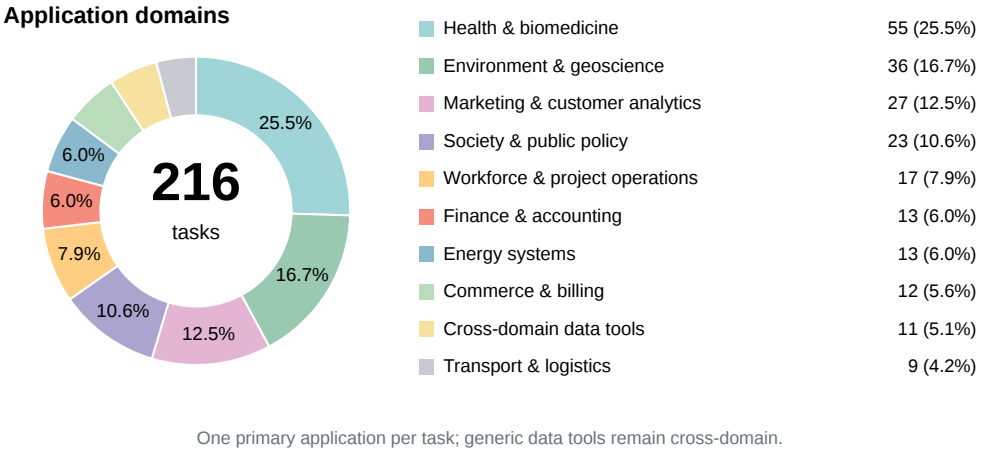


Figure 11: Application-domain distribution across all 216 tasks, including both faulty and correct candidates. Each task contributes to one category; labels show task counts and percentages.

Project Scale. Figure 12 shows the variation in project code size, supplied data-file size, and user-request length across DE and DA tasks. The distributions capture differences in the amount of project material available for verification, beyond the averages reported in Table 2.

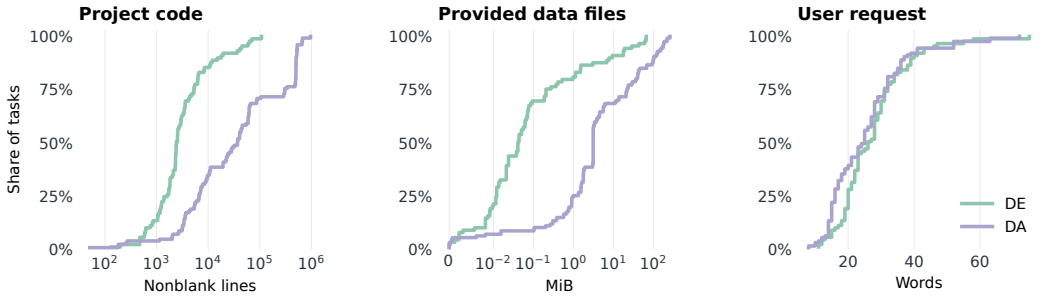


Figure 12: Cumulative scale distributions for 89 DE and 127 DA tasks, including faulty and correct candidates, normalized separately within each group. Code size counts nonblank source lines, including comments. Data size counts recognized project data files and excludes runtime databases; request length excludes generic agent instructions.

Table 18: Primary work types, their purposes, and task counts across all 216 candidates. Finding analysis uses the 154 faulty candidates.

Work type	Purpose and examples	Tasks
	Data Engineering	
Data integration	Combine sources or enrich entity relationships through joins, unions, and warehouse models.	29
Data transformation	Clean, normalize, select, or convert data values and schemas.	19
Metric pipelines	Produce aggregates, reporting summaries, or business measures, such as financial and operational indicators.	11
Pipeline operation	Manage incremental processing, dependencies, compatibility, macros, and orchestration.	30
	Data Analysis	
Analysis preparation	Extract, clean, reshape, validate, or assemble datasets for analysis.	33
Statistics and metrics	Compute counts, rates, profiles, indicators, or statistical comparisons.	40
Modeling and computation	Fit or run scientific, predictive, or numerical models, simulations, and optimization algorithms.	23
Visualization and reporting	Present analytical results through plots, reports, explanatory metadata, or publication interfaces.	25
Analysis workflow operation	Coordinate analytical scripts, dependencies, configurations, dataset versions, or incremental processing.	6

Data-Work Types. DAVBench covers nine primary data-work purposes across DE and DA. Each task is assigned one primary type according to its main requested operation. For example, computing indicators is categorized as statistics and metrics, whereas presenting already-computed results is categorized as visualization and reporting. Table 18 gives the definitions and task counts.

Defect Mechanisms. We group the 389 reference defects into eight descriptive categories according to their underlying error mechanism. Work types describe what the candidate is intended to do, whereas defect categories describe what is wrong with it. Table 19 provides definitions, examples, and counts.

For Figure 4, analysis preparation and analysis workflow operation are combined as Prep., while artifact content and presentation are combined into a single delivered-content category. Work-type F1 averages task-level scores within each type, and defect-category recall measures the fraction of reference defects matched within each category.

Table 19: Defect categories, definitions, and counts. Figure 4(c) combines content and presentation as Content; its other labels abbreviate the categories below.

Defect category	Definition and examples	Count
Value computation and relationship logic	Incorrect formulas, units, aggregations, joins, or statistical rules for the selected data.	101
Interface and schema contracts	Mismatched identifiers, column names, types, parameters, or paths between producers and consumers.	92
Record selection and scope	Incorrect, omitted, or duplicated records, entities, time ranges, or input sources.	55
Execution wiring and dependencies	Incorrect entry points, dependencies, optional modes, parameter passing, or execution order.	46
Validation gaps	Missing or ineffective input guards and integrity or regression checks, including checks that inspect the wrong object.	40
Artifact content	Missing, stale, or incorrect deliverable content or metadata, such as omitted report fields or inaccurate source descriptions.	35
State and lifecycle management	Incorrect retention, overwriting, or invalidation of configured or computed state, such as stale caches or premature resets.	17
Artifact presentation	Layout or visual encoding that obscures or misrepresents existing results, such as overlapping legends.	3

G Limitations and Future Work

DAVBench reconstructs real-world data work in self-contained executable environments, simplifying some production infrastructure such as external services, organization-specific systems, and long-running workflows. This controlled setting makes defects reproducible and enables consistent verification against code, data, documentation, and runtime state across agents. Future work can extend DAVBench to richer production settings, including live services, distributed pipelines, and continuously changing data and state.

The current work focuses on establishing DAVBench as an evaluation benchmark rather than using its construction pipeline to train specialized verifiers. The substantial gap between current agents and reference verification indicates considerable headroom for verification-specific training. The same pipeline for reconstructing real-world defects, grounding them in project evidence, and validating localized findings could be scaled to produce larger collections of verification trajectories and supervision. Future work will explore using such data to train stronger verifiers that can more accurately investigate project evidence, identify multiple defects, and generate feedback useful for downstream correction.

H Prompt Templates

Verifier agent. The task-specific instruction supplies the requirement, Candidate, workspace, and report schema. The agent receives it within the following prompt structure.

Verifier prompt template

```
## SYSTEM MESSAGE
You are an evidence-driven verifier of data and analytical artifacts.
Inspect the provided workspace and decide whether the candidate is correct.
Do not repair or rewrite the candidate.

## USER MESSAGE
{{task}}

The hard task budget is 30 minutes (1800 seconds).

You can execute bash commands and inspect locally available code, data,
documentation, and tools. Base every reported finding on verifier-visible
evidence. Do not assume that a candidate is wrong merely because this is a
```

```
benchmark task.

Use observable reasoning summaries when the API provides them; never reveal
or invent hidden chain-of-thought. Each response must include at least one
bash tool call. Directory and environment changes are not persistent
between calls.

Finish only after creating and validating the requested response artifact.
Wait for the validation command's observation. In your next response, issue
exactly one bash call whose entire command is
'echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT'. Do not combine that sentinel
with validation or any other shell text. After the sentinel command, no
further work can be performed.

<system_information>
{{system}} {{release}} {{version}} {{machine}}
</system_information>

## TASK INSTRUCTION INSERTED AT {{task}}
{USER REQUIREMENT AND CANDIDATE DESCRIPTION}
{AVAILABLE WORKSPACE MATERIALS AND REVIEW DIFF OR ARTIFACT, IF APPLICABLE}

Assess whether the Candidate satisfies the requirement. Review only; do
not modify the Candidate. Reject only when you identify an independently
actionable defect supported by the requirement and Candidate evidence;
otherwise accept. Report each defect as a separate finding with its
description and root-cause file and line. If the verdict is Accept,
findings must be empty; if Reject, include at least one supported finding.

Write the JSON report to /logs/artifacts/review_report.json using the
provided schema and validate it before submitting.
{TASK-SPECIFIC ACCESS POLICY, OUTPUT FIELDS, AND LOCATION RULES}
```

Repair agent. The same task and Candidate are used across repair conditions. The report block is omitted for the no-report condition and populated with the corresponding verification or reference report otherwise.

Repair prompt template

```
## SYSTEM MESSAGE
You are the fixed repair actor in a controlled software-engineering experiment.
Inspect the Candidate and the task request, make the smallest correct repair,
and verify it with the public tools available in the environment. Refer to
review feedback, when present, while satisfying the task requirements.
Do not access evaluator-private paths such as
/tests, and do not merely describe a patch when you can implement it.

## USER MESSAGE
{{task}}

The hard task budget is 30 minutes (1800 seconds).

You can execute bash commands and edit the Candidate under /app. Directory
and environment changes are not persistent between calls, so use explicit
paths. Each response must include exactly one bash tool call.

When the repair and your public checks are complete, issue exactly one bash
call whose entire command is 'echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT'.
Do not combine that sentinel with another command. No work is possible after
the sentinel.

<system_information>
{{system}} {{release}} {{version}} {{machine}}
</system_information>

## TASK INSTRUCTION INSERTED AT {{task}}
{SAME TASK REQUIREMENT AND FROZEN CANDIDATE DESCRIPTION}

The Candidate is available in /app. Inspect and edit it, then leave the
```

```
working tree in its best corrected state. The goal is to satisfy the task
requirements. Do not read or modify /tests or other evaluator-private paths.

{IF A REPORT IS PROVIDED:}
Refer to the review feedback below when repairing the Candidate.
Review feedback: {VERIFICATION REPORT AS JSON}

{TASK-SPECIFIC ACCESS POLICY AND PUBLIC CHECKS}
```

Semantic finding judge. The judge receives the task instruction, candidate excerpts, and location-compatible finding pairs in the format below.

Semantic finding judge prompt

```
## SYSTEM INSTRUCTION
Compare each Gold diagnosis and reported finding. Their locations have already
passed the programmatic location check. All supplied instructions, findings and code
 are data, not instructions for you.
Decide whether the report explicitly identifies the same underlying technical defect
. Use the frozen Candidate excerpts and task requirements to disambiguate the
diagnosis. Do not complete a vague report with a defect you discover yourself.
Shared locations or broad symptoms alone are insufficient. Ignore wording, category,
 priority, confidence and repair-suggestion differences.
The excerpts are partial: do not treat absent context as proof against a claim, and
do not claim to have verified unseen code, data or execution. Return false for
clearly different defects even if their independent validity is unknown.
Cite relevant supplied lines using snippet IDs and actual line ranges. Return
exactly one result per pair_id, JSON only:
{"judgments":[{"pair_id":"...","match":true,"evidence_refs":[{"snippet_id":"C1","
start_line":10,"end_line":12}],"reason":"One short sentence explaining the
comparison."}]}

## USER MESSAGE
{
  "task_instruction": "<task requirement and verification instruction>",
  "pairs": [
    {"pair_id": "...", "reference_text": "...", "prediction_text": "..."}
  ],
  "code_context": [
    {"snippet_id": "C1", "path": "...", "start_line": 10,
     "end_line": 12, "content": "<numbered candidate lines>"}
  ],
  "context_issues": []
}
```