BEYOND THE TEXT: VERIFYING THAT AGENT-WRITTEN PAPERS ARE BACKED BY THEIR ARTIFACTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents are increasingly capable of conducting research autonomously, producing research documents alongside the code and experiments that ostensibly support them. Yet whether the reported findings are consistently supported by corresponding implementations and execution evidence remains largely unexplored. Existing review practices primarily assess textual quality and cannot reliably identify inconsistencies such as hard-coded metrics, unimplemented methods, or unsupported experimental results. We present ReAgent, an automated auditing framework for assessing the consistency between agent-generated research documents and their associated repositories. ReAgent constructs structured representations of scientific claims from research documents and uses them to guide repository analysis and evidence collection. Static auditing examines whether claimed methodologies, implementations, and experimental configurations are consistently reflected in the repository. Dynamic auditing executes relevant experiments and collects execution evidence to assess empirical findings. By combining static analysis with dynamic evidence, ReAgent identifies inconsistencies that may remain hidden under either perspective alone. The collected evidence and audit decisions are organized into a structured repository-level audit report, enabling transparent evidence traceability. We collect and manually annotate a benchmark of agent-generated research document–repository pairs, and evaluate ReAgent against representative static and reproduction-based baselines. Experimental results demonstrate that ReAgent effectively identifies inconsistencies between reported research findings and their supporting repository evidence.

1 INTRODUCTION

In recent years, rapid advances in Large Language Models (LLMs) have significantly expanded the capabilities of intelligent agents (Wang et al., 2024; Jia & Li, 2026). In scientific research, these agents can now perform literature review, hypothesis generation, code implementation, experiment execution, result analysis, and paper writing, covering increasingly large portions of the research pipeline.

Existing studies primarily evaluate whether autonomous research agents can generate novel scientific ideas and high-quality research outputs (Si et al., 2024; Chen et al., 2025a; Wang et al., 2026; Chen et al., 2025c; Tang et al., 2025). As these systems increasingly automate end-to-end scientific workflows, a more fundamental yet largely overlooked question arises:

Are the scientific claims reported by autonomous research agents trustworthy?

This issue is not unique to autonomous research. In traditional scientific practice, inconsistencies between papers and their implementations have long contributed to the reproducibility crisis (Baker, 2016; Gundersen & Kjensmo, 2018; Pineau et al., 2021). Autonomous research further amplifies this challenge because research documents, code, and experiments are often generated and iteratively refined independently. Foundation-model limitations, including hallucination (Ji et al., 2023; Tian et al., 2025), weak alignment between natural language and executable code (Baumgärtner & Gurevych, 2026), and limited self-verification (Huang et al., 2024; Tyen et al., 2024), can cause scientific claims to diverge from their implementations and execution evidence. These characteristics motivate dedicated mechanisms to verify their alignment. **Scientific research is trustwor-**

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.

thy only when its claims are supported by empirical evidence (Wadden et al., 2020; Luo et al., 2025). Methodological claims, experimental protocols, and reported results should be grounded in corresponding implementations, configurations, and execution evidence. Discrepancies arise when proposed methods are absent, results cannot be reproduced, or experimental settings differ from the paper, undermining the credibility of autonomous research outputs even without malicious intent.

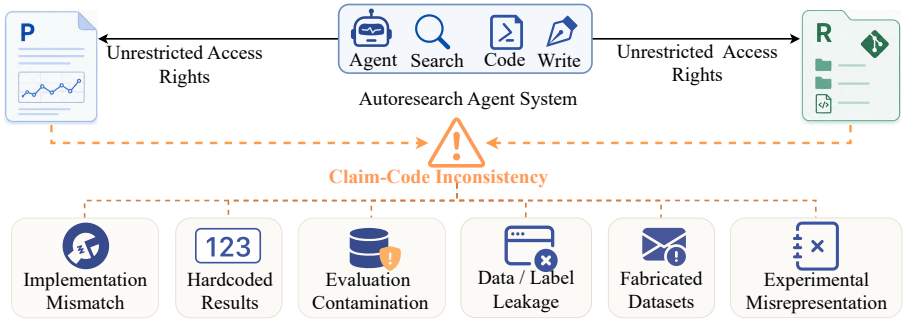

Figure 1: Problem definition. An automated research agent generates a research document P and an associated repository R. During autonomous generation and iterative refinement, claim-code inconsistencies may arise across the research workflow.

Regrettably, existing work has yet to establish a systematic evaluation framework targeting this issue. While recent studies on automated paper replication and autonomous research have advanced scientific automation, they primarily evaluate code quality or peer-review scores (Starace et al., 2025; Chen et al., 2025b; Tang et al., 2025), rather than whether scientific claims are supported by both implementation and experimental evidence. This challenge stems from the intrinsic dual nature of scientific claim verification: **trustworthy assessment requires verifying both that the implementation matches the claimed methodology and that the reported results are supported by executable evidence**. Static analysis can examine whether a repository contains implementations corresponding to described methods, but cannot determine whether reported findings were actually produced by the current implementation. Conversely, dynamic re-execution can verify whether results are reproducible, but cannot establish whether the executed implementation faithfully matches the scientific claims. Therefore, neither perspective alone is sufficient. As illustrated in Figure 1, autonomous research requires a unified framework that jointly reasons over implementation consistency and executable evidence.

This paper introduces ReAgent, an automated framework for auditing the consistency between agent-generated research documents and their associated repositories. Central to the framework are structured claim contracts that connect scientific statements to implementation evidence and executable verification targets. Static auditing checks whether the repository implements the claimed methodology, while dynamic verification tests whether execution supports the reported results. The two stages produce a traceable audit report that distinguishes implementation discrepancies from execution barriers. We evaluate this approach on a manually annotated benchmark of 96 auditable document–repository pairs from AutoSOTA and find that the two verification stages expose complementary risks. Our main contributions are:

- We introduce **Claim-Code Consistency** as a new auditing problem for autonomous research, defining how scientific claims should be verified against both code implementations and executable evidence.
- We propose **ReAgent**, an automated claim verification framework that combines static implementation auditing with dynamic evidence collection to systematically verify research documents and repositories generated by autonomous research agents.
- We build and manually annotate a benchmark from 188 AutoSOTA repositories (Li et al., 2026), yielding 96 auditable document–repository pairs and a taxonomy of static and dynamic inconsistencies for systematic claim–code evaluation.

2 Related Work

Autonomous Research Agents. Recent advances in large language models have enabled autonomous research agents that produce coupled bundles of manuscripts, source code, configurations, logs, and result artifacts (Lu et al., 2026; Schmidgall et al., 2025; Tang et al., 2025; Su et al., 2025; Li et al., 2026; Meng et al., 2026; Liu et al., 2026). Existing evaluations emphasize workflow completion, scientific quality, or productivity, while giving less attention to whether released artifacts faithfully support the claims in the manuscript. We therefore treat the document and repository as auditable objects and verify their consistency at the claim level.

Trustworthy LLM Agents. The increasing autonomy of LLM agents has raised concerns about trustworthiness, including deceptive reasoning, specification gaming, and hidden coordination (Huang et al., 2025; Mathew et al., 2025). In scientific artifacts, the central concern is whether code, configuration, and execution traces actually support the reported claims; failures can arise from context loss or tool errors without deliberate deception. We therefore focus on claim support and evidence provenance rather than behavioral safety during task execution.

AI-assisted Scientific Auditing and Verification. Recent work has employed LLM agents to reconstruct implementations or reproduce reported results, including PaperBench, DiscoveryBench, and ScienceAgentBench (Starace et al., 2025; Majumder et al., 2024; Chen et al., 2025c). These benchmarks primarily measure an agent's ability to explore repositories, set up dependencies, generate or recover code, and reproduce metrics; they do not directly audit whether the agent's released code is consistent with each scientific claim. A similar number may come from a different algorithm or split, while a correct implementation may be blocked by an unavailable checkpoint. Auditing work adapted from Luo et al. (2025) identifies risks such as leakage and benchmark misuse, but does not construct an explicit claim contract or link static findings to target-bound execution evidence. ReAgent combines these forms of evidence while preserving failure causes and provenance.

3 Method

3.1 Problem Definition

Autonomous research agents can now generate both research documents and repositories, but their outputs may contain intentional or accidental inconsistencies. We study these inconsistencies without assuming adversarial behavior, focusing on discrepancies that emerge as agents plan, implement, execute, and revise research workflows. Here, P denotes the textual research output containing scientific claims to be audited, including conventional papers as well as automatically generated research reports, while R denotes the associated repository containing implementation code and experimental configurations. An overview of this problem setting is illustrated in Figure 1. Such inconsistencies can arise unintentionally—from planning errors, execution failures, incomplete implementations, or hallucinated experimental results—or intentionally, such as when an agent optimizes the narrative for persuasiveness without ensuring alignment with the repository.

Assuming the agent possesses unrestricted access to both the research document and the repository, we categorize the resulting inconsistencies into two major classes:

- **Static Repository Inconsistencies:** Discrepancies where the repository fails to align with the claims made in the research document without requiring code execution.
  - Hardcoded Results: Reported figures are hardcoded into scripts or output templates rather than being generated through actual computation.
  - Substantive Implementation Discrepancies: The reported module exists and runs, but its core algorithmic steps diverge substantially from the textual description.
  - Omitted Implementation: Proposed methods or modules mentioned in the research document are absent from the repository.
  - Shell/Dummy Implementation: Modules are defined, but they are short-circuited or never invoked during actual runtime.
  - Fabricated Datasets: The research document claims to use a specific dataset, but the code generates synthetic data or uses a minor subset instead.

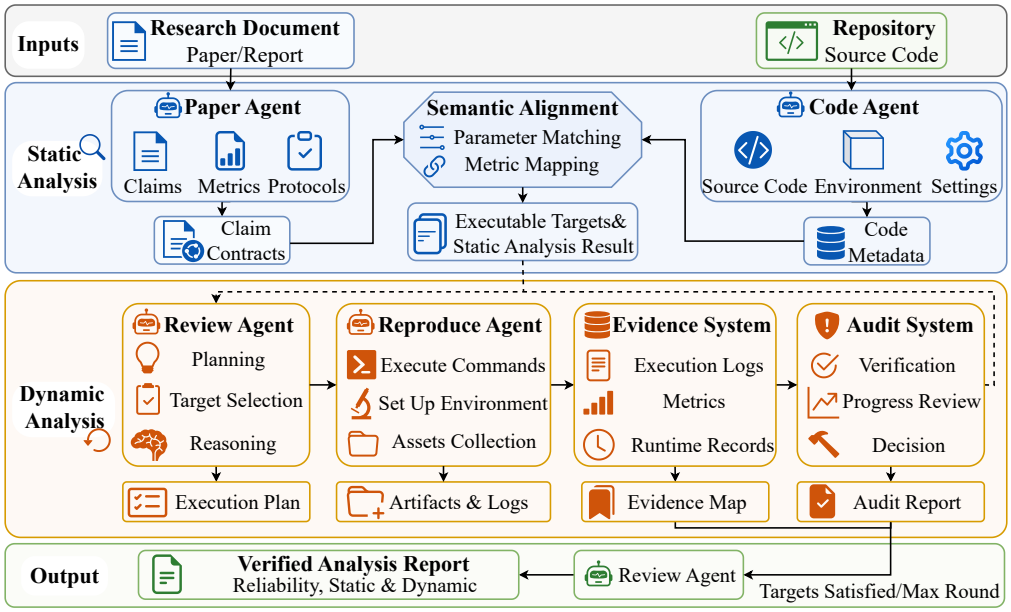

Figure 2: ReAgent framework. Static verification constructs claim contracts, repository metadata, and semantic bindings to align scientific claims with implementation evidence and generate executable verification targets. Dynamic verification performs review-guided reproduction and organizes execution feedback into provenance-aware evidence, enabling evidence-grounded audit decisions.

  - Fabricated Experimental Hyperparameters: Claimed configurations or parameters are either omitted in code or fixed to arbitrary constant values.
  - Data / Label Leakage: The test set, answers, or labels are exposed during training, hyperparameter tuning, or random seed selection.
- **Dynamic Execution Inconsistencies:** Discrepancies between the actual execution outputs and the claims in the research document, typically involving exaggerated performance. Under identical configurations, the executed code yields performance far below the claimed results or fails to execute entirely.

These inconsistencies arise when a reported method, configuration, or result is not supported by the corresponding repository or its execution evidence. Our task is to determine, for a research document P and repository R, whether the claims in P are backed by verifiable evidence in R. We address this task with ReAgent, which combines static claim–code analysis with dynamic execution-based verification.

3.2 Static Verification

As shown in Figure 2, Static Claim Verification performs claim-guided analysis of the correspondence between scientific claims and repository implementations without executing code. It constructs structured claim representations, retrieves and aligns relevant implementation evidence, identifies potential inconsistencies, and establishes executable targets for subsequent dynamic verification. This phase comprises four steps: Claim Contract Construction, Coarse-to-Targeted Repository Analysis, Semantic Binding, and Executable Target Construction. Detailed procedures and schemas are provided in Appendix A.1.

The Paper Agent extracts structured claim contracts from the research document, systematically decoupling verifiable claims into implementation claims checked through code and experimental claims verified through execution. For implementation claims, it records fine-grained component specifications, enabling static analyses of entry-point reachability, structural role, and downstream usage. For experimental claims, heuristic searches isolate empirical content and bind each item to

a dedicated contract identifier together with its source context. Every contract preserves the original supporting text and provenance, while validation enforces substantive grounding and surfaces unresolved fields as evidence gaps rather than silently omitting them.

The Code Agent begins by constructing a lightweight repository manifest and recording the environment, dependency graph, executable entry points, configuration interfaces, and visible result artifacts. It then expands analysis along imports and calls from candidate entry points, tracing the data loaders, evaluators, and result writers that can produce the claimed outputs. Code Metadata records definition, instantiation, call, and output-consumer locations; a reflection pass validates and supplements these records. The analysis stores facts supported by code evidence and does not infer correctness merely from the paper claim.

Semantic binding aligns claim contracts with Code Metadata using normalized identifiers and local execution context, preserving source locations for each correspondence. The Review Agent combines these bindings with claim contracts, Code Metadata, and targeted code inspection to identify static inconsistencies under the predefined risk taxonomy. Each finding must cite concrete code locations and explain the discrepancy with the claim. A programmatic validator checks the report for evidence specificity, completeness, and separation from dynamic failures before accepting it. Insufficient evidence is recorded as an auditing limitation rather than a clean result. These bindings also prepare dynamic verification by connecting experimental claims to candidate execution paths. The system constructs targets that link each claim to its execution setting, required resources, expected results, and evidence requirements; targets without a valid execution path retain an explicit blocker.

3.3 Dynamic Verification

As also illustrated in Figure 2, Dynamic Claim Verification validates empirical claims by executing the repository and collecting execution evidence. Unlike static verification, which examines implementation consistency without execution, this stage evaluates whether experimental claims can be supported through actual execution. Appendix A.1 gives the full execution protocol and stopping criteria. Starting from the executable targets generated in the static stage, the system iteratively plans, executes, and evaluates verification until sufficient evidence is collected or a predefined stopping condition is reached.

For each executable target derived from a claim, the Review Agent receives the target, its unresolved evidence fields, the evidence collected so far, and the remaining budgets. Its planning prompt requests one next action, a command, expected observation, and a stopping condition; it cannot assign the final claim status. The Reproduce Agent then prepares the environment, resolves dependencies, acquires required datasets when available, and launches the corresponding experiment. It may perform only logged compatibility fixes, such as correcting file paths or environment configuration; changes to the algorithm, data split, metric, or evaluation aggregation are prohibited.

During execution, the evidence system maintains two linked paths. The execution-audit path records commands, return codes, runtime logs, callbacks, intermediate artifacts, and generated outputs. The evidence path uses the semantic bindings to select fact windows from those command products; an LLM converts each window into a structured evidence record, a reverse check verifies that the record is grounded in the original output, and a separate rule-based audit checks whether it advances the intended executable target. The records retain execution commands, model configurations, datasets, timestamps, artifact locations, and source hashes, so every verification result can be traced back to its origin.

After each round, a deterministic round-audit procedure checks whether new traces or artifacts satisfy the target evidence contract and whether they are bound to the intended code path. It marks the target as satisfied, unsatisfied, progressed, not checkable, or hard blocked, and records the gap and blocker reason. The audit preserves specific failure causes to distinguish execution barriers from contradictions of the claim. The planner may then refine the search or select another permitted execution path; it cannot close a target without the required evidence. Verification stops when every target is closed, a claim is contradicted, a blocker cannot be recovered within budget, or the global round/time budget is exhausted (Appendix A.1.6).

Table 1: Repositories excluded during benchmark corpus construction.

Reason	Description	Count
Missing report	Report unavailable	3
Missing repository	Source code unavailable	10
Unavailable assets	Dataset or checkpoint unavailable	79

Table 2: Static issues found by manual auditing.

Issue Type	Count
Substantive Implementation Discrepancies	10
Data / Label Leakage	7
Fabricated Experimental Hyperparameters	4
Omitted Implementation	4
Shell/Dummy Implementation	1
Hardcoded Results	1
Fabricated Datasets	1
Others	6

The output of this stage is a collection of execution-based verification results together with claim-level evidence, which are subsequently integrated with the static verification results during the final assessment.

3.4 Final Assessment

Finally, static verification results and execution evidence are jointly considered to produce the final audit assessment. Static verification identifies potential implementation-level inconsistencies, while dynamic verification provides execution-based evidence for empirical claims. For empirically verifiable claims, the system assigns dynamic verification outcomes, including *Matched*, *Mismatched*, *Blocked*, or *Not Reproduced*, together with supporting evidence. These findings are aggregated into a repository-level audit report, providing an overall assessment of artifact trustworthiness with complete evidence traceability.

4 Experiments

We evaluate whether ReAgent can reliably audit claim-code consistency in research artifacts generated by automated research agents. Our evaluation addresses three research questions:

- **RQ1.** How accurately can ReAgent audit claim-code consistency compared with existing approaches?
- **RQ2.** What types of unsupported claims commonly appear in agent-generated research artifacts?
- **RQ3.** How does ReAgent perform on representative real-world auditing cases?

4.1 Experimental Setup

Our evaluation consists of three components: the evaluation corpus, the claim auditing protocol for constructing ground truth, and comparison methods. Additional implementation details are provided in the supplementary material.

4.1.1 Evaluation Corpus

We construct our benchmark from AutoSOTA (Li et al., 2026), which releases 188 repositories generated by automated research agents. Each repository contains an agent-generated report, source code, and experimental results. We assess the auditability of all 188 repositories and identify 92 repositories with insufficient evidence for reliable verification. These cases are treated as *unverifiable* rather than inconsistent, leaving 96 auditable repositories for our analysis (Table 1). The filtering procedure and inclusion criteria are detailed in Appendix A.3.2. Because this filtering may introduce selection bias, our reported inconsistency rates characterize the auditable subset rather than the full collection.

Table 3: Verification-stage ablation.

Ablation setting	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Dynamic only	82.29	90.91	84.51	87.59
Static only	50.00	79.49	43.66	56.36
Dynamic + Static	83.33	83.95	95.77	89.47

Table 4: Comparison of auditing performance; Acc. denotes Accuracy.

Method	Static			Reproduction	Dynamic	Overall
	Acc. (%)	Recall (%)	F1 (%)	Success Rate (%)	Acc. (%)	Acc. (%)
Static-Audit	31.25	100.00	45.90	–	–	–
Codex	41.67	67.86	40.43	60.42	79.17	35.42
ReAgent	78.13	82.14	68.66	76.04	90.63	70.83

4.1.2 Comparison Methods

We compare ReAgent with two baselines. All methods use GPT-5.4 (OpenAI, 2026b) as the underlying model. Since different methods adopt different risk taxonomies, we evaluate them under a unified repository-level binary detection setting: a repository is positive if it contains at least one consistency issue.

Luo et al. (2025) propose an auditing framework for automated scientific discovery that detects implementation issues such as data leakage, benchmark misuse, and methodological flaws. Since their framework does not explicitly verify claim-code consistency, we adapt it as a static verification baseline, referred to as Static-Audit.

Codex (OpenAI, 2026a) is a general-purpose coding agent with strong software engineering capabilities. It receives the same paper, repository, task description, and execution environment as ReAgent.

4.1.3 Evaluation Protocol

We manually audit all repositories to establish ground truth; the candidate-discovery, adjudication, and annotation procedure is described in Appendix A.2. For static verification, Claude Code is used only for candidate issue discovery. Three Claude models independently analyze each repository using the same risk taxonomy and return file, function, and line evidence. Three authors then verify every candidate against the paper, implementation, and experimental configuration; a candidate is retained only when its code evidence supports the assigned category. Disagreements are resolved through discussion, and repositories without detected issues are additionally inspected to reduce false negatives.

Execution traces are manually examined to verify whether reported results are supported by actual execution. Failed reproductions are further analyzed, with manual reproduction performed when necessary.

4.2 RQ1: How accurately can ReAgent audit claim-code consistency?

RQ1 evaluates whether ReAgent can correctly audit agent-generated repositories. We compare the predicted auditing results against the manually established ground truth described in Section 4.1.
Metrics.

Static Verification Accuracy measures the proportion of repositories whose static verification decisions agree with manual annotations. The static verification task is formulated as a binary classification problem, where the system predicts whether a repository contains static claim-code inconsistencies. We additionally report Recall and F1-score due to the imbalanced distribution of inconsistent repositories.

Reproduction Success Rate measures the proportion of repositories for which the auditing system successfully establishes an executable environment and completes experiment execution.

Dynamic Verification Accuracy measures the proportion of repositories for which the binary dynamic verification decisions agree with the manually established ground truth. Since our objective is to determine whether a scientific claim is supported by executable evidence, only Matched is regarded as a positive outcome. Mismatched, Blocked, and Not Reproduced are conservatively mapped to the negative class because none of them provides sufficient evidence to support the reported claim. Nevertheless, these outcomes are preserved as diagnostic labels in the audit report to differentiate distinct causes of verification failure.

Overall Audit Accuracy measures the proportion of repositories for which both static verification and dynamic verification decisions agree with manual annotations. The metric is computed over all evaluated repositories.

As shown in Table 4, ReAgent consistently outperforms existing methods across all evaluation dimensions. For static verification, ReAgent achieves 78.13% accuracy, 82.14% recall, and 68.66% F1-score, substantially improving over Codex and Static-Audit. In particular, compared with Static-Audit, which achieves high recall but a low F1-score, ReAgent provides a better balance between identifying inconsistent repositories and avoiding false alarms. For dynamic verification, ReAgent achieves an accuracy of 90.63% over all evaluated repositories, outperforming Codex by 11.46 percentage points. The improvement mainly comes from our execution-oriented verification pipeline, which converts empirical claims into executable verification targets and collects execution evidence through an integrated auditing and evidence management process. By grounding verification decisions on experimental evidence rather than textual reasoning alone, ReAgent reduces unreliable judgments caused by incomplete or hallucinated evidence.

Overall, ReAgent achieves 70.83% audit accuracy, improving over Codex by 35.41 percentage points. The result reflects the benefit of combining implementation evidence with target-bound execution traces.

Verification-stage ablation. We first combine the manually annotated static and dynamic ground truths to define a common repository-level risk label: a repository is no-risk only if static auditing finds no risk and dynamic verification successfully reproduces its reported result. This yields 25 no-risk repositories and 71 risk repositories. Table 3 treats risk repositories as the positive class and compares static verification alone, dynamic verification alone, and their combination against these labels. The combined prediction is no-risk only when both stages pass. Combining the two stages detects more potential risks, achieving a recall of 95.77% compared with 84.51% for dynamic verification alone and 43.66% for static verification alone. It also achieves higher accuracy (83.33%) than either stage alone (82.29% and 50.00%, respectively), although precision decreases relative to dynamic verification alone.

4.3 RQ2: What consistency issues are prevalent in agent-generated research artifacts?

We next analyze what types of unsupported scientific claims are commonly produced by autonomous research agents. Unlike RQ1, which evaluates auditing accuracy, this analysis focuses on the characteristics of the detected inconsistencies.

Among the 96 repositories, 28 contain at least one static consistency issue. Since one repository may contain multiple issues, category counts do not sum to the number of affected repositories. Table 2 summarizes the distribution: the most common issues are Substantive Implementation Discrepancies (10) and Data / Label Leakage (7), suggesting that autonomous research agents may modify implementations for performance improvement without faithfully documenting these changes. Less frequent patterns are grouped under Others; detailed examples and their evidence are reported in Appendix A.2.1.

Dynamic auditing reproduces the reported metrics for 33 repositories, but only 25 also pass static consistency checks. The remaining eight cases show why execution alone is insufficient: their metrics are recovered despite unsupported implementations, and static verification detects seven of them.

4.4 RQ3: How does ReAgent perform on representative real-world auditing cases?

4.4.1 Static verification identifies implementation inconsistencies

One repository claims that propensity-feature augmentation with robust ensembling improves IHDP PEHE from 0.1829 to 0.1539. Static verification revealed that the evaluation pipeline implemented an S-learner augmentation strategy rather than the claimed propensity-feature augmentation, indicating a substantive implementation discrepancy. This case shows that execution alone may miss inconsistencies when the repository remains runnable but deviates from the reported methodology.

4.4.2 Dynamic verification avoids false negatives

Another repository reports that Anchorage AUROC improves from 99.31 to 99.42 after twelve optimization iterations. ReAgent reproduced the official evaluation pipeline and obtained the reported value of 99.42. In contrast, Codex used a simplified reproduction strategy due to computational constraints and incorrectly concluded that the improvement was unverifiable, highlighting the importance of faithful execution for reliable auditing.

4.4.3 Dynamic verification distinguishes blocked reproduction

Beyond binary claim-code consistency classification, dynamic verification further distinguishes genuine claim-code inconsistencies from cases where reproduction is blocked by missing or unusable execution artifacts.

Another repository reports a PSNR of 32.24 obtained from its official seven-seed optimization pipeline. Static verification confirmed that the documented workflow and implementation were consistent with the reported methodology. However, during dynamic verification, ReAgent could only execute the evaluation stage, reproducing a PSNR of 30.66 instead of the reported 32.24. Further investigation revealed that the complete generation pipeline failed because the required model checkpoint (`weights/PFCNet.pth`) was an invalid HTML page rather than a valid PyTorch checkpoint. ReAgent therefore classified this case as *Blocked* rather than *Mismatched*, distinguishing missing executable artifacts from genuine claim-code inconsistencies.

5 Discussion and Limitations

Claim-code inconsistency can arise during iterative agent revisions even without intentional fabrication, which motivates evidence-grounded auditing. ReAgent does not assess scientific validity or novelty, and execution may be blocked by missing assets, obsolete dependencies, or computational limits. The AutoSOTA-only benchmark may also introduce distribution and survivorship bias. These limitations motivate evaluation on broader benchmarks with independently maintained artifacts and richer provenance records.

6 Conclusion

This paper presents ReAgent, an automated framework that combines claim-guided static verification with execution-based dynamic verification. Structured claim contracts and provenance-aware traces test whether agent-generated documents are supported by their repositories. On 96 manually audited repositories, ReAgent achieves 70.83% overall audit accuracy, compared with 35.42% for Codex. Static checks and dynamic verification expose complementary risks. Recovering a reported metric alone does not establish that the claimed method was implemented. The audit also preserves failure causes, separating blocked execution from measured contradictions. These results support auditing both implementations and execution evidence when assessing agent-generated research. Future work will evaluate broader domains and richer evidence sources.

AI USE STATEMENT

We used generative AI tools to assist with manuscript language editing, LaTeX formatting, experiment-audit support, and candidate dataset-risk labeling. The authors manually reviewed and verified all AI-assisted text, claims, analyses, labels, and artifacts, and take responsibility for the final content.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide the implementation of ReAgent together with the claim-contract schema, audit prompts, risk taxonomy, execution policies, runtime budgets, filtering criteria, and benchmark-construction protocol used in this study. The reported results are computed from fixed repository selections, evaluation labels, and decision rules described in the paper and appendix. Reproduction may require external datasets, checkpoints, or system dependencies that are not redistributable or are no longer available; such cases are recorded as explicit *Blocked* outcomes rather than repaired by changing the claimed method or evaluation.

ETHICS STATEMENT

This work analyzes publicly released research documents and code repositories to study claim–code consistency. We do not collect personal data, interact with human participants, or make decisions about individuals. Repository execution is performed in isolated environments under logged, least-privilege policies; network access and file modifications are constrained, and any exposed credentials or sensitive artifacts encountered during auditing are not disclosed. The benchmark and annotations are used to evaluate auditing methods, not to attribute misconduct to individual authors, and we report reproduction failures as evidence limitations when missing assets or dependencies prevent verification.

REFERENCES

Monya Baker. 1,500 scientists lift the lid on reproducibility, 2016.

Tim Baumgärtner and Iryna Gurevych. Scicoqa: Quality assurance for scientific paper-code alignment. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 38740–38770. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.acl-long.1795/.

Hui Chen, Miao Xiong, Yujie Lu, Wei Han, Ailin Deng, Yufei He, Jiaying Wu, Yibo Li, Yue Liu, and Bryan Hooi. Mlr-bench: Evaluating ai agents on open-ended machine learning research, 2025a. URL https://arxiv.org/abs/2505.19955.

Pengcheng Chen, Ning Yan, Zihan Zhao, Yixiao Lin, Huaibo Chen, Yue Hu, Qinbo Bai, Xiang Li, and Masood S Mortazavi. Deep-reproducer: From paper understanding to code generation. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025b.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025c. URL https://openreview.net/forum?id=6z4YKr0GK6.

Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In Sheila A. McIlraith and Kilian Q. Weinberger (eds.), *Proceedings of the Thirty-Second*

AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pp. 1644–1651. AAAI Press, 2018. doi: 10.1609/AAAI.V32I1.11503. URL https://doi.org/10.1609/aaai.v32i1.11503.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Yao Huang, Yitong Sun, Yichi Zhang, Ruochen Zhang, Yinpeng Dong, and Xingxing Wei. Deceptionbench: A comprehensive benchmark for ai deception behaviors in real-world scenarios. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/55494d8756b72c2219027edc9de1ee5a-Paper-Datasets_and_Benchmarks_Track.pdf.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 55(12):248:1–248:38, 2023. doi: 10.1145/3571730. URL https://doi.org/10.1145/3571730.

Jingyi Jia and Qinbin Li. Autotool: Efficient tool selection for large language model agents. In Sven Koenig, Chad Jenkins, and Matthew E. Taylor (eds.), *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*, pp. 31265–31273. AAAI Press, 2026. doi: 10.1609/AAAI.V40I37.40389. URL https://doi.org/10.1609/aaai.v40i37.40389.

Yu Li, Chenyang Shao, Xinyang Liu, Ruotong Zhao, Peijie Liu, Hongyuan Su, Zhibin Chen, Qinglong Yang, Anjie Xu, Yi Fang, Qingbin Zeng, Tianxing Li, Jingbo Xu, Fengli Xu, Yong Li, and Tie-Yan Liu. Autosota: An end-to-end automated research system for state-of-the-art ai model discovery, 2026. URL https://arxiv.org/abs/2604.05550.

Jiaqi Liu, Shi Qiu, Mairui Li, Bingzhou Li, Haonian Ji, Siwei Han, Xinyu Ye, Peng Xia, Zihan Dong, Meng Chen, Congyu Zhang, Letian Zhang, Guiming Chen, Haoqin Tu, Xinyu Yang, Lu Feng, Xujiang Zhao, Haifeng Chen, Jiawei Zhou, Xiao Wang, Weitong Zhang, Hongtu Zhu, Yun Li, Jieru Mei, Hongliang Fei, Jiaheng Zhang, Linjie Li, Linjun Zhang, Yuyin Zhou, Sheng Wang, Caiming Xiong, James Zou, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Autoresearchclaw: Self-reinforcing autonomous research with human-ai collaboration, 2026. URL https://arxiv.org/abs/2605.20025.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of ai research. *Nature*, 651(8107):914, 2026.

Ziming Luo, Atoosa Kasirzadeh, and Nihar B. Shah. The more you automate, the less you see: Hidden pitfalls of ai scientist systems, 2025. URL https://arxiv.org/abs/2509.08713.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi Mishra, Abhijeetsingh Meena, Aryan Prakhar, Tirth Vora, Tushar Khot, Ashish Sabharwal, and Peter Clark. Discoverybench: Towards data-driven discovery with large language models, 2024. URL https://arxiv.org/abs/2407.01725.

Yohan Mathew, Ollie Matthews, Robert McCarthy, Joan Velja, Christian Schröder de Witt, Dylan Cope, and Nandi Schoots. Hidden in plain text: Emergence & mitigation of steganographic collusion in llms. In Kentaro Inui, Sakriani Sakti, Haofen Wang, Derek F. Wong, Pushpak Bhattacharyya, Biplab Banerjee, Asif Ekbal, Tanmoy Chakraborty, and Dhirendra Pratap Singh (eds.), *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, IJCNLP-AACL 2025, Mumbai, India, December 20-24, 2025*, pp. 585–624. The Asian Federation

of Natural Language Processing and The Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.IJCNLP-LONG.34. URL https://doi.org/10.18653/v1/2025.ijcnlp-long.34.

Rui Meng, Bhavana Dalvi Mishra, Jiefeng Chen, Chun-Liang Li, Palash Goyal, Mihir Parmar, Yiwen Song, Yale Song, Rajarishi Sinha, Parthasarathy Ranganathan, Burak Gokturk, Jinsung Yoon, and Tomas Pfister. Scientistone: Towards human-level autonomous research via chain-of-evidence, 2026. URL https://arxiv.org/abs/2605.26340.

OpenAI. OpenAI Codex: The Next Era of Knowledge Work and Code Generation. https://openai.com/index/codex-for-knowledge-work/, 2026a. Accessed: 2026-07-29.

OpenAI. GPT-5.4: Technical Report and Model Card. https://openai.com/index/introducing-gpt-5-4/, 2026b. Accessed: 2026-07-29.

Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *Journal of machine learning research*, 22(164):1–20, 2021.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using LLM agents as research assistants. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pp. 5977–6043. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-EMNLP.320. URL https://doi.org/10.18653/v1/2025.findings-emnlp.320.

Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. Can llms generate novel research ideas? a large-scale human study with 100+ nlp researchers, 2024. URL https://arxiv.org/abs/2409.04109.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate ai research, 2025. URL https://arxiv.org/abs/2504.01848.

Haoyang Su, Renqi Chen, Shixiang Tang, Zhenfei Yin, Xinzhe Zheng, Jinzhe Li, Biqing Qi, Qi Wu, Hui Li, Wanli Ouyang, Philip Torr, Bowen Zhou, and Nanqing Dong. Many heads are better than one: Improved scientific idea generation by A llm-based multi-agent system. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 28201–28240. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.ACL-LONG.1368. URL https://doi.org/10.18653/v1/2025.acl-long.1368.

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. Ai-researcher: Autonomous scientific innovation. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 9481–9520. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/0d904d300a105809a2114d727851e759-Paper-Conference.pdf.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. Codehalu: Investigating code hallucinations in llms via execution-based verification. In Toby Walsh, Julie Shah, and Zico Kolter (eds.), *Thirty-Ninth AAAI Conference on Artificial Intelligence, Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence, Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2025, Philadelphia, PA, USA, February 25 - March 4, 2025*, pp. 25300–25308. AAAI Press, 2025. doi: 10.1609/AAAI.V39I24.34717. URL https://doi.org/10.1609/aaai.v39i24.34717.

Gladys Tyen, Hassan Mansoor, Victor Carbune, Peter Chen, and Tony Mak. Llms cannot find reasoning errors, but can correct them given the error location. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, volume ACL 2024 of *Findings of ACL*, pp. 13894–13908. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.826. URL https://doi.org/10.18653/v1/2024.findings-acl.826.

David Wadden, Shanchuan Lin, Kyle Lo, Lucy Lu Wang, Madeleine van Zuylen, Arman Cohan, and Hannaneh Hajishirzi. Fact or fiction: Verifying scientific claims. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 7534–7550, 2020.

Jiayu Wang, Weijiang Lv, Bowen Fu, Jing Fu, Jiayi Song, Lingyu Zhang, Lanxuan Xue, Luodi Chen, Zepeng Xin, Kaiyu Li, and Xiangyong Cao. Act as a real researcher: A suite of benchmarks evaluating frontier llms and agentic harnesses in research lifecycle, 2026. URL https://arxiv.org/abs/2606.07462.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

A Appendix

A.1 Method Details

ReAgent follows a sequential auditing pipeline. It converts the research document into structured claim contracts, associates these contracts with repository metadata through semantic binding, and asks the Review Agent whether the static evidence is sufficient. Unresolved targets are passed to the Reproduce Agent, which executes the repository and collects additional evidence until a final decision can be made.

A.1.1 Claim Contract

The paper and repository are unstructured and can contain many pages and files. ReAgent therefore transforms each auditable statement into a structured claim contract, which is the unit used for semantic binding, target construction, execution, and final auditing.

Claim contract example

```
{
  "claim_id": "C17",
  "claim_type": "performance",
  "paper_id": "paper-10-SuitabilityFilter",
  "paper_section": "main_text",
  "statement": "The optimized method reaches an OOD suitability score
  → of 0.9838.",
  "method": "autosota_optimized",
  "baseline": "autosota_reported_baseline",
  "dataset": "autosota_public_eval",
  "split": "test",
  "expected_metric": "OOD suitability score",
  "expected_value": 0.9838,
  "metric_direction": "higher_is_better",
  "evidence": null,
  "source_provenance": {
    "section": "main_text",
    "table_id": "paper-10-SuitabilityFilter_autosota_optimized_metric"
  }
}
```

The core fields retained in each claim contract are summarized in Table 5. The extraction prompt asks for claims that can be checked independently and supplies the allowed claim types, the required JSON schema, and the instruction to copy the smallest complete text span. A claim is created when a sentence or table cell asserts a method, dataset, configuration, comparison, or numerical result; purely motivational or speculative text is ignored. Regular expressions identify numerical values, units, and comparison operators before the LLM fills the semantic fields. The parser checks that every value has a metric and provenance, normalizes percentages and scientific notation, and sends malformed records through one repair pass. Claims with unresolved fields remain in the contract with an explicit "unknown" value, so missing information becomes an auditable gap rather than an implicit assumption.

The extraction prompt is: "Extract independently testable method, dataset, configuration, comparison, and performance claims from this segment. Copy the minimal supporting span, fill the claim schema, preserve exact values and provenance, and return an empty list when no claim is present. Do not infer omitted settings."

A claim contract does not itself determine whether a claim is supported; instead, it explicitly specifies what should be audited. For example, in `paper-10-SuitabilityFilter`, the contract binds the reported `OOD suitability score`, the reported value of 0.9838, the baseline score of 0.9687, and the corresponding public evaluation setting into a single auditing target. If execution later produces a score of 0.9834, the system can unambiguously determine which reported claim this observation should be compared against.

A.1.2 Coarse-to-Targeted Repository Analysis

Repository analysis begins with a deterministic manifest. ReAgent records the file tree, language, package and environment declarations, executable scripts, configuration files, result files, and references to external assets. It then searches normalized claim terms, aliases, metric names, and entry-point names in this manifest and in source text. A candidate file is retained when it contains a match or is imported by a matched file; the targeted pass follows imports and calls from the candidate entry point to the data loader, evaluator, and result writer. Each selected region is summarized with its path, line span, symbol name, and content hash.

The search is deliberately shallow at first and expands only when the current evidence contract remains open. README-only matches, generated files, and result files without a producing command are marked as weak evidence. Common failure causes include a missing report or source tree, an entry point that cannot be resolved, a parser failure, a result file with no producer, and an external asset whose location or format is unavailable. These cases are emitted as typed gaps so that the Review Agent can request a targeted execution attempt instead of treating a lexical match as implementation support.

A.1.3 Semantic Binding

Once claim contracts have been constructed, the next challenge is to associate paper-side claims with repository-side evidence. Semantic binding addresses this problem. Rather than relying solely on string matching, embedding retrieval, or BM25 search, ReAgent combines rule-based normalization with LLM-assisted contextual reasoning to establish semantic correspondences between terminology used in the paper and implementation artifacts in the repository.

For parameter matching, identifiers such as `lr`, `learning_rate`, `Learning Rate`, and `--learning-rate` are first normalized into candidate equivalents. Similarly, `batchsize`, `batch_size`, and `batch-size` are treated as potential aliases. The LLM then determines, using local contextual information, whether these candidates indeed refer to the same experimental configuration. When multiple candidates exist within a repository, ReAgent prioritizes objects that co-occur with the target entry point, dataset, method, or evaluation metric, rather than simply selecting the highest lexical similarity.

Metric matching additionally verifies metric direction, aggregation semantics, dataset split, and output location. For example, a reported `Accuracy` may correspond to repository outputs such as `acc`, `top1`, or `test_acc`; however, a successful binding requires that the evaluation split, top-k

Table 5: Core fields of a claim contract.

Field	Description
claim_id	Unique identifier of the claim
claim_type	Claim category (e.g., method, performance, configuration, dataset)
statement	Natural-language claim extracted from the paper
method	Method or model referenced by the claim
baseline	Baseline used for comparison
dataset	Dataset or evaluation setting associated with the claim
expected_metric	Reported evaluation metric
expected_value	Reported numerical value
metric_direction	Whether higher or lower values indicate better performance
source_provenance	Source location of the claim within the paper

Table 6: Typical objects involved in semantic binding.

Object	Paper side	Repository side
Parameter	learning rate, batch size	configuration key, CLI argument, default value
Metric	Accuracy, AUROC, MSE	stdout key, evaluator output, result file
Dataset	CIFAR-10 test split	data path, data loader, split file
Method	proposed module	class, function, training pipeline
Asset	checkpoint or feature cache	local file, download link, bundled artifact

definition, and aggregation procedure are also consistent. Likewise, if the paper reports the average, best, or final performance, the repository output must correspond to the same aggregation semantics.

Typical paper–repository objects used in semantic binding are summarized in Table 6. Each semantic binding produces a structured record containing the paper_term, repo_term, binding_type, confidence, and rationale. These bindings do not constitute the final audit decision. Instead, they serve as inputs to target construction, enabling the system to identify which code paths, metric outputs, data assets, or configuration items require subsequent dynamic verification.

The binding prompt contains one claim contract, the candidate repository objects with short code windows, and the candidate entry-point context. It asks the LLM to return one of supports, contradicts, or ambiguous, together with the exact file and line evidence and a short rationale: "Given the claim and these candidate objects, decide whether each object supports, contradicts, or is ambiguous for the claim. Check reachability, split, direction, aggregation, and override order; cite only supplied file and line evidence." Rule-based normalization supplies candidates but never resolves an ambiguous tie. A binding is considered strong only when the candidate is reachable from the selected entry point and agrees on the dataset split, metric direction, and aggregation. Otherwise the record is retained with low confidence and a target gap for execution.

An important design principle of semantic binding is that uncertainty is explicitly preserved. If a paper-side term can only be matched to a README description but cannot be traced to the actual execution path, the resulting binding is not treated as strong supporting evidence. Likewise, if a repository object exists but its default value, override order, or execution entry point differs from the paper, the binding is marked as a target gap and deferred to the dynamic verification stage for further validation.

A.1.4 Round Audit

After semantic binding establishes candidate relationships between paper claims and repository evidence, the system needs to determine after each execution round whether the newly collected evidence satisfies the current target's evidence contract, and whether the target should be progressed, closed, blocked, or further recovered in the next round. This process is handled by the round audit module rather than being freely decided by the Review Agent.

Table 7: Core target states updated by round audit.

Target status	Meaning
`satisfied`	Required evidence for the current target is fully closed.
`unsatisfied`	Evidence indicates that target requirements are not met or major gaps remain.
`progressed`	New valid evidence is collected, but the target is not yet closed.
`not_checkable`	Required assets, entry points, permissions, or other prerequisites are unavailable.
`hard_blocked`	The blocker is identified and cannot be resolved within the allowed scope or budget.

The Review Agent is responsible for proposing the next verification plan, while the round audit module converts execution traces, callbacks, patch records, and evidence maps into structured and traceable target state updates.

The target states produced by round audit are summarized in Table 7. The inputs of round audit include the current target specification, round audit contract, callbacks, runtime records, artifact records, patch records, and the existing evidence state. It first checks whether the round produces new evidence required by the contract, such as additional execution traces, metric artifacts, failure logs, asset availability evidence, or patch provenance. It then verifies whether the collected evidence is actually bound to the current target rather than originating from unrelated execution paths. Finally, it updates the target gap, supporting evidence references, blocker reasons, and closure conditions for subsequent rounds.

This design separates planning from auditing. The Review Agent can propose subsequent actions based on the current verification state, but it cannot independently determine whether a target is supported or unsupported. Target closure is determined by round audit based on structured evidence and the predefined evidence contract. The final assessment then aggregates target-level states into claim-level or repository-level decisions.

The planning prompt is intentionally target-bound. It contains the claim contract, the current binding records, unresolved evidence fields, prior round outcomes, allowed modification policy, and remaining budgets. It asks for a single next action, command or inspection scope, expected observation, and stop condition. The planner must cite the evidence gap that motivates the action and may select an alternative path only when it tests the same target. This prevents a convenient but unrelated successful run from closing the claim.

A.1.5 Reproduce Agent Constraints

When static evidence is insufficient, ReAgent executes the repository to collect additional evidence. Dynamic execution, however, introduces another challenge: if the agent silently modifies the algorithm or evaluation logic to reproduce the reported results, the resulting evidence can no longer be used to verify the original scientific claim. The capabilities of the Reproduce Agent must therefore be explicitly constrained.

In our implementation, the Reproduce Agent has write access only to an isolated execution workspace and a mutable checkout of the repository. The original source repository and protected system directories remain read-only to preserve provenance and auditing integrity. All file modifications and command executions are checked against predefined policies and recorded in audit logs together with patch records.

The default modification policy is summarized in Table 8. Under the default policy, at most two file-level modifications are permitted during reproduction. Each patch record includes the iteration number, file path, modification type, summary, rationale, pre- and post-modification hashes, a diff excerpt, execution impact, and an assigned risk level. These records are subsequently incorporated into the final reliability assessment. Results obtained without any modifications are regarded as the most directly comparable to the original repository. Results requiring only path or configuration fixes remain comparable but are reported with appropriate caveats. In contrast, if the reproduction process modifies algorithmic or evaluation logic, the resulting evidence is no longer treated as strong support for the original scientific claim.

Table 8: Default modification policy for the Reproduce Agent.

Change type	**Default status**	**Example**
`config_or_path_fix`	Allowed	Adapting file paths, configuration files, or working directories.
`build_or_script_fix`	Allowed	Repairing build scripts or shell entry points.
`entrypoint_exposure_fix`	Allowed	Exposing existing training, evaluation, or main entry points.
`runtime_recovery_source_fix`	Allowed	Minor import fixes or compatibility shims.
`algorithmic_logic_change`	Forbidden	Modifying models, loss functions, optimizers, or training loops.
`metric_or_eval_change`	Forbidden	Modifying metric definitions, dataset selection, or evaluation aggregation.
`source_code_change`	High risk	General source-code modifications outside the approved low-risk categories.

Table 9: Primary stopping conditions for iterative verification.

Condition	Action
Target metric, evaluation setting, and provenance are all resolved	Mark as Matched/Satisfied
Execution evidence contradicts the reported claim	Mark as Mismatched/Unsatisfied
Required public assets or authorization credentials are unavailable	Mark as Blocked/Not Checkable
The same blocker recurs without producing new evidence	Stop attempting recovery for the current target
Iteration or modification budget is exhausted	Return a Partial/Progressed/Blocked status
Only policy-restricted or unschedulable targets remain	Finalize the audit

A.1.6 Iterative Verification

After each execution round, the Review Agent reassesses the collected evidence. Depending on the audit status, it may conclude that the available evidence is sufficient, identify a recoverable blocker, or request additional execution. Consequently, the dynamic verification stage naturally forms an iterative review–reproduction loop. Although each iteration relies on the Reproduce Agent, the loop itself is fundamentally an *evidence-closure protocol*: every round must either produce new auditable evidence or explicitly explain why no further progress can be made.

For each round, the auditor selects the target with the largest unresolved evidence gap, emits the plan and local budgets, executes only the target-bound command, records commands, artifacts, logs, and patches, and runs the round audit. A round must either add a new evidence record or produce an explicit blocker; repeated output with no new record is treated as non-progress. The loop stops when the target is supported, contradicted, blocked, or outside the remaining budget.

The primary stopping conditions are summarized in Table 9. By default, the review–reproduce loop runs for at most six iterations, although this limit can be adjusted through the `MVP_REVIEW_MAX_ROUNDS` parameter. Each review decision additionally specifies local execution budgets, including `retry_budget`=5, `search_budget`=5, and `modification_budget`=2 by default. The round audit contract further constrains the verification process by defining the allowed evidence classes, required new records, success criteria, acceptable blocker types, and forbidden evidence patterns.

This protocol constrains the agent to evidence collection rather than unrestricted repository modification. A failed execution does not necessarily imply that a claim is unsupported; it may simply reveal missing dependencies or incorrect execution entry points. Conversely, a successful execution does not automatically imply that a claim is supported. ReAgent further verifies whether the reproduced metric is bound to the correct target, remains unaffected by high-risk patches, and covers all required verification units.

A.2 Experimental Protocol

Having described the auditing pipeline, we next explain how ReAgent itself is evaluated. The experimental protocol addresses three questions: how the ground truth is constructed, how system outputs are mapped to evaluation labels, and why the adopted execution budgets and stopping criteria constitute a reasonable evaluation protocol.

A.2.1 Static Ground Truth

The static ground truth is established through manual auditing. We first use Claude Code to identify candidate issues, after which human reviewers determine the final labels. The purpose of this procedure is not to allow an LLM to generate the ground truth directly, but rather to improve candidate coverage while relying on human reviewers to verify the underlying code evidence, corresponding scientific claims, and risk categories.

During candidate generation, Claude Code independently analyzes each repository using three models: `claude-4.7-opus`, `claude-4.8-opus`, and `claude-fable-5`. The prompt used for candidate generation is shown below.

Static-audit prompt

```
You are an expert software auditor.
Analyze each repository in [path to repository] independently and identify whether it contains any of the following static risks:

- Hardcoded Results: Reported figures are hardcoded into scripts or output templates rather than being generated through actual computation.
- Substantive Implementation Discrepancies: The reported module exists and runs, but its core algorithmic steps diverge substantially from the textual description.
- Omitted Implementation: Proposed methods or modules mentioned in the paper are absent from the codebase.
- Shell/Dummy Implementation: Modules are defined, but they are short-circuited or never invoked during actual runtime.
- Fabricated Datasets: The paper claims to use a specific dataset, but the code generates synthetic data or uses a minor subset instead.
- Fabricated Experimental Hyperparameters: Claimed configurations or parameters are either omitted in code or fixed to arbitrary constant values.
- Test set / Answer / Label leakage: Directly using the test set for training, parameter tuning, and seed selection.
- Others: Does not fall into the above 7 categories, but you consider it risky

For each repository:

1. Inspect the project structure, source code, configs, training scripts, evaluation scripts, and datasets.
2. Search for evidence of each risk.
3. Report only findings supported by concrete code evidence. Do not speculate.
4. If no evidence is found, report "Not Found".

Return a JSON object for each repository in the following format:

{
  "repository": "...",
  "has_risk": true,
  "risks": [
    {
```

Table 10: Repository statistics during static ground-truth construction.

Stage	Repositories
LLM candidate merge	71
Additional repositories identified during manual review	1
Unanimous author agreement	25
Two-author agreement	3
Retained after discussion	2
Final static-risk repositories	28
Final no-risk repositories	68

```
      "type": "...",
      "status": "Found",
      "confidence": "High",
      "file": "...",
      "function": "...",
      "line": 123,
      "reason": "..."
    }
  ]
}

Requirements:

- Analyze repositories independently.
- Only report the predefined risk types.
- Every finding must include supporting code locations.
- Output valid JSON only.
```

After merging the candidate findings from the three models, 71 repositories were flagged as containing at least one potential static risk (Table 10). We then manually re-examined the remaining repositories to identify potential false negatives, discovering one additional repository with a previously missed risk.

Finally, three authors independently reviewed every candidate issue proposed by the models. The review followed three principles. First, reviewers inspected the reported source locations (i.e., file, function, and line number) and compared the corresponding implementation against the associated research document. A reported issue was rejected if the claimed behavior could not be verified in the source code or if the paper had already explicitly disclosed the implementation detail. Second, reviewers verified whether each issue had been assigned to the correct risk category, with particular attention paid to repositories flagged as risky by the models. Third, reviewers were allowed to introduce additional candidate issues if new evidence was discovered during manual inspection.

After merging the three independent reviews, 25 repositories were unanimously identified as containing genuine static risks. An additional three repositories were judged as risky by two of the three reviewers. Following discussion, two of these repositories were retained as true positive cases. Together with the one repository discovered during the false-negative review, the final benchmark contains $25 + 2 + 1 = 28$ repositories with confirmed static risks. We further reviewed the risk categorization of the 25 unanimously accepted repositories. Only three repositories exhibited disagreement regarding the specific risk category, and all remaining disagreements were resolved through discussion.

The final static ground truth contains 96 repositories, of which 28 contain at least one confirmed static risk. In total, we identified 34 confirmed risk items, including 24 high-confidence and 10 medium-confidence findings. The distribution of confirmed risk categories is summarized in Table 11.

The `Others` category contains six confirmed risk items that do not fit cleanly into the seven predefined risk categories but nevertheless represent implementation issues relevant to repository auditing.

Table 11: Distribution of confirmed static risk categories.

Risk type	Confirmed items
Substantive Implementation Discrepancies	10
Test Set / Answer / Label Leakage	7
Others	6
Omitted Implementation	4
Fabricated Experimental Hyperparameters	4
Hardcoded Results	1
Fabricated Datasets	1
Shell/Dummy Implementation	1
Total	34

Table 12: Breakdown of confirmed risk items in the `Others` category.

Reason group	Count
Credential leakage / agent safety risk	1
Issues affecting dynamic verification	2
Code-level correctness bug	2
Undocumented parameter choice / provenance gap	1
Total	6

Rather than discarding these findings, we retain them as an auxiliary category while distinguishing them from the core claim–code inconsistency taxonomy.

The breakdown of the `Others` category is shown in Table 12. The six repositories in the `Others` category correspond to the following issues:

- Scene-specific Gaussian budgets are hardcoded in the evaluation code but are neither fully documented nor explained in the accompanying research document.
- The repository depends on the `diffusion_feature` module, while the released snapshot lacks both the corresponding implementation and the `requirements.txt` file referenced in the report.
- `HogwildCache.get_seq_length()` may return incorrect sequence lengths after cache merging, introducing a potential correctness issue in cache length and position handling.
- A Hugging Face access token is hardcoded in the repository, representing a credential leakage and agent safety risk.
- During global parameter updates, the implementation averages dual variables from all clients, including stale variables from clients that were not selected in the current round, resulting in behavior inconsistent with the standard FedDyn algorithm.
- The report claims an AUDRC improvement from 89.58% to 91.07%, but no corresponding values can be located in the released `results.csv`, and the associated dataset and experimental run are not documented, preventing provenance verification.

A.2.2 Dynamic Ground Truth

Constructing dynamic ground truth entirely through manual reproduction would require substantial effort. We therefore adopt an agent-assisted reproduction protocol followed by human verification. The agent is first responsible for creating an isolated execution environment, installing dependencies, downloading required assets, and attempting repository execution. Human annotators then selectively reuse the generated environments and downloaded assets during manual verification. All repositories are executed in independent virtual environments created with `uv` to avoid dependency contamination across repositories.

We distinguish three verification scenarios. First, if the repository successfully produces the reported metric, we verify whether the execution assets and experimental configuration are consistent with

Table 13: Summary of the dynamic ground truth.

Item	Count	Percentage
Total claim-level records	96	100%
Reproducible claim values	33	34.4%
Not reproducible claim values	63	65.6%

Table 14: Runtime parameters used for dynamic verification.

Parameter	Value	Description
`MVP_REVIEW_MAX_ROUNDS`	6	Maximum review–reproduce iterations
`retry_budget`	5	Retry attempts per iteration
`search_budget`	5	Repository search budget per iteration
`modification_budget`	2	Allowed modifications per iteration
`max_modified_files`	2	Maximum modified files
`time_budget`	8 h	Maximum execution time per repository
Allowed change types	4	Path/config, script, entrypoint, runtime recovery
High-risk change types	3	Algorithmic, metric/evaluation, generic source-code changes

the research document. When they are consistent, the execution environment is reused directly for manual confirmation. Second, if execution succeeds but the configuration differs from the paper, only the execution environment and downloaded assets are reused, while the experiment is rerun under the correct configuration. Third, if the required assets differ from those used in the paper, the corresponding assets are collected manually before reproduction. For repositories that fail to produce valid metrics, we first identify the underlying failure causes and then manually attempt to overcome these issues whenever possible.

For the combined repository-level ground truth, a repository is labeled no-risk only when it has no confirmed static risk and its relevant dynamic claim values are successfully reproduced. Thus, the 25 repositories satisfying both conditions define the final no-risk set used for the joint evaluation; static-only and dynamic-only labels remain diagnostic.

The final dynamic ground truth contains 96 claim-level records. Each record includes the `paper_id`, `claim_value`, `metric`, `is_claim_value_reproducible`, `manual_reference_value`, and `threshold`. Thirty-three claim values are labeled as reproducible. The remaining 63 are labeled as not reproducible, with the audit record retaining whether the cause was a measured mismatch, a blocked execution, or an unavailable target. Table 13 summarizes these dynamic ground-truth counts.

A.2.3 Runtime Configuration

This section reports the runtime parameters used in our experiments rather than the algorithmic motivation described earlier. These parameters are part of the experimental protocol because they determine how verification is executed, when execution terminates, and whether different methods are evaluated under comparable computational budgets.

Rather than repeatedly executing entire repositories without restriction, ReAgent performs *claim-targeted verification*. Execution is limited to code paths associated with the constructed claim contracts and semantic bindings, and terminates once sufficient evidence has been collected or the predefined budget has been exhausted.

By default, dynamic verification performs at most six review–reproduce iterations. The complete runtime configuration is reported in Table 14. Each iteration is further constrained by retry, search, and modification budgets. These limits bound the search and modification capabilities of the Reproduce Agent, ensuring that verdicts such as `Blocked` and `Not Reproduced` reflect the evidence obtainable within a fixed computational budget rather than the outcome of unbounded reproduction attempts.

For the Codex baseline, we use a single prompt instructing the model to inspect the repository, reproduce the reported results, and summarize its findings in both structured JSON and a textual report.

Codex baseline prompt

```
You need to review the statements in the code and files to see if they are consistent with the results listed in the (path to research document) file under this directory (path to repository). You need to create your own virtual environment using uv and try to reproduce the results in this file by running the experiments. You can access the network via port ****. You should organize the results into a report.md file and describe your findings.
And return a JSON object for each repository in the following format:
{
  "repository": "...",
  "has_static_risk": true,
  "metric": "...",
  "metric_direction": "higher_is_better or lower_is_better",
  "claim_value": ...,
  "observed_value": ...,
  "is_claim_match_code": true,
  "evidence":"..."
}
```

The structured JSON output is used to extract reproduced metrics, while the accompanying `report.md` provides qualitative evidence for manual inspection. During evaluation, we observed four repositories for which Codex did not actually execute the reported experiments. Instead, it inferred claim–code consistency solely from existing logs or released artifacts. Although these repositories contained non-empty `observed_value` fields that matched the reported `claim_value`, we did not regard them as successful reproductions. They were therefore labeled as unsuccessful dynamic reproductions in our evaluation.

A.3 Additional Analyses

A.3.1 Repository-Level vs. Claim-Level Evaluation

Each AutoSOTA repository corresponds to one autonomous research attempt whose primary objective is to improve a target evaluation metric beyond a reported baseline. Consequently, each repository in our main benchmark can typically be reduced to a single primary performance claim. To remain consistent with the evaluation unit of AutoSOTA artifacts, the main paper reports repository-level results. Internally, however, ReAgent performs auditing at the claim level through claim contracts, verification targets, and evidence maps.

This design has one limitation: the AutoSOTA benchmark does not fully expose the diversity of claim-level inconsistencies that arise in papers containing multiple independent scientific claims. To complement the main evaluation, Section D reports additional claim-level case studies and analyses.

A.3.2 Dataset Construction

The evaluation benchmark was constructed from the 188 optimization repositories released in AutoSOTA (version v0.3.0-13-ge29faa98).

During a pilot study, we observed that reproduction frequently failed because required external assets (e.g., pretrained checkpoints or datasets) were unavailable or undocumented, rather than because of claim–code inconsistencies. Since AutoSOTA does not provide standardized asset preparation instructions, many repositories exhibited nearly identical failure modes caused by missing dependencies. Such homogeneous failures provide limited insight into the consistency problem studied in this work.

Table 15: Average computational cost per repository.

Method	Mean tokens	Median tokens	Mean time	Median time
Static-Audit	6.9K	5.9K	9.51 s	9.14 s
Codex	2.07M	867.5K	46 m 12 s	27 m 06 s
ReAgent	10.75M	9.42M	3 h 47 m	2 h 47 m

To reduce this source of noise, we performed a lightweight filtering step using Codex with `gpt-5.5`. Each repository was automatically analyzed to determine

- whether external assets were required for execution; and
- whether these assets could be located or downloaded together with corresponding access links.

We subsequently performed manual verification on a random sample of 30 repositories. During this process, we identified two additional classes of incomplete artifacts. Some repositories contained only intermediate code-analysis files without a final research document, suggesting unfinished autonomous research outputs. Others provided a final research document but no corresponding source repository. Neither case satisfies the minimum inputs required for Claim–Code Consistency auditing, and we could not reliably determine whether these artifacts were incomplete releases or genuinely unfinished research products.

After filtering, we excluded 79 repositories with unavailable external assets, three repositories without a final research document, and ten repositories without corresponding source code. The resulting benchmark contains 96 repositories.

Importantly, this filtering procedure was not intended to increase the reproduction success rate. Instead, it ensured that every benchmark instance satisfied the minimum prerequisites for claim auditing while reducing the large number of homogeneous failures caused solely by missing asset preparation instructions.

A.3.3 Cost Analysis

Table 15 summarizes the average computational cost of the evaluated methods in terms of token consumption and wall-clock execution time.

As expected, ReAgent incurs substantially higher computational cost than both baselines because it performs claim extraction, semantic binding, iterative execution, evidence collection, and review before producing the final verdict. In contrast, Static-Audit performs only lightweight static inspection, while Codex typically executes a single end-to-end reproduction attempt.

The additional cost reflects a deliberate design choice rather than implementation overhead. ReAgent is designed to maximize auditing reliability by collecting sufficient evidence before making a decision, which explains the higher token consumption and execution time observed in Table 15.

A.3.4 Discussion of Experimental Results

Table 4 reveals several interesting observations.

First, both Static-Audit and Codex achieve relatively high recall in static verification but show limited precision, resulting in lower overall accuracy. Manual analysis indicates that these methods tend to identify a broad range of potential inconsistencies but may lack sufficient evidence filtering. In particular, Codex sometimes treats implementation-level differences as claim inconsistencies without fully considering experimental context, intended usage, or whether the difference affects the reported claim.

Second, Codex follows a more execution-oriented but less constrained reproduction strategy. In several cases, it concludes verification based on existing logs or released artifacts without fully reconstructing the claimed experimental pipeline. Although this strategy reduces reproduction cost, it may produce unreliable judgments when released artifacts are incomplete, generated under different configurations, or inconsistent with the methodology described in the research document.

Table 16: AAAI 2024 benchmark summary.

Status	Count
Audit workflow completed	37
Auditing failed	20
Filtered out	23

Table 17: Causes of auditing failures.

Failure reason	Count
Execution failure	9
Missing or inaccessible assets	6
Unsupported metric computation	2
Insufficient hyperparameter specification	1
large performance gap	1
Execution timeout	1

Finally, ReAgent achieves stronger overall performance by explicitly separating evidence collection from claim judgment. Instead of treating static differences or successful execution as direct verification signals, ReAgent makes decisions based on the consistency among claim contracts, semantic bindings, execution evidence, and repository provenance. This evidence-driven process enables ReAgent to distinguish genuine claim–code inconsistencies from common reproduction obstacles, leading to more reliable auditing results.

A.4 Additional Experiments

A.4.1 Experimental Setup

To further evaluate ReAgent on human-authored scientific artifacts, we constructed an additional benchmark consisting of AAAI 2024 papers and their official repositories.

To ensure that the evaluation remained computationally tractable, we applied two filtering criteria before repository selection.

- Repositories requiring datasets or checkpoints larger than 10 GB were excluded.
- Repositories whose estimated execution time exceeded eight hours were excluded. Execution time was estimated through static repository analysis using Codex after cloning the repository.

Unlike the AutoSOTA benchmark, where the accompanying research document is an agent-generated report, the AAAI 2024 benchmark consists of complete human-authored scientific papers. This additional experiment therefore evaluates whether ReAgent generalizes beyond autonomous research artifacts.

Since these repositories are not generated by automated research agents, we do not manually construct claim-level ground truth. Instead, we apply ReAgent directly to the benchmark and report descriptive auditing statistics, providing a complementary evaluation of the framework in real-world research repositories.

A.4.2 Results and Analysis

We collected 80 accepted papers from AAAI 2024 together with their official repositories. After applying the filtering criteria described in Appendix A.4.1, 57 repositories were included in the evaluation, while the remaining 23 repositories were excluded. Table 16 summarizes these inclusion and audit outcomes.

For 37 repositories, ReAgent completed the audit workflow. This count measures workflow completion and does not imply that all scientific claims in these repositories were consistent with the code or supported by reproduced results. Claim-level verification outcomes are reported separately below. Table 17 summarizes the primary causes of auditing failures.

The 57 evaluated repositories contain a total of 105 scientific claims, corresponding to an average of 1.84 claims per repository. Among them, 23 claims were fully verified, 22 were partially progressed, 46 were classified as not checkable, and 14 failed during dynamic verification.

A *not checkable* claim indicates that the required verification target could not be constructed after static analysis, typically because the corresponding execution entry point or required assets were

unavailable. A *partially progressed* claim refers to a claim for which execution produced partial evidence but failed to satisfy all verification targets. A *failed* claim denotes a claim whose execution terminated without producing valid metric evidence and whose failure cause was determined to be unrecoverable.

Among the 46 not-checkable claims, 19 lacked executable entry points identified during static analysis, nine depended on unavailable external assets, and the remaining 18 could not be evaluated because the corresponding experiments were never successfully executed.

Compared with the AutoSOTA benchmark, the AAAI 2024 benchmark presents substantially greater auditing complexity. In AutoSOTA, each repository typically corresponds to a single optimization objective with one primary performance claim. In contrast, human-authored research papers usually contain multiple independent claims evaluated under different datasets, metrics, and experimental settings.

Across the 105 claims, ReAgent constructed a total of 526 verification units, averaging approximately five units per claim. Each verification unit may correspond to a different dataset, evaluation metric, experimental configuration, or execution target. Consequently, auditing real scientific papers requires considerably more evidence collection and target verification than auditing AutoSOTA repositories, explaining the increased verification difficulty observed in this benchmark.

A.4.3 Case Studies

We further examine one successful case and one failure case to illustrate how ReAgent performs evidence-based auditing.

The successful case is `paper-14`, corresponding to the paper *Reliable Conflictive Multi-View Learning*. In this repository, both primary targets, `table_2__t0` and `table_3__t0`, were marked as `satisfied`, corresponding to Accuracy reproduction on normal test sets and conflictive test sets, respectively.

Rather than treating successful execution as direct evidence of claim verification, ReAgent decomposes each table-level target into fine-grained closure units defined by the dataset, method, and metric combinations. A target is considered satisfied only when all required units are matched with executable metric evidence. This case demonstrates that ReAgent can align fine-grained experimental claims from paper tables with repository execution results and provide a positive verdict only after the corresponding evidence requirements are fulfilled.

The failure case is `paper-395`, corresponding to the paper *Decoupling Degradations with Recurrent Network for Video Restoration in Under-Display Camera*. The target `table_1__t0` requires reproducing multiple metrics on the VidUDC33K dataset, including `RT(s)`, `#P(M)`, PSNR, SSIM, and LPIPS. However, the final target status was `unsatisfied`.

Further analysis by ReAgent revealed that the failure was not caused by ordinary execution errors. Instead, the repository lacked required execution assets, including `dataset/VidUDC33K/Input` and `dataset/VidUDC33K/GT`. These assets require additional preparation through the data links and procedures provided by the authors. Therefore, ReAgent did not incorrectly mark the claim as verified, nor did it simply report an unspecified failure. Instead, it identified the failure as a traceable asset availability gap that prevented complete verification.

This case demonstrates that ReAgent not only evaluates whether a scientific claim can be reproduced, but also provides actionable failure attribution when verification cannot be completed.