EVIDENCE IMPLEMENTATION ALIGNMENT FOR PAPER-TO-CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Faithfully translating scientific papers into complete code repositories remains challenging even when the relevant algorithmic requirements have already been correctly identified during paper analysis. During coding, these requirements may still be omitted, partially realized, or implemented inconsistently. We refer to this mismatch between identified requirements and their code realizations as the Analysis-to-Code Grounding Gap. To address it, we introduce Evidence Implementation Alignment (EIA), which maintains a unit-level correspondence between paper-grounded implementation requirements and their concrete repository realizations. EIA organizes these requirements into Algorithm Units and verifies their paper support before code generation, then traces the same units into the generated repository and diagnoses their alignment to guide targeted repair. We evaluate EIA on PaperBench CodeDev and Paper2CodeBench using DeepSeek-v4-Flash and Qwen2.5-Coder-32B-Instruct-AWQ. On PaperBench CodeDev, EIA improves over PaperCoder from 52.40% to 63.75% and from 35.98% to 43.48%, respectively. On Paper2CodeBench, EIA achieves the highest average score among the evaluated methods under DeepSeek-v4-Flash and remains close to PaperCoder under Qwen2.5-Coder-32B-Instruct-AWQ. Stage-level ablations and unit-level diagnostics further show incremental gains from both stages and lower measured evidence-supported implementation misalignment after targeted repair.

1 INTRODUCTION

Modern machine learning and artificial intelligence papers increasingly describe complex algorithms whose reproduction requires complete, repository-level implementations rather than isolated code snippets. A usable reproduction may need to coordinate model components, data processing, optimization objectives, training procedures, evaluation protocols, and the interfaces that connect them across files. Recent large language model (LLM)-based Paper-to-Code systems seek to automate this process through a pipeline that moves from a paper to planning, analysis, coding, and ultimately a generated repository (Seo et al., 2026). The objective is therefore more demanding than producing syntactically valid code: the resulting repository should faithfully implement the method described in the paper while remaining structurally coherent and executable.

One difficulty lies in the transition from analysis to coding. Requirements identified during analysis—including algorithmic components, equations, losses, preprocessing steps, and interfaces—may still be omitted, only partially implemented, or realized inconsistently across files. We call the resulting discrepancy between requirements identified during paper analysis and their faithful realization in repository-level code the *Analysis-to-Code Grounding Gap*. This gap is not equivalent to incorrect paper understanding or conventional runtime failure. A repository can be syntactically valid and may even execute successfully while remaining algorithmically unfaithful to the paper. Consequently, successful execution alone is insufficient to establish implementation fidelity.

We formulate the Analysis-to-Code Grounding Gap as a unit-level correspondence problem between paper-grounded implementation requirements and their repository realizations. Existing Paper-to-Code systems increasingly improve requirement extraction, specification, verification, or generation; nevertheless, a paper-supported requirement may still be omitted, partially realized, or implemented inconsistently in the generated repository. EIA therefore preserves and inspects the correspondence between the same Algorithm Units on the paper and repository sides. Each unit receives a paper-

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

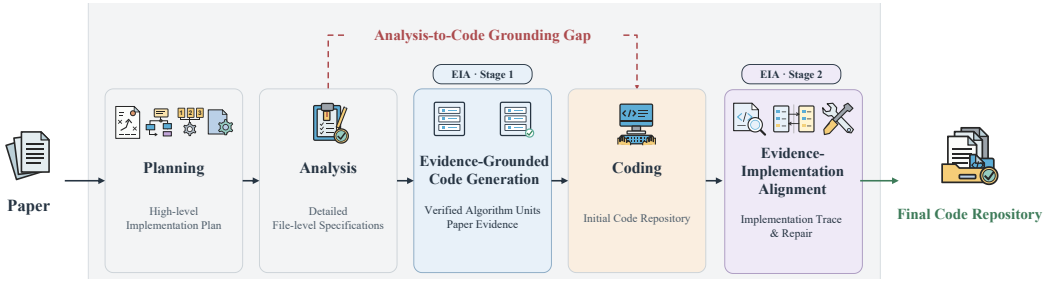


Figure 1: High-level overview of EIA. The framework augments the existing Paper-to-Code pipeline with Blueprint and Verification before coding and Trace and Targeted Repair after coding to address the Analysis-to-Code Grounding Gap.

side evidence state and a repository-side implementation state. Their joint state makes evidence-supported implementation misalignment explicit and provides a concrete target for repair.

We introduce Evidence Implementation Alignment (EIA) to track how paper-grounded requirements are carried into repository-level code. An Evidence-Aware Blueprint represents implementation requirements as fine-grained Algorithm Units, and Evidence Verification establishes their paper-side support and clarifies implementation constraints before code generation. After generation, Implementation Trace maps the same units to repository realizations and assigns implementation states, while Alignment Diagnosis jointly evaluates the paper- and repository-side states. Trace-Guided Targeted Repair then revises diagnosed misalignments, followed by re-tracing and Safety Gate.

Figure 1 summarizes how EIA augments the existing Paper-to-Code pipeline at the Analysis-to-Code transition.

We evaluate EIA on PaperBench and Paper2CodeBench with multiple backbone models (Starace et al., 2025; Seo et al., 2026). On PaperBench, EIA improves over PaperCoder under both backbones. On Paper2CodeBench, EIA achieves the highest average score under DeepSeek-v4-Flash and remains close to PaperCoder under Qwen2.5-Coder-32B-Instruct-AWQ. Fine-grained Algorithm Unit diagnostics further show a reduction in measured evidence-supported implementation misalignment after targeted repair.

Our contributions are summarized as follows:

- We formalize the Analysis-to-Code Grounding Gap as a unit-level correspondence problem between paper-grounded implementation requirements and their repository realizations. This formulation separates paper-side evidence states from repository-side implementation states, making post-generation implementation misalignment explicitly diagnosable.
- We use Algorithm Units as a shared alignment representation for the analysis-to-code transition: each unit links an implementation requirement to its supporting paper evidence, intended repository target, and interface constraints, so the same requirement can be traced from paper analysis to code realization.
- We operationalize Evidence Implementation Alignment through joint evidence and implementation states. Implementation Trace localizes the realization of each Algorithm Unit, Alignment Diagnosis identifies evidence-supported misalignment, and trace-guided repair revises the affected realization followed by re-tracing and Safety Gate.

2 Related Work

2.1 Repository-Level Code Generation and Software Agents

Repository-level code generation has explored cross-file context, retrieval, dependency-aware planning, and tool-assisted repository reasoning (Liu et al., 2024; Bairi et al., 2023; Ding et al., 2023; Shrivastava et al., 2023; Zhang et al., 2023; 2024a; Wang et al., 2024; Pan et al., 2024). Software

agents address repository-scale development, issue resolution, and program repair (Jimenez et al., 2024; Wang et al., 2025; Xia et al., 2024; Yang et al., 2024; Zhang et al., 2024b; Bouzenia et al., 2025).

2.2 Paper-to-Code and Automated Reproduction

PaperCoder established a staged Planning–Analysis–Coding paradigm for repository-level Paper-to-Code, while PaperBench broadened evaluation toward research replication (Seo et al., 2026; Starace et al., 2025). Subsequent systems and benchmarks extend this setting through multimodal evidence, repository-generation agents, paper-lineage knowledge, reproducibility evaluation, hierarchical coordination, tacit-knowledge recovery, and state-aware replanning (Lin et al., 2025; Gandhi et al., 2025; Zhao et al., 2026; Hong et al., 2026; Li et al., 2026; Song et al., 2026; Siegel et al., 2024; Xiang et al., 2025). Broader agent-based ML experimentation is evaluated by MLAgentBench and MLE-bench (Huang et al., 2024; Chan et al., 2025). These methods mainly improve information recovery, planning, or repository generation. EIA instead targets the transition from paper analysis to code realization in staged Paper-to-Code pipelines, asking whether requirements identified during analysis are faithfully carried into the generated repository.

2.3 Verification, Fidelity, and Semantic Alignment

Recent studies have shifted from executability alone toward fine-grained assessment of implementation fidelity. RePro extracts atomic paper fingerprints as supervisory criteria for mathematical formulas, algorithmic logic, and implementation details, then checks and refines generated code in an iterative verification loop (Zhou et al., 2025). Thus, fine-grained verification and iterative refinement are already explicit objectives in Paper-to-Code reproduction. Complementing this generation-oriented approach, SA-Bench characterizes silent divergence from paper specifications as semantic drift and evaluates repositories through atomic Semantic Alignment Units (Hu et al., 2026a). Its 1,491 units across 30 papers cover numerical settings, methods and algorithms, experimental protocols, and procedure ordering. As a diagnostic benchmark rather than a generation framework, it provides direct evidence that executability does not necessarily establish scientific fidelity. In contrast, the Analysis-to-Code Grounding Gap studied by EIA is more specific: it concerns requirements that have already been identified during paper analysis but are not faithfully carried into the generated repository.

A related line of work strengthens the specifications carried through repository construction. ReproAgent maintains an implementation contract across its Prepare–Plan–Generate–Repair pipeline (Hu et al., 2026b). Separate implementation-requirement and reference-evidence channels are bound to work packages and projected into file-level contracts that persist through generation and repair. This design is closely related to the Evidence-Aware Blueprint in its structured requirements, persistent constraints, and file-level implementation obligations. Concurrently, PaperCompiler observes that downstream coding agents may ignore, compress, or reinterpret free-form plans and instead compiles paper-grounded evidence into explicit repository-level specifications (Liu et al., 2026). These specifications preserve provenance; distinguish paper-supported, inferred, externally delegated, and unresolved information; and encode ownership, cross-file dependencies, file-level constraints, and non-degradation requirements. Together, these approaches highlight the importance of structured requirements and repository-level constraints for preserving paper-grounded information throughout repository construction.

2.4 Post-Generation Evidence–Implementation Correspondence

Prior work has mainly focused on constructing specifications, verifying generated code, or repairing repositories during reproduction. EIA makes the post-generation correspondence between paper-grounded requirements and repository realizations explicit by recording where each requirement is implemented and assessing whether its realization preserves the intended behavior and constraints. PaperCompiler emphasizes compiling paper evidence into repository-level specifications before generation (Liu et al., 2026), while ReproAgent maintains implementation contracts throughout reproduction (Hu et al., 2026b). RePro verifies and refines generated code against atomic paper fingerprints (Zhou et al., 2025), whereas EIA explicitly traces each paper-grounded implementation

requirement to its concrete repository realization and uses this requirement-to-realization correspondence to localize misalignment and guide targeted repair.

EIA separates the paper-side evidence state from the repository-side implementation state for each Algorithm Unit and evaluates their correspondence after generation through Alignment Diagnosis. This joint evaluation makes evidence-supported implementation misalignment explicit. Implementation Trace localizes each code realization, Alignment Diagnosis assesses the correspondence, and Trace-Guided Targeted Repair uses this correspondence to revise the affected realization, followed by re-tracing and Safety Gate.

3 METHOD

3.1 PROBLEM SETUP AND GROUNDING GAP

Given a research paper P, Paper-to-Code generation seeks a repository R that faithfully implements the methods described in the paper; we write this mapping as $R = F(P)$. Modern systems typically proceed from planning and paper analysis to repository-level coding. The analysis stages identify algorithmic components, implementation requirements, and structural dependencies, while the coding stage realizes these intermediate representations as executable repository artifacts.

This transition is commonly mediated by coarse-grained natural-language descriptions or intermediate plans, so even a correctly identified requirement is not guaranteed to be faithfully realized in the repository. We call this discrepancy the *Analysis-to-Code Grounding Gap*: the mismatch between algorithmic requirements identified during paper analysis and their actual repository-level realizations. A requirement may be omitted, only partially realized, or implemented in conflict with the paper, an interface constraint, or a related component. Because such a repository may remain syntactically valid or executable, successful execution alone is insufficient to establish implementation fidelity.

We represent this gap at the level of individual Algorithm Units, where each unit links one paper-side implementation requirement to its intended repository realization. The Evidence-Aware Blueprint constructs a set of Algorithm Units from the implementation requirements identified during paper analysis:

$$\mathcal{U} = \{u_1, u_2, \ldots, u_n\}, \qquad u_i = (r_i, e_i, t_i, d_i), \tag{1}$$

where r_i is an implementation requirement, e_i its supporting paper evidence, t_i its intended target (e.g., a file, function, or module), and d_i the relevant dependency, interface, or protocol constraints.

Granularity of Algorithm Units. An Algorithm Unit is a semantically coherent and independently assessable implementation obligation that can be assigned a single Evidence State, a single Implementation State, and a localized repository target. Its boundary is semantic rather than tied to a fixed syntactic structure: depending on the requirement, a unit may correspond to a mathematical operation, an algorithmic step, or a function-level behavior. A unit may contain multiple tightly coupled conditions when they jointly define one implementation behavior, share a repository target, and admit a common evidence-state assessment. Such a unit is considered `IMPLEMENTED` only when all required conditions are sufficiently and consistently realized. It is considered `PARTIAL` when substantial behavior is present but some required conditions remain unrealized, provided that no conflicting behavior or constraint violation warrants a `CONFLICTING` assessment. Appendix B provides a concrete example.

For each Algorithm Unit, the paper-side evidence state is

$$E_i \in \{\texttt{VERIFIED}, \texttt{INFERRED}, \texttt{UNRESOLVED}\}. \tag{2}$$

These states indicate, respectively, explicit paper support, a reasonable contextual inference without direct support, or insufficient evidence to determine the requirement reliably. The repository-side implementation state is

$$I_i \in \{\texttt{IMPLEMENTED}, \texttt{PARTIAL}, \texttt{MISSING}, \texttt{CONFLICTING}\}, \tag{3}$$

indicating complete and consistent realization, partial realization, absence of a corresponding implementation, or a realization that conflicts with the requirement or related constraints.

Evidence Implementation Alignment jointly considers E_i and I_i. A verified requirement is aligned when it is fully and consistently realized. In contrast, we define an evidence-supported implementation misalignment by the condition

$$E_i = \texttt{VERIFIED} \quad \wedge \quad I_i \in \{\texttt{PARTIAL}, \texttt{MISSING}, \texttt{CONFLICTING}\}. \tag{4}$$

This condition is the observable manifestation of the grounding gap at the Algorithm Unit level and makes the correspondence among paper evidence, algorithmic requirements, and code realizations traceable, inspectable, and diagnosable.

3.2 Overview

EIA uses shared Algorithm Units to connect paper-side requirements with their repository realizations throughout generation. Before generation, Blueprint and Verification establish the paper-side requirements and their evidence support. After generation, Trace, Diagnosis, and Targeted Repair map the same units to repository realizations and correct detected misalignments, as summarized in Figure 2. EIA is implemented as an agentic workflow. LLM-based components perform Blueprint construction, Evidence Verification, Implementation Trace, and Targeted Repair, while Alignment Diagnosis and the final Safety Gate are implemented programmatically. Detailed prompts and implementation logic are provided in Appendix C.

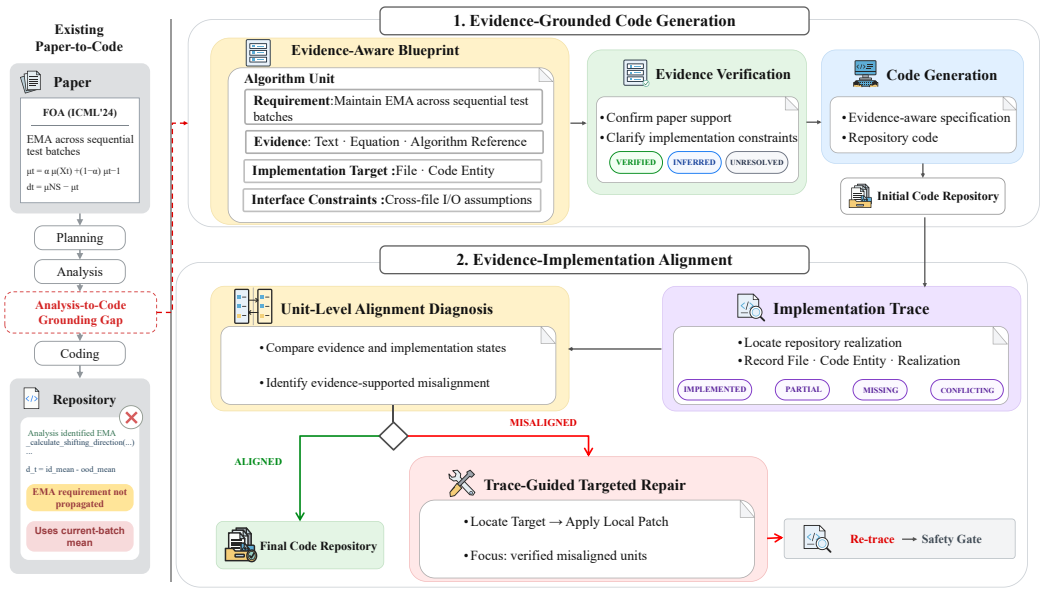


Figure 2: Detailed Evidence Implementation Alignment framework. EIA consists of two stages: (1) Evidence-Grounded Code Generation, where paper-grounded requirements are organized into Algorithm Units and verified before repository generation; and (2) Evidence Implementation Alignment, where the generated repository is traced against the same units, diagnosed for evidence-supported misalignment, and locally repaired when needed. Re-tracing and a programmatic Safety Gate are then used to assess the repaired repository.

3.3 Evidence-Aware Blueprint

The Evidence-Aware Blueprint decomposes the algorithmic information identified during paper analysis into the Algorithm Units defined in Section 3.1, providing a finer-grained representation for repository implementation. Instead of passing a coarse summary directly to the coding stage, the Blueprint organizes this information into a structured form that can be reused during verification and post-generation alignment.

For each unit, the Blueprint records the implementation requirement together with its provenance, intended repository location, and relevant cross-file constraints. It also assigns a provisional evidence

state based on the available paper context, which is independently checked and may be revised by the subsequent Evidence Verification stage. Repository-level requirements may span files, functions, modules, or interfaces and can depend on data formats, tensor shapes, calling conventions, and other cross-file protocols. The resulting representation is shared by subsequent Evidence Verification and Evidence Implementation Alignment, while the final Evidence State and code correctness are not determined by the Blueprint itself.

3.4 Evidence Verification

Structuring a requirement does not imply that it is explicitly supported by the paper. Evidence Verification examines each Algorithm Unit against the paper context before code generation and determines its final Evidence State, confirming or revising the provisional evidence state produced by the Blueprint. It distinguishes directly supported requirements from contextual inferences and unresolved cases. This separation prevents inferred implementation choices from being treated as explicit paper requirements.

When sufficient evidence is available, Verification clarifies the requirement together with any paper-supported validation criteria and dependency, interface, or protocol constraints. It also checks cross-unit consistency for relationships such as input–output contracts, tensor shapes, data representations, signatures, calling conventions, and preprocessing protocols. When the paper does not provide enough information, the uncertainty is retained rather than resolved by assumption. The resulting specification forms the paper-evidence side of Evidence Implementation Alignment and guides code generation without treating all extracted requirements as equally certain. By default, only VERIFIED units are treated as implementation obligations and repair targets; INFERRED and UNRESOLVED units retain their uncertainty rather than being enforced as requirements.

3.5 Implementation Trace and Alignment Diagnosis

After code generation, Implementation Trace maps each Algorithm Unit to its concrete repository realization. It identifies relevant locations and code entities, records code evidence for the correspondence, and assigns the Implementation State defined in Section 3.1. The trace also records relevant dependencies and interfaces, providing a localized view of how each requirement is realized in the repository.

A unit is assessed as IMPLEMENTED when the available code evidence sufficiently realizes the requirement and its associated constraints; it is PARTIAL when related code realizes only part of the requirement, MISSING when sufficient corresponding code cannot be found, and CONFLICTING when the realization is inconsistent with the requirement, an interface constraint, or a related component.

Alignment Diagnosis then jointly considers the paper-side Evidence State and the traced Implementation State. In particular, a VERIFIED requirement assessed as PARTIAL, MISSING, or CONFLICTING is treated as an evidence-supported implementation misalignment and becomes a target for repair.

3.6 Trace-Guided Targeted Repair

Trace-Guided Targeted Repair uses the diagnosed unit-level misalignments to guide localized code revisions. For each affected Algorithm Unit, the repair context combines its paper evidence, implementation requirement, interface and dependency constraints, current realization, and alignment diagnosis. The repair policy prioritizes VERIFIED requirements assessed as PARTIAL, MISSING, or CONFLICTING and uses the traced locations to avoid treating the repository as a black box.

This localized context allows repair to focus on the affected realization rather than regenerating the entire repository. The repair aims to preserve unrelated implementations while correcting the diagnosed mismatch. After repair, EIA re-runs Implementation Trace and applies a programmatic Safety Gate; failed repairs are rolled back to the pre-repair repository snapshot and trace (Appendix C.4).

4 EXPERIMENTS

We evaluate EIA at three levels: repository-level performance, stage-level contribution, and Algorithm Unit-level implementation misalignment before and after repair.

4.1 EXPERIMENTAL SETUP

Datasets and Benchmarks. We evaluate repository-level Paper-to-Code generation on PaperBench and Paper2CodeBench. PaperBench contains 20 papers. Paper2CodeBench contains 90 papers, with 30 from each of ICLR 2024, ICML 2024, and NeurIPS 2024. **Baselines.** We compare EIA with PaperCoder and HiRAS under the corresponding benchmark settings.

Backbone Models. We evaluate all methods with DeepSeek-v4-Flash and Qwen2.5-Coder-32B-Instruct-AWQ. **Evaluation Metrics.** For PaperBench, we report repository-level CodeDev scores using GPT-4o-mini as the judge. For Paper2CodeBench, we report reference-based scores on a five-point scale using DeepSeek-v4-Pro as the judge. We prioritize the reference-based protocol because it compares generated repositories with author implementations and is sensitive to paper-specific methodological deviations. We additionally report PaperBench token usage as an inference-cost analysis in Appendix D.

4.2 MAIN RESULTS

PaperBench. Table 1 reports the repository-level CodeDev score. EIA obtains 63.75 with DeepSeek-v4-Flash, improving over PaperCoder and HiRAS by 11.35 and 31.59 points. With Qwen2.5-Coder-32B-Instruct-AWQ, EIA reaches 43.48, gains of 7.50 and 23.11 points, respectively. Figure 3 shows the per-paper breakdown: EIA outperforms PaperCoder on 17 of 20 papers under DeepSeek-v4-Flash and on 14 of 20 papers under Qwen2.5-Coder-32B-Instruct-AWQ.

Paired statistical analysis further shows that EIA's improvements over PaperCoder on PaperBench are statistically significant under both backbones: the paired-bootstrap 95% confidence intervals exclude zero, and two-sided Wilcoxon signed-rank tests yield $p < 0.05$. Full paired statistical results are reported in Appendix A.

Table 1: PaperBench repository-level CodeDev score (%).

Method	Qwen2.5-Coder-32B-Instruct-AWQ	DeepSeek-v4-Flash
PaperCoder	35.98	52.40
HiRAS	20.37	32.16
EIA (ours)	**43.48**	**63.75**

Paper2CodeBench. Table 2 reports results for 90 papers. Under DeepSeek-v4-Flash, EIA has the highest score on all three conference subsets and improves the overall average from 1.81 for the strongest baseline to 1.94. Under Qwen2.5-Coder-32B-Instruct-AWQ, EIA ties PaperCoder on ICLR at 1.07, is 0.03 lower on ICML, and is 0.01 higher on NeurIPS; its 1.08 average is close to PaperCoder's 1.09.

4.3 ABLATION STUDY

We use a stage-level progressive ablation over EIA's two major stages: *Evidence-Grounded Code Generation* and *Evidence Implementation Alignment*. The analysis measures stage-level effects rather than isolating every internal subprocedure.
Table 3 reports three progressive variants on the full 20-paper PaperBench CodeDev evaluation under Qwen2.5-Coder-32B-Instruct-AWQ and DeepSeek-v4-Flash.

Table 3: Stage-level ablation

Variant	Qwen	DeepSeek
A. Base	35.98	52.40
B. + EIA stage1	41.49	62.36
C. Full EIA	**43.48**	**63.75**

Variant A corresponds to the underlying PaperCoder pipeline without either EIA stage. Variant B adds *Evidence-Grounded Code Generation*, and Variant C further adds *Evidence Implementation Alignment*, corresponding to the complete EIA framework.

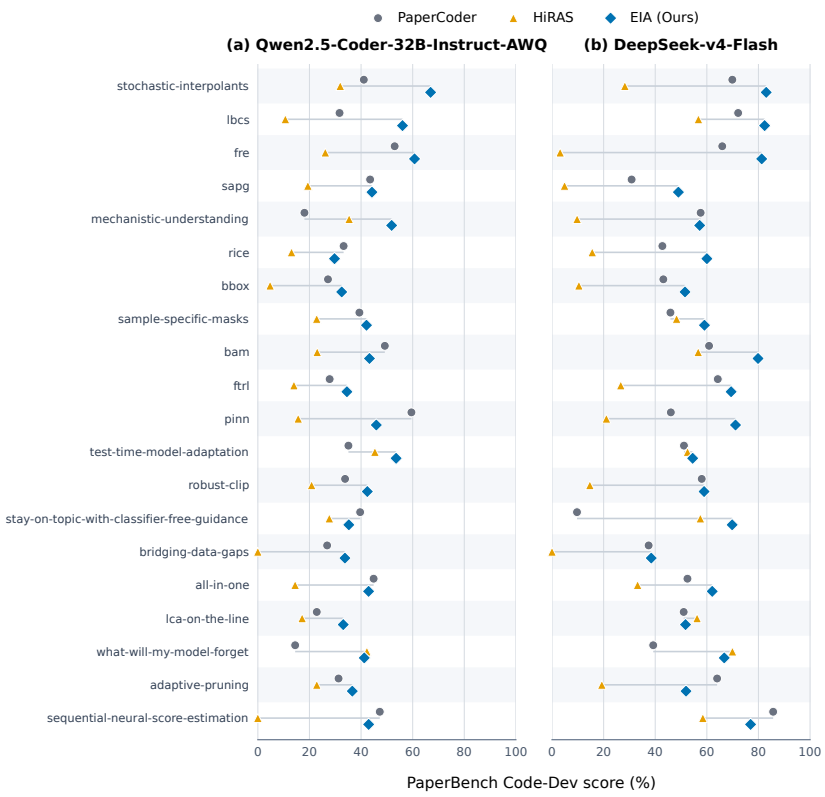


Figure 3: Per-paper PaperBench CodeDev scores under Qwen2.5-Coder-32B-Instruct-AWQ and DeepSeek-v4-Flash. Horizontal segments span the three method scores for each paper. Papers are ordered by EIA's mean improvement over the stronger baseline across the two backbones.

Table 2: Reference-based results on Paper2CodeBench. Each conference subset contains 30 papers. We report mean ± standard deviation across paper-level scores, where each paper-level score is averaged over three independent judge evaluations. Avg. denotes the overall mean for the 90-paper benchmark. Bold indicates the highest reported mean within each backbone; equal means are both bold.

Backbone	Method	ICLR'24 (30)	ICML'24 (30)	NeurIPS'24 (30)	Avg. (90)
DeepSeek-v4-Flash	PaperCoder	1.79 ± 0.46	2.04 ± 0.52	1.60 ± 0.40	1.81
	HiRAS	1.83 ± 0.54	1.78 ± 0.45	1.64 ± 0.46	1.75
	EIA (ours)	$\mathbf{1.97} \pm 0.55$	$\mathbf{2.12} \pm 0.50$	$\mathbf{1.74} \pm 0.37$	**1.94**
Qwen2.5-Coder-32B -Instruct-AWQ	PaperCoder	$\mathbf{1.07} \pm 0.16$	$\mathbf{1.12} \pm 0.24$	1.08 ± 0.21	**1.09**
	HiRAS	1.03 ± 0.10	1.01 ± 0.06	1.01 ± 0.06	1.02
	EIA (ours)	$\mathbf{1.07} \pm 0.16$	1.09 ± 0.20	$\mathbf{1.09} \pm 0.21$	1.08

Both backbones show the same progressive pattern. Under Qwen2.5-Coder-32B-Instruct-AWQ, Stage 1 improves the score from 35.98% to 41.49%, and Stage 2 further increases it to 43.48%. Under DeepSeek-v4-Flash, the corresponding scores are 52.40%, 62.36%, and 63.75%. Thus, Stage 1 provides the larger gain, while *Evidence Implementation Alignment* contributes an additional improvement under both backbones.

4.4 Fine-Grained Alignment Diagnostics

VMR, VRSR, and VRR are used as fine-grained internal alignment diagnostics. To examine evidence-supported implementation misalignment at the Algorithm Unit level, we compare each

unit's Implementation Trace before targeted repair with its final trace after repair, re-tracing, and safety handling.

Diagnostic Measures. Let $N(\cdot)$ count traced Algorithm Units satisfying a condition and let $\mathcal{M} = \{\texttt{PARTIAL}, \texttt{MISSING}, \texttt{CONFLICTING}\}$.

The Verified Misalignment Rate (VMR) is

$$\mathrm{VMR} = \frac{N(E_i = \texttt{VERIFIED},\ I_i \in \mathcal{M})}{N(E_i = \texttt{VERIFIED})}. \tag{5}$$

VMR measures the fraction of verified units diagnosed as misaligned.

The Verified Repair Success Rate (VRSR) is

$$\mathrm{VRSR} = \frac{N\big(E_i = \texttt{VERIFIED},\ I_i^{\mathrm{before}} \in \mathcal{M},\ I_i^{\mathrm{after}} = \texttt{IMPLEMENTED}\big)}{N(E_i = \texttt{VERIFIED},\ I_i^{\mathrm{before}} \in \mathcal{M})}. \tag{6}$$

VRSR measures the fraction of pre-repair verified misaligned units that become `IMPLEMENTED` after repair.

The Verified Regression Rate (VRR) is

$$\mathrm{VRR} = \frac{N\big(E_i = \texttt{VERIFIED},\ I_i^{\mathrm{before}} = \texttt{IMPLEMENTED},\ I_i^{\mathrm{after}} \in \mathcal{M}\big)}{N(E_i = \texttt{VERIFIED},\ I_i^{\mathrm{before}} = \texttt{IMPLEMENTED})}. \tag{7}$$

VRR measures regressions among previously `IMPLEMENTED` verified units after repair.

Table 4: Fine-grained alignment diagnostics across Paper2CodeBench and PaperBench (%). Before and After denote traces before and after targeted repair. Lower VMR and VRR are better; higher VRSR is better. PaperBench results use micro aggregation over the full 20-paper evaluation.

Dataset	Backbone	VMR Before	VMR After	VRSR	VRR
Paper2CodeBench	Qwen2.5-Coder-32B-Instruct-AWQ	27.87	19.24	41.59	4.10
	DeepSeek-v4-Flash	9.36	6.70	56.31	2.88
PaperBench	Qwen2.5-Coder-32B-Instruct-AWQ	28.80	17.39	43.40	1.53
	DeepSeek-v4-Flash	17.01	14.34	26.51	2.22

Table 4 shows that targeted repair reduces measured verified implementation misalignment across both benchmarks and backbones. On Paper2CodeBench, VMR decreases from 27.87% to 19.24% under Qwen2.5-Coder-32B-Instruct-AWQ and from 9.36% to 6.70% under DeepSeek-v4-Flash; on PaperBench, it decreases from 28.80% to 17.39% and from 17.01% to 14.34%, respectively. Across the four settings, the automatic diagnostics report VRSR values of 26.51%–56.31% and VRR values of 1.53%–4.10%.

Figure 2 provides a concrete example of such an implementation misalignment in FOA: although analysis identifies the requirement to maintain EMA statistics across sequential test batches, the initial code uses current-batch statistics in the relevant execution path, resulting in a state-propagation mismatch. Appendix B provides a complementary FTRL example involving a configuration-interface mismatch.

5 Conclusion

We formulate the Analysis-to-Code Grounding Gap as the mismatch between paper-grounded implementation requirements and their concrete repository realizations, and introduce Evidence Implementation Alignment (EIA) to make this correspondence explicit and repair misaligned realizations. Across PaperBench CodeDev and Paper2CodeBench, EIA improves repository-level performance in several settings. Stage-level ablations show incremental gains from both stages, while unit-level diagnostics report lower measured evidence-supported implementation misalignment after targeted repair. Together, these results highlight the value of explicitly aligning paper-grounded requirements with their code realizations in Paper-to-Code generation.

AI USE STATEMENT

Generative AI tools were used to assist with literature discovery, research ideation, and language polishing. All research methodology, experimental design, analyses, results, and conclusions were developed, reviewed, and verified by the authors, who take full responsibility for the final content.

REFERENCES

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. *arXiv preprint arXiv:2309.12499*, 2023. URL https://arxiv.org/abs/2309.12499.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2188–2200. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00157. URL https://doi.org/10.1109/ICSE55347.2025.00157.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://arxiv.org/abs/2310.11248.

Shubham Gandhi, Dhruv Shah, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. ResearchCodeAgent: An LLM multi-agent system for automated codification of research methodologies. *arXiv preprint arXiv:2504.20117*, 2025. URL https://arxiv.org/abs/2504.20117.

Hanhua Hong, Yizhi LI, Jiaoyan Chen, Sophia Ananiadou, Xiaoli Li, Jung-jae Kim, and Chenghua Lin. HiRAS: A hierarchical multi-agent framework for paper-to-code generation and execution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 7624–7652. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.377. URL https://aclanthology.org/2026.findings-acl.377/.

Xue Hu, Zewei Pan, Zeli Su, Zhou Liu, and Wentao Zhang. SA-Bench: Evaluating semantic alignment in LLM-based paper reproduction. *arXiv preprint arXiv:2608.24252*, 2026a. doi: 10.48550/arXiv.2608.24252. URL https://arxiv.org/abs/2608.24252.

Xue Hu, Zewei Pan, Zhongyuan Wang, Zhou Liu, Zeli Su, and Wentao Zhang. ReproAgent: Contract-guided paper-to-code reproduction. *arXiv preprint arXiv:2608.24291*, 2026b. doi: 10.48550/arXiv.2608.24291. URL https://arxiv.org/abs/2608.24291.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 20271–20309. PMLR, 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Lehui Li, Ruining Wang, Haochen Song, Yaoxin Mao, Tong Zhang, Yuyao Wang, Jiayi Fan, Yitong Zhang, Jieping Ye, Chengqi Zhang, and Yongshun Gong. What papers don't tell you: Recovering tacit knowledge for automated paper reproduction. *arXiv preprint arXiv:2603.01801*, 2026. doi: 10.48550/arXiv.2603.01801. URL https://arxiv.org/abs/2603.01801.

Zijie Lin, Yiqing Shen, Qilin Cai, He Sun, Jinrui Zhou, and Mingjun Xiao. AutoP2C: An LLM-based agent framework for code repository generation from multimodal content in academic papers. *arXiv preprint arXiv:2504.20115*, 2025. URL https://arxiv.org/abs/2504.20115.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2306.03091.

Yunhao Liu, Hong Phuc Pham, and Jaehong Yoon. PaperCompiler: Faithful paper-to-code generation via repository-level specification compilation. *arXiv preprint arXiv:2609.02272*, 2026. doi: 10.48550/arXiv.2609.02272. URL https://arxiv.org/abs/2609.02272.

Zhiyuan Pan, Xing Hu, Xin Xia, and Xiaohu Yang. Enhancing repository-level code generation with integrated contextual information. *arXiv preprint arXiv:2406.03283*, 2024. doi: 10.48550/arXiv.2406.03283. URL https://arxiv.org/abs/2406.03283.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating code generation from scientific papers in machine learning. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=3DcaUTjdKc.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. RepoFusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*, 2023. URL https://arxiv.org/abs/2306.10998.

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv:2409.11363*, 2024. URL https://arxiv.org/abs/2409.11363.

Hongru Song, Ruqing Zhang, Jiafeng Guo, Xueqi Cheng, and Maarten de Rijke. DeepRepro: State-aware subplanning for paper-to-code reproduction in evolving repositories. *arXiv preprint arXiv:2608.26557*, 2026. doi: 10.48550/arXiv.2608.26557. URL https://arxiv.org/abs/2608.26557.

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. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 56843–56873. PMLR, 2025. URL https://proceedings.mlr.press/v267/starace25a.html.

Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. Teaching code LLMs to use autocompletion tools in repository-level code generation. *arXiv preprint arXiv:2401.06391*, 2024. doi: 10.48550/arXiv.2401.06391. URL https://arxiv.org/abs/2401.06391.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. SciReplicate-Bench: Benchmarking LLMs in agent-driven algorithmic reproduction from research papers. *arXiv preprint arXiv:2504.00255*, 2025. URL https://arxiv.org/abs/2504.00255.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13643–13658. Association for Computational Linguistics, 2024a. doi: 10.18653/v1/2024.acl-long.737. URL https://aclanthology.org/2024.acl-long.737/.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604. Association for Computing Machinery, 2024b. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoReproduce: Automatic AI experiment reproduction with paper lineage. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21920–21942. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.1001. URL https://aclanthology.org/2026.acl-long.1001/.

Mingyang Zhou, Quanming Yao, Lun Du, Lanning Wei, and Da Zheng. Reflective paper-to-code reproduction enabled by fine-grained verification. *arXiv preprint arXiv:2508.16671*, 2025. doi: 10.48550/arXiv.2508.16671. URL https://arxiv.org/abs/2508.16671.

A Paired Statistical Analysis

We compare EIA and PaperCoder using paired statistics on both benchmarks. For Paper2CodeBench, the three independent judge scores for each paper are averaged before analysis. We report the mean paired difference, a 95% confidence interval from 100,000 paired bootstrap resamples, a two-sided Wilcoxon signed-rank test, and win/tie/loss counts (W/T/L).

Table 5: Paired statistical comparison between EIA and PaperCoder. Mean Δ denotes EIA minus PaperCoder. Bootstrap confidence intervals are computed from 100,000 paired resamples. Wilcoxon tests are two-sided.

Benchmark	Backbone	Mean Δ	95% CI	p	W/T/L
PaperBench	DeepSeek-v4-Flash	$+11.35$ pp	$[+5.30, +18.48]$	$\mathbf{0.00071}$	17/0/3
PaperBench	Qwen2.5-Coder-32B	$+7.50$ pp	$[+2.29, +13.04]$	$\mathbf{0.0153}$	14/0/6
Paper2CodeBench	DeepSeek-v4-Flash	$+0.133$	$[+0.004, +0.270]$	0.156	34/25/31
Paper2CodeBench	Qwen2.5-Coder-32B	-0.007	$[-0.048, +0.037]$	0.653	11/65/14

On PaperBench, EIA shows statistically significant paired improvements over PaperCoder under both backbones, with bootstrap confidence intervals excluding zero and Wilcoxon p-values below 0.05. On Paper2CodeBench with DeepSeek-v4-Flash, the mean paired difference is positive and the bootstrap confidence interval lies slightly above zero, whereas the Wilcoxon test does not reach statistical significance, indicating stronger evidence for a positive mean gain than for a consistent rank-based shift across papers. Under Qwen2.5-Coder-32B-Instruct-AWQ, both analyses are consistent with comparable overall performance between EIA and PaperCoder.

B QUALITATIVE CASE STUDY

Table 6: Algorithm Unit U2 and its alignment lifecycle in the PaperBench `ftrl` case study.

Field	U2 example
Unit ID	`knowledge_retention.py::U2`
Unit name	Apply EWC regularization
Requirement	Compute the Fisher Information Matrix from 10,000 batches of pre-trained data and integrate the EWC auxiliary loss with the main RL objective
Paper evidence	EWC auxiliary-loss formulation in the paper; the 10,000-batch setting is specified in the accompanying PaperBench addendum
Target	`knowledge_retention.py` / `KnowledgeRetention::apply_EWC`
Final Evidence State	`VERIFIED`
Initial ImplementationState	`PARTIAL`
Trace diagnosis	The EWC scaling factor is read from `environment.net_hack` rather than the EWC configuration interface
Repair	A localized configuration-lookup change in `apply_EWC`
Final ImplementationState	`IMPLEMENTED`
Safety Gate	Accepted

This case illustrates that the boundary of an Algorithm Unit is semantic rather than fixed to a particular code structure. Here, U2 corresponds approximately to the EWC-related behavior implemented by `KnowledgeRetention::apply_EWC`, while other behaviors in the same file are represented by separate Algorithm Units.

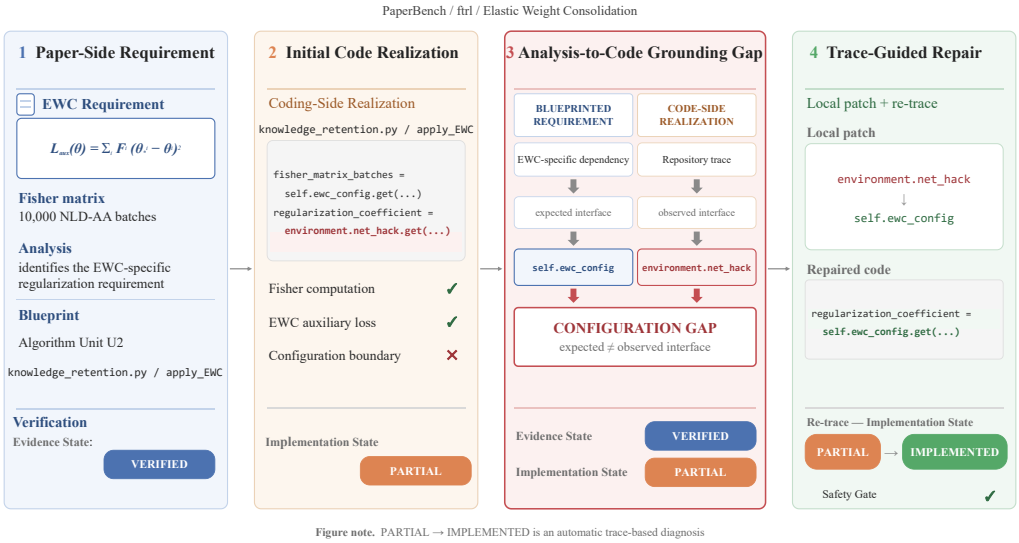


Figure note. PARTIAL → IMPLEMENTED is an automatic trace-based diagnosis

Figure 4: Qualitative case study of an EWC unit in the PaperBench `ftrl` paper. Analysis identifies the EWC-related requirement, which is organized into an Algorithm Unit by the Blueprint and verified against the paper evidence. The initial coding realization nevertheless connects an EWC-related dependency through an inconsistent configuration interface. EIA localizes this configuration-interface inconsistency and performs a targeted repair, after which re-tracing changes the automatically diagnosed implementation state from `PARTIAL` to `IMPLEMENTED`.

We examine an Elastic Weight Consolidation (EWC) unit from the PaperBench `ftrl` paper. The paper requirement is identified and the main EWC mechanism is already present in the generated repository, but one dependency is connected through an inconsistent configuration interface. The

failure occurs not in identifying EWC itself, but in how one of its dependencies is connected to the surrounding repository.

Evidence verification marks Algorithm Unit U2 in `knowledge_retention.py` as `VERIFIED`. The unit targets the `apply_EWC` function and requires an EWC auxiliary objective based on a diagonal Fisher matrix. The accompanying PaperBench addendum specifies 10,000 NLD-AA batches for estimating the Fisher matrix. The generated implementation already computes the Fisher matrix, stores the pretrained parameter values, and defines the parameter-distance penalty. The initial trace assigns the implementation state `PARTIAL`.

The diagnosed mismatch concerns configuration ownership rather than a missing EWC implementation or an incorrect default coefficient. The Fisher-batch count is read from the EWC configuration, whereas the regularization coefficient is obtained from the environment configuration. In the generated repository, these correspond to `self.ewc_config` and `environment.net_hack`, respectively. This connects an EWC-specific dependency to an environment-specific namespace. The hardcoded fallback remains $2{,}000{,}000$ before and after repair; the repair changes only the configuration interface.

The repair is localized to this dependency. The environment-specific lookup is replaced with the existing EWC configuration interface, `self.ewc_config`. After re-tracing, the evidence state remains `VERIFIED`, while the automatically diagnosed implementation state changes from `PARTIAL` to `IMPLEMENTED`. The Safety Gate accepts the repaired repository after re-tracing.

The transition from `PARTIAL` to `IMPLEMENTED` is an automatic diagnostic result rather than human-verified semantic ground truth. This case is used to illustrate how EIA identifies a configuration-interface mismatch and performs a localized repair at the Algorithm Unit level.

C Prompt Templates and Implementation Details

This appendix summarizes the prompts and programmatic checks introduced by EIA. The templates are faithful abridgements of the implementation: bracketed fields denote run-time inputs, while repeated boilerplate, instance-specific content, and nonessential schema fields are omitted. General planning and analysis prompts are not reproduced.

C.1 Evidence-Aware Blueprint and Evidence Verification

The Blueprint converts the paper and upstream analysis into fine-grained Algorithm Units. It assigns each unit a *provisional* evidence state; this state is not the final Evidence State.

Prompt excerpt for Evidence-Aware Blueprint

```
SYSTEM: Construct an implementation blueprint for [Target File]
from [Paper] and [Analysis Output]. Decompose independently
verifiable formulas, transformations, updates, protocols, and
control-flow requirements into Algorithm Units.

For each Algorithm Unit, specify:
  - implementation requirement and ordered behavior;
  - paper evidence or provenance;
  - target file and class/function;
  - validation criteria;
  - dependency, interface, and protocol constraints; and
  - Provided / Consumed / Produced interface hints.

Assign a provisional evidence state:
  VERIFIED   -- explicitly supported by the supplied paper;
  INFERRED   -- required or plausible but not explicitly specified;
  UNRESOLVED -- unsupported, ambiguous, missing, or conflicting.

Do not invent paper evidence, equations, references, or interfaces.
Plausibility alone is not evidence.

ABRIDGED OUTPUT:
{
  "unit_id": "...",
  "file": "...",
```

```
  "symbol": "...",
  "provisional_evidence_state":
    "VERIFIED | INFERRED | UNRESOLVED",
  "paper_evidence": "...",
  "implementation_requirement": "...",
  "validation_criteria": ["..."],
  "interfaces": {
    "provided": ["..."],
    "consumed": ["..."],
    "produced": ["..."],
    "config_keys": ["..."],
    "cross_file_assumptions": ["..."]
  }
}
```

Evidence Verification then checks every provisional unit independently against the supplied paper. Its output determines the final Evidence State.

Prompt excerpt for Evidence Verification

```
SYSTEM: Independently verify each Algorithm Unit in
[Algorithm Units] against [Paper]. VERIFIED requires explicit
paper support. Distinguish VERIFIED, INFERRED, and UNRESOLVED
without using external knowledge as paper evidence.

Do not invent references and do not omit units.

For each unit, return one verdict:
  CONFIRMED | CORRECTED | DOWNGRADED | REJECTED

ABRIDGED OUTPUT:
{
  "unit_id": "...",
  "verdict": "CONFIRMED | CORRECTED | DOWNGRADED | REJECTED",
  "evidence_state": "VERIFIED | INFERRED | UNRESOLVED",
  "paper_support": ["..."],
  "corrected_implementation_requirement": "..."
}
```

The implementation validates unit identity and the evidence vocabulary, rejects unsupported `VERIFIED` evidence states, and reconciles cross-file interface hints into an interface manifest. The coding stage receives the readable Blueprint together with the Algorithm Units after Evidence Verification and the cross-file interface manifest. `VERIFIED` requirements are treated as implementation obligations, while `INFERRED` and `UNRESOLVED` units retain their uncertainty.

C.2 Implementation Trace and Alignment Diagnosis

Implementation Trace receives the Algorithm Units after Evidence Verification, together with the interface manifest and repository source. It locates the concrete realization of each unit and assigns an Implementation State from executable code evidence.

Prompt excerpt for Implementation Trace

```
SYSTEM: Audit [Repository Source] against [Algorithm Units]
and [Interface Manifest]. For every unit, locate its concrete
repository realization, identify the file and code entity,
and record executable code evidence.

Comments, names, and docstrings alone are not implementation.

Assign one Implementation State:
  IMPLEMENTED -- the required executable behavior is present;

  PARTIAL     -- substantial behavior exists but part of the
                 requirement is missing;

  MISSING     -- no executable realization is present;

  CONFLICTING -- behavior contradicts the requirement or
```

```
                repository interface.

ABRIDGED OUTPUT:
{
  "unit_id": "...",
  "implementation_state": "IMPLEMENTED | PARTIAL | MISSING | CONFLICTING",
  "implementation_symbols": ["..."],
  "code_evidence": ["..."],
  "reason": "..."
}
```

Alignment Diagnosis is implemented programmatically rather than through a separate prompt. By default, an Algorithm Unit is treated as an evidence-supported implementation misalignment, and therefore a repair target, when its Evidence State is `VERIFIED` and its Implementation State is `PARTIAL`, `MISSING`, or `CONFLICTING`. An optional configuration can additionally admit `INFERRED` units; it is disabled in the default setting.

C.3 Trace-Guided Targeted Repair

For each programmatically selected repository file, the repair prompt receives paper evidence, verified Algorithm Units, diagnosed traces, the interface manifest, and the current source. File ranking and selection occur outside the prompt.

Prompt excerpt for Trace-Guided Targeted Repair

```
SYSTEM: Repair the selected repository file using
[Paper Evidence], [Verified Algorithm Units],
[Diagnosed Misalignment], [Interface Manifest], and
[Repository Source].

Repair the diagnosed VERIFIED units whose Implementation State
is PARTIAL, MISSING, or CONFLICTING, and focus on executable
behavior at the traced locations.

Preserve repository interfaces unless verified constraints
require a change, and preserve unrelated behavior.

Do not fabricate paper facts, introduce fake data or TODO stubs,
or broadly suppress exceptions.

Return the complete corrected file.

INPUT:
  [Paper Evidence]
  [Selected Repository File]
  [Verified Algorithm Units]
  [Implementation Trace]
  [Interface Manifest]
  [Current Source]
```

Repair operates only on the selected files, but the model returns each corrected file in full rather than as a structured patch.

C.4 Re-tracing and Safety Gate

Re-tracing has no independent prompt: after repair, EIA reuses the same Implementation Trace template on the modified repository. The Safety Gate is implemented programmatically and rejects repairs that increase the aggregate number of non-`IMPLEMENTED` `VERIFIED` units.

If a repair fails the Safety Gate, EIA restores the pre-repair repository and trace; otherwise, the repaired repository and re-trace are accepted. The Safety Gate provides a conservative regression safeguard for the repair process.

D TOKEN USAGE ANALYSIS

We report token usage on PaperBench as an additional measure of inference cost. All token counts are obtained from provider-reported usage records rather than estimated from prompt length. PaperCoder and EIA have valid records for all 20 papers. For HiRAS, valid records are available for 19 papers under DeepSeek-v4-Flash and 18 papers under Qwen2.5-Coder-32B-Instruct-AWQ. The missing HiRAS runs are excluded from the averages rather than imputed.

Table 7: Average token usage per paper on PaperBench. N denotes the number of papers with valid cost records. Token counts are obtained from provider-reported usage records.

Backbone	Method	N	Input Tokens	Output Tokens	Total Tokens
DeepSeek-v4-Flash	HiRAS	19/20	5,579,285	149,515	5,728,800
DeepSeek-v4-Flash	PaperCoder	20/20	746,222	85,494	831,716
DeepSeek-v4-Flash	EIA	20/20	2,230,680	240,568	2,471,248
Qwen2.5-Coder-32B	HiRAS	18/20	6,224,472	115,969	6,340,441
Qwen2.5-Coder-32B	PaperCoder	20/20	431,414	25,970	457,384
Qwen2.5-Coder-32B	EIA	20/20	1,495,910	77,513	1,573,423

EIA uses approximately 3.0× and 3.4× the tokens of PaperCoder under DeepSeek and Qwen, respectively, while remaining below HiRAS in both settings.

HiRAS uses the default agent-policy configuration from the official implementation. We do not specifically adjust the step budget, manager–worker structure, or prompts. For HiRAS, valid token records are unavailable for three runs because those runs terminated before end-of-run cost logging was completed. Inspection of the execution logs shows repeated planning, context inspection, tool-error recovery, and manager–worker coordination, and some runs terminate with AgentMaxStepsError before a complete repository is produced. These observations may partly explain the lower HiRAS scores in our setting, since incomplete repository generation directly affects repository-level evaluation. We therefore view these results as reflecting the sensitivity of the hierarchical agent pipeline to backbone capability and execution limits in our setting, rather than as evidence that multi-agent methods are generally less effective.

E ADDITIONAL EXPERIMENTAL DETAILS

Baseline selection. We use PaperCoder and HiRAS as the primary baselines because both adopt staged Paper-to-Code pipelines that separate planning, paper analysis, and code generation. This setting directly matches the analysis-to-code transition studied by EIA. ReproAgent and PaperCompiler appeared shortly before our submission, when our main experiments were already underway. We therefore discuss them as closely related concurrent work in Section 2 rather than include them in the primary empirical comparison.

Evaluation setting. Within each benchmark, all methods are evaluated with the same backbone and judge settings. PaperBench uses GPT-4o-mini as the judge for repository-level CodeDev evaluation, while Paper2CodeBench uses DeepSeek-v4-Pro for reference-based scoring. These evaluator choices are held fixed across PaperCoder, HiRAS, and EIA so that the comparison within each benchmark is consistent.

F LIMITATIONS

EIA has a few limitations. The evidence states and implementation traces are generated automatically and can therefore be imperfect, especially in classifying or locating a mismatch. Our evaluation is also limited to two benchmarks and two backbone models, so the observed gains may not carry over to other domains or generation settings. More broadly, faithful repository implementation does not by itself guarantee full scientific reproduction, which may depend on unavailable data, environment details, compute, randomness, or experimental choices omitted from the paper. EIA targets this narrower evidence-to-implementation problem rather than the full reproducibility pipeline.