ReproAgent: Contract-Guided Paper-to-Code Reproduction

Xue Hu*[1], Zewei Pan*[2], Zhongyuan Wang[1], Zhou Liu[3], Zeli Su[4], Wentao Zhang†[3,5,6]
[1]Beihang University, [2]Shanghai Jiao Tong University, [3]Peking University
[4]Minzu University of China, [5]Zhongguancun Academy
[6]Beijing Key Laboratory of Data Intelligence and Security (Peking University)
kernel@buaa.edu.cn, pzp0057@sjtu.edu.cn, zywang111@buaa.edu.cn
zhouliu25@stu.pku.edu.cn, rickamorty@muc.edu.cn, wentao.zhang@pku.edu.cn
*Equal contribution. †Corresponding author.

Abstract

Paper-to-code reproduction asks scientific AI agents to turn research papers into executable repositories that preserve the paper's method, protocol and artifacts. This is difficult because the specification is split: explicit paper content such as algorithms, metrics and artifacts is often lost across long agent trajectories, while implicit details such as framework defaults and conventions inherited from related work are absent from the paper. We introduce REPROAGENT, a four-stage Prepare–Plan–Generate–Repair pipeline built around a persistent implementation contract with two channels: an implementation-requirement channel that turns paper snippets into code obligations, and a reference-evidence channel that retrieves content and structure evidence from related repositories. Both are bound to work packages, projected into file-level contracts, and consumed across generation and repair. On PaperBench Code-Dev (Starace et al., 2025), ReproAgent reaches the highest mean score among same-backbone scaffolds under both Claude-Sonnet-4.5 and Gemini-3-Flash. End-to-end channel ablations and per-paper cases support the contribution of both channels. Code and experimental artifacts are publicly available.[1]

1 Introduction

Large language models are increasingly used to assist scientific research, but a long-standing reproducibility crisis persists (Baker, 2016; Pineau et al., 2021). Systematic reviews of NLP reproduction studies report that only a small fraction of attempted reproductions recover the original scores (Belz et al., 2021), and even strong LLM judges detect only 46.7% of real-world discrepancies between papers and their accompanying code (Baumgärtner and Gurevych, 2026). Together, faster method production and persistent reproducibility gaps make it harder to validate new methods, compare baselines fairly, and build on prior work. An automated framework that turns each new paper into a faithful, runnable repository, rather than merely executable code, would make paper-specific methods, protocols, and artifacts easier to inspect and reuse.

Recent paper-to-code agents take a step in this direction (Seo et al., 2026; Xiang et al., 2025; Zhao et al., 2025; Zhou et al., 2025), but a substantial fidelity gap remains. A recurring failure mode is runnable but unfaithful code: a generated repository may import correctly, pass smoke tests, and even produce plausible experimental results, while still dropping or misimplementing key paper-specific details, such as substituting a generic training loop for the paper's algorithm, omitting the decisive metric or artifact, or altering a loss term. Runtime success alone is therefore an insufficient proxy for faithful paper reproduction.

We identify a split-specification problem behind these failures. A paper states some implementation requirements explicitly, including algorithms, metrics, artifacts and evaluation protocols. However, many decisions needed to build a faithful repository are implicit in framework defaults, cited implementations and community conventions. Existing paper-to-code agents mainly condition generation on the paper text and transient planning context, without a persistent object that carries paper-derived obligations or binds implicit details to external evidence. As a result, explicit requirements drift out of context across long trajectories, while implicit details fall back on model priors rather than grounded evidence. Faithful reproduction therefore needs a persistent implementation contract that preserves paper-derived obligations and attaches external repository evidence to the implementation decisions it supports.

REPROAGENT instantiates this idea with a global implementation contract containing two channels. The implementation-requirement chan-

[1]https://github.com/kernel-14/ReproAgent.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 3681–3702
October 24-29, 2026 ©2026 Association for Computational Linguistics

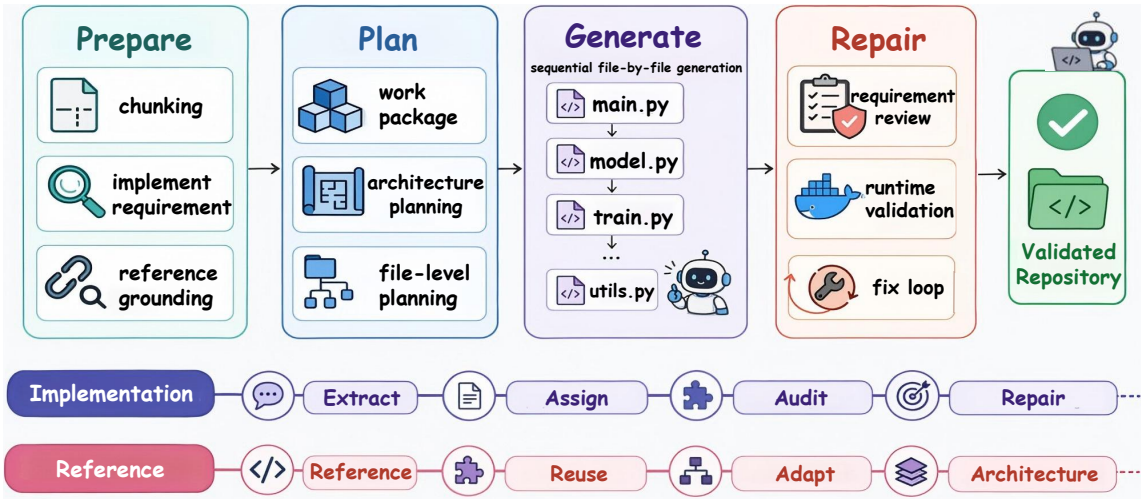

Figure 1: Overview of REPROAGENT. A four-stage pipeline, Prepare, Plan, Generate, and Repair, carries an implementation-requirement channel and a reference-evidence channel into a shared file-level implementation contract. The implementation-requirement channel preserves paper-derived code obligations across generation and repair, while the reference-evidence channel supplies package-local content and structure evidence from related repositories.

nel extracts explicit paper knowledge as code obligations, while the reference-evidence channel grounds implicit repository knowledge in related implementations. Both channels are bound to work packages and projected into file-level contracts, so the same contract guides generation and serves as the audit target for detecting missing obligations, unsupported implementation choices, broken interfaces, and runtime failures.

REPROAGENT carries this contract through a four-stage pipeline. Prepare segments the paper, extracts implementation requirements, and builds a reference-repository index. Plan organizes requirements into work packages, binds package-local reference evidence, derives the target architecture, and freezes file-level contracts over obligations, evidence, validation checks, and file responsibilities. Generate produces the repository file by file under these local contracts. Repair audits the result against the same contract together with runtime checks, so failures are reported as violated paper-derived obligations, missing evidence use, broken interfaces, or execution failures rather than only as crashing commands.

We evaluate REPROAGENT on PaperBench Code-Dev (Starace et al., 2025), using repository-level rubric grading to test whether generated code implements the target paper rather than merely executing. We run REPROAGENT with two backbones, Claude-Sonnet-4.5 and Gemini-3-Flash, and under each backbone REPROAGENT reaches the highest mean score among same-backbone scaffolds. Ablations of the two contract channels together with a per-paper case analysis support that both the implementation-requirement and reference-evidence channels contribute to faithful reproduction.

Our main contributions are:

- We formulate paper-to-code reproduction as contract-guided repository construction under partial paper specifications, where faithful generation requires preserving explicit paper obligations and grounding implicit implementation choices.
- We introduce a two-channel implementation contract over paper-derived requirements and related-repository evidence, with work-package bindings, file-level obligations, provenance records, and typed requirement-review/runtime reports that preserve both traces across generation and repair.
- We instantiate this contract in REPROAGENT, a four-stage Prepare–Plan–Generate–Repair pipeline, and evaluate it on the full 20-paper PaperBench Code-Dev suite with same-backbone comparisons and ablations that remove each contract channel.

2 Related Work

LLM agents for research and code. LLM agents have been applied to research workflows (Baek et al., 2024; Lu et al., 2024; Schmidgall

et al., 2025), ML engineering (Chan et al., 2024; Tang et al., 2024), and code-generation or repository-level coding settings (Qian et al., 2024; Islam et al., 2024; Hong et al., 2024; Yang et al., 2024; Xia et al., 2025; Zhang et al., 2024). Paper reproduction differs because the target is a scientific procedure described only partially in natural language: the repository must be faithful to paper-specific methods, protocols, and artifacts, not merely pass a generic execution check.

Paper-to-code reproduction. Following DLPaper2Code (Sethi et al., 2018), recent systems extend reproduction to whole repositories: PaperCoder, the system introduced by Seo et al. (Seo et al., 2026), stages planning, analysis, and coding; Sci-Reproducer (Xiang et al., 2025) highlights missing-algorithm bottlenecks; AutoP2C (Lin et al., 2025) and HiRAS (Hong et al., 2026) push multimodal parsing and hierarchical supervision; AutoReproduce (Zhao et al., 2025) retrieves along citation graphs; DeepCode (Li et al., 2025b) treats reproduction as agentic coding. To our knowledge, prior systems do not make both paper-derived obligations and retrieved repository evidence a persistent, file-level audit object shared across planning, generation, and repair, which is the traceability layer REPROAGENT targets.

Reference retrieval and contracts. Retrieval-augmented code generation depends on retrieval granularity (Chen et al., 2024) and pulls evidence from documentation (Zhou et al., 2023; Pimparkhede et al., 2024), repositories (Zhang et al., 2023; Cheng et al., 2024), or code graphs (Li et al., 2025a), but does not tie each snippet to the scientific requirement it supports. Our contract view connects to requirements traceability and design by contract (Gotel and Finkelstein, 1994; Meyer, 1992; Cleland-Huang et al., 2014), adapted to settings where requirements are extracted from paper snippets rather than written formally. REPROAGENT therefore splits reference evidence into content evidence for reuse or adaptation and structure evidence for architecture planning, and binds both to work-package and file-level contracts that persist across the pipeline.

Evaluation and fidelity. PaperBench (Starace et al., 2025) grades reproductions against rubrics, with related benchmarks studying reproducibility and paper-code mismatch (Yan et al., 2025; Siegel et al., 2024; Baumgärtner and Gurevych, 2026). RePro (Zhou et al., 2025) applies paper-derived criteria as post-hoc refinement, after a candidate repository exists. REPROAGENT differs by placing the same paper-derived requirements into the forward generation contract, so the audit target is fixed before generation rather than reconstructed afterwards.

3 Method

REPROAGENT turns a paper into a repository by maintaining two persistent contract channels, shown in Figure 2. The implementation-requirement channel preserves what the target paper explicitly says must be implemented. The reference-evidence channel supplies evidence for implicit repository knowledge: implementation conventions, file structure, entry points, and protocol details from cited or related repositories. Both channels are bound to work packages and projected into file-level contracts, so information extracted early remains visible during planning, generation, validation, and repair. A work package is a small planning unit that bundles related requirements with the evidence supporting them; we give its full schema in §3.3.

3.1 Implementation Requirements

An implementation requirement is a paper snippet that must be realized in code, and we represent it as a unit

$$u = (\mathrm{id}, \mathrm{src}, \mathrm{stmt}, \mathrm{cite}, \mathrm{surf}, \mathrm{art}, \mathrm{status}).$$

Here id is a stable identifier, src records the paper source span the unit is drawn from, and stmt is a normalized statement of the obligation. The remaining four fields specify how the unit travels through the rest of the pipeline: cite stores explicit reference cues when they are present in the paper, and the reference-evidence channel attaches prior-work snippets through unit IDs, source anchors, and match records (§3.2); surf names the expected code surfaces, such as a model class, loss function, optimizer, metric, CLI option, config field, training script, or artifact writer; art enumerates the artifacts the unit should produce; and status is a lifecycle flag that stays active until the unit is realized. A concrete example of such a unit is given in Appendix A.1.

Extraction. We first segment the paper into section-level chunks, falling back to paragraph-level splits when a section is too long for a sin-

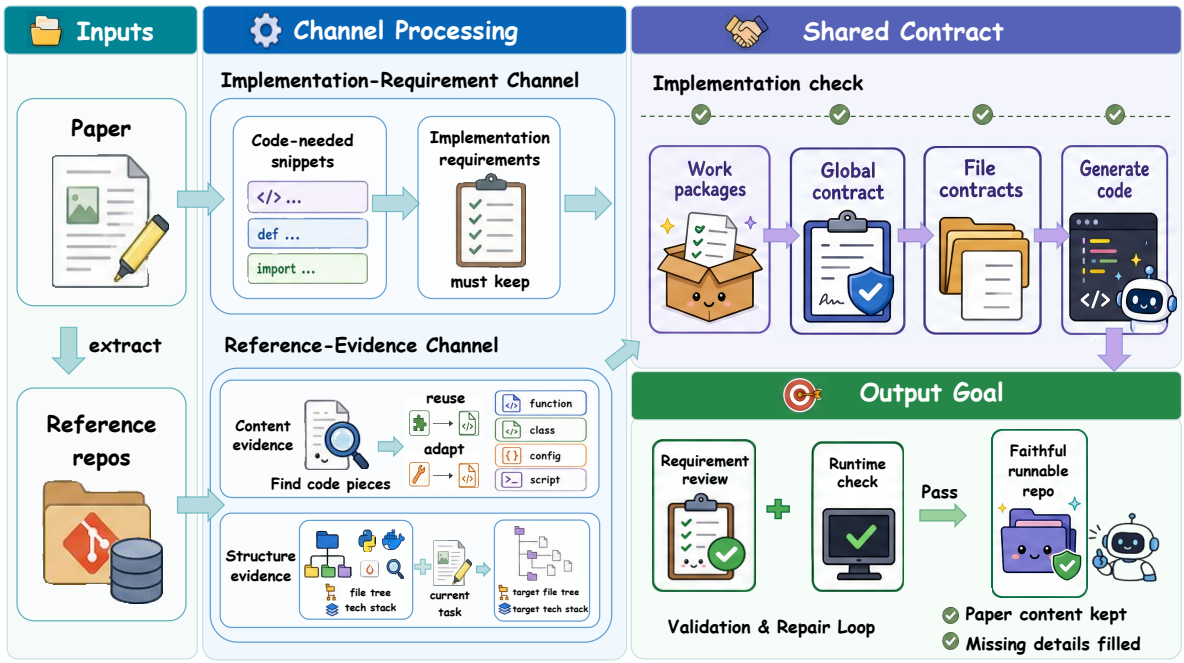

Figure 2: REPROAGENT overview. The implementation-requirement channel extracts code-facing obligations from the target paper, while the reference-evidence channel retrieves content and structure evidence from related repositories. The shared contract binds both channels into work packages, global and file-level contracts, generation, and validation, yielding a repository that is checked for both requirement fidelity and runtime correctness. Representative unit and evidence records appear in Appendix A.

gle LLM pass. From each chunk, we extract only snippets that imply concrete implementation work, such as algorithmic steps, loss formulations, optimizer and training settings, dataset and preprocessing rules, and reported metrics or artifacts; pure motivation, related work narration, and discussion are skipped.

Preservation invariant. To make this preservation operational, we attach a coverage invariant to every active unit so that any silent loss surfaces as a checkable failure inside the pipeline. For every active unit $u \in \mathcal{U}_{\text{active}}$, the contract requires

$$\text{own}(u),\ \text{proj}(u),\ \text{prov}(u) \neq \emptyset,$$
$$\text{verdict}(u) \in \mathcal{Y},$$

where $\mathcal{Y}$ is the fixed verdict set returned by requirement review, with four possible values: pass, missing, wrong-surface and inconsistent-with-source. The four predicates correspond to four pipeline stages:

- **Ownership.** Plan assigns the unit to at least one work package.
- **File projection.** File planning assigns the unit to at least one file contract whose declared surfaces cover the unit's expected surfaces.
- **Provenance.** Generation emits a record linking the produced file region back to the unit.
- **Verdict.** Requirement review returns a value in $\mathcal{Y}$.

A failed predicate creates a contract violation report and a repair obligation; if the repair budget is exhausted, the unresolved verdict is retained in the diagnostics rather than silently absorbed.

3.2 Reference Evidence

Papers often leave implementation details implicit: a paper may cite a prior method without restating its update rule, or refer to a benchmark's "standard" data augmentation whose canonical form lives in a reference repository. REPROAGENT therefore collects reference evidence from related repositories listed in the paper's bibliography, and represents each evidence item as

$$e = (\text{repo}, \text{path}, \text{region}, \text{units}, \text{role}, \text{basis}).$$

Here repo identifies the source repository, path the file inside it, and region the matched line range; units lists the requirement IDs the snippet supports, role records whether the snippet is meant for local reuse or for adaptation as an interface pattern, and basis is a structured match record explaining why

the snippet is relevant. Each item is bound to the work package that owns its supported units, so reference code enters generation through a specific package contract rather than as global context. A concrete evidence item is given in Appendix A.2.

Content evidence. For each candidate reference repository, REPROAGENT indexes file paths, imports, exported symbols, scripts, configurations, and entry points described in the README. A snippet is retained when one of three signals connects it to an owned unit, an expected artifact, or a declared implementation surface: a symbol or interface name match, a path or configuration cue, or a structured LLM judgment. The judgment is stored in basis as the supported unit IDs, the matching surface or artifact, and whether the snippet should be reused locally or adapted as an interface pattern, rather than as a free-form note. The role field then acts as a use constraint: local reuse covers reusable formulas or transforms, while interface adaptation covers structural patterns that should be re-implemented under the package's own interface, not copied wholesale.

Structure evidence. Reference repositories also expose where code should live: file trees, entry scripts, module boundaries, configuration conventions, dependency files, and technology-stack hints. Architecture planning combines these structural priors with the task and requirements to decide the target file tree, stable interfaces, entry points, and dependency graph. Content evidence therefore informs how to implement an obligation, while structure evidence informs where the implementation should live.

3.3 Work Packages and the Global Contract

A work package is the local planning unit that binds requirements to evidence. Each package records its goal, the requirement IDs it owns, its dependencies on other packages, the public interface it exposes, the artifacts it should produce, and the evidence items attached to it.

After evidence binding, the planner freezes a global contract

$$\mathcal{G} = (\mathcal{U}, \mathcal{W}, \mathcal{E}, \mathcal{A}, \mathcal{V}),$$

where $\mathcal{U}$ is the active requirement set, $\mathcal{W}$ the work packages, $\mathcal{E}$ the package-bound evidence, $\mathcal{A}$ the target architecture, and $\mathcal{V}$ the validation plan. Once $\mathcal{G}$ is frozen, later stages can refine local implementation choices and add derived helper files, but the active requirements, package ownerships, and evidence-package bindings carried by $\mathcal{G}$ stay fixed. File planning then projects $\mathcal{G}$ into file contracts $\mathcal{C}_F = \{c_f\}$, where each c_f specifies its target path, owner package, owned units, imported and exported symbols, attached evidence, produced artifacts, and review checks. Every generated file therefore has a per-file specification grounded in the global contract.

3.4 Generation, Validation, and Repair

Generation proceeds file by file in dependency order. Each call is conditioned on the file contract c_f, the evidence attached to its owner package, and the already-generated dependency files needed for the local interface; reference code from sibling packages is not injected as global context. This package-local scoping keeps each produced file auditable: a training loop is generated against training requirements and training evidence, while an evaluation script is generated against metric and artifact requirements. Each generated file returns a provenance record linking file regions back to the requirement and evidence IDs they realize, and lists implementation choices that are inferred rather than directly specified by the contract.

Validation produces two typed reports. Requirement review evaluates each active unit against its file contract and provenance record, and returns a verdict in $\mathcal{Y}$. Runtime validation runs dependency installation, import checks, declared entry points, smoke commands, and benchmark commands when available. The two reports surface different failures: runtime validation catches broken code and missing artifacts, while requirement review catches runnable but paper-inconsistent implementations.

When either report fails, REPROAGENT issues a repair ticket rather than regenerating the repository. The ticket carries the failed units, target files, attached evidence, the edits required, and the units that already passed and must remain protected. Repair is file-local unless the ticket identifies an interface-level failure that requires coordinating two or more files, and repository regions whose contracts are unchanged skip redundant validation. The repair loop is bounded by a fixed budget; we use the same budget within each backbone setting and report the setting-specific value in §4.3.

4 Experimental Setup

4.1 Benchmark and Backbones

We evaluate on **PaperBench Code-Dev** (Starace et al., 2025), which contains 20 ICML 2024 papers and hierarchical rubrics for repository-level code reproduction; the rubrics grade whether a generated repository implements paper-specific methods, protocols, and artifacts rather than only running. On this benchmark, the evaluation has three layers. The full-suite layer compares the primary REPROAGENT run with reported PaperBench Code-Dev systems, using Claude-Sonnet-4.5[2] as the generation backbone. The same-backbone layer compares REPROAGENT with simpler scaffolds under Gemini-3-Flash[3], the backbone shared with BasicAgent, IterAgent and AiScientist. The ablation layer removes each contract channel under the same Gemini-3-Flash backbone.

4.2 Comparisons and Metrics

External rows reuse the model, scaffold, and runtime budget reported by their source. The AiScientist engineering report provides the BasicAgent, IterAgent, and AiScientist scaffold rows used in the same-backbone comparison. The aggregate full-suite figure includes only Code-Dev rows, and the per-paper table expands the main comparison to BasicAgent, IterAgent, AiScientist, DeepCode, and both REPROAGENT backbones; Appendix B lists the additional external rows.

We follow the Code-Dev judging protocol and report macro-averaged PaperBench score as the primary metric: weighted leaf scores aggregate to a per-paper score, and the 20 paper scores are averaged. Token, time, cost, and judge-provenance diagnostics are reported in the appendix.

4.3 Implementation and Ablations

REPROAGENT is implemented as a four-stage pipeline: the paper is chunked by section, reference repositories come from PaperBench metadata and related-code search subject to the benchmark blacklist on official target repositories, and planning freezes a global contract before file-level generation. Repair combines requirement review with sandboxed runtime validation, so failures become targeted tickets that patch only affected files.

[2] https://platform.claude.com/docs/en/about-claude/models/model-ids-and-versions.

[3] https://ai.google.dev/gemini-api/docs/gemini-3.

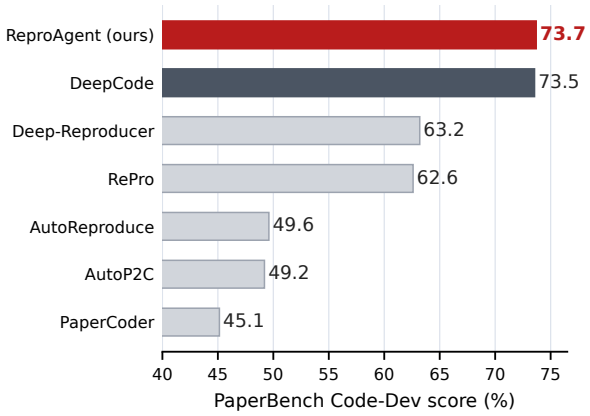

Figure 3: Full-suite PaperBench Code-Dev comparison across reported paper-to-code systems.

We cap stage-review repair at three attempts and repository-level repair at five rounds, uniformly across the Claude-Sonnet-4.5 main run and all Gemini-3-Flash full and ablation runs; holding this budget fixed isolates the channel effect from repair-budget effects. In total we report 20 papers $\times$ 4 REPROAGENT settings = 80 reproduction runs, one run per (paper, setting) pair.

Channel ablations. On the same Gemini-3-Flash backbone, No Implementation Requirements skips requirement extraction and requirement review, so architecture planning falls back on the paper's section outline and reference-derived module structure; No Reference Evidence skips reference-repository surveying, so generation relies on requirement-derived contracts without bound content or structure evidence. All other components and repair budgets are held fixed.

5 Results and Analysis

5.1 Full-Suite Comparison

Figure 3 reports mean PaperBench Code-Dev scores across published paper-to-code systems. REPROAGENT with Claude-Sonnet-4.5 reaches 73.7, above PaperCoder (Seo et al., 2026) (45.1), AutoP2C (Lin et al., 2025) (49.2), AutoReproduce (Zhao et al., 2025) (49.6), and Deep-Reproducer (Chen et al., 2025) (63.2), and edges past the concurrent state-of-the-art DeepCode at 73.5.

Tables 1 and 2 expand this view into the 20 targets for each backbone; additional external rows appear in Appendix B. We hypothesize that the gain comes from the contract itself: requirement and evidence bindings that survive across files keep

Paper	BasicAgent	IterAgent	DeepCode	REPROAGENT (ours)
Adaptive Pruning	16.6	22.4$_{\uparrow 5.8}$	54.4$_{\uparrow 37.8}$	**67.8**$_{\uparrow 51.2}$
All in One	24.8	11.5$_{\downarrow 13.3}$	75.9$_{\uparrow 51.1}$	**76.9**$_{\uparrow 52.1}$
BAM	25.2	18.7$_{\downarrow 6.5}$	**74.8**$_{\uparrow 49.6}$	61.8$_{\uparrow 36.6}$
BBOX	10.7	15.2$_{\uparrow 4.5}$	64.4$_{\uparrow 53.7}$	**86.4**$_{\uparrow 75.7}$
Bridging Data Gaps	15.8	8.1$_{\downarrow 7.7}$	58.1$_{\uparrow 42.3}$	**66.0**$_{\uparrow 50.2}$
FRE	22.7	17.5$_{\downarrow 5.2}$	**81.4**$_{\uparrow 58.7}$	76.8$_{\uparrow 54.1}$
FTRL	9.3	4.3$_{\downarrow 5.0}$	59.8$_{\uparrow 50.5}$	**63.9**$_{\uparrow 54.6}$
LBCS	19.8	20.7$_{\uparrow 0.9}$	**74.7**$_{\uparrow 54.9}$	65.8$_{\uparrow 46.0}$
LCA on the Line	14.0	13.3$_{\downarrow 0.7}$	**74.9**$_{\uparrow 60.9}$	64.9$_{\uparrow 50.9}$
Mechanistic Understanding	18.1	3.7$_{\downarrow 14.4}$	**92.5**$_{\uparrow 74.4}$	75.7$_{\uparrow 57.6}$
PINN	24.8	22.5$_{\downarrow 2.3}$	**91.0**$_{\uparrow 66.2}$	81.5$_{\uparrow 56.7}$
RICE	19.8	7.8$_{\downarrow 12.0}$	70.2$_{\uparrow 50.4}$	**75.1**$_{\uparrow 55.3}$
Robust CLIP	29.6	24.7$_{\downarrow 4.9}$	**73.3**$_{\uparrow 43.7}$	63.0$_{\uparrow 33.4}$
Sample-specific Masks	30.9	25.7$_{\downarrow 5.2}$	67.1$_{\uparrow 36.2}$	**82.0**$_{\uparrow 51.1}$
SAPG	5.4	6.3$_{\uparrow 0.9}$	73.8$_{\uparrow 68.4}$	**86.5**$_{\uparrow 81.1}$
Sequential Neural Score Estimation	41.4	32.7$_{\downarrow 8.7}$	**87.0**$_{\uparrow 45.6}$	78.0$_{\uparrow 36.6}$
Stay on Topic with Classifier-Free Guidance	10.8	16.2$_{\uparrow 5.4}$	**70.5**$_{\uparrow 59.7}$	58.5$_{\uparrow 47.7}$
Stochastic Interpolants	12.4	26.3$_{\uparrow 13.9}$	**81.5**$_{\uparrow 69.1}$	80.8$_{\uparrow 68.4}$
Test-Time Model Adaptation	14.3	14.2$_{\downarrow 0.1}$	64.9$_{\uparrow 50.6}$	**75.6**$_{\uparrow 61.3}$
What Will My Model Forget	29.7	11.4$_{\downarrow 18.3}$	80.8$_{\uparrow 51.1}$	**87.6**$_{\uparrow 57.9}$
Mean score	19.8	16.2$_{\downarrow 3.6}$	73.5$_{\uparrow 53.7}$	**73.7**$_{\uparrow 53.9}$

Table 1: **Per-paper PaperBench Code-Dev comparison with Claude backbones.** Claude columns for BasicAgent and IterAgent follow the PaperBench report (Starace et al., 2025); DeepCode follows its concurrent report (Li et al., 2025b). BasicAgent and IterAgent use Claude-3.5-Sonnet, while DeepCode and REPROAGENT (ours) use Claude-4.5-Sonnet. Subscripts show absolute-point differences from BasicAgent on the same paper; red upward and green downward arrows indicate the sign of the difference. **best** and second-best mark the row-best and row-second-best scores among the shown Claude columns.

dependent artifacts, such as method, training, and evaluation code, consistent during generation and repair instead of being re-derived per stage.

5.2 Same-Backbone Comparison

Holding the backbone fixed at Gemini-3-Flash, REPROAGENT reaches 39.7 against 19.3 for BasicAgent, 20.6 for IterAgent, and 30.5 for AiScientist. The advantage over the strongest external Gemini row is on the order of 9 absolute points, consistent with the contract effect we hypothesize in the full-suite comparison.

5.3 Ablation Study

We next isolate the contribution of each contract channel under Gemini-3-Flash. Table 3 summarizes the aggregates: removing reference evidence drops the mean by 18.1 and removing implementation requirements drops it by 14.1. The full setting beats both ablations on every one of the 20 targets, so each channel contributes capability that the other cannot recover.

Requirement channel: Sample-specific Masks. Sample-specific Masks scores 53.8 with the full

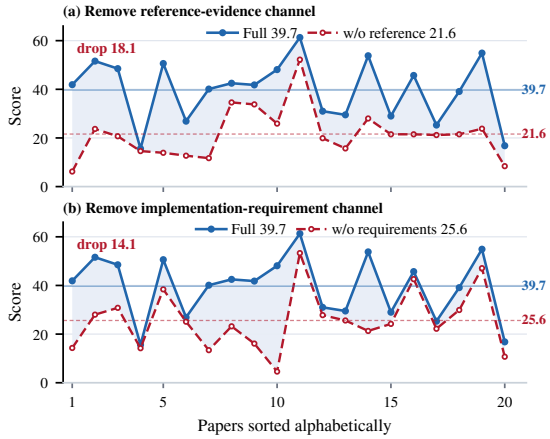

Figure 4: Gemini channel ablations on all 20 papers. Removing reference evidence (top) or implementation requirements (bottom) lowers the mean; judge provenance and token diagnostics are in the appendix.

contract, 21.3 without implementation requirements, and 28.0 without reference evidence: the requirement drop is the dominant one. The paper's contribution is a small set of named, implementation-critical obligations, including a per-sample mask generator, interpolation from a

Paper	BasicAgent	IterAgent	AiScientist	REPROAGENT (ours)
Adaptive Pruning	24.5	$3.0_{\downarrow 21.5}$	$27.2_{\uparrow 2.7}$	$\mathbf{41.9}_{\uparrow 17.4}$
All in One	20.9	$45.1_{\uparrow 24.2}$	$46.3_{\uparrow 25.4}$	$\mathbf{51.6}_{\uparrow 30.7}$
BAM	48.5	$45.0_{\downarrow 3.5}$	$\mathbf{56.6}_{\uparrow 8.1}$	48.5
BBOX	15.4	$8.3_{\downarrow 7.1}$	$\mathbf{33.8}_{\uparrow 18.4}$	$15.6_{\uparrow 0.2}$
Bridging Data Gaps	12.6	$12.4_{\downarrow 0.2}$	$23.1_{\uparrow 10.5}$	$\mathbf{50.6}_{\uparrow 38.0}$
FRE	21.7	$23.9_{\uparrow 2.2}$	$\mathbf{35.2}_{\uparrow 13.5}$	$26.9_{\uparrow 5.2}$
FTRL	5.9	$4.2_{\downarrow 1.7}$	$10.1_{\uparrow 4.2}$	$\mathbf{40.1}_{\uparrow 34.2}$
LBCS	17.8	$15.3_{\downarrow 2.5}$	$27.9_{\uparrow 10.1}$	$\mathbf{42.5}_{\uparrow 24.7}$
LCA on the Line	13.0	$18.3_{\uparrow 5.3}$	$30.2_{\uparrow 17.2}$	$\mathbf{41.8}_{\uparrow 28.8}$
Mechanistic Understanding	14.9	$21.9_{\uparrow 7.0}$	$29.9_{\uparrow 15.0}$	$\mathbf{48.1}_{\uparrow 33.2}$
PINN	26.6	$30.8_{\uparrow 4.2}$	$49.9_{\uparrow 23.3}$	$\mathbf{61.3}_{\uparrow 34.7}$
RICE	10.4	$8.9_{\downarrow 1.5}$	$10.9_{\uparrow 0.5}$	$\mathbf{31.0}_{\uparrow 20.6}$
Robust CLIP	15.4	$10.4_{\downarrow 5.0}$	$18.3_{\uparrow 2.9}$	$\mathbf{29.5}_{\uparrow 14.1}$
Sample-specific Masks	25.4	$33.3_{\uparrow 7.9}$	$36.8_{\uparrow 11.4}$	$\mathbf{53.8}_{\uparrow 28.4}$
SAPG	11.4	$12.7_{\uparrow 1.3}$	$19.9_{\uparrow 8.5}$	$\mathbf{29.0}_{\uparrow 17.6}$
Sequential Neural Score Estimation	53.5	$60.2_{\uparrow 6.7}$	$\mathbf{64.9}_{\uparrow 11.4}$	$45.7_{\downarrow 7.8}$
Stay on Topic with Classifier-Free Guidance	8.4	$13.7_{\uparrow 5.3}$	$20.1_{\uparrow 11.7}$	$\mathbf{25.3}_{\uparrow 16.9}$
Stochastic Interpolants	17.0	$17.4_{\uparrow 0.4}$	$18.8_{\uparrow 1.8}$	$\mathbf{39.1}_{\uparrow 22.1}$
Test-Time Model Adaptation	15.3	$18.1_{\uparrow 2.8}$	$32.5_{\uparrow 17.2}$	$\mathbf{54.9}_{\uparrow 39.6}$
What Will My Model Forget	6.6	$9.0_{\uparrow 2.4}$	$\mathbf{17.9}_{\uparrow 11.3}$	$16.8_{\uparrow 10.2}$
Mean score	19.3	$20.6_{\uparrow 1.3}$	$30.5_{\uparrow 11.2}$	$\mathbf{39.7}_{\uparrow 20.4}$

Table 2: **Per-paper PaperBench Code-Dev comparison with Gemini.** Gemini columns for BasicAgent, IterAgent, and AiScientist follow the AiScientist engineering report (Chen et al., 2026); all Gemini columns use Gemini-3-Flash. Subscripts show absolute-point differences from BasicAgent on the same paper; red upward and green downward arrows indicate the sign of the difference. **best** and second-best mark the row-best and row-second-best scores among the shown Gemini columns.

Setting	Mean	Median	Δ
Full contract	39.7	41.8	—
− Reference evidence	21.6	21.4	−18.1
− Implementation requirements	25.6	24.6	−14.1

Table 3: Gemini-3-Flash ablation results over 20 papers; Δ is the mean drop from the full contract.

low-resolution mask up to the input image size, joint optimization of a shared pattern and the mask-generator parameters, the PAD/NARROW/MEDIUM/FULL baselines, and ablations over shared-pattern and mask channels. The full contract pins these obligations to specific work packages and files, so the generated repository keeps the baseline and ablation method registries that the rubric checks. Without the requirement channel, the planner falls back on the section outline and reference-derived modules, and the generated repository keeps runnable scaffolding but drops the named baselines, ablations, and mask-channel switches, since none of them are pinned to a file. The case-level code excerpt and a per-paper score table for additional cases appear in Appendix C.

Evidence channel: Bridging Data Gaps. Bridging Data Gaps scores 50.6 with the full contract, 38.4 without implementation requirements, and 13.9 without reference evidence: the evidence drop is the dominant one. The paper names the high-level task and adaptor objectives, but its concrete protocol surface, including file layout, training-loop wiring, checkpoint and trace artifacts, and configuration handling, is supplied by reference repositories rather than the paper text. With the full contract, the generated repository emits the expected checkpoint, training-trace, and resolved-configuration artifacts under conventional output paths, matching the pattern of checkpointed training and reproducible configuration logging. Without the evidence channel, paper obligations remain visible in the plan, but the generated repository under-implements file layout, data-handling conventions, experiment wiring, and artifact naming, so it fails to produce these expected artifacts. Appendix C gives the corresponding code-level trace and Appendix Table 6 lists the per-paper scores behind these aggregates.

5.4 Discussion

Two observations stand out. First, the same-backbone advantage and the full-suite gain move together, so the lift attaches to the contract rather than to any single backbone. Second, the two channels are complementary: removing either drops the mean by more than 14 points, and the failures fall into two classes, paper-specific obligations dropped without the requirement channel and repository-level scaffolding lost without the evidence channel. Together, these support a mechanistic claim: REPROAGENT benefits from a global contract whose requirement channel preserves explicit paper obligations and whose evidence channel grounds implicit repository knowledge.

6 Conclusion

We presented REPROAGENT, a contract-guided pipeline for paper-to-code reproduction. Its global contract binds explicit paper obligations and implicit reference-repository evidence to work packages, file contracts, and repair decisions. On PaperBench Code-Dev, REPROAGENT scores 73.7 across all 20 papers; same-backbone rows support scaffold-level gains, while channel ablations and case traces support the role of both contract channels. The takeaway is that paper obligations and reference-repository evidence are most useful when persisted as a contract: once both are bound, the same artifact anchors planning, exposes coverage failures, and scopes repair.

7 Limitations

REPROAGENT targets repository-level reproduction in which the paper and benchmark-allowed reference material jointly specify the implementation target. Extending the approach to other reproduction settings, modalities, and evaluation protocols is a natural direction for future work.

We report token, time, and call diagnostics in Appendix G to make the process auditable rather than to compare deployment cost; further engineering optimizations are compatible with the contract design and are also left to future work.

Acknowledgments

This work is supported by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113), National Natural Science Foundation of China (92470121, 62402016), National Key R&D Program of China (2024YFA1014003), Zhongguancun Academy (C20250204, C20250602), Beijing Major Science and Technology Project (Z251100008125043, Z251100008425023), and High-performance Computing Platform of Peking University.

References

Jinheon Baek, Sukhdeep Singh Jauhar, Silviu Cucerzan, and Sung Ju Hwang. 2024. ResearchAgent: Iterative research idea generation over scientific literature with large language models. *arXiv preprint arXiv:2404.07738*.

Monya Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454.

Tim Baumgärtner and Iryna Gurevych. 2026. SciCoQA: Quality assurance for scientific paper–code alignment. *arXiv preprint arXiv:2601.12910*.

Anya Belz, Shubham Agarwal, Anastasia Shimorina, and Ehud Reiter. 2021. A systematic review of reproducibility research in natural language processing. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 381–393.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2024. MLE-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.

Guoxin Chen, Jie Chen, Lei Chen, Jiale Zhao, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Cheng Chen, Ji-Rong Wen, and Kai Jia. 2026. Toward autonomous long-horizon engineering for ML research. *arXiv preprint arXiv:2604.13018*.

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

Tong Chen, Hongwei Wang, Sihao Chen, Wenhao Yu, Kaixin Ma, Xinran Zhao, Hongming Zhang, and Dong Yu. 2024. Dense X retrieval: What retrieval granularity should we use? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7957–7977, Bangkok, Thailand. Association for Computational Linguistics.

Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: Trends and future directions. In *Proceedings of the Future of Software Engineering (FOSE)*, pages 55–69.

Orlena C. Z. Gotel and Anthony C. W. Finkelstein. 1994. An analysis of the requirements traceability problem. In *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE)*, pages 94–101.

Hanhua Hong, Yizhi Li, Jiaoyan Chen, Sophia Ananiadou, Xiaoli Li, Jung-jae Kim, and Chenghua Lin. 2026. HiRAS: A hierarchical multi-agent framework for paper-to-code generation and execution. *arXiv preprint arXiv:2604.17745*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation mirroring human cognitive processes. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Jia Li, Xianjie Shi, Kechi Zhang, Ge Li, Zhi Jin, Lei Li, Huangzhao Zhang, Fang Liu, Yuwei Zhang, Zhengwei Tao, Yihong Dong, Yuqi Zhu, and Chongyang Tao. 2025a. GraphCodeAgent: Dual graph-guided LLM agent for retrieval-augmented repo-level code generation. *arXiv preprint arXiv:2504.10046*.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. 2025b. DeepCode: Open agentic coding. *arXiv preprint arXiv:2512.07921*.

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

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*.

Bertrand Meyer. 1992. Applying "design by contract". *Computer*, 25(10):40–51.

Sameer Pimparkhede, Sameer Kammakomati, Srikanth Tamilselvam, Prince Kumar, Ashok Bhatt, and Pushpak Kumar. 2024. DocCGen: Document-based controlled code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*.

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

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings*

of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. 2025. Agent laboratory: Using LLM agents as research assistants. *arXiv preprint arXiv:2501.04227*.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2026. Paper2Code: Automating code generation from scientific papers in machine learning. In *The Fourteenth International Conference on Learning Representations*.

Akshay Sethi, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. 2018. DLPaper2Code: Auto-generation of code from deep learning research papers. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, page 899. AAAI Press.

Zachary S. Siegel, Sayash Kapoor, Nitya Nagdir, Benedikt Stroebl, and Arvind Narayanan. 2024. CORE-Bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *Transactions on Machine Learning Research (TMLR)*.

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. 2025. PaperBench: Evaluating AI's ability to replicate AI research. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Liang Chen, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yin Fang, and 5 others. 2024. ML-Bench: Evaluating large language models and agents for machine learning tasks on repository-level code. *Preprint*, arXiv:2311.09835.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE).

Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. 2025. SciReplicate-Bench: Benchmarking LLMs in agent-driven algorithmic reproduction from research papers. In *Proceedings of the Conference on Language Modeling (COLM)*. ArXiv:2504.00255.

Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, Juntong Ni, George Michalopoulos, Yue Zhang, Ziyang Zhang, Mian Zhang, Zhiyu Chen, and Xinya Du. 2025. LMR-BENCH: Evaluating LLM agent's ability on reproducing language modeling research. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 6164–6186, Suzhou, China. Association for Computational Linguistics.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. Curran Associates Inc.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Chen, and Wenqiang Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025. AutoReproduce: Automatic AI experiment reproduction with paper lineage. *arXiv preprint arXiv:2505.20662*.

Mingyang Zhou, Quanming Yao, Lun Du, Lanning Wei, and Da Zheng. 2025. Reflective paper-to-code reproduction enabled by fine-grained verification. *arXiv preprint arXiv:2508.16671*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Wang, Zhiruo Jiang, and Graham Neubig. 2023. DocPrompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

A Contract Case Study: Unit and Evidence Records

This appendix gives two paired schema records for a continual-RL paper that introduces a forgetting-aware fine-tuning protocol. The first illustrates the implementation-requirement schema in §3.1; the second illustrates the reference-evidence schema in §3.2 and is anchored to the unit from the first. The records are compact schema views that preserve source anchors and evidence bindings used by review and repair.

A.1 An Implementation-Requirement Unit

The unit below illustrates how Prepare extracts a requirement, Plan assigns it to a method work package, Generate projects it into a model-method file contract, and Repair verifies it with requirement review.

Schema-aligned `units.json` view for the EWC running example

```
{
  "unit_id": "unit_004",
  "type": "method",
  "statement": "implement Elastic Weight
    Consolidation (EWC)",
  "paper_evidence": [
    "L_EWC = L_RL + (lambda/2) * sum_i F_i (
    theta_i - theta_*_i)^2",
    "where theta_* is the pre-trained model
    and F is the diagonal of the Fisher
    matrix."
  ],
  "source_paragraph_ids": ["chunk_004_02", "
    chunk_024"],
  "verification_targets": [
    {
      "kind": "artifact",
      "description": "Fisher matrix
    calculation and EWC loss implementation"
    }
  ],
  "implementation_surfaces": ["
    model_or_method"],
  "code_obligations": [
    "Implement diagonal Fisher Information
    Matrix from the pre-trained policy on its
     training distribution.",
    "Implement the EWC loss term (lambda/2) *
     sum_i F_i (theta_i - theta_*_i)^2.",
    "Use lambda=10000 as the default
    coefficient."
  ],
  "status": "active"
}
```

What each field stores.

- **unit_id** is a stable identifier; repair tickets and provenance records both index by this value, so the same unit remains addressable across stages.
- **type** categorizes the unit (here: method) and drives how Plan groups it with related units into a work package.
- **statement** is the normalized one-duty implementation requirement, derived from the paper rather than copied verbatim.
- **paper_evidence** keeps the originating paper quotes, so review can cross-check the statement against the source.
- **source_paragraph_ids** are anchors into the segmented paper (chunk IDs), used to locate the unit in the addendum or main body and to guide review and repair.
- **verification_targets** declares what reviewers should check; each target has a kind (artifact, surface, or metric) and a description.
- **implementation_surfaces** declares where the obligation should appear in code; file planning must project the unit onto at least one file whose declared surfaces cover this set.
- **code_obligations** expand the statement into concrete sub-duties that generation and review can verify pointwise.
- **status** is the lifecycle flag; review writes verdicts (pass, missing, wrong-surface, inconsistent-with-source) back into this field.

How the preservation invariant fires on this unit. Plan cannot proceed unless the EWC unit is bound to at least one owner work package; in this run it is owned by a method package together with the related PPO-fine-tuning unit. File planning cannot proceed unless the unit is projected into at least one file contract whose declared surface includes `model_or_method`; in this run the unit lands in a single dedicated method file. Generate cannot finalize that file unless the file emits a provenance record citing `unit_004`, so the unit id surfaces in the per-file provenance map alongside its source paragraphs. Repair cannot mark the unit `pass` until requirement review verifies all three code obligations against the file contract; otherwise it issues a repair ticket carrying the unit id, the matched evidence ids attached to the work package, and the protected units that already passed.

The unit is therefore not a free-form paragraph or a planning-time prompt fragment. It is an addressable record with a fixed schema that planning, file projection, generation, validation, and repair are all conditioned on, which is what allows the same paper requirement to remain visible across

the four-stage pipeline rather than being silently dropped.

A.2 A Reference-Evidence Item

The evidence item below illustrates how a selected reference snippet is scoped to the method work package that owns the EWC unit `unit_004`.

Schema-aligned `evidence_bundles.json` view tied to unit_004


```
{
  "work_package_id": "wp_methods",
  "unit_id": "unit_004",
  "ref_id": "ref_001",
  "file_path": "model.py",
  "snippet_preview": "model.py:13:     def
    __init__(self, in_features, out_features,
     sigma0=0.5):\nmodel.py:14:         super
    ().__init__()\nmodel.py:15:         self.
    in_features = in_features\nmodel.py:16:
           self.out_features = out_features\
    nmodel.py:17:         self.weight = nn.
    Parameter(torch.Tensor(out_features,
    in_features))\nmodel.py:18:         self.
    bias = nn.Parameter(torch.Tensor(
    out_features))\nmodel.py:19:
    ...",
  "why_relevant": "function `__init__` in `
    model.py` aligns with units
    paper_method_core, unit_004 via parameter
     construction patterns reusable for the
    EWC reference network.",
  "confidence": 1.0,
  "matched_keywords": ["nn.Parameter", "
    in_features", "out_features"]
}
```


What each field stores.

- **work_package_id** names the package this evidence is attached to; in §3.2 the same identifier serves as the binding point between requirements and evidence.
- **unit_id** names the supported requirement; here it is the EWC unit `unit_004` from Appendix A.1, which is what makes the evidence consumable by the same package contract.
- **ref_id**, **file_path**, and **snippet_preview** together pin down (repo, path, region): the source repository, the file inside it, and the matched line range, recorded so review and repair can locate the snippet again.
- **why_relevant** is the structured match basis: it explains, in one sentence, which symbol or interface aligns the snippet with the supported unit, rather than leaving the LLM judgment as a free-form note.
- **confidence** and **matched_keywords** record the auxiliary signals used by the retain rule in §3.2, namely a symbol-level match strengthened by an LLM judgment.
- The role field, omitted from this stripped record but tracked elsewhere in the run state, marks this snippet as reusable structural evidence for parameter construction.

How the evidence enters generation. The item is bound to `wp_methods` during Plan, so when Generate writes the file contract that owns `unit_004`, the prompt for that file sees this snippet but no unrelated reference code from sibling work packages. If requirement review later returns a non-pass verdict on `unit_004`, the repair ticket carries the same `ref_id` and `file_path`, allowing repair to re-anchor the edit on this specific reference snippet rather than searching the reference repository again.

B External Baselines and Cost Notes

Table 4 records aggregate external rows that are useful context but not part of the primary same-backbone comparison. Rows include official PaperBench results, recent paper-to-code systems, and separately collected external reference rows. They differ in scaffold, model, runtime budget, and reporting protocol, and provide contextual reported scores rather than a primary controlled comparison.

C Gemini Ablation Case Studies

We keep qualitative ablation cases in the appendix so that the main results focus on aggregate behavior. The examples below illustrate the channel roles behind the aggregate ablation pattern in Section 5.3. Scores in this section provide mechanism-oriented evidence for the aggregate analysis.

C.1 Tracing One Unit in Test-Time Model Adaptation

We trace Test-Time Model Adaptation because its FOA/CMA unit is a strong trace for the implementation-requirement channel and a useful contrast for how reference evidence affects repository organization. In our consolidated 2026-05-24 PaperBench judging snapshot, the full Gemini run scores 54.9, while removing implementation requirements scores 47.1 and removing reference evidence scores 23.8. The trace is taken from the corresponding Gemini full-run artifacts for `test-time-model-adaptation`.

Method	Backbone / setting	Reported score
BasicAgent (Starace et al., 2025)[a]	Gemini-2.0-Flash	5.0
	o3-mini	5.1
	GPT-4o	7.7
	o1	19.5
	Claude-3.5-Sonnet	35.4
IterativeAgent (Starace et al., 2025)[a]	o3-mini	16.4
	Claude-3.5-Sonnet	27.5
	o1	43.3
PaperCoder (Seo et al., 2026)	o3-mini-high	45.1
AutoP2C (Lin et al., 2025)	source-reported setting	49.2
AutoReproduce (Zhao et al., 2025)	source-reported setting	49.6
Deep-Reproducer (Chen et al., 2025)	GPT-5	63.2
RePro (Zhou et al., 2025)[b]	o3-mini	62.6
DeepCode (Li et al., 2025b)[c]	Claude-Sonnet-4.5	73.5
BasicAgent (Chen et al., 2026)[d]	Gemini-3-Flash	19.3
IterAgent (Chen et al., 2026)[d]	Gemini-3-Flash	20.6
AiScientist (Chen et al., 2026)[d]	Gemini-3-Flash	30.5
REPROAGENT (ours)	Claude-Sonnet-4.5	73.7
	Gemini-3-Flash	39.7

Table 4: Supplemental aggregate PaperBench baselines. Reported scores are percentages. Rows differ in scaffold, backbone, runtime budget, and reporting protocol, and provide context for the full-suite comparison rather than a primary controlled comparison.

[a] Official 20-paper PaperBench Code-Dev scaffold rows; standard errors are reported in the cited source, and this table reports means only for compact contextual comparison. The o1 IterativeAgent row uses two seeds. [b] RePro reports PRroot@5, so we list it as external PRroot context separate from the Code-Dev rows. [c] Collected 20-paper mean over three runs per paper; the reported mean standard error across papers is 2.8 and the mean cost is $8.88. [d] Collected AiScientist engineering aggregates; mean costs are $6.25, $27.44, and $15.67 for BasicAgent, IterAgent, and AiScientist.

Extraction. `unit_002` is extracted as "Implement Forward-Only Prompt Adaptation with CMA". Its obligations require prompt insertion into the ViT input sequence, a CMA-ES optimizer, a population size $K \in [2, 28]$, and a forward-only prompt update loop. `unit_003` is extracted as the activation-distribution discrepancy fitness function, requiring mean/variance statistics for test CLS tokens and a fitness value passed to CMA.

Work package and contract. Planning binds both units to `wp_foa_core`, whose goal is to implement the FOA method and CMA optimizer. The work package produces `methods/foa.py`, `optimizers/cma_es.py`, and `results/source_stats.json`. Its global contract preserves the FOA loop, CMA-ES interface, source-statistics collector, fitness function, method registry, and parameter inventory for K, prompt length, alignment weights, batch size 64, and momentum 0.9. This is the implementation-requirement channel in concrete form: the paper obligations become file-owned duties rather than prose context.

Evidence binding. The evidence bundle for `wp_foa_core` records paper chunks for FOA, source-statistics calculation, and the evaluation protocols. In this case, the package is marked `self_contained` and has no repository `evidence_links`, because the decisive FOA/CMA obligations are already present in the paper and addendum evidence. This clarifies the ablation: the no-reference setting still recovers much of the requirement surface, while the full setting adds stronger repository-level organization, protocol wiring, and comparison surfaces.

Generated files and provenance. Generation materializes the contract in three files. `methods/foa.py` contains the hyperparameter sweeps, method/baseline registries, a CMA-ES implementation, source-statistics loading, and FOA routines. `optimizers/cma_es.py` exposes a standalone CMA-ES `ask/tell` optimizer and method selectors. `src/foa/utils/cma_optimizer.py` contains a forward-only `FOA.adapt` route under `torch.no_grad()`. The generation file-provenance record assigns all three files to `wp_foa_core`, lists `unit_002` and `unit_003` as owned units, and attaches validation checks for

implementation requirements, formula/algorithm contracts, evidence-matrix coverage, and route closure. The key point is that the requirement IDs, work-package owner, generated files, and validation checks remain connected throughout the trajectory; the ablation removes one of these channels before the file-level contract is formed.

C.2 Tracing Reference Evidence in Bridging Data Gaps

Bridging Data Gaps is a complementary case because the same snapshot shows an asymmetric ablation: the full Gemini run scores 50.6, the run without implementation requirements scores 38.4, and the run without reference evidence scores 13.9. The case therefore stresses the second channel in Figure 2: the paper requirement channel keeps the high-level adaptation and artifact-writing obligations visible, while reference evidence helps convert those obligations into a concrete repository protocol.

Requirement channel. The requirement units preserve the paper-derived duties that must survive into code: train the adaptation components, expose a runnable experiment path, save model/adaptor checkpoints, and write result artifacts that make the experiment inspectable. These duties are projected into file contracts that expect a training route and artifact writer rather than a single monolithic script. This explains the asymmetric score pattern: the remaining reference and generic task context can still support some repository structure, but the paper-specific obligations are less explicit without the requirement channel.

Reference-evidence channel. The reference channel supplies the implementation texture that the paper does not spell out line by line. In the full run, the generated repository writes `checkpoints/adaptor.pth`, `checkpoints/trained_model.pth`, `results/ant_training_trace.json`, and `results/config_resolved.json`, matching the expected pattern of checkpointed training and reproducible configuration logging. When reference evidence is removed, the paper obligations remain visible, but the run has less support for file layout, data-handling conventions, experiment wiring, and artifact naming. This case provides a concrete trace of why the reference-evidence channel can matter even when the target paper already names the high-level task.

C.3 Additional Ablation Patterns

The ablation pattern is not driven by a single paper. The excerpts below are illustrative traces from the full Gemini repositories, not additional scoring criteria. They show how the contract is materialized as file-owned implementation choices rather than remaining only in planning prose.

Sample-specific Masks: method and ablation factory excerpt

```
BASELINES = ["PAD", "NARROW", "MEDIUM", "FULL
    "]
ABLATIONS = ["ONLY_delta", "ONLY_f_mask",
             "SINGLE_CHANNEL_f_mask_s", "OURS
    "]

def get_method_config(method_name: str) ->
    MethodConfig:
    if method_name == "ONLY_delta":
        return MethodConfig(name="ONLY_delta",
     use_delta=True,
                            use_f_mask=False)
    elif method_name == "ONLY_f_mask":
        return MethodConfig(name="ONLY_f_mask
    ", use_delta=False,
                            use_f_mask=True)
    elif method_name in ["ours", "OURS", "SMM
    "]:
        return MethodConfig(name="OURS",
    use_delta=True,
                            use_f_mask=True,
    mask_channels=3)
    elif method_name in ["PAD", "NARROW", "
    MEDIUM", "FULL"]:
        return MethodConfig(name=method_name,
     use_delta=True,
                            use_f_mask=False)
```

Bridging Data Gaps: training artifact excerpt

```
for step in range(num_iterations):
    loss_val = train_ant_step(batch, config,
    model, adaptor,
                              classifier,
    optimizer)
    trace.append({"step": step, "loss":
    loss_val})

os.makedirs("checkpoints", exist_ok=True)
os.makedirs("results", exist_ok=True)
torch.save(adaptor.state_dict(), "checkpoints/
    adaptor.pth")
torch.save(model.state_dict(), "checkpoints/
    trained_model.pth")

with open("results/ant_training_trace.json",
    "w") as f:
    json.dump(trace, f, indent=2)
with open("results/config_resolved.json", "w")
     as f:
    json.dump(config, f, indent=2)
```

Paper	Full	w/o impl.	w/o ref.	Case role
Sample-specific Masks	53.8	21.3	28.0	Clear two-channel benefit: the full run preserves SMM-specific optimization, mask generation, interpolation, baselines, and ablations.
BAM	48.5	30.8	20.7	Large positive case: both removed-channel settings score substantially below the full contract, with the largest gap for reference evidence.
Bridging Data Gaps	50.6	38.4	13.9	Reference-heavy case: paper obligations remain useful, but removing reference evidence sharply reduces concrete protocol and repository-structure recovery.
PINN	61.3	53.3	52.2	Standardized-prior case: all variants remain strong because PINN objectives and PDE-loss structure are common, but the full contract is still best.
What Will My Model Forget	16.8	10.7	8.4	Scale and coverage stress case: the task has many long-tail details, and the full contract still leads both ablations.
LCA-on-the-Line	41.8	16.1	33.8	Boundary case: no-reference keeps a visible checklist for ObjectNet/LCA details, while the full run remains the strongest setting.

Table 5: Representative Gemini ablation cases. The case role column summarizes inspection observations alongside the score columns. "w/o impl." removes the implementation-requirement channel; "w/o ref." removes the reference-evidence channel.

Large two-channel gains. Sample-specific Masks and BAM are representative qualitative traces covering large two-channel gains. In Sample-specific Masks, the full run keeps the paper's required surfaces connected: the mask generator, interpolation from a low-resolution mask to the input image size, joint optimization of the shared pattern and mask-generator parameters, PAD/NARROW/MEDIUM/FULL baselines, and ablations over shared pattern and mask channels. Removing implementation requirements disrupts this preservation path, so the repository retains runnable scaffolding but covers fewer paper-specific obligations. Removing reference evidence keeps the obligations but reduces the file and experiment organization needed to expose baselines and ablation routes in a concrete implementation. BAM shows a similar pattern at a coarser level: the full score is 48.5, while no-implementation and no-reference fall to 30.8 and 20.7, respectively. This indicates that the reference-evidence channel recovers practical implementation and evaluation structure, while the implementation-requirement channel keeps paper-specific modeling and protocol obligations explicit.

Reference-sensitive cases. Bridging Data Gaps is useful because the two ablations degrade asymmetrically. The full run reaches 50.6, the no-implementation run remains at 38.4, and the no-reference run drops to 13.9. The artifact trace shows that the implementation-requirement channel preserves high-level tasks and expected outputs, while reference evidence supplies much of the concrete protocol surface: file layout, scripts, data handling conventions, and experiment wiring. This is the kind of paper where related repositories supply implementation texture that is not explicit enough in the paper text alone.

Standardized tasks. PINN is a high-scoring but smaller-margin case. The full run obtains 61.3, while the two ablations remain at 53.3 and 52.2. This pattern is consistent with the channel hypothesis. PINN-style repositories have strong public priors: collocation points, PDE residual losses, boundary losses, optimizer loops, and diagnostic plots are common implementation patterns. As a result, Gemini can recover a reasonable repository even with one channel removed. The full contract still gives the best score because it binds these common priors to the paper-specific objective, PDE setup, loss-landscape diagnostics, and expected artifacts.

Scale and coverage stress. What Will My Model Forget is a scale-sensitive case for the Gemini backbone. The full setting scores 16.8 and both ablations are lower. The target has broad long-tail coverage requirements and many benchmark-specific details, so this case separates channel utility from backbone capacity while preserving the same ordering pattern.

C.4 Why Ablation Gaps Vary

The per-paper ablation gaps are heterogeneous, and this variation is itself informative. For this

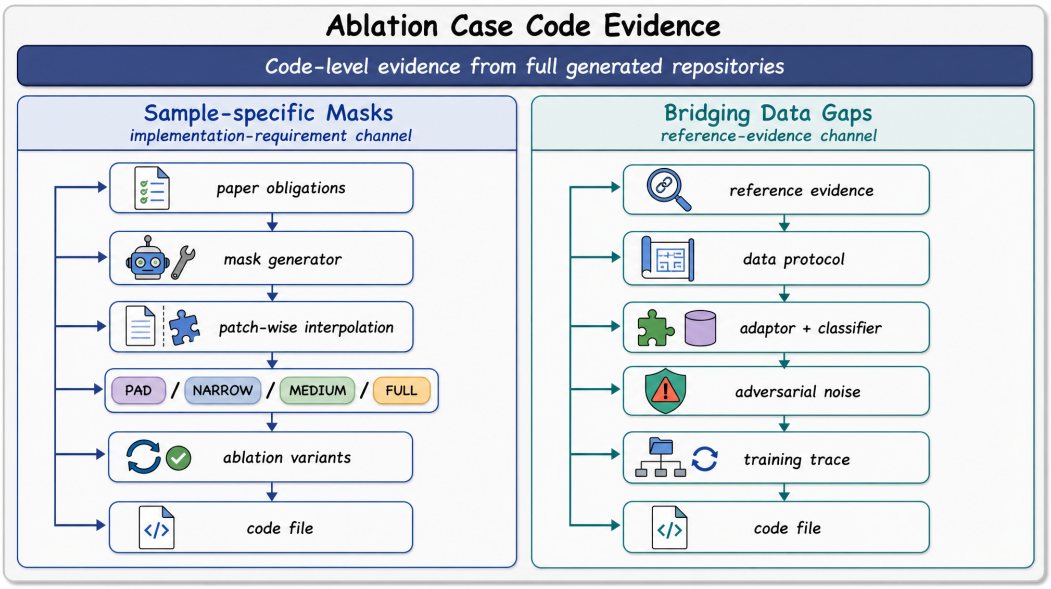

Figure 5: Code-level evidence for two representative ablation cases. The full generated repositories expose different concrete surfaces: Sample-specific Masks relies on implementation-requirement preservation for method-specific masks, interpolation, baselines, and ablations, while Bridging Data Gaps relies more heavily on reference evidence for protocol and training-artifact wiring.

discussion, we define the reference-evidence gain as the difference between the full Gemini score and the score without reference evidence, and the implementation-requirement gain as the difference between the full Gemini score and the score without implementation requirements. The largest reference-evidence gains occur on Bridging Data Gaps (+36.7), Adaptive Pruning (+35.7), Test-Time Adaptation (+31.1), FTRL (+28.4), All-in-One (+27.9), and BAM (+27.8). The largest implementation-requirement gains occur on Mechanistic Understanding (+43.5), Sample-Specific Masks (+32.5), Adaptive Pruning (+27.6), FTRL (+26.7), LCA-on-the-Line (+25.7), and All-in-One (+23.6). These rankings suggest that the two channels help for related but distinguishable reasons.

Reference transferability. Reference evidence helps most when external artifacts provide transferable implementation structure: dataset protocols, baseline organization, metric wiring, experiment scripts, configuration conventions, or repository layout. This is broader than simply counting cloned repositories. For example, BBOX has multiple reference or dataset repositories in the preparation artifacts, yet its reference-evidence gain is small (+1.0), showing that repository count alone is not the key factor. Conversely, Bridging Data Gaps shows a large reference-evidence gain even with a compact prepared reference-repository list, because the useful signal is whether reference evidence can be fused into concrete protocol and repository-structure decisions. In reference-sensitive cases, removing this channel leaves the paper obligations visible but makes the generated repository less able to recover executable data handling, experiment wiring, and comparison surfaces.

Requirement hardness. The implementation-requirement channel helps most when a paper's contribution is compact but implementation-critical: a small set of named mechanisms, constraints, losses, ablations, metrics, or artifacts determines whether the repository is faithful. Sample-Specific Masks is a representative case: the decisive obligations include mask generation, interpolation from a low-resolution mask to image size, joint optimization of shared patterns and mask-generator parameters, PAD/NARROW/MEDIUM/-FULL baselines, and ablations over shared-pattern and mask channels. Mechanistic Understanding, Adaptive Pruning, FTRL, and LCA-on-the-Line exhibit the same pattern at different scales: without explicit implementation requirements, the model can still produce plausible scaffolding, but it tends to omit or dilute the paper-specific mechanisms and protocol obligations that decide the score.

Small-gap cases. Small ablation gaps do not necessarily imply that the removed channels are irrele-

vant. They usually fall into one of three categories. First, some papers have strong public implementation priors; PINN is the clearest example, where collocation points, PDE residual losses, boundary losses, optimizer loops, and diagnostic plots are common enough that both ablations remain strong. Second, some papers have compressed gaps because all variants face the same central implementation bottleneck; BBOX is a case where the three Gemini variants remain close. Third, some tasks are scale or coverage stress cases, such as What Will My Model Forget, where the full contract still leads but the target contains many long-tail requirements.

Overall, the ablation gaps are best interpreted as indicators of where each channel provides leverage. Reference evidence is most valuable when external artifacts are transferable into implementation structure, while implementation requirements are most valuable when fidelity depends on non-obvious paper-specific obligations. Small gaps are therefore ambiguous: they may reflect strong public priors, limited reference transferability, or shared bottlenecks across variants, rather than evidence that the channels are unnecessary.

D Prompting and Structured Outputs

The production prompts use a shared system-user pattern. The current pipeline uses schema-constrained JSON for planning stages, deterministic evidence grounding for package-level evidence bundles, and repository-edit prompts for generation and repair. We therefore report compact schema excerpts rather than the full prompt text. The released repository contains the full prompt templates and structured-output schemas.

Prepare: ExtractedUnit schema excerpt

```
{
  "unit_id": "unit_002",
  "type": "method | protocol | claim |
    artifact | task",
  "statement": "paper-specific implementation
     obligation",
  "hypothesis": "what claim or mechanism this
     unit tests",
  "decision_value": "what implementation
    decision depends on it",
  "paper_evidence": ["source paper/addendum
    excerpt"],
  "source_paragraph_ids": ["chunk_006_01"],
  "reference_query_terms": ["optional
    explicit method or repository cue"],
  "implementation_surfaces": ["
    model_or_method", "metric_formula"],
  "code_obligations": ["concrete executable
    obligation"],
  "runtime_interfaces": ["CLI/function/config
     surface"],
  "expected_artifacts": ["results/metrics.
    json"],
  "suggested_module_kinds": ["training_loop",
     "artifact_writer"],
  "implementation_notes": ["inventory or
    scope notes"],
  "tags": ["method", "evaluation"]
}
```

Plan: contract and file-planning schema excerpt

```
{
  "work_package": {
    "work_package_id": "wp_foa_core",
    "owned_unit_ids": ["unit_002", "unit_003
     "],
    "inventories": {
      "obligation_matrix": ["Method -> file
     path"],
      "method_inventory": ["named method or
     baseline"],
      "parameter_inventory": ["paper-visible
     values"]
    },
    "scope_boundary": {
      "preserve": ["unit-level obligations
     that must survive"],
      "implementation_focus": ["active routes
      and artifacts"]
    },
    "method_obligations": ["what code must
     expose"]
  },
  "file_plan": {
    "target_file": "methods/foa.py",
    "work_package_id": "wp_foa_core",
    "owned_units": ["unit_002", "unit_003"],
    "reference_ids": ["ref or paper chunk ids
     "],
    "implementation_surfaces": ["
     model_or_method"],
    "defines_symbols": ["FOA", "CMAES"],
    "calls_symbols": ["fitness_function"],
    "writes_artifacts": ["results/
     adaptation_trace.json"],
    "scope_boundary": {"preserve": ["file-
     owned obligations"]},
    "generation_prompt": "file-local
     implementation instruction",
    "review_points": ["semantic and route-
     closure checks"]
  }
}
```

Evidence grounding and repair schema excerpt

```
{
  "evidence_bundle": {
    "work_package_id": "wp_foa_core",
    "focus": "implementation duty",
    "owned_unit_ids": ["unit_002", "unit_003
     "],
    "evidence_links": [
      {"unit_id": "unit_002", "ref_id": "
     ref_001", "file_path": "repo/file.py",
       "snippet_preview": "...", "
     why_relevant": "...", "confidence": 0.0}
    ],
    "context_summary": ["paper or reference
     evidence summary"],
    "grounding_status": "grounded |
     self_contained | weak | ungrounded"
  },
  "repair_ticket": {
    "failure_type": "semantic | runtime |
     integration",
    "required_fix_targets": ["methods/foa.py
     "],
    "allowed_changes": ["targeted repair
     scope"],
    "forbidden_changes": ["requirements
     already preserved"],
    "next_fix_scope": ["files to patch"]
  }
}
```

E Additional Algorithmic Details

E.1 Algorithmic Summary

Algorithm 1 REPROAGENT pipeline

Require: Paper $\mathcal{P}$, request r, references $\mathcal{R}$, rounds T
Ensure: Repository $\hat{\mathcal{C}}$

$\mathcal{S} \leftarrow \text{ChunkPaper}(\mathcal{P})$
$\mathcal{Q} \leftarrow \text{SurveyRepos}(\mathcal{R})$
$\mathcal{U} \leftarrow \text{ExtractRequirements}(\mathcal{S})$
$\mathcal{U} \leftarrow \text{Deduplicate}(\mathcal{U})$
$\mathcal{W} \leftarrow \text{PlanWorkPackages}(\mathcal{U})$
$\text{CheckPreserved}(\mathcal{U}, \mathcal{W})$
$\mathcal{E} \leftarrow \text{CollectContentEvidence}(\mathcal{W}, \mathcal{Q})$
$\mathcal{Z} \leftarrow \text{CollectStructureEvidence}(\mathcal{Q})$
$\mathcal{G} \leftarrow \text{BuildContract}(\mathcal{U}, \mathcal{W}, \mathcal{E}, \mathcal{Z})$
$\mathcal{F} \leftarrow \text{PlanFiles}(\mathcal{G})$
for $f \in \text{TopologicalOrder}(\mathcal{F})$ **do**
  $\hat{f} \leftarrow \text{GenerateFile}(f, \mathcal{G}, \mathcal{E})$
end for
for $t = 1$ to T **do**
  $v_q \leftarrow \text{RequirementReview}(\hat{\mathcal{C}}, \mathcal{U})$
  $v_r \leftarrow \text{RuntimeValidation}(\hat{\mathcal{C}})$
  if $v_q.\text{passed} \wedge v_r.\text{passed}$ **then**
    break
  end if
  $\tau \leftarrow \text{RepairTicket}(v_q, v_r, \mathcal{E})$
  $\hat{\mathcal{C}} \leftarrow \text{ApplyRepair}(\hat{\mathcal{C}}, \tau)$
end for
return $\hat{\mathcal{C}}$

E.2 Repair Loop

Algorithm 2 Contract-preserving repair loop

Require: Repository $\hat{\mathcal{C}}$, requirements $\mathcal{U}$, contracts $\mathcal{G}$, evidence $\mathcal{E}$, budget T
Ensure: Patched repository $\hat{\mathcal{C}}$

for $t = 1$ to T **do**
  $s \leftarrow \text{ReviewRequirements}(\hat{\mathcal{C}}, \mathcal{U}, \mathcal{G})$
  $r \leftarrow \text{RunValidation}(\hat{\mathcal{C}})$
  if $s.\text{passed} \wedge r.\text{passed}$ **then**
    break
  end if
  $\tau \leftarrow \text{BuildRepairTicket}(s, r, \mathcal{G}, \mathcal{E})$
  $\tau_{\text{ban}} \leftarrow \text{PreservedReqs}(s)$
  $\hat{\mathcal{C}} \leftarrow \text{PatchAffectedFiles}(\hat{\mathcal{C}}, \tau)$
end for
return $\hat{\mathcal{C}}$

F Per-Paper Scores and Generated Repository Sizes

This section gives the complete per-paper scores and generated repository sizes behind the full and ablated runs reported in the main results.

G Token, Time, and Cost Diagnostics

Tables 7 and 8 report usage metadata for interpreting the reproduction process. Scoring-token counts come from PaperBench judge summaries; when a judge summary stores token usage at leaf level, we sum the recoverable per-leaf metadata from the same graded task tree. Pipeline-token counts, wall-clock time, and agent-call counts use exported run logger summaries when available. Runtime metadata is exported at different granularity across settings; the Claude pipeline-token cell is estimated from matched run logs and is reported only as process metadata, while hours and calls use the recoverable run metadata directly. For the Gemini family, the full run uses more pipeline tokens than either ablation, which is expected because the full setting carries both implementation requirements and reference evidence through planning and repair. Wall-clock time varies with target-paper difficulty and available runtime metadata.

H Full Pipeline Stage Sequence

REPROAGENT comprises 15 sub-stages organized into four phases. Table 9 lists every stage with its phase assignment and a brief description.

The Repair phase operates as an iterative loop with a fixed budget within each backbone setting. In each iteration, the system executes validation, diagnosis, and regeneration in sequence. The loop terminates when all validation checks pass or the budget is exhausted.

H.1 Process Trace and Readiness

Each run records intermediate artifacts for auditing each phase, not only the final repository and score. The stable audit records include prepared units, reference-repository records, work-package plans, contract assertions, file-planning artifacts, validation reports, stage status, quality status, and usage summaries. Table 10 summarizes coarse audit statistics from the 20 Gemini full runs.

These logged counts make the generation process more concrete. Prepare extracts implementation units from the paper itself rather than from the PaperBench rubric, then attempts to ground those units with related repositories when actionable evidence is available. Prepared reference repositories are concentrated in a subset of papers, reflecting the benchmark's mix of papers with different amounts of reusable prior-work structure. Plan then turns prepared units into work packages and contract assertions, which are contract-shaping objects rather than file-count targets. Generate materializes the file plans into repositories, and Repair audits requirement, trace, artifact, integration, and runtime checks.

These logged counts are not inputs to PaperBench judging. All reported Gemini full runs completed repository generation and were submitted to the PaperBench judge. We report them alongside PaperBench scores to make the pipeline trace auditable.

Paper	Claude		Gemini		w/o ref.		w/o req.	
	Score	Files	Score	Files	Score	Files	Score	Files
Adaptive Pruning	67.8	46	41.9	92	6.2	23	14.3	24
All in One	76.9	77	51.6	48	23.7	16	28.0	36
BAM	61.8	37	48.5	47	20.7	18	30.8	38
BBOX	86.4	280	15.6	70	14.6	15	14.2	75
Bridging Data Gaps	66.0	151	50.6	23	13.9	17	38.4	19
FRE	76.8	23	26.9	32	12.7	14	25.1	20
FTRL	63.9	28	40.1	71	11.7	23	13.4	51
LBCS	65.8	40	42.5	51	34.6	24	23.2	22
LCA on the Line	64.9	30	41.8	22	33.8	18	16.1	15
Mech. Understanding	75.7	201	48.1	26	25.9	14	4.6	20
PINN	81.5	163	61.3	35	52.2	14	53.3	18
RICE	75.1	34	31.0	81	19.9	16	27.8	25
Robust CLIP	63.0	51	29.5	19	15.7	14	25.6	20
Sample-specific Masks	82.0	80	53.8	50	28.0	17	21.3	33
SAPG	86.5	141	29.0	25	21.5	16	24.2	24
Sequential Neural Score Estimation	78.0	30	45.7	14	21.5	13	42.6	26
Stay on Topic with Classifier-Free Guidance	58.5	31	25.3	21	21.2	20	22.2	27
Stochastic Interpolants	80.8	73	39.1	25	21.5	19	29.9	24
Test-Time Model Adaptation	75.6	49	54.9	20	23.8	12	47.1	26
What Will My Model Forget	87.6	23	16.8	22	8.4	12	10.7	19
Mean	**73.7**	**79.4**	**39.7**	**39.7**	**21.6**	**16.8**	**25.6**	**28.1**

Table 6: Per-paper scores and generated repository sizes for our full and ablated runs. File counts come from the corresponding run metadata and are reported as process metadata rather than scoring inputs.

Setting	Papers	Score	Judge tok.	Pipe tok.	Hours	Calls
ReproAgent / Claude	20	73.7	12,868,729	17,833,795	9.3	35.4
ReproAgent / Gemini full	20	39.7	6,243,588	17,459,007	5.6	11.7
Gemini w/o reference evidence	20	21.6	6,582,787	4,034,007	1.5	13.8
Gemini w/o implementation requirements	20	25.6	4,668,761	4,429,321	1.3	12.6

Table 7: Usage metadata for our runs. Judge-token cells use root-level or summed leaf metadata depending on how the PaperBench summary stores usage; pipeline-token cells use exported run metadata or the estimate described above.

Paper	Claude judge	Gemini judge	Gemini pipe	No-ref judge	No-ref pipe	No-req judge	No-req pipe
Adaptive pruning	5,423,412	2,868,933	23,047,479	1,980,736	2,236,269	3,341,264	5,919,631
All-in-one	5,538,759	3,780,967	9,264,024	3,429,304	3,936,893	2,250,662	2,630,553
BAM	30,766,863	7,768,083	16,097,296	8,827,504	4,969,452	1,892,690	15,027,467
BBOX	8,567,577	774,669	58,268,936	4,903,571	4,131,573	4,373,037	4,509,759
Bridging data gaps	4,746,750	2,300,086	8,820,069	1,864,915	4,264,829	2,412,905	4,406,997
FRE	37,387,691	13,780,748	8,425,041	10,006,741	2,975,272	12,737,673	3,925,572
FTRL	11,530,535	12,388,835	23,625,213	3,976,499	5,255,585	291,099	4,136,571
LBCS	35,890,914	17,312,480	9,390,925	20,577,036	5,455,156	9,045,550	6,665,934
LCA on the line	25,328,353	613,700	8,697,850	18,609,830	4,585,214	4,880,558	3,644,005
Mech. understanding	2,176,841	1,589,700	7,923,157	1,340,680	3,134,806	1,383,850	4,118,063
PINN	7,360,909	5,691,891	13,620,939	7,053,399	4,120,951	6,536,328	4,348,687
RICE	16,389,279	1,550,040	74,534,103	6,929,485	3,739,967	5,148,614	3,261,044
Robust CLIP	5,588,405	2,147,322	16,285,677	3,131,589	3,445,426	2,689,292	2,283,182
Sample-specific masks	9,405,505	2,230,104	18,789,684	2,932,425	4,813,964	604,261	2,426,577
SAPG	4,553,396	3,507,987	8,202,509	3,339,036	3,793,693	2,585,659	4,160,601
Sequential NSE	3,293,860	2,661,812	6,486,552	2,075,601	4,250,141	2,723,479	5,711,514
Stay on topic	3,206,265	1,684,783	13,880,313	2,100,836	4,674,944	159,171	2,992,827
Stochastic interpolants	3,641,790	2,171,919	8,435,217	1,364,707	4,452,320	1,731,008	1,227,333
Test-time adaptation	7,102,606	3,326,170	7,495,412	3,511,711	3,539,717	336,977	5,935,940
What will my model forget	29,474,870	36,721,521	7,889,747	23,700,128	2,903,969	28,251,151	1,254,158

Table 8: Per-paper token metadata. "Judge" is PaperBench grading-token usage, with leaf metadata summed for summaries that store usage at leaf level; "pipe" is generation-token usage from exported run metadata when available.

Phase	Stage	Engineering role
Prepare	Input normalization	Normalize the target, task type, entities, and explicit constraints.
Prepare	Paper chunking	Segment paper text into section-level chunks with overflow handling.
Prepare	Requirement extraction	Extract implementation requirements that need concrete code, artifacts, interfaces, or validation targets.
Prepare	Reference acquisition	Collect reference repositories and survey file trees, README files, and likely reusable modules.
Plan	Work-package planning	Group prepared requirements into implementation-oriented packages such as data, method, training, evaluation, and artifact writing.
Plan	Package reference evidence	Attach package-local content and structure evidence; this supports package decisions but does not decide the final file tree.
Plan	Reference selection	Select actionable reference repositories and evidence items for the current package needs.
Plan	Pipeline plan	Specify package dependencies, runnable routes, and ordering constraints.
Plan	Global contract synthesis	Freeze requirement ownership, evidence bindings, result targets, artifact paths, interfaces, validation checks, and cross-package dependencies.
Plan	Architecture planning	Decide file tree, stack, entrypoints, config surfaces, dependency rules, and package-to-file layout.
Plan	Package-file planning	Refine architecture into per-file contracts with owned requirements, evidence, interfaces, artifacts, and validation hooks.
Generate	Canonical representation synthesis	Consolidate requirements, packages, contracts, architecture, file plans, evidence links, and validator expectations.
Generate	Local file generation	Generate source code file by file following file contracts and dependency ordering.
Repair	Repair validation	Validate through requirement review, contract checks, preflight checks, and sandboxed runtime execution.
Repair	Repair regeneration	Build a targeted repair plan and patch affected files while preserving requirements that already pass.

Table 9: Complete stage sequence of the REPROAGENT pipeline. Stages are executed in order; within-phase stages may share intermediate artifacts.

Stage	Audit statistic	Mean	Median	Range
Prepare	Extracted implementation units per paper	33.8	34.0	28–38
Prepare	Candidate reference repositories requested per paper	24.6	25.0	10–30
Prepare	Prepared reference repositories per paper	1.1	0.0	0–9
Prepare	Implementation units grounded by references per paper	6.2	0.0	0–29
Plan	Work packages per paper	19.6	21.5	3–38
Plan	Contract assertions per paper	6.9	7.0	5–10
Generate	Generated files per paper	35.5	26.5	14–80
Runtime metadata	Pipeline agent calls per paper	11.7	6.0	3–29

Table 10: Coarse audit statistics from the 20 Gemini full runs. These logged counts summarize the scale of preparation, planning, generation, and runtime interaction recorded during execution; they are not inputs to PaperBench judging.