

When the Specification Emerges: Benchmarking Faithfulness Loss in Long-Horizon Coding Agents

Lu Yan

Purdue University
yan390@purdue.edu

Xuan Chen

Purdue University
chen4124@purdue.edu

Xiangyu Zhang

Purdue University
xyzhang@purdue.edu

Abstract

Current coding-agent benchmarks usually provide the full task specification upfront. Research-code implementation may instead require an agent to preserve design commitments disclosed progressively across a long session. We introduce SLUMP, a controlled benchmark for testing implementation faithfulness when a mathematically structured machine-learning target is disclosed through synthetic, paper-grounded multi-turn interactions rather than in a single canonical implementation specification. The benchmark contains 20 recent ML papers, 371 atomic verifiable components, and approximately 60 coding requests per paper. An exposure audit verifies whether scored components are recoverable from the visible interaction. On Claude Code and Codex, the single-shot control achieves higher overall implementation fidelity on 16/20 and 14/20 papers, respectively. Structural integration degrades under progressive disclosure on both platforms, while component-level design faithfulness degrades substantially on Claude Code and remains approximately stable on Codex. These conclusions persist under stricter exposure filters and a targeted human correction of judge false-absence errors. As a mitigation case study, ProjectGuard maintains external semantic and structural project state and recovers 90% of the aggregate Claude Code MCF gap. Code and benchmark artifacts are available at [GitHub](#).

1 Introduction

AI coding agents are increasingly used to implement research ideas in practice. Recent benchmarks evaluate whether agents can translate papers or task descriptions into working code, including PaperBench (Starace et al., 2025), MLR-Bench (Chen et al., 2025), LMR-Bench (Yan et al., 2025), and RECODE-H (Miao et al., 2025). These benchmarks have substantially advanced evaluation of coding agents, but they largely share a common assumption: the target specification is available at

the start of the session, whether as a paper, a task description, or a test suite. They do not measure whether an agent can faithfully implement a system whose specification is distributed across a long interaction.

Real research workflows often violate this assumption. A researcher rarely begins with a complete implementation blueprint. Early turns frame the idea and compare alternatives, middle turns implement components under partial specification, and later turns refine and integrate those components into a unified training and evaluation pipeline. The intended method is therefore not written once in a canonical prompt. It is assembled across many turns, interleaved with exploratory alternatives, local revisions, and integration decisions. Faithful implementation in this setting requires specification tracking: the agent must retain durable design commitments introduced earlier in the interaction and preserve them as the codebase evolves (Zhu et al., 2026; Liu et al., 2026).

Scope. SLUMP is a controlled progressive-disclosure stress test for mathematically structured ML research code. Its scripts are synthetic, paper-grounded artifacts that support a matched single-shot control and component-level recoverability audit; they are not historical author traces or live human-agent sessions. Accordingly, our results establish the effect in the benchmark and platforms studied here, rather than its prevalence across software engineering, research domains, or real-world workflows.

We study faithfulness Loss Under eMergent sPecification (SLUMP): the reduction in final implementation faithfulness that occurs when the target design is disclosed progressively through interaction rather than provided upfront. SLUMP is an endpoint-based notion. We do not attempt to measure temporal drift inside a session. Instead, we ask whether the final repository produced under

emergent specification is less faithful than the final repository produced by the same platform under a single-shot specification control. We analyze this loss along two complementary but non-orthogonal diagnostic dimensions: component-level design faithfulness and global structural integration of earlier modules into the final system.

To measure SLUMP, we construct a benchmark from 20 randomly sampled ML papers (10 from ICML 2025 and 10 from NeurIPS 2025). For each paper, we extract a canonical specification from the original text, decompose it into atomic verifiable components, and derive an interaction script of approximately 60 coding requests that progressively discloses the target design without revealing the paper itself. Across the 20 papers, this yields 371 frozen components for evaluation. We score the final repository against a five-level component faithfulness rubric and complement scoring with an exposure audit that checks, for each component, whether it is recoverable from the agent-visible interaction. This audit is necessary because emergent-specification evaluation is only meaningful if the target components are actually inferable from the visible dialogue.

We instantiate the benchmark on two long-horizon coding platforms, Claude Code and Codex, under two conditions: emergent specification and a single-shot specification control. The single-shot control achieves higher overall implementation fidelity on 16 of 20 papers on Claude Code and 14 of 20 papers on Codex, providing direct evidence of SLUMP. The decomposition of this gap differs across platforms: structural integration degrades under emergent specification on both systems, while component-level design-faithfulness loss is substantial on Claude Code and small on Codex. These results show that progressive specification disclosure imposes a measurable cost on final implementation fidelity in the studied benchmark setting when the underlying platform and tooling are held fixed.

As a case study in targeted mitigation, we introduce ProjectGuard, an external project-state layer for specification tracking. ProjectGuard maintains a semantic view of committed project knowledge and a structural view of repository organization, then injects a compatibility-aware project brief before each coding turn or proactive restart. On Claude Code, where SLUMP is substantial, ProjectGuard recovers 90% of the aggregate Claude Code MCF gap between emergent speci-

fication and the single-shot specification control, increases the number of fully faithful components by 53%, and reduces severe failures by 32%.

This paper makes three contributions. First, it introduces a benchmark and evaluation methodology for specification tracking under emergent-specification coding, combining long interaction scripts, component-level faithfulness scoring, and an exposure-based fairness audit. Second, it provides empirical evidence that emergent specification reduces final implementation fidelity relative to a single-shot control on the studied platforms and benchmark. Third, it presents ProjectGuard as a targeted mitigation case study, showing that externally maintained project state can substantially reduce SLUMP without modifying the underlying model.

The benchmark is open-sourced on [Github](#).

2 Problem Statement

2.1 Emergent-Specification Coding

We study emergent-specification coding, a multi-turn coding setting in which the target system is not given in a single initial prompt but is progressively disclosed through user requests. Early turns may explore alternatives or defer details, middle turns implement components under partial specification, and later turns refine and integrate those components into a complete system. The implementation target is therefore distributed across the interaction rather than stated upfront.

This setting requires the agent to retain both local task state and project-level specification state. Local task state includes recent files, current errors, and the immediate edit under discussion. Project-level specification state includes durable design commitments such as equations, architectural constraints, interface contracts, and evaluation requirements. Faithful implementation requires tracking both across a long interaction.

We evaluate agents against the final committed design: the set of design facts that are established by the end of the interaction as part of the intended system. Exploratory alternatives, abandoned branches, and unresolved suggestions are not part of the target. The benchmark construction in Section 3 operationalizes this target using a canonical specification derived from the source paper and an interaction script that reveals it progressively without exposing the paper itself.

2.2 SLUMP: Faithfulness Loss Under Emergent Specification

Let p denote a task and a an agent platform. Let $R_{p,a}^{\text{em}}$ be the final repository produced when the agent receives the emergent multi-turn interaction, and let $R_{p,a}^{\text{ss}}$ be the final repository produced when the same agent receives the frozen canonical implementation specification in a single shot upfront. For any implementation-faithfulness metric F , we define **faithfulness Loss Under emergent Specification (SLUMP)** as

$$\text{SLUMP}_F(p, a) = F(R_{p,a}^{\text{ss}}) - F(R_{p,a}^{\text{em}}).$$

Positive values indicate that progressive disclosure reduces final implementation faithfulness relative to the single-shot specification control.

The single-shot control receives the same frozen canonical implementation specification from which the component checklist and progressive-disclosure script are derived; it does not receive the source paper or PDF. This matched control removes the need to assemble the target across turns while holding the platform and tooling fixed. Note it is a diagnostic comparison condition, not a universal upper bound on achievable performance. The contrast changes both information timing and the incremental development trajectory; it estimates the effect of the full benchmark protocol rather than isolating memory loss or any single internal mechanism.

This definition is deliberately endpoint-based. We do not attempt to measure temporal drift within a session. Instead, we measure whether the final implementation under emergent specification is less faithful than the final implementation obtained when the full design is available from the start.

We analyze SLUMP along two complementary but non-orthogonal diagnostic dimensions. **Component-level design faithfulness** asks whether each scored unit preserves the intended algorithmic behavior and its local architectural realization. **Structural integration** asks whether the final system globally reuses earlier modules rather than bypassing or reimplementing them along incompatible paths. These dimensions are measured with Mean Component Faithfulness (MCF) and Public-Symbol Reuse Ratio (PSRR), respectively; overlap is possible because the component rubric itself includes local architecture.

3 Benchmark

Our benchmark is designed to evaluate specification tracking under emergent-specification coding. It targets four properties. First, the target system should emerge through interaction rather than being given upfront in a single prompt. Second, sessions should be long enough to require the agent to preserve design commitments across a sustained implementation arc rather than only across local edits. Third, evaluation should be component-level and tied to a frozen target specification derived from the source paper. Fourth, the papers should be recent and high-value so that contamination risk is reduced, though not eliminated. The benchmark therefore aims to be neither trivially unfair nor trivially explicit: most scored components must be recoverable from the visible interaction, but many should only become clear after integrating information across turns.

We construct the benchmark from 20 recent ML papers: 10 randomly sampled from ICML 2025 and 10 from NeurIPS 2025. Each paper is converted into an emergent-specification coding task together with a frozen component checklist for evaluation.

3.1 Construction Pipeline

All benchmark assets are produced with a single frozen model snapshot (CLAUDE-SONNET-4-5-20250929) using fixed prompt templates and schema-constrained outputs. The model serves as a controlled annotation instrument rather than an oracle for scientific truth: the original paper remains the sole authority for all paper-grounded content. Exact prompts, snapshot identifiers, and output schemas are provided in the appendix.

Phase 1: specification grounding. For each paper, we first extract a canonical specification from the PDF. This artifact contains a detailed method description with explicit mathematical content, the datasets and metrics used in evaluation, a step-by-step implementation plan, and plausible local design alternatives for major components. The purpose of the canonical specification is not to reconstruct the authors' historical development process, but to define a single frozen implementation target against which final repositories can be scored.

A valid design alternative is local and task-preserving: it retains the paper-level task, domain, input/output role, and surrounding module interface while varying one component-level design axis.

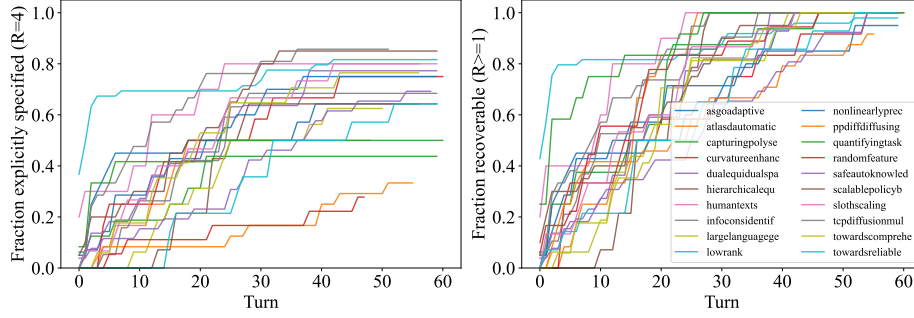


Figure 1: Cumulative exposure over turns. **Left:** fraction of components explicitly specified ($R=4$). **Right:** fraction recoverable ($R \geq 1$). Each line is one paper. The gap between panels reflects the ambiguous middle where components are inferable but not yet fully specified.

We then decompose the canonical specification into atomic verifiable components. Each component records its intended functionality, exact formula when applicable, expected inputs and outputs, and module-level connections to the rest of the system. We focus on algorithmic structure and inter-module wiring. Hyperparameters are excluded unless they are part of the mathematical identity of the method. This extraction is performed once per paper and frozen before any agent run. Across the 20 papers, the benchmark contains 371 verifiable components in total.

Phase 2: interaction scaffolding. Given the canonical specification, we synthesize a latent trajectory of 10 to 20 intermediate project versions. The first version begins from a deliberately under-specified idea, such as a loose objective family or a vague architectural template, and the final version matches the canonical method exactly. Each transition introduces one meaningful refinement to the design. We then convert these refinements into natural-language coding requests under a strict agent-blindness constraint: the agent sees neither the source paper nor the hidden canonical specification. Each paper yields approximately 60 user requests.

These trajectories are synthetic rather than historical. They are constrained to satisfy temporal coherence, meaningful refinement, and exact convergence to the canonical specification. This controlled paper-grounded stress test measures whether an agent can recover and preserve a distributed target design; it does not claim to reproduce live human-agent workflows.

3.2 Scoring Protocol

After each run, the final repository is scored component-wise against the frozen checklist using the five-level rubric in Table 1. Scoring is performed by a bounded LLM judge with at most eight repository-inspection tool calls per component. The judge is given the component specification and may inspect the repository, but it does not have access to hidden benchmark artifacts beyond the frozen component description. The judge is CLAUDE-SONNET-4-6; Appendix C gives invocation details.

The distinction between *equivalent* and *faithful* is important. A component receives score 3 if the implementation differs in form but preserves the same functional behavior and system role. It receives score 4 only when the implementation closely matches both the intended formulation and its architectural realization. This distinction matters because SLUMP is not limited to outright omission or failure; it can also appear as a structurally weaker or differently realized implementation that still partially works.

3.3 Benchmark Fairness Validation

An emergent-specification benchmark raises an immediate validity question: are the scored components actually recoverable from the interaction the agent observed? If not, low scores would reflect benchmark unfairness rather than a failure of specification tracking. We address this with an exposure audit conducted independently of any agent run.

The exposure audit establishes recoverability and benchmark fairness, not ecological validity: synthetic scripts may still differ from live workflows in pacing, ambiguity, noise, and revision style.

Score	Label	Definition
0	Absent	No code attempts this component.
1	Wrong	Code attempts the component but implements logic inconsistent with the specification.
2	Simplified	Code implements a degraded or approximate version of the intended component.
3	Equivalent	Code differs in form but preserves the same functional behavior and system role.
4	Faithful	Code closely matches the intended formulation and architectural realization.

Table 1: Component-level faithfulness rubric used to score final repositories against the frozen benchmark specification.

One model-based annotator (CLAUDE-SONNET-4-5-20250929) assigns each component’s recoverability from the complete frozen request sequence before agent runs. It sees no agent-produced repository, response, or score, and no human adjudication is used. The level $R(c) \in \{0, 1, 2, 3, 4\}$ denotes non-recoverable, weak, ambiguous, clear, and explicit exposure, respectively; Appendix C gives the full protocol.

We summarize this audit with two corpus-level statistics. The Recoverable Component Rate is

$$\text{RCR} = \frac{1}{|C|} \sum_c \mathbf{1}[R(c) \geq 1],$$

the fraction of scored components that are at least weakly inferable from the visible interaction. The Explicitly Specified Rate is

$$\text{ESR} = \frac{1}{|C|} \sum_c \mathbf{1}[R(c) = 4],$$

the fraction stated without ambiguity. Across the 20 papers, the macro-averaged RCR is 0.994, while the macro-averaged ESR is 0.666. In total, 368 of 371 components have $R(c) \geq 1$. Thus, nearly all scored components are available from the visible dialogue, but many must be assembled across turns rather than copied from a single explicit instruction.

Figure 1 plots the cumulative fraction of components that are explicitly specified ($R=4$, left) and recoverable ($R \geq 1$, right) as a function of turn number, for each paper. Two features are visible. First, the curves are spread across papers: some methods are largely specified by turn 20, while others continue introducing new components past turn 40.

Second, the right panel converges faster than the left, indicating that many components enter an ambiguous window (recoverable but not yet explicit) before their final form is committed. This temporal structure is by design and ensures that the benchmark stresses long-horizon retention rather than testing only the agent’s response to the final few turns.

3.4 Benchmark and Judge Calibration

Benchmark-asset audit. We audit all 20 papers, treating each source PDF as the sole authority and never inspecting agent outputs. For method and mathematical content, 335/371 evaluated components pass without qualification and 364/371 pass or need only a minor notation, shape, or scope correction. For component-decomposition support, the corresponding counts are 356/371 and 366/371; for evaluation fields, 65/71 and 68/71. Of 147 design alternatives, 140 satisfy the local, task-preserving constraint. Appendix C gives the criteria and full results; Section 4.3 reports how filtering by this audit affects the evaluation results.

Judge calibration. Because both benchmark construction and scoring involve LLM-based annotation, we calibrate the automated judge against human labels (Zhuge et al., 2025) on 120 component-run pairs stratified across papers, platforms, and score levels. Two independent annotators, each blind to the judge and to each other, inspect the repository and assign scores using the same five-level rubric. Table 2 reports agreement between each annotator and the automated judge, as well as inter-annotator agreement. Agreement is high: weighted Cohen’s κ is 0.882–0.927 between annotators and judge, inter-annotator agreement is 0.930, and 96.7%–99.2% of pairwise scores fall within ± 1 . The dominant residual error is conservative false absence: in four cases per annotator, the judge assigns score 0 despite relevant code existing in the repository, with three such cases shared across both annotators. This suggests that remaining disagreement is driven more by bounded repository search than by rubric ambiguity.

Full details about the benchmark can be found in Appendix A.

4 Experiments

4.1 Setup and Metrics

We evaluate Claude Code and OpenAI Codex with one run per paper and paired condition ($20 \times 2 \times$

Metric	A1-Judge	A2-Judge	A1-A2
Weighted κ (quadratic)	0.927	0.882	0.930
Exact-match accuracy	91.7%	86.7%	88.3%
Spearman ρ	0.938	0.899	0.932
Within ± 1	97.5%	96.7%	99.2%
Boundary disagr. (2-3)	1.8%	4.9%	5.2%
Boundary disagr. (3-4)	3.3%	6.2%	7.9%

Table 2: Pairwise agreement among the automated component judge and two independent annotators (each blind to the other scores) on 120 stratified component-run pairs.

2 = 80 baseline runs). The commercial backends were not snapshot-pinned; exact backend IDs, CLI versions, and timestamps were not retained. Paired conditions used identical scaffolds, permissions, and native context settings without human edits. Papers are the unit of paired Wilcoxon inference. Appendix C gives protocol and cost details.

We report three metrics. Mean Component Faithfulness (MCF) measures component-level design faithfulness, including algorithmic behavior and local architectural realization under Table 1. Let $s_i \in \{0, 1, 2, 3, 4\}$ be the rubric score for component i and let N be the number of components in the paper. Then

$$\text{MCF} = \frac{1}{N} \sum_{i=1}^N s_i.$$

MCF ranges from 0 to 4.

Public-Symbol Reuse Ratio (PSRR), introduced here, measures structural integration:

$$\text{PSRR} = \frac{|E_{\text{used}}|}{|E_{\text{total}}|},$$

where E_{total} is the set of public symbols exported by standalone modules and E_{used} is the subset actually reused by downstream integration code. PSRR ranges from 0 to 1.

Operationally, PSRR uses static imports among top-level public Python symbols. Package re-exports, dynamic imports, and test-only usage do not count; no eligible exports yields PSRR 0. Table 3 macro-averages paper-level values. Because the denominator depends on agent-created modules, PSRR can reflect modularity as well as required integration; Appendix C specifies all cases.

To summarize both dimensions, we define

$$\text{IF}_\lambda = \lambda \frac{\text{MCF}}{4} + (1 - \lambda) \text{PSRR}.$$

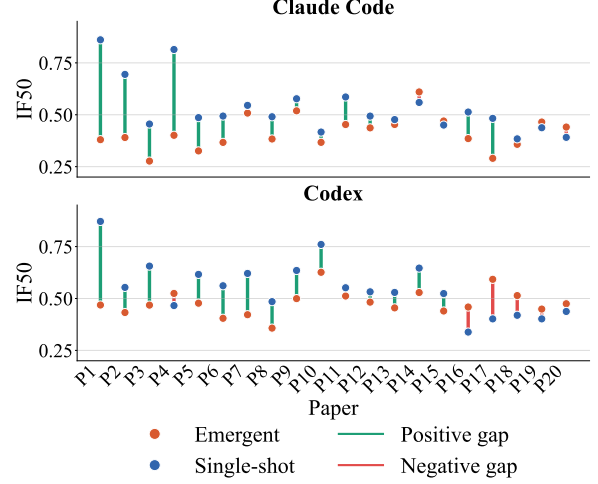


Figure 2: Per-paper IF50 under emergent specification and the single-shot specification control.

The equal-weight summary IF_{50} denotes $\lambda = 0.5$ and ranges from 0 to 1.

IF50 is a transparent summary of normalized component-level design faithfulness ($\text{MCF}/4$) and structural integration, not a uniquely optimal utility function. PSRR is a coarse structural diagnostic and does not prove design correctness; conversely, MCF does not fully capture whether earlier modules remain connected to the final pipeline. The metrics are complementary rather than orthogonal. We therefore interpret MCF and PSRR separately and use IF50 only as a compact endpoint summary.

4.2 Results

Figure 2 shows per-paper IF50 under emergent specification and the single-shot specification control. On Claude Code, the single-shot control outperforms emergent specification on 16 of 20 papers, with a mean IF50 gap of +0.116 and Wilcoxon signed-rank $p = 0.0003$. On Codex, the same pattern holds on 14 of 20 papers, with a mean gap of +0.071 and $p = 0.012$. This is the primary evidence for SLUMP: when the same platform receives the canonical implementation specification upfront rather than progressively through interaction, final implementation fidelity is higher.

The decomposition of this gap differs across platforms. On Claude Code, emergent specification lowers both component-level design faithfulness and structural integration: MCF drops from 3.031 under the single-shot control to 2.718 under emergent specification, while PSRR drops from 0.303 to 0.149. On Codex, the MCF gap is negligible

Platform / Condition	MCF	PSRR	IF50
Claude Code / Emergent	2.718	0.149	0.414
Claude Code / Single-shot	3.031	0.303	0.530
Codex / Emergent	3.242	0.148	0.479
Codex / Single-shot	3.245	0.289	0.550

Table 3: Aggregate faithfulness under emergent specification and the single-shot specification control.

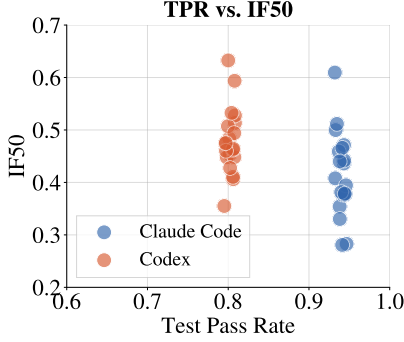


Figure 3: Agent-authored test pass rate is a weak proxy for IF50 within emergent runs.

(3.245 versus 3.242), but the PSRR gap remains substantial (0.289 versus 0.148). Thus, SLUMP is robust at the overall IF50 level and robust on structural integration across both platforms, while the MCF loss is concentrated on Claude Code.

This platform asymmetry is important. It shows that SLUMP is not tied to a single narrow failure pattern. Progressive specification disclosure can reduce final implementation fidelity through different mixtures of component-level design loss and structural non-reuse on different systems. For this reason, we treat MCF and PSRR as diagnostic views of SLUMP, while using IF50 as the primary summary measure.

Agent-authored test pass rate (TPR) is a weak proxy for IF50 within emergent runs. TPR is the fraction of collected agent-authored tests that pass; failures, collection errors, and timeouts are non-passing, and no collectable tests yields TPR 0. Figure 3 is a within-condition diagnostic: Spearman $\rho = -0.451$ ($p = 0.046$) for Claude Code and 0.274 ($p = 0.243$) for Codex, so it does not test the condition-level SLUMP gap.

Qualitative example. One representative example comes from the DualEquiNet task (Xu et al., 2025). By user request 40 of 60, the visible interaction clearly commits to a three-stream residual update: the scalar state receives the sum of the EU and SH deltas, coordinates receive an

EU coordinate delta, and SH features receive an SH delta. In the emergent run, the agent initially implements this design as a dedicated module (`equivariant_coordinate_update.py`), but the final consolidated model later bypasses that module, drops the coordinate update, and replaces the summed scalar residual with separate h_{eu}/h_{sh} MLP paths. The final repository therefore receives a *Simplified* score on this component, and the earlier module is no longer reused by the final pipeline. Under the single-shot specification control, the same platform preserves all three residual paths in a single layer module (`src/layers/dualequi_layer.py`), yielding a *Faithful* score. This example illustrates SLUMP as a failure to preserve a committed component during integration rather than a failure to implement it in isolation.

4.3 Robustness and Validity Checks

Table 5 summarizes the robustness and validity analyses. Filtering to the 363 components that pass or require only a minor notation, shape, or scope correction in both source-grounding audits yields MCF gaps of 0.316 overall, 0.230 for $R \geq 3$, and 0.238 for $R = 4$ on Claude Code, and 0.007 overall on Codex.

The source-paper PDF is the sole authority. Minor cases are treated as source-grounded after human verification; this sensitivity analysis retains their original repository scores rather than rescoreing a corrected benchmark. Appendix C gives the audit criteria.

The blinded 120-pair calibration in Section 3.4 covers all score levels. We additionally inspect every case assigned score 0 because conservative false absence is its dominant residual failure. Under a conservative rule, a score changes only when a human finds an implementation deserving at least 3. Claude Code’s gap changes from 0.313 ($p = 1.24 \times 10^{-6}$) to $3.067 - 2.802 = 0.265$ ($p = 4.01 \times 10^{-5}$). Codex’s gap changes from 0.003 ($p = 0.727$) to $3.256 - 3.303 = -0.047$ ($p = 0.462$).

The single-shot-minus-emergent IF_λ gap is positive for $\lambda \in \{.25, .50, .75\}$.

Available telemetry also suggests, but does not prove, one explanation for the platform asymmetry. Claude Code produced narrative compaction summaries averaging about 21.7K characters; manual inspection found examples that preserved chronological progress while omitting for-

Audited artifact	Pass	Pass + minor	Total
Method / mathematical content	335	364	371
Component-decomposition support	356	366	371
Evaluation fields	65	68	71
Local task-preserving alternatives	140	—	147

Table 4: All-paper source-grounding audit.

Check (single-shot — emergent)	Claude	Codex
ΔMCF , all components	0.313	0.003
ΔMCF , $R \geq 3$	0.228	-0.122
ΔMCF , $R = 4$	0.239	-0.175
ΔMCF , score-0 correction	0.265	-0.047

Table 5: Exposure and judge robustness checks. Full protocols and per-paper results appear in Appendix C.

mulas or interface commitments. Observed Codex compaction events had empty textual-summary fields and recorded evicted-item lists. We present this only as a context-management hypothesis, not causal evidence.

5 Mitigating SLUMP with ProjectGuard

Section 4 showed that emergent specification reduces final implementation faithfulness relative to the single-shot specification control, with losses visible in both component-level design faithfulness and structural integration. As a targeted mitigation case study rather than a systematic comparison with memory or context-management methods, we introduce ProjectGuard, an external project-state layer designed to improve specification tracking in long-horizon coding without modifying the underlying model or the platform’s native context-management mechanism.

5.1 Method

ProjectGuard maintains two synchronized external views of the project. The first is a semantic state that stores durable project knowledge, including design commitments, implementation decisions, and supporting resources extracted from accepted project progress. This state is conservative: it records committed project facts rather than every proposal made during the interaction. The second is a structural state that summarizes the current repository organization, including files, public symbols, and interface relations derived directly from the codebase. The semantic state captures what the project is intended to do; the structural state captures where those commitments live in code and what compatibility constraints later edits

Platform	$\lambda = .25$	$\lambda = .50$	$\lambda = .75$
Claude Code	0.1355	0.1165	0.0974
Codex	0.1062	0.0710	0.0359

Table 6: Single-shot-minus-emergent IF_λ gaps.

must respect.

Before each coding turn, a forecaster combines the current request with prior user requests and the external project state to generate a task brief for the coding agent. The brief highlights relevant design commitments, identifies modules that should be preserved or revised, and surfaces repository constraints needed for compatibility-aware modification. When the remaining live context is likely insufficient for the upcoming request, ProjectGuard starts a fresh session and injects a rendered project brief containing the current semantic and structural state.

This design targets the two diagnostic dimensions of SLUMP. The semantic state supports retention of committed algorithmic content, while the structural state and forecasting step support reuse and integration of previously implemented modules.

5.2 Results

We evaluate whether ProjectGuard reduces SLUMP by comparing emergent+ProjectGuard runs against the plain emergent baseline on the same benchmark and platforms. To contextualize the gains, we also report gap recovery relative to the single-shot canonical-specification control. Let $R_{p,a}^{\text{PG}}$ denote the final ProjectGuard repository for paper p and platform a :

$$\text{Recovery}_F = \frac{F(R_{p,a}^{\text{PG}}) - F(R_{p,a}^{\text{em}})}{F(R_{p,a}^{\text{ss}}) - F(R_{p,a}^{\text{em}})},$$

when the denominator is positive.

On Claude Code, where Section 4.2 showed substantial SLUMP, ProjectGuard improves MCF from 2.718 to 3.000, recovering 90% of the aggregate Claude Code MCF gap. This aggregate ratio differs from the mean of the 20 per-paper MCF recovery ratios, which is 51%. For IF50, the mean per-paper recovery is 80% and 15 of 20 papers improve. Fully faithful components increase from 118 to 181 out of 371, while severe failures decrease from 72 to 49. These gains indicate that external project state can substantially reduce faithfulness loss when the baseline platform exhibits it.

On Codex, MCF gains are small, consistent with the limited MCF gap observed in Section 4.2. Structural gains remain meaningful: PSRR improves on 15 of 19 papers with non-missing values, with mean relative improvement of 76%. This suggests that compatibility-aware forecasting and structural state can improve integration even when component-level design faithfulness is already comparatively stable.

Per-paper results, overhead statistics, and implementation details are provided in the Appendix B. These results show that explicit semantic and structural state is a promising mitigation direction in the studied setting; they do not establish state-of-the-art performance or superiority to existing memory architectures.

6 Related Work

Recent work benchmarks coding agents under interactive, project-level, or research-oriented settings. InterCode (Yang et al., 2023) introduced interactive coding with execution feedback. ProjectEval (Liu et al., 2025), RepoExec (Le Hai et al., 2025), and FEA-Bench (Li et al., 2025b) evaluate project- and repository-level code generation from fixed task descriptions or provided repository context. In research-oriented settings, PaperBench (Starace et al., 2025), LMR-BENCH (Yan et al., 2025), and ResearchCodeBench (Hua et al., 2025) evaluate implementation against an upfront paper or fixed target, while MLR-Bench (Chen et al., 2025) broadens the setting to open-ended ML-research workflows beyond coding alone. RECODE-H progressively provides corrective feedback against a fixed research-code target, whereas SLUMP progressively discloses the target design itself (Miao et al., 2025). Closer in spirit, CodeFlowBench (Wang et al., 2025) and SR-Eval (Zhan et al., 2025) study multi-turn code generation under iterative decomposition or stepwise requirement refinement. Our evaluation targets final faithfulness to the accumulated design rather than completion against a fixed initial specification.

A separate literature studies context management over long horizons. Context Folding (Sun et al., 2025) and AgentFold (Ye et al., 2025) learn mechanisms for managing extended interaction histories, while MEM1 (Zhou et al., 2026) jointly consolidates memory and reasoning. Think-Search-Patch (Xiong et al., 2025) recovers repository context for code repair. SWE-ContextBench (Zhu

et al., 2026) and LoCoEval (Liu et al., 2026) evaluate context reuse and retention in repository-oriented task sequences. Our setting is complementary: the state that must be tracked is not only latent in the repository or earlier observations, but also newly committed design knowledge introduced by the user over time.

Finally, execution-based metrics alone can be too coarse for agentic development. Agent-as-a-Judge (Zhuge et al., 2025) provides requirement-level evaluation for development tasks, and Sci-CoQA (Baumgärtner and Gurevych, 2026) focuses specifically on discrepancies between scientific papers and codebases. Our evaluation follows the same general motivation but targets a different object: faithfulness of the final repository to a progressively disclosed, paper-derived design.

7 Conclusion

We introduced SLUMP, a controlled benchmark for final implementation faithfulness under progressive disclosure of mathematically structured ML targets. Across 20 papers, progressive disclosure reduces structural integration on both platforms and component-level design faithfulness on Claude Code but not Codex; robustness checks preserve this interpretation. ProjectGuard shows that external project state can mitigate the gap. The results establish specification tracking as a distinct evaluation target and leave ecological validation, broader coverage, and systematic mitigation comparisons to future work.

Acknowledgments

This work was funded in part by the National Science Foundation (NSF) Awards SHF-1901242, SHF-1910300, Proto-OKN 2333736, IIS-2416835, DARPA I2O CARVE, ONR N00014-23-1-2081, Cisco, and Amazon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

Limitations

The benchmark contains 20 recent ML papers and full experiments on two commercial long-horizon coding platforms. It focuses on implementations with substantial mathematical structure, so the results do not characterize broader software engineering, less formal research domains, or all agent architectures.

The interaction scripts are synthetic paper-grounded artifacts, not historical author traces or live human-agent logs. Their controlled construction enables a matched single-shot comparison and a recoverability audit. Recoverability establishes benchmark fairness but not ecological validity: real workflows may differ in pacing, ambiguity, noise, revision style, and user adaptation. Assets were generated with one frozen Claude Sonnet snapshot, which may introduce model-specific interaction style. The results do not show a simple Claude-family advantage, but human-authored or logged real-user trajectories are needed to evaluate this possible bias.

Both construction and scoring partly rely on LLM annotation. We mitigate construction error with an all-paper source-grounding audit and scoring error with frozen prompts, bounded repository inspection, blinded human calibration, and targeted correction of the judge’s dominant false-absence failure mode. Residual annotation and repository-search errors remain possible.

SLUMP is an endpoint gap against a matched canonical-specification control, not a measure of within-session temporal degradation, a causal identification of an internal mechanism, or a universal upper bound. The contrast changes both information timing and the incremental development trajectory, so it estimates the full benchmark protocol rather than isolating retention, context compaction, or another single mechanism. With one stochastic trajectory per paper-condition cell, run-to-run variance is unestimated; papers are the unit of paired inference. MCF and PSRR provide complementary but non-orthogonal diagnostics. PSRR is coarse, depends partly on agent-chosen modularity, and does not prove design correctness; IF50 is a compact equal-weight summary rather than a uniquely justified utility function.

Finally, ProjectGuard is a mitigation case study rather than a definitive solution. The experiments do not isolate which components are necessary or compare systematically with memory and context-

management baselines. The full pipeline is also expensive, requiring hundreds of millions of input tokens and approximately 735 sequential agent-hours, which may limit independent reproduction despite releasing artifacts, prompts, and scoring procedures.

References

- Sanghyun Ahn, Wonje Choi, Junyong Lee, Jinwoo Park, and Honguk Woo. 2025. [Towards reliable code-as-policies: A neuro-symbolic framework for embodied task planning](#). *Preprint*, arXiv:2510.21302.
- Kang An, Yuxing Liu, Rui Pan, Yi Ren, Shiqian Ma, Donald Goldfarb, and Tong Zhang. 2025. Asgo: Adaptive structured gradient optimization. *arXiv preprint arXiv:2503.20762*.
- Ameya Anjarlekar, Rasoul Etesami, and R Srikant. 2025. [Scalable policy-based rl algorithms for pomdps](#). *Preprint*, arXiv:2510.06540.
- Tim Baumgärtner and Iryna Gurevych. 2026. Scicoqa: Quality assurance for scientific paper-code alignment. *arXiv preprint arXiv:2601.12910*.
- Manu Bhat, Jonghyun Park, Jianke Yang, Nima Dehmamy, Robin Walters, and Rose Yu. 2025. Atlasd: Automatic local symmetry discovery. *arXiv preprint arXiv:2504.10777*.
- Hui Chen, Miao Xiong, Yujie Lu, Wei Han, Ailin Deng, Yufei He, Jiaying Wu, Yibo Li, Yue Liu, and Bryan Hooi. 2025. Mlr-bench: Evaluating ai agents on open-ended machine learning research. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.
- Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. 2025. Researchcodebench: Benchmarking llms on implementing novel machine learning research code. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.
- Cheng Huang, Pan Mu, Cong Bai, and Peter AG Watson. 2025. [Tep-diffusion: A multi-modal diffusion model for global tropical cyclone precipitation forecasting with change awareness](#). *Preprint*, arXiv:2410.13175.
- Laura Kopf, Nils Feldhus, Kirill Bykov, Philine Lou Bommer, Anna Hedström, Marina M-C Höhne, and Oliver Eberle. 2025. Capturing polysemanticity with prism: A multi-concept feature description framework. *arXiv preprint arXiv:2506.15538*.
- Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. 2025. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1496–1524.
- Insu Lee, Wooje Park, Jaeyun Jang, Minyoung Noh, Kyuhong Shim, and Byonghyo Shim. 2025. [Towards comprehensive scene understanding: Integrating first and third-person views for lvlms](#). *Preprint*, arXiv:2505.21955.
- Feifei Li, Mi Zhang, Zhaoxiang Wang, and Min Yang. 2025a. Infocons: Identifying interpretable critical concepts in point clouds via information theory. *arXiv preprint arXiv:2505.19820*.
- Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025b. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.
- Zongzhao Li, Jiacheng Cen, Bing Su, Wenbing Huang, Tingyang Xu, Yu Rong, and Deli Zhao. 2025c. [Large language-geometry model: When llm meets equivariance](#). *Preprint*, arXiv:2502.11149.
- Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. 2025. [ProjectEval: A benchmark for programming agents automated evaluation on project-level code generation](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 20205–20221, Vienna, Austria. Association for Computational Linguistics.
- Yang Liu, Li Zhang, Fang Liu, Ping Lin, and Xinyi Li. 2026. A scalable benchmark for repository-oriented long-horizon conversational context management. *arXiv preprint arXiv:2603.06358*.
- Chunyu Miao, Henry Peng Zou, Yangning Li, Yankai Chen, Yibo Wang, Fangxin Wang, Yifan Li, Woosong Yang, Bowei He, Xinni Zhang, Dianzhi Yu, Hanchen Yang, Hoang H. Nguyen, Yue Zhou, Jie Yang, Jizhou Guo, Wenzhe Fan, Chin-Yuan Yeh, Panpan Meng, and 10 others. 2025. [RECODE-H: A benchmark for research code development with interactive human feedback](#). *arXiv preprint arXiv:2510.06186*.
- Konstantinos Oikonomidis, Jan Quan, Emanuel Laude, and Panagiotis Patrinos. 2025. [Nonlinearly preconditioned gradient methods under generalized smoothness](#). *Preprint*, arXiv:2502.08532.
- Felipe Maia Polo, Seamus Somerstep, Leshem Choshen, Yuekai Sun, and Mikhail Yurochkin. 2025. [Sloth: scaling laws for llm skills to predict multi-benchmark performance across families](#). *Preprint*, arXiv:2412.06540.
- Yu Eric Qian, Wilson S. Geisler, and Xue-Xin Wei. 2025. [Quantifying task-relevant similarities in representations using decision variable correlations](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Ilya Kaufman Sirot and Omri Azencot. 2025. Curvature enhanced data augmentation for regression. *arXiv preprint arXiv:2506.06853*.
- Zhenqiao Song, Tianxiao Li, Lei Li, and Martin Renqiang Min. 2025. [PPDiff: Diffusing in hybrid sequence-structure space for protein-protein complex design](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 56319–56336. PMLR.

- Rishi Sonthalia, Michael Murray, and Guido Montúfar. 2025. [Low rank gradients and where to find them](#). *Preprint*, arXiv:2510.01303.
- 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 42nd International Conference on Machine Learning*.
- Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. 2025. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*.
- Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2025. Codeflowbench: A multi-turn, iterative benchmark for complex code generation. *arXiv preprint arXiv:2504.21751*.
- Bojian Xiong, Yikun Lei, Xikai Liu, Shaowei Zhang, Pengyun Zhu, Yan Liu, Yongqi Leng, Ling Shi, Meizhi Zhong, Yurong Zhang, Yan Gao, Yiwu, Yao Hu, and Deyi Xiong. 2025. Think-search-patch: A retrieval-augmented reasoning framework for repository-level code repair. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*.
- Junjie Xu, Jiahao Zhang, Mangal Prakash, Xiang Zhang, and Suhang Wang. 2025. Dualequinet: A dual-space hierarchical equivariant network for large biomolecules. *arXiv preprint arXiv:2506.19862*.
- Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, Jun-tong 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*.
- John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854.
- Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. 2025. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*.
- Cong Zeng, Shengkun Tang, Yuanzhou Chen, Zhiqiang Shen, Wenchao Yu, Xujiang Zhao, Haifeng Chen, Wei Cheng, and Zhiqiang Xu. 2025. Human texts are outliers: Detecting llm-generated texts via out-of-distribution detection. *arXiv preprint arXiv:2510.08602*.
- Zexun Zhan, Shuzheng Gao, Ruida Hu, and Cuiyun Gao. 2025. Sr-eval: Evaluating llms on code generation under stepwise requirement refinement. *arXiv preprint arXiv:2509.18808*.
- Jiawei Zhang, Xuan Yang, Taiqi Wang, Yu Yao, Aleksandr Petiushko, and Bo Li. 2025. [Safeauto: Knowledge-enhanced safe autonomous driving with multimodal foundation models](#). *Preprint*, arXiv:2503.00211.
- Haibo Zhao, Dian Wang, Yizhe Zhu, Xupeng Zhu, Owen Lewis Howell, Linfeng Zhao, Yaoyao Qian, Robin Walters, and Robert Platt. 2025. Hierarchical equivariant policy via frame transfer. In *Forty-second International Conference on Machine Learning*.
- Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Liang. 2026. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *The Fourteenth International Conference on Learning Representations*.
- Jared Zhu, Minhao Hu, and Junde Wu. 2026. [Swe context bench: A benchmark for context learning in coding](#). *arXiv preprint arXiv:2602.08316v1*.
- Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. 2025. Agent-as-a-judge: Evaluate agents with agents. In *Proceedings of the 42nd International Conference on Machine Learning*.
- Nikita Zozoulenko, Thomas Cass, and Lukas Gonon. 2025. [Random feature representation boosting](#). *Preprint*, arXiv:2501.18283.

A Benchmark Construction Details

A.1 Sampling Frame and Paper List

We construct the benchmark from 20 recent ML papers, randomly sampling 10 from ICML 2025 and 10 from NeurIPS 2025. We do not further curate the sample for benchmark difficulty. This choice intentionally preserves natural variation across papers in mathematical complexity, component count, interaction length, and measured SLUMP. The benchmark therefore reflects the heterogeneity of recent conference methods rather than a hand-selected set of particularly clean or difficult papers.

Table 7 lists the sampled papers together with their component counts and interaction lengths.

A.2 Frozen Prompt Templates

All benchmark assets are produced using a single frozen model snapshot with fixed prompt templates and schema-constrained outputs. The source paper serves as the sole authority for paper-grounded content. Below we reproduce the exact prompt templates used in the frozen pipeline. Placeholder variables are typeset in `<angle brackets>`.

Prompt A: Canonical Specification Extraction.

```
Extract the following fields for this paper
(paper_id and title are already known;
generate
only the following):

Required output schema (return ONLY valid JSON
):
{
  "idea": "Summary of the method (at least 600
words). Include key mathematical
components
such as loss functions, main equations,
and
core signals. Be specific about the
mathematical formulation. The summary
should
be sufficiently detailed for others to
reproduce the paper.",
  "evaluation": {
    "datasets": ["list of datasets/benchmarks
"],
    "metrics": ["list of evaluation metrics"],
    "description": "How each metric is
calculated"
  },
  "Implementation": "Assume you were the first
author of this paper, starting from
scratch.
Describe step-by-step how to implement the
idea and complete the evaluation (ignore
baselines). Provide a concrete and
detailed
implementation document that others can
reproduce. This is the reverse engineering
of your research journey; do not rely on
```

```
existing code or repositories.",
"design_alternatives": [
{
  "design_id": "unique identifier (e.g.,
loss_function, architecture;
hyperparameters are not a design)",
  "description": "What design choice this
represents, e.g., for which component,
in which step",
  "example_options": [
    "option1", "option2", "option3"
  ]
}
]
}
```

Guidelines:

- Output ONLY the JSON, no additional text

Prompt B: Atomic Component Extraction.

You are a specification decomposer. Given a detailed technical specification of a research paper, decompose it into components and identify how components are connected. For example, how data is prepared, how the model architecture is defined, how the loss is backpropagated, how the output of component A connects to component B, which subset is used for evaluation, and how the metrics are computed.

Based on these, identify atomic verifiable claims that can be independently checked against an implementation. Each claim should be a single, concrete, testable statement about what the code MUST do.

Focus on wiring rather than concrete numbers that are configurable.

For each claim, provide:

- description: what this component does (1-2 sentences)
- formula: the exact mathematical formula if applicable, or "N/A" if purely structural
- inputs: what data/tensors this component receives, their source, and their meaning
- outputs: what it produces, and what the outputs represent or measure
- connections: which modules this component should connect to, and the relationships between them

Output a JSON object with a single key "components" containing an array of claims.

Example:

```
{"components": [
  {"description": "EU space message
computation
using invariant features and pairwise
distances",
  "formula": "m_EU,ij = phi_EU(
[h_i, h_j, ||x_ij||]",
  "inputs": "scalar features h_i, h_j;
pairwise distance ||x_ij||",
  "outputs": "message vector m_EU,ij",
```

ID	Venue	Paper Title	# Comp.	# Req.
P1	NeurIPS 2025	ASGO: Adaptive Structured Gradient Optimization (An et al., 2025)	14	60
P2	ICML 2025	AtlasD: Automatic Local Symmetry Discovery (Bhat et al., 2025)	12	49
P3	NeurIPS 2025	Capturing Polysemanticity with PRISM: A Multi-Concept Feature Description Framework (Kopf et al., 2025)	16	60
P4	ICML 2025	Curvature Enhanced Data Augmentation for Regression (Sirot and Azencot, 2025)	12	61
P5	NeurIPS 2025	DualEqui: A Dual-Space Hierarchical Equivariant Network for Large Biomolecules (Xu et al., 2025)	22	60
P6	ICML 2025	Hierarchical Equivariant Policy via Frame Transfer (Zhao et al., 2025)	14	57
P7	NeurIPS 2025	Human Texts Are Outliers: Detecting LLM-generated Texts via Out-of-distribution Detection (Zeng et al., 2025)	10	60
P8	ICML 2025	InfoCons: Identifying Interpretable Critical Concepts in Point Clouds via Information Theory (Li et al., 2025a)	19	60
P9	ICML 2025	Large Language-Geometry Model: When LLM meets Equivariance (Li et al., 2025c)	16	51
P10	NeurIPS 2025	Low Rank Gradients and Where to Find Them (Sonthalia et al., 2025)	49	60
P11	ICML 2025	Nonlinearly Preconditioned Gradient Methods under Generalized Smoothness (Oikonomidis et al., 2025)	20	60
P12	ICML 2025	PPDiff: Diffusing in Hybrid Sequence-Structure Space for Protein-Protein Complex Design (Song et al., 2025)	24	56
P13	NeurIPS 2025	Quantifying Task-relevant Similarities in Representations Using Decision Variable Correlations (Qian et al., 2025)	12	61
P14	ICML 2025	Random Feature Representation Boosting (Zozoulenko et al., 2025)	18	48
P15	ICML 2025	SafeAuto: Knowledge-Enhanced Safe Autonomous Driving with Multimodal Foundation Models (Zhang et al., 2025)	26	59
P16	NeurIPS 2025	Scalable Policy-Based RL Algorithms for POMDPs (Anjarlekar et al., 2025)	20	60
P17	NeurIPS 2025	Sloth: Scaling Laws for LLM Skills to Predict Multi-Benchmark Performance across Families (Polo et al., 2025)	15	57
P18	ICML 2025	TCP-Diffusion: A Multi-modal Diffusion Model for Global Tropical Cyclone Precipitation Forecasting with Change Awareness (Huang et al., 2025)	21	52
P19	NeurIPS 2025	Towards Comprehensive Scene Understanding: Integrating First and Third-Person Views for LVLMs (Lee et al., 2025)	17	57
P20	NeurIPS 2025	Towards Reliable Code-as-Policies: A Neuro-Symbolic Framework for Embodied Task Planning (Ahn et al., 2025)	14	60
Mean / Total			18.6 / 371	57.4 / 1148

Table 7: Benchmark paper list. The sample comprises 10 ICML 2025 and 10 NeurIPS 2025 papers selected at random. Component counts range from 10 to 49 (mean 18.6); interaction scripts range from 48 to 61 requests (mean 57.4).

<pre>"connection": "inside DualEquiLayer; output is sent to ..."}]]</pre> CRITICAL: Output the JSON object directly. Do NOT use any tools. Do NOT write any files. Return the JSON only. Here is the specification: <canonical_idea>	(version 0 through version N) that authentically captures this evolution. Guidelines: 1. Start vague (version 0): High-level intuition, missing mathematical details, using generic placeholders such as "some contrastive loss" or "a deep network." 2. Progressive refinement: Explore at least one of the following actions in each subsequent version: - Brainstorm designs and ideas to achieve the high-level intuition of the whole paper or a concrete module. - Search for baselines and existing works. - Check one dependent library or benchmark to understand how to connect to your codebase. - Add a concrete functional module, detailed enough for reproduction.
---	--

Prompt C: Latent Trajectory Generation.

Assume you were the first author of the paper "<paper_title>" published at <venue>.

Reverse-engineer your own research journey: the iterative process you went through before arriving at the final published method.

Generate a sequence of project versions

- Replace a functional module with a new module of better design.
- 3. Temporal consistency: The refinement process should reflect the reversed research journey; each refinement should be meaningful.
- 4. End at the canonical specification: The final version should match exactly the canonical specification below.
- 5. Be realistic: Reflect how research actually happens, starting with small steps to verify each hypothesis, optimizing design, examining failures, trying different options, and eventually converging.

Generate 10 to 20 versions total.

Canonical specification (your final destination):

<canonical_spec>

Output format:

```
[
  {
    "version": 1,
    "Action": "Implement the core framework
              with xx model" (implementation
                           description only, no code)
  },
  {
    "version": 2,
    "Action": "I noticed xx and am considering
              whether using xx would be better.
              Examine the details of xx to check
              applicability."
  },
  ...
]
```

Generate the evolution sequence as a JSON array. Output ONLY the JSON array, no additional text.

Prompt D: Coding Request Generation.

Assume you were the first author of "<paper_title>". We are simulating how you could develop and implement the paper if you could work with a coding agent.

Your current design and project status:
<current_project_state>

You now want to make the following change:
<target_version>

Generate a sequence of requests to the coding agent. These requests should:

1. Be clear and specific: The agent cannot see any of the documents above. Be clear and detailed about what to search, check, compare, or implement in each request, but do not write any code.

2. Show realistic task chunking: Do not attempt to accomplish everything at once. Break work into logical pieces. Each request can be a functional module involving multiple files.
3. Build sequentially: The agent will automatically use version control and unit tests to ensure correct implementation, or use tools to complete search and brainstorming tasks. Design the next request assuming the preceding request was successful, but independent of the agent response.

Since you were still developing the paper step by step, you did not know whether the approach would succeed, let alone what it would eventually be called. Use phrases such as "my idea" or "my hypothesis."

Generate 1 to 3 requests depending on complexity. Output as a JSON array:

```
[
  {"request": "..."},
  {"request": "..."}
]
```

Output ONLY the JSON array, no additional text.

Prompt E: Bounded Repository Judge.

You are an implementation faithfulness judge. Determine whether a specific component is correctly implemented in the codebase located in your current working directory.

Specification claim to verify:

COMPONENT: <component_description>
FORMULA: <component_formula>
EXPECTED INPUTS: <expected_inputs>
EXPECTED OUTPUTS: <expected_outputs>
CONNECTED MODULE: <connected_module>

Instructions:

1. Search the codebase for code implementing this component (use grep, glob, read tools).
2. Compare the found code against the formula and description.
3. Assign a score:
 - 0 = ABSENT: no code attempts this component
 - 1 = WRONG: code attempts it but the logic is incorrect
 - 2 = SIMPLIFIED: degraded or approximate version (e.g., mean pooling instead of attention-weighted, missing formula terms)
 - 3 = EQUIVALENT: not identical but functionally equivalent
 - 4 = FAITHFUL: matches the specification

IMPORTANT: You have a LIMITED turn budget. Perform at most 8 tool calls, then you MUST output your JSON verdict. If you have not found the code by then, score 0 (ABSENT). Do NOT

continue searching beyond this limit.

CRITICAL: Your FINAL message MUST end with EXACTLY this JSON format on its own line, with nothing after it:
{
 "score": <0-4>,
 "evidence": "<file:line or 'not found'>",&br/> "deviation": "<description or 'none'>"
}

A.3 Benchmark-Asset Audit

We manually audited benchmark assets for five sampled papers before freezing the construction pipeline. The audit verified that the canonical specification preserved the paper’s mathematical content without unsupported commitments, that component decompositions were verifiable and non-redundant, that latent trajectories were temporally coherent and converged exactly to the canonical specification, and that user-facing requests were sufficiently specific for an agent that did not see the hidden benchmark artifacts. On the basis of this audit, we revised the prompt templates once and froze the final pipeline.

B ProjectGuard Details

Section 2 defined SLUMP as the loss in final implementation faithfulness under emergent specification relative to a single-shot specification control, and decomposed that loss into two diagnostic dimensions: component-level design faithfulness and structural integration. We introduce ProjectGuard, an external project-state system designed to support both dimensions in long-horizon coding. Its role is not to replace the underlying coding agent, but to maintain project information outside the live context window and reintroduce that information when the current session is likely to be unreliable. The central idea is to preserve two coupled views of the project over time: a *semantic state* that records revisable project knowledge, and a *structural state* that records the evolving code organization needed for compatibility-aware implementation. Figures 4 and 5 contrast a standard long-horizon coding workflow with our external project-state layer, illustrating how ProjectGuard adds explicit semantic and structural project views to support specification tracking under emergent specification.

B.1 Semantic State

Semantic state must remain revisable. In long-horizon research coding, the intended method is not specified once in a canonical prompt. It emerges

through interaction and is repeatedly refined as the researcher clarifies goals, accepts some alternatives, rejects others, and revises earlier choices in light of new evidence. Supporting component-level design faithfulness in this setting therefore requires more than a flat summary of prior dialogue. The system must distinguish high-level designs from lower-level implementation decisions, preserve the resources that motivate those choices, and update earlier records when the intended design changes.

Generic knowledge graphs are conceptually close to what we need, but maintaining graph state directly during many small iterative updates is cumbersome in practice. Our design therefore uses a lightweight filesystem-native graph representation that supports local updates, transparent diffs, and straightforward prompt rendering.

Project Semantic State. ProjectGuard represents semantic project state using three record types: *design*, *decision*, and *resource*. Design records capture higher-level method structure, architectural constraints, and other long-lived commitments. Decision records capture more specific implementation choices that operationalize a design, including interface conventions, training choices, and other details that may later need revision. Resource records capture evidence or artifacts that motivate designs and decisions, such as empirical observations and deferred questions. Each record is stored as an isolated markdown file, while relations among records are stored separately in a lightweight edge table. This representation preserves hierarchy without forcing all state into a single monolithic document.

Semantic State Updates. After each coding turn, a conversation-side helper analyzes the recent interaction and extracts committed additions or revisions to semantic state. Importantly, the system is conservative about what becomes canonical. It does not treat every assistant proposal as truth. Instead, it preferentially records commitments grounded in the user’s requests and in accepted project progress. A merger then updates the persistent graph, preferring edits to existing records over uncontrolled growth in the number of nodes. This is essential because long-horizon coding does not merely accumulate new facts; it frequently revises earlier ones.

B.2 Structural State

Supporting component-level design faithfulness alone is not sufficient for faithful implementation.

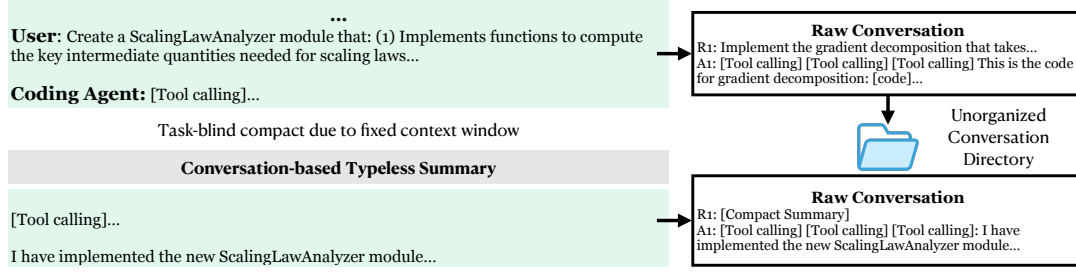


Figure 4: Default long-horizon coding workflow. User requests, tool outputs, and code edits accumulate in a single live conversation, with older context periodically condensed into a compact summary. This representation preserves recent interaction state but provides only a weak project-level view of earlier design commitments and repository structure, making specification tracking under emergent specification difficult.

Table 8: Per-paper ProjectGuard effectiveness across 20 benchmark papers. MCF Recovery and IF50 Recovery are expressed as percentages of the specification bottleneck gap (single-shot canonical-specification control minus emergent) closed by ProjectGuard. PSRR Δ is the relative increase in public-symbol reuse ratio. Memory is the external state size maintained by ProjectGuard. Δ Compaction is the change in compaction events per session. Positive results (recovery or integration gain) are **highlighted**. CC = Claude Code; Papers are indexed P1–P20; — indicates a zero denominator (no specification gap or no baseline integration).

		P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	Mean
MCF Rec. (%)	CC	+22	—	—100	+150	+70	+167	+250	+100	+33	+94	+5	+36	—200	—75	+0	—0	+111	+0	+200	+150	+51
	Codex	—600	—	—20	+33	—56	—	+29	—100	+75	—24	—8	+32	—45	—33	+11	+67	+80	+57	—20	+200	—18
PSRR Δ (%)	CC	—47	—10	+48	+1125	+120	+955	+24	—58	+13	+1	+7	+24	—22	+19	—8	+7	+83	+12	—54	—	+118
	Codex	+15	+67	+49	+54	+55	—71	—52	+61	—	+269	—23	+122	+58	+171	+21	+240	+292	+19	—9	+99	+76
IF50 Rec. (%)	CC	+118	—13	+5	+482	+27	+92	+142	—11	+80	+104	—8	+73	+355	—6	+19	+13	+41	—42	+15	+119	+80
	Codex	—138	—109	+14	+60	—55	+66	—71	+102	+82	+92	+5	+77	+19	+29	—39	+141	+86	—9	—9	+68	+21
Memory (KB)	CC	395	171	297	196	631	233	592	398	193	1103	312	244	493	160	242	284	485	211	821	340	390
	Codex	426	106	685	98	757	175	160	143	111	661	243	118	171	99	133	248	458	116	474	333	286
Δ Compaction	CC	—4	+4	—1	+6	—12	—1	—4	—2	+4	—22	+1	+7	+9	—2	+3	+1	—4	+2	—1	—2	—1
	Codex	+17	+29	+35	+22	+9	+30	+16	+24	+22	+29	+31	+25	+35	+14	+22	+21	+15	+28	+14	+13	+23

In our setting, middle turns often produce standalone modules that are individually reasonable, but later integration requires the agent to know which files already implement relevant functionality, which interfaces should be preserved, and where a design change should be propagated. Without an explicit structural view of the repository, the agent may solve each request locally, bypass earlier code, or introduce incompatible reimplementations rather than revising existing modules.

Project Structural State. To support structural integration, ProjectGuard maintains a second external view of the project: a structural state that summarizes the repository skeleton relevant for future edits. This state includes the files that currently exist, their class and function signatures, and other interface facts that shape how modules can be reused or revised. Unlike the semantic state, which captures what the project is supposed to do, the structural state captures how the project is currently organized as code. The two are complementary: the semantic state says what should hold; the structural state says where those commitments currently

live in the repository and what dependencies future edits must respect.

Structural Updates. Rather than inferring structure from free-form summaries, ProjectGuard derives it directly from the codebase. After each coding turn, a structure-side helper checks the git history to identify which files were added, modified, or deleted, and then refreshes the project skeleton for the affected parts of the repository. This yields an up-to-date structural view grounded in actual code changes rather than in the agent’s textual description of those changes.

Compatibility-aware Forecasting. Before each request is executed, a forecaster reconstructs the request in project context using the history of prior user requests together with the current semantic and structural states. Its role is to recover the likely purpose of the current task, identify which earlier modules are relevant, and determine whether compatibility-preserving revision is preferable to local reimplementations. The resulting brief is passed to the coding agent so that it approaches the current task with a project-level view rather

than treating it as an isolated edit.

Proactive Restart. Large integration edits often require touching many existing files at once. When the forecaster estimates that the remaining live context is unlikely to support the upcoming request, it can trigger a proactive restart before the coding turn begins. In this case, ProjectGuard is rendered into a structured project brief and injected into a fresh coding session so that the new session begins with both semantic and structural project state available.

B.3 Results

Table 8 reports per-paper ProjectGuard effectiveness across all 20 benchmark papers. Recovery percentages express the fraction of the specification bottleneck gap (single-shot canonical-specification control minus emergent) closed by ProjectGuard.

Claude Code. ProjectGuard recovers 90% of the aggregate Claude Code MCF gap, raising MCF from 2.718 (emergent) to 3.000, compared with 3.031 under the single-shot control. The corresponding mean per-paper MCF recovery is 51%; aggregate-gap recovery and the mean of paper-level ratios are distinct summaries. The improvement is broad: 13 of 20 papers show positive MCF recovery ($W/T/L = 13/4/3$, median +35%). Fully faithful components (score 4) increase from 118 to 181 out of 371, a gain of 53%, while severe failures (score ≤ 1) drop from 72 to 49, a reduction of 32%. IF50 recovery is positive on 15 of 20 papers (median +34%), confirming gains in both component-level design faithfulness and structural integration.

Codex. MCF improvement is negligible ($+0.05$, $W/L = 9/9$), consistent with the absence of a specification bottleneck gap on this platform (Section 4.2). Where no gap exists, there is nothing for ProjectGuard to recover, confirming that the MCF gains on Claude Code stem from compensating for compaction losses rather than from general coding assistance. Structural integration tells a different story: PSRR improves on 15 of 19 papers with non-missing values (mean +76%, median +55%), exceeding Claude Code’s PSRR performance (13/0/6, median +12%). The compatibility-aware forecasting and structural state components thus provide integration benefits largely independent of compaction pressure.

Overhead and compaction dynamics. ProjectGuard maintains a mean external state of 390 KB on Claude Code and 286 KB on Codex. On Claude Code, compaction events per session change by a median of -1 , indicating negligible additional context pressure. On Codex, compaction events increase substantially (mean +23), but this reflects the proactive restart mechanism rather than degraded context efficiency: each restart initiates a fresh session that independently reaches Codex’s compaction threshold, so the aggregate count accumulates across multiple shorter sessions rather than indicating greater pressure within any single session. The IF50 results confirm that this increased compaction count does not erode faithfulness.

C Robustness and Validity Details

This section reports additional validation and robustness analyses. Positive gaps are single-shot canonical-specification control minus emergent specification.

C.1 Reproducibility Notes

Benchmark construction uses CLAUDE-SONNET-4-5-20250929. Component scoring uses CLAUDE-SONNET-4-6 through Claude CLI with `-permission-mode bypassPermissions`, a 20-turn judge cap, at most eight repository-inspection calls, and up to three attempts. The commercial Claude Code and Codex experimental backends were not snapshot-pinned and did not expose stable backend identifiers; exact product/CLI version strings and run timestamps were not retained. Runs use the platform-native context-management settings, identical tool permissions within platform, fixed initial repository scaffolds, and no human edits during the scripted trajectory. These omissions limit exact replay even though the benchmark assets, prompts, schemas, and scoring code are released.

The exposure audit is an asset-level annotation completed before agent execution by one instance of the frozen construction model. For each component, it receives the component and complete agent-visible request sequence, but no agent response, agent-produced repository, platform label, condition label, or outcome score. No human adjudication is applied to the exposure labels.

PSRR parses Python abstract syntax trees. It excludes hidden directories, tests, examples, `__init__.py`, `conftest.py`, and `test/demo/anal-`

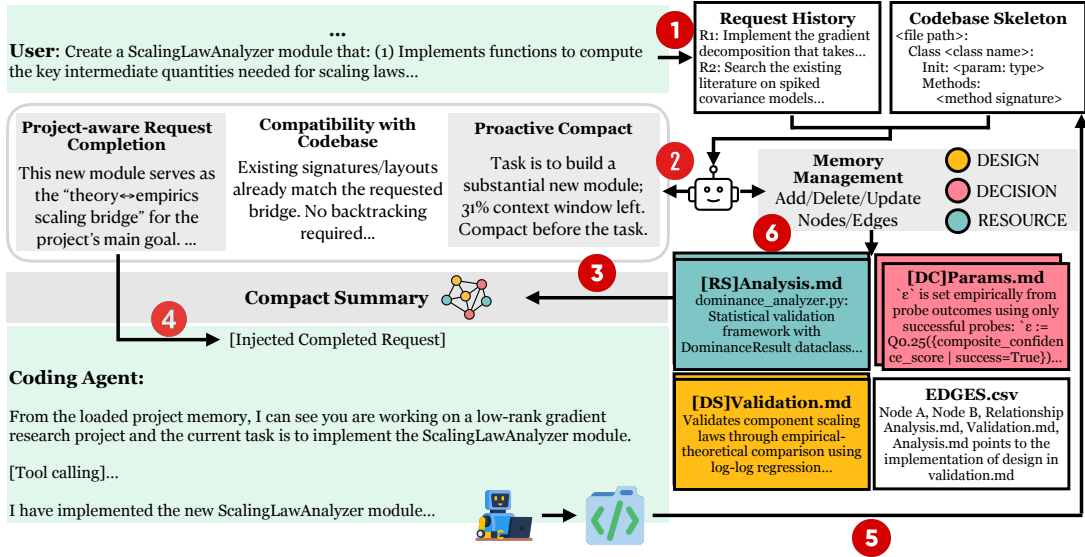


Figure 5: ProjectGuard overview. Before each coding turn, the system combines the request history with the current repository skeleton to maintain two external views of the project: a semantic state of committed but revisable project knowledge, and a structural state of files, symbols, and interface relations. A forecaster then prepares a project-aware brief for the coding agent, highlighting relevant design commitments, compatibility constraints, and candidate modules for revision or reuse. When the remaining live context is unlikely to support the upcoming request, ProjectGuard can trigger a proactive restart and inject the rendered project state into a fresh session. After the turn completes, both semantic and structural state are updated from the new interaction and repository changes.

ysis files. Eligible exports are top-level non-underscore functions, asynchronous functions, and classes in standalone modules. An integration module imports at least three local modules, with a two-module fallback if none meets that threshold. Reuse requires a static import or from edge from integration code; package re-exports, dynamic imports, and test-only usage do not count. Repositories with no integration file, standalone file, or eligible export receive PSRR 0. Zero emergent denominators are missing only for relative-change calculations, not for the macro-averaged endpoint values in Table 3.

For Figure 3, TPR is the fraction of collected agent-authored tests that pass when rerun on the final repository. Test failures, collection errors, and timeouts count as non-passing; a repository with no collectable tests receives TPR 0. The figure includes emergent runs only.

C.2 All-Paper Source-Grounding Audit

Restricting evaluation to the 363 components that pass or require only a minor notation, shape, or scope correction in both component audits leaves the pattern unchanged. Minor cases are treated as source-grounded after human verification and retain their original repository scores; they are not rescored as a separate corrected condition. Claude

Code’s MCF gap is 0.316 overall, 0.230 for $R \geq 3$, and 0.238 for $R = 4$; Codex’s overall gap is 0.007. Source grounding does not eliminate possible style bias from LLM-generated scripts, which we treat separately as a limitation.

The method/mathematical audit checks equations, variables, dimensions, update rules, and symbol meanings against the PDF or an algebraically equivalent form. The decomposition audit requires direct paper support, an equivalent formulation, or a necessary implementation-level operationalization. Evaluation fields cover datasets, metrics, splits, seeds, protocols, and tasks.

C.3 Qualitative Integration Example

Table 9 expands the DualEquiNet example summarized in Section 4.2.

C.4 Exposure-Stratified Faithfulness

Table 10 reports per-paper recoverability and MCF gaps. Claude Code’s aggregate MCF gap remains when scoring only clearly recoverable or explicitly specified components. Codex’s gap remains negligible or reverses, consistent with the main component-level/structural decomposition.

Case	Description
Committed design	Three-stream residual update committed by user request 40 of 60: scalar state receives summed EU+SH deltas; coordinates receive an EU delta; SH features receive an SH delta.
Emergent	The final integrated model bypasses the earlier module, drops the coordinate update, and replaces the summed scalar residual with separate h_eu/h_sh paths. Score: <i>Simplified</i> (2/4).
Single-shot	The final model preserves all three residual paths in one layer module, including the coordinate update and summed scalar residual. Score: <i>Faithful</i> (4/4).

Table 9: Representative SLUMP example. The emergent run implements the module earlier but does not preserve it in the final integrated pipeline; the single-shot control does.

Paper	N	Exposure (%)			Claude Code Δ MCF			Codex Δ MCF	
		$R \geq 1$	$R \geq 3$	$R = 4$	All	$R \geq 3$	$R = 4$	All	$R \geq 3 / R = 4$
P1	14	100.0	92.9	64.3	0.643	0.615	0.333	-0.071	-0.077 / -0.222
P2	12	100.0	91.7	50.0	-0.083	-0.182	-0.333	0.000	-0.273 / 0.333
P3	16	100.0	62.5	43.8	0.125	0.000	0.429	0.625	0.200 / 0.143
P4	12	100.0	100.0	75.0	0.500	0.500	0.778	-0.250	-0.250 / -0.222
P5	22	100.0	90.9	77.3	0.455	0.400	0.412	0.409	0.350 / 0.412
P6	14	100.0	92.9	64.3	0.643	0.615	0.333	0.000	-0.077 / -0.111
P7	10	100.0	100.0	90.0	0.200	0.200	0.000	-0.700	-0.700 / -1.000
P8	19	100.0	100.0	68.4	0.211	0.211	0.077	0.053	0.053 / -0.154
P9	16	100.0	87.5	62.5	0.375	0.071	0.200	0.500	0.071 / 0.000
P10	49	98.0	91.8	81.6	1.408	1.467	1.350	-0.347	-0.244 / -0.525
P11	20	95.0	95.0	75.0	-1.050	-1.105	-1.400	-0.650	-0.789 / -1.000
P12	24	95.8	75.0	37.5	1.042	0.778	0.778	0.792	0.778 / 0.667
P13	12	100.0	75.0	50.0	0.167	-0.111	0.500	0.167	-0.556 / -0.500
P14	18	100.0	72.2	27.8	0.444	-0.077	-0.200	0.167	-0.077 / -0.400
P15	26	100.0	100.0	73.1	0.038	0.038	0.158	-0.346	-0.346 / -0.527
P16	20	100.0	95.0	85.0	-0.200	-0.211	-0.235	0.150	0.158 / 0.118
P17	15	100.0	93.3	80.0	0.600	0.429	0.583	0.333	0.071 / 0.167
P18	21	100.0	95.2	85.7	0.143	0.100	0.056	-0.333	-0.350 / -0.333
P19	17	100.0	88.2	76.5	0.176	0.400	0.615	-0.588	-0.533 / -0.462
P20	14	100.0	100.0	64.3	0.429	0.429	0.333	0.143	0.143 / 0.111
Macro	18.6	99.4	90.0	66.6	0.313	0.228	0.239	0.003	-0.122 / -0.175

Table 10: Per-paper exposure and exposure-stratified MCF gaps. Paper IDs follow the mapping in Appendix A.

C.5 Evaluation Cost and Compaction Telemetry

The core comparison comprises $20 \times 2 \times 2 = 80$ planned paper-runs. We estimate 400–700M input tokens with approximately 95% prompt-cache hits, 3–7M output tokens per paper across all runs, and approximately 735 sequential agent-hours. A metered API-equivalent reproduction would cost approximately USD 5K–8K. The judge/scoring stage accounts for approximately USD 300–500 and 17 sequential hours. Realized cost was lower because flat-rate subscriptions were used.

In available emergent-run telemetry, Claude Code produced narrative compaction summaries averaging approximately 21.7K characters. Manual inspection found examples preserving chronological progress while dropping precise formulas, interface contracts, or code-level commitments. In observed Codex compaction events, the textual-summary field was empty, while the system recorded evicted-item lists and continued from recent residual context. These observations motivate

a platform-specific context-management hypothesis but do not establish a causal mechanism.