FRONTIERREFACTOR: BENCHMARKING MULTI-SCALE PERFORMANCE-ORIENTED CODEBASE REFACTORING

Anonymous authors
Paper under double-blind review

ABSTRACT

Autonomous codebase refactoring requires coding agents to restructure working software while strictly preserving externally observable behavior, a process driven in production by execution performance and compute efficiency rather than stylistic preference. However, existing benchmarks evaluate these requirements in isolation: refactoring benchmarks evaluate functional correctness alone while overlooking severe runtime regressions, whereas code performance benchmarks target isolated micro-kernels that ignore codebase-scale complexity. We observe that software refactoring scale is governed by the scope of the *observable contract*, the caller-visible interface that fixes behavioral invariants while freeing private implementations for optimization. Building on this insight, we introduce FRONTIERREFACTOR, a benchmark of 60 qualified engineering tasks spanning four scales from computational kernels to full repositories across in-place optimization and cross-language rewriting. Evaluating frontier coding agents through a reference-free cascaded protocol across 1,080 execution runs reveals that correctness and speedup are largely decoupled: 41.2% of candidate submissions satisfy functional equivalence yet fail to deliver verified performance improvements, with 19.3% suffering active performance regressions and 21.9% trapped in measurement noise. Furthermore, refactoring paradigms reveal orthogonal engineering bottlenecks: while cross-language rewrites secure substantial speedup dividends from native compilation upon satisfying functional checks, agents struggle with contract fidelity, dropping from a 65.0% pass rate on modules to 32.6% at package scale; conversely, in-place optimization reliably preserves behavioral contracts at 86.5% but rarely achieves acceleration without physical performance profiling. Trajectory analysis demonstrates that agents frequently fall victim to uncalibrated micro-benchmarking illusions, an operational blind spot that multi-turn execution timing feedback directly resolves, raising strict success from 16.0% to 40.0%.

1 INTRODUCTION

Autonomous codebase refactoring has emerged as a crucial task for evaluating whether AI agents can restructure complex software while maintaining contractual invariants (Xu et al., 2026b; Shi et al., 2026; Hong et al., 2026; Yan & Kjolstad, 2026). Recent efforts have made foundational progress in evaluating modular reorganization and dependency modernizations. Yet, they overlook the single most pervasive driver of software rewriting in practice: *performance-oriented refactoring*. Production engineering rarely tolerates the risk of modifying working code solely for stylistic elegance. Instead, refactoring is propelled by physical operational constraints: meeting strict latency requirements, curtailing millions of dollars in compute infrastructure, and breaking architectural bottlenecks (OpenAI, 2025a).

Evaluating coding agents on performance refactoring presents three foundational challenges (Figure 1): (1) *Speed-blind refactoring vs. single-scale optimization*: Existing refactoring suites evaluate correctness alone (Xu et al., 2026b; Shi et al., 2026; Hong et al., 2026), ignoring runtime regressions, while performance suites isolate micro-kernels (Ouyang et al., 2025; Shypula et al., 2024; Shetty et al., 2025; He et al., 2026; Ma et al., 2026) that ignore multi-file systems. (2) *Cold-path edits and flaky test suites*: Arbitrary edits often target non-critical paths with negligible runtime shares,

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

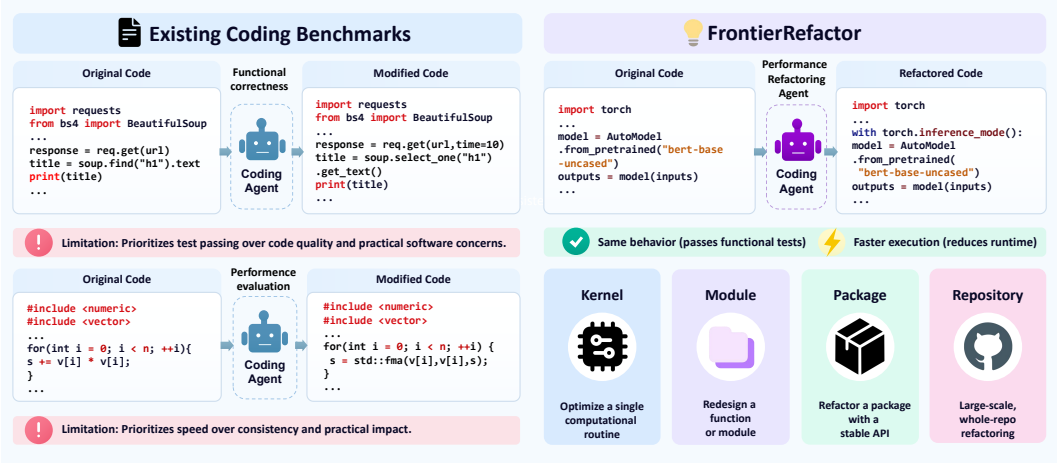


Figure 1: Conceptual comparison between existing benchmarks and FRONTIERREFACTOR. Left: Existing coding benchmarks evaluate functional correctness alone (blind to latency regressions) or isolate narrow micro-kernels (ignoring multi-file complexity). **Right**: FRONTIERREFACTOR formalizes refactoring through the *observable contract*, jointly enforcing behavioral equivalence and noise-calibrated speedup from kernels to repositories.

while raw test suites contain non-deterministic assertions triggering false rejections (Alshammari et al., 2024; Brown et al., 2024). (3) *Hardware volatility and reward hacking*: Wall-clock measurements fluctuate with machine jitter (Maricq et al., 2018; Henning et al., 2025), while agents exploit shortcuts such as inter-trial caching or timing harness introspection.

To address these challenges, we introduce FRONTIERREFACTOR, a multi-scale benchmark of 60 qualified tasks across seven languages, four contract granularities, and two operational forms (in-place optimization and cross-language rewriting). FRONTIERREFACTOR overcomes these barriers via three core designs: (1) *Observable contracts* define software scale through caller-visible interfaces (O), preserving behavioral invariants while granting complete internal redesign freedom across four granularities: *Kernel*, *Module*, *Package*, and *Repository* (Section 3.3); (2) *Amdahl qualification* verifies that targeted components dominate execution under Amdahl's Law (Amdahl, 1967), filtering non-deterministic probes across repeated runs; and (3) *Cascaded reference-free gating* enforces dependency compliance, zero-partial-credit functional equivalence, and noise-calibrated runtime speedup without artificial human patch ceilings.

Evaluating frontier coding agents across foundation models and production harnesses (Anthropic Claude Code (Anthropic, 2025), OpenAI Codex (OpenAI, 2025b), DeepSeek (Xu et al., 2026a)) across 1,080 execution runs yields three critical insights. First, behavioral preservation and physical acceleration decouple: among 774 functionally passing submissions, 57.5% fail the performance gate (208 active regressions, 237 trapped in noise). Second, refactoring exhibits an architectural trade-off: in-place edits reliably preserve contracts (86.5% pass rate) but struggle to locate Amdahl bottlenecks (17.8% strict success), whereas cross-language rewrites readily accelerate once compiled but suffer contract fragility (dropping from 65.0% on modules to 32.6% on packages). Third, trajectory audits expose an operational profiling blind spot where agents optimize for syntactic readability over empirical costs; multi-turn timing feedback directly resolves this deficit, boosting strict success from 16.0% to 40.0%.

In summary, this work makes three primary contributions:

- **Conceptual Paradigm:** We formalize multi-scale refactoring through the *observable contract*, decoupling caller-visible invariants from private implementation freedom across four software scales.
- **FRONTIERREFACTOR Benchmark:** We curate 60 Amdahl-qualified tasks across seven languages, paired with a reference-patch-free cascaded protocol featuring automated probe distillation and noise calibration.
- **Empirical Discoveries & Diagnostics:** Evaluations across 1,080 runs show correctness and speedup decouple (41.2% non-accelerating), diagnose an operational profiling blind spot, and confirm execution timing feedback resolves it.

Table 1: Comparison with existing performance optimization and codebase refactoring benchmarks. **Code scope**: measured in lines of code (LoC). **Timed**: requires measured runtime speedup. **Behavior**: enforces strict functional equivalence. **Multi-scale**: spans 10^1–10^4+ LoC. **Cross-language**: supports cross-language rewriting. **No ref patch**: reference-free evaluation without ground-truth patches. **Timing defense**: robust against measurement noise and cheat exploits.

Benchmark	Code scope (LoC)	Timed	Behavior	Multi-scale	Cross-language	No ref patch	Timing defense
Code performance optimization							
KernelBench (Ouyang et al., 2025)	10–100	✓	✓	✗	✗	✓	✗
PIE (Shypula et al., 2024)	50–200	✓	✓	✗	✗	✗	✗
GSO (Shetty et al., 2025)	100–500	✓	✓	✗	✗	✓	✗
SWE-Perf (He et al., 2026)	50–300	✓	✓	✗	✗	✓	✗
SWE-fficiency (Ma et al., 2026)	50–500	✓	✓	✗	✗	✓	✗
Behavior-preserving codebase refactoring							
SWE-Refactor (Xu et al., 2026b)	10–100	✗	✓	✗	✗	✗	✗
SWE-Bench ProMax (Shi et al., 2026)	100–1,000+	✗	✓	✗	✗	✗	✗
SWE Refactor Bench (Hong et al., 2026)	1,000–10,000+	✗	✓	✗	✓	✓	✗
FRONTIERREFACTOR (Ours)	**10–10,000+**	✓	✓	✓	✓	✓	✓

2 RELATED WORK

Code performance optimization benchmarks. Evaluating coding agents on execution speedup spans standalone routines (Shypula et al., 2024; Chen et al., 2024), GPU operator kernels (Ouyang et al., 2025), and repository-level tasks in Python (Shetty et al., 2025; He et al., 2026; Ma et al., 2026; Sehgal et al., 2026; Jing et al., 2026; Cui et al., 2026; Nangia et al., 2026). However, existing benchmarks restrict evaluation to a single codebase scale (Press et al., 2026), while raw timing without noise calibration exhibits severe hardware sensitivity that flips machine rankings (Chen et al., 2026).

Behavior-preserving codebase refactoring. Conversely, behavior-preserving refactoring benchmarks evaluate method-level Java edits (Xu et al., 2026b), multi-file cross-repository coordination (Shi et al., 2026), or full-stack migrations (Hong et al., 2026) under software quality standards (IEC, 2011; Song et al., 2026). Crucially, none measures runtime: functionally passing yet catastrophically slow code passes unconditionally, and identity patches succeed trivially. Furthermore, agents routinely bypass assertions via caching or test tampering in unhardened harnesses (Chen et al., 2026; Zhong et al., 2026; Deng et al., 2026). Table 1 positions FRONTIERREFACTOR against prior suites, contrasting its multi-scale coverage, reference-free gating, and timing defenses (Appendix A.9, Table 9).

3 THE FRONTIERREFACTOR BENCHMARK

3.1 TASK FORMULATION

The goal of performance refactoring is to accelerate an existing codebase while strictly preserving its observable behavior (Zhao et al., 2026c; Fazel et al., 2026; Klepl et al., 2026; Zhao et al., 2026b). Formally, each refactoring task τ is defined as a 4-tuple containing an exposed workspace and a held-out evaluation suite:

$$\tau = \big(S,\ O,\ \mathcal{T},\ \mathcal{D}\big), \tag{1}$$

where S is the original codebase, O is its caller-visible interface (the *observable contract*), $\mathcal{T}$ is a held-out functional test suite, and $\mathcal{D}$ is a held-out set of benchmark workloads. To prevent test leakage and benchmark overfitting (Zhao et al., 2026a; Zhang et al., 2026), the evaluation harness withholds $\mathcal{T}$ and $\mathcal{D}$ out-of-band, exposing only (S, O) to autonomous coding agent $\mathcal{A}$, which generates a candidate refactored implementation S':

$$S' \leftarrow \mathcal{A}(S, O). \tag{2}$$

The contract O establishes the legal optimization boundary: it fixes caller-visible functional invariants (such as function signatures, return types, and CLI flags) along with operational compliance policies $\Pi = (\Pi_{\text{allow}}, \Pi_{\text{prohib}})$ governing permitted dependencies and prohibited symbols that candidate S' must satisfy, while leaving internal implementation details (algorithms, data structures, and programming language) entirely unconstrained. The agent's objective is to ensure candidate S' satisfies contract compliance Π, passes all functional probes in $\mathcal{T}$, and maximizes the worst-case runtime

Table 2: FRONTIERREFACTOR statistics across the four scales. **In-place / Cross-lang**: task distribution across the two optimization forms. **Original LoC**: lines of code in refactoring scope (median and range). **Probes**: frozen differential checks (median and range; at Repository scale, each probe represents an integration suite aggregating up to 5,588 assertions). **Workloads**: benchmark workloads per task. **Nominal noise floor** $\bar{\theta}$: median 99th percentile of baseline self-speedup across $K = 50$ null calibration runs. **Target share**: baseline runtime in targeted component. **SNR**: achievable speedup over null spread.

Scale	Tasks	In-place / Cross	Original LoC	Probes	Workloads	Nominal noise $\bar{\theta}$	Target share	SNR
Kernel	18	9 / 9	48 (18–96)	57 (57–57)	3 (2–4)	1.07 (1.05–1.12)	98.2% (92.0%–100.0%)	12.4× (7.8–24.1)
Module	18	8 / 10	186 (64–420)	57 (56–57)	3 (2–5)	1.07 (1.05–1.10)	89.6% (71.0%–98.0%)	8.6× (4.5–18.2)
Package	15	7 / 8	1,420 (580–3,850)	166 (57–197)	4 (3–6)	1.07 (1.04–1.09)	82.1% (65.0%–95.0%)	7.2× (3.9–15.0)
Repository	9	6 / 3	14,280 (3,400–48,000)	13 (10–57)	4 (3–5)	1.11 (1.08–1.13)	74.5% (58.0%–91.0%)	5.8× (3.2–11.4)
Total	60	30 / 30	420 (18–48,000)	57 (10–197)	3 (2–6)	1.07 (1.04–1.13)	86.4% (58.0%–100.0%)	8.1× (3.2–24.1)

speedup of S' over baseline S (formalized in Section 3.2). While universal program equivalence is theoretically undecidable for arbitrary Turing-complete code (Rice, 1953), FRONTIERREFACTOR's dense, calibrated differential probe suites $\mathcal{T}$ empirically verify caller-facing behavioral invariants across return values, exceptions, standard streams, exit codes, and boundary inputs (Section 3.3).

3.2 EVALUATION PROTOCOL

Unlike bug-fixing benchmarks that rely on human reference patches, performance refactoring admits no single canonical solution: human patches can be sub-optimal, public patches risk benchmark memorization, and cross-language rewrites share zero syntactic overlap. FRONTIERREFACTOR evaluates candidates reference-free, measuring behavioral fidelity and runtime acceleration directly against original codebase S through three cascaded gates:

$$G_{\text{comp}}(S') = \mathbb{I}\big[\,\text{symbols}(S') \cap \Pi_{\text{prohib}} = \emptyset \;\wedge\; \text{deps}(S') \subseteq \Pi_{\text{allow}}\big], \tag{3}$$

$$G_{\text{func}}(S') = \mathbb{I}\big[\forall x \in \mathcal{T},\; S'(x) \equiv S(x)\big], \tag{4}$$

$$G_{\text{perf}}(S') = \mathbb{I}\left[r(S') > \theta_{\text{trial}}\right], \quad \text{where} \quad r(S') = \min_{d \in \mathcal{D}} \frac{\text{Runtime}(S, d)}{\text{Runtime}(S', d)}. \tag{5}$$

Gate 1: Compliance (G_{comp}) audits candidate S' against contract-permitted dependencies Π_{allow} and prohibited symbols Π_{prohib} via static AST parsing and container isolation, intercepting unauthorized third-party libraries, timer introspection, inter-trial state caching, and baseline tampering.

Gate 2: Functionality (G_{func}) tests differential behavioral equivalence across frozen probe suite $\mathcal{T}$ with zero partial credit across function signatures, return values, exceptions, standard streams, and exit codes.

Gate 3: Performance (G_{perf}) enforces that worst-case speedup across workloads $\mathcal{D}$ strictly clears the empirical noise floor. Each workload $d \in \mathcal{D}$ executes after two warmup runs across $R = 5$ repetitions; $\text{Runtime}(S, d)$ measures core workload execution time using high-resolution monotonic clocks (excluding container boot and cold JIT overhead), taking the 20% trimmed mean across repetitions to suppress transient outliers. To protect against machine jitter, the evaluation harness deploys a two-tier calibration architecture: at the macro level, we record the nominal median noise floor $\bar{\theta}$ from offline paired baseline self-comparisons (detailed in Table 2); at the micro evaluation level, each candidate execution dynamically computes an instantaneous batch threshold $\theta_{\text{trial}} = \max\left(\bar{\theta},\, \text{Quantile}_{0.95}(\mathcal{D}_{\text{batch}})\right)$ via $M = 5$ interleaved paired runs, ensuring 0.0% false positive rate under empirical null calibration (formal derivation in Appendix A.5).

Continuous scoring and outcome partition. A candidate achieves verified *strict success* if and only if it clears all three gates sequentially:

$$G_{\text{strict}}(S') = G_{\text{comp}}(S') \times G_{\text{func}}(S') \times G_{\text{perf}}(S') \in \{0, 1\}. \tag{6}$$

The evaluation order is strict: compliance prevents cheating, functionality comes before timing because the speed of a broken program is meaningless, and timing executes last under clean physical conditions. To reward granular progress while penalizing unaccelerated code, the evaluation proto-

col pairs strict success with a continuous **Overall Score**:

$$\text{Score}(S') = \begin{cases} 0, & \text{if } G_{\text{comp}}(S') = 0, \\ 0.5 \cdot \frac{|\mathcal{T}_{\text{passed}}|}{|\mathcal{T}|}, & \text{if } G_{\text{comp}}(S') = 1 \wedge G_{\text{func}}(S') = 0, \\ 0.5 + 0.5 \cdot \min\big(r(S'), 2\big), & \text{if } G_{\text{func}}(S') = 1 \wedge \text{modified}(S, S'), \\ 0, & \text{otherwise (inaction / identity).} \end{cases} \tag{7}$$

where $\mathcal{T}_{\text{passed}} = \{x \in \mathcal{T} \mid S'(x) \equiv S(x)\}$ denotes the subset of differential probes passed by candidate S'. This continuous formulation maps partial functional coverage to $[0, 0.5)$, anchors baseline functional equivalence at 1.0 ($r(S') = 1.0$), and scales linearly with speedup up to a cap of 1.5 ($r = 2.0$), while $\text{modified}(S, S')$ filters out empty identity edits. Crucially, whereas $\text{Score}(S')$ provides a dense progress signal for model optimization and training guidance, the binary gate G_{strict} serves as the sole authoritative criterion for benchmark certification, requiring worst-case speedup to decisively clear empirical hardware noise floor θ_{trial}. While the evaluation protocol formally partitions attempts into five mutually exclusive outcomes (including Compliance Failure, formalized in Appendix A.5), zero candidate submissions violated sandboxing constraints in primary evaluation ($G_{\text{comp}} = 1$ across all 1,080 runs). Consequently, Figure 2 visualizes the four non-empty outcome categories, which sum to exactly 100.0% across systems.

3.3 Dataset Statistics

FrontierRefactor curates 60 qualified tasks balanced between in-place optimizations and cross-language rewrites (30 each; Table 2; catalog in Appendix A.3). Rather than line counts, software scale is formalized by the scope of observable contract O across four expanding granularities: (1) *Kernel* (18 tasks, median 48 LoC): restricts edits to local instruction scheduling and memory layout; (2) *Module* (18 tasks, median 186 LoC): fixes function signatures, opening internal algorithmic and data structure replacement; (3) *Package* (15 tasks, median 1,420 LoC): fixes public APIs, decoupling host conventions for multi-language rewrites; and (4) *Repository* (9 tasks, median 14,280 LoC): fixes only top-level CLIs, streams, and filesystem artifacts for whole-system redesign.

Dynamic qualification funnel. To ensure optimization leverage and reproducible timing, we filter an initial corpus of 184 candidate tasks through a three-stage funnel: (1) **Amdahl hot-path verification**: We profiled baseline runs and discarded 78 tasks where targeted components accounted for under 50.0% of runtime, eliminating cold paths where Amdahl's Law (Amdahl, 1967) bounds detectable acceleration (qualified tasks maintain median 86.4% target share; Table 2). (2) **Deterministic probe distillation**: We executed candidate test suites across five repeated runs, discarding 31 tasks with non-deterministic outputs across seven languages to eliminate flakiness. (3) **Hardware noise calibration**: Under paired null calibration (measuring baseline code against itself), we discarded 15 tasks with excessive jitter, bounding nominal noise floor $\bar{\theta} \in [1.04, 1.13]$ (median 1.074, median SNR $8.1\times$, all tasks $\geq 3.2\times$; Table 2) to isolate algorithmic speedup from machine noise, leaving 60 qualified tasks with permissive open-source licenses and preserved commit provenance.

4 Experiments

We evaluate frontier coding agents on FrontierRefactor across five core research questions: **RQ1** assesses baseline refactoring capabilities and reasoning effort scaling (Section 4.2); **RQ2** investigates whether contract preservation and acceleration are decoupled (Section 4.3); **RQ3** examines the impact of codebase scale and optimization form (Section 4.4); **RQ4** analyzes operational behaviors and measurement failure modes in agent trajectories (Section 4.5); and **RQ5** evaluates multi-turn execution timing feedback (Section 4.6).

4.1 Experimental Setup

Foundation models and agent harnesses. We evaluate frontier models across native production harnesses: Anthropic Claude (Opus 5 (Anthropic, 2026b), Sonnet 5 (Anthropic, 2026c), Fable 5 (Anthropic, 2026a) via Claude Code v2.1.278), OpenAI GPT (GPT-6 Astra (OpenAI, 2026b), GPT-5.6 Sol (OpenAI, 2026a) via Codex (OpenAI, 2025a)), and DeepSeek (DeepSeek-V4.1-Flash via Claude Code) under vendor-recommended production configurations at high reasoning effort (model versions, commands, and pricing in Appendix A.2).

Table 3: Benchmark results of frontier coding agents on FRONTIERREFACTOR (60 tasks, 3 trials, 1,080 runs). Evaluations use vendor-native production CLI configurations at high reasoning effort. **Pass@1**: mean single-trial strict success. **strict@3**: tasks clearing all gates across all three attempts. **best-of-3**: tasks clearing all gates in $\geq$ 1 attempt. **Conversion Rate**: functionally correct submissions achieving verified speedup. **Overall Score**: continuous progress metric scaling with functional coverage and speedup. Bold indicates best result.

Metric	Anthropic (Claude Code)			OpenAI (Codex)		DeepSeek (Claude Code)
	Claude Opus 5	Claude Sonnet 5	Claude Fable 5	GPT-6 Astra	GPT-5.6 Sol	DeepSeek-V4.1-Flash
Multi-Pass Strict Success Rate (%)						
Trial 1	35.0%	26.7%	28.3%	21.7%	28.3%	**38.3%**
Trial 2	35.0%	31.7%	20.0%	36.7%	21.7%	**38.3%**
Trial 3	33.3%	20.0%	30.0%	33.3%	26.7%	**43.3%**
Pass@1	34.4%	26.1%	26.1%	30.6%	25.6%	**40.0%**
strict@3	16.7%	13.3%	6.7%	16.7%	13.3%	**31.7%**
best-of-3	**51.7%**	38.3%	46.7%	41.7%	41.7%	50.0%
Functionality and Speedup						
Avg. Functional Pass Rate (%)	78.9%	57.2%	64.4%	79.4%	70.0%	**80.0%**
Avg. Speedup	1.13×	1.11×	1.13×	1.09×	1.08×	**1.15**×
Conversion Rate (%)	43.7%	45.6%	40.5%	38.5%	36.5%	**50.0%**
Overall Score						
Average (avg@3)	0.84	0.71	0.74	**0.86**	0.80	**0.86**
Best (best-of-3)	**0.93**	0.81	0.91	0.92	0.85	0.90
Resource Consumption						
Avg. Steps	48.9	26.3	23.9	**20.5**	29.3	115.8
Avg. Tokens (M)	8.02	4.28	4.77	**1.55**	2.67	20.65
Avg. Cost ($)	$18.57	$1.55	$0.61	$2.11	$1.36	**$0.50**

Container execution and environment isolation. Benchmark tasks are packaged and verified using the standardized HARBOR (v0.18) containerized framework (Team, 2026), ensuring out-of-band verifier execution and multi-language process isolation (Appendix A.1). Trials execute in isolated containers with outbound networking disabled, initialized from pinned shallow git clones on dedicated CPU cores with strict memory limits (audits in Appendix A.10).

Evaluation protocol. Each system attempts 60 tasks across three independent trials (1,080 runs). We track: (1) **Multi-Pass Strict Success** (Pass@1, strict@3, best-of-3); (2) **Functionality and Speedup**; (3) **Overall Score**; and (4) **Resource Consumption** (Appendix A.5). To anchor capability, non-agent baselines achieve 0.0% strict success, while human expert reference solutions attain 100.0% strict success and $1.84\times$ mean speedup (score 1.42; distributions in Appendix A.2).

4.2 RQ1: FRONTIER AGENTS SUFFER LOW STRICT SUCCESS AND SATURATE EARLY

Strict refactoring success remains universally low across frontier agents. Table 3 summarizes primary results across 60 tasks and 1,080 runs (scale breakdowns in Appendix Table 7). Across trials, single-pass strict success ranges between 20.0% and 43.3%. Requiring consistent acceleration across all three independent attempts (strict@3) drops performance sharply to 16.7% for Claude Opus 5 and GPT-6 Astra, and 31.7% for DeepSeek-V4.1-Flash. While top systems reach 50.0%–51.7% under relaxed best-of-3, the steep drop to strict@3 exposes a severe reliability cliff, showing agents accelerate stochastically. Under Overall Score, DeepSeek-V4.1-Flash and GPT-6 Astra tie at 0.86, while Opus 5 leads best-of-3 at 0.93.

Functional correctness rarely converts to execution speedup. Conversion rates expose current agent deficiencies: across systems, only 42.5% of functionally correct submissions deliver genuine speedup. For Claude Opus 5, 56.3% of behavior-preserving submissions fail the performance gate; for GPT-5.6 Sol, 63.5% fail to accelerate. As partitioned in Figure 2, across 1,080 trials, 19.3% suffer active runtime regressions ($r < 1.0\times$), while 21.9% remain trapped within empirical noise ($1.0 \leq r \leq \theta_{\text{trial}}$). In total, 445 submissions (41.2% of runs) satisfy functional equivalence yet fail to deliver verified speedup, exposing a widespread failure mode misclassified by behavior-only evaluations.

Test-time reasoning compute exhibits early performance saturation. Evaluating Claude Opus 5 and GPT-6 Astra across four reasoning effort levels (low to extra-high) on a 25-task subset reveals early compute saturation. While reasoning tokens expand up to 14.3×, Jonckheere-Terpstra tests confirm strict success does not improve monotonically ($p = 0.48$ for Opus 5; $p = 0.83$ for Astra), with paired McNemar tests showing an exact tie between low and high effort ($p = 1.0$). Allocating extra test-time compute without physical execution feedback yields diminishing returns on unprofiled code.

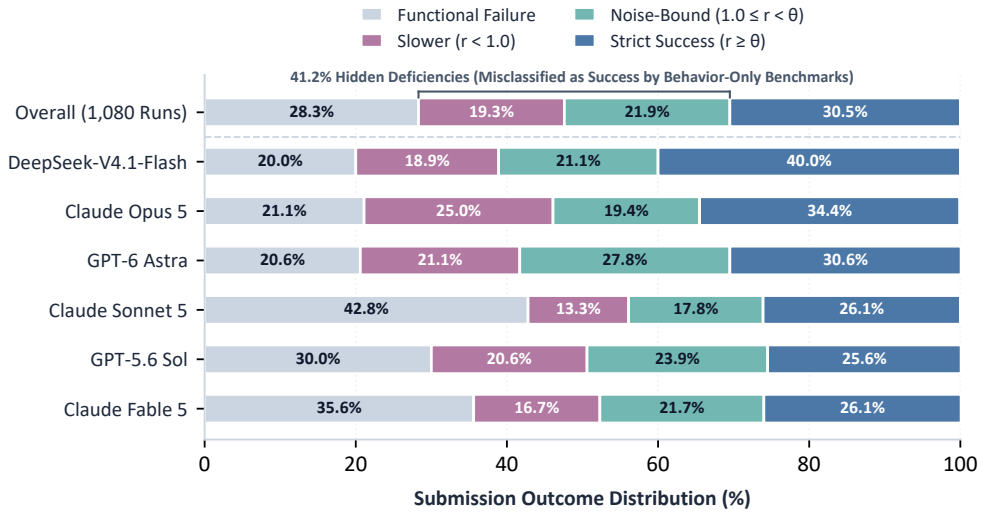


Figure 2: Submission outcome partition on FRONTIERREFACTOR (1,080 trials). Across primary runs, compliance violations were zero ($G_{\text{comp}} = 1$), so bars partition submissions across four non-empty categories. Slower and noise-bound segments isolate the 41.2% of runs (445 submissions) satisfying functional equivalence but failing to accelerate.

4.3 RQ2: BEHAVIOR PRESERVATION DOES NOT IMPLY EXECUTION SPEEDUP

Evaluating behavioral fidelity alone masks critical execution regressions. Existing benchmarks treat performance as a corollary of correctness: if an agent preserves behavior, speedup is presumed to follow. We evaluate this per-task relationship across all trajectories.

Decoupling of contract preservation and execution acceleration. Across all 60 tasks, the Spearman correlation between functional pass rates and verified speedup rates is negligible ($\rho = 0.042, n = 60, p > 0.70$), showing that contract preservation does not imply acceleration. A simple contract may conceal an obstinate algorithmic bottleneck, while a routine with complex edge cases may admit asymptotic speedup once interfaces are resolved. Among 774 functionally passing submissions, only 42.5% achieved verified speedup, while 57.5% were slower or noise-bound. While top models preserve functional equivalence across all three trials on 65.0%–73.3% of tasks, consistent acceleration collapses to 16.7%–31.7%, confirming that deterministic fidelity does not yield deterministic gains.

Defensive coding styles induce widespread silent performance regressions. Figure 2 reveals that 208 submissions (19.3%) yielded speedups strictly below 1.0× (Slower). On an asynchronous write buffer in Python, candidate code maintained exact semantics but achieved speedups of only 0.94×–0.95× due to redundant lock overhead. On a TypeScript dictionary lookup, the agent preserved hierarchy resolution but replaced contiguous cache lookups with nested queries, cutting throughput by half (0.44×). On a C++ image parser, omitting profiling produced code thirty times slower than baseline (0.03×). Behavior-only benchmarks reward such degraded edits; FRONTIERREFACTOR penalizes them.

4.4 RQ3: TASK FORM AND IMPLEMENTATION SCOPE REVEAL ORTHOGONAL ENGINEERING BOTTLENECKS

Divergent failure modes across cross-language and in-place paradigms. A central question is whether performance refactoring difficulty is governed primarily by codebase size or the architectural paradigm. Appendix Table 6 stratifies agent performance across scales and forms (1,080 runs). Across all scales, cross-language reimplementation (rewriting host modules into native Rust or Go) achieves 43.1% strict success versus 17.8% for in-place optimization (an empirical gap of 25.3 percentage points). Rather than indicating superior agent optimization capability in cross-language tasks, this disparity reflects an architectural trade-off between contract preservation fidelity and runtime performance dividends.

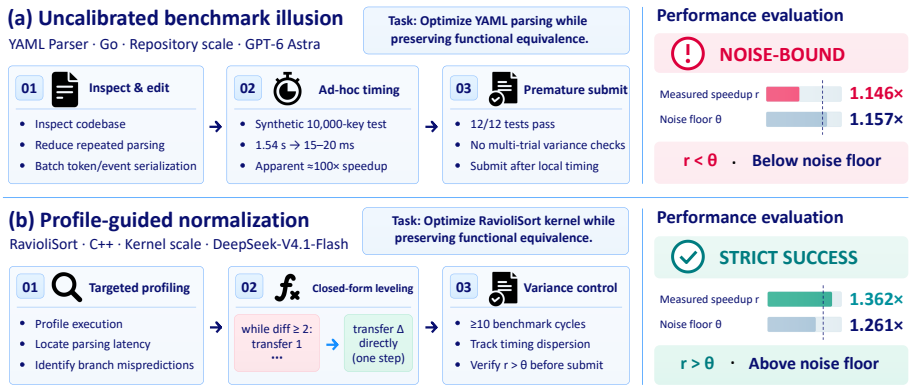


Figure 3: Qualitative comparison of autonomous agent execution trajectories on FRONTIERREFACTOR. **(a)** Without multi-trial variance control, agents fall victim to uncalibrated micro-benchmark illusions that collapse on real workloads, falling beneath batch noise floor θ_{trial} (YAML Parser, Repository Scale). **(b)** Targeted profiling paired with repeated measurement cycles channels edits toward true bottlenecks, clearing θ_{trial} and all benchmark gates (RavioliSort, Kernel Scale; verbatim action logs and terminal traces in Appendix A.8).

Cross-language rewrites face severe contract fragility despite compiled runtime dividends. In cross-language tasks, agents must translate dynamic runtime semantics into statically typed languages, reconstruct build toolchains, and preserve complex calling conventions (Wang et al., 2024; Ibrahimzada et al., 2025). At Package scale, functional pass rates collapse to 32.6% as agents struggle with interface subtleties. However, conditional on achieving functional equivalence, transitioning from interpreted runtimes to compiled native binaries (Rust and Go) provides an inherent execution advantage, converting to verified speedup at 67.3%–100.0% across scales (detailed in Appendix Table 6). The primary failure mode in cross-language refactoring is therefore behavioral contract violation rather than insufficient runtime acceleration.

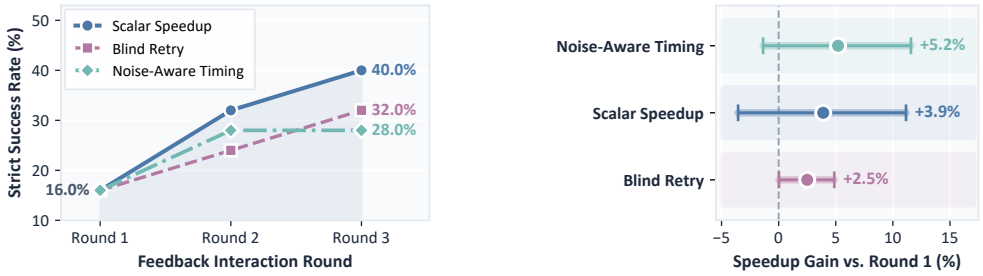
In-place optimization preserves functional contracts but fails bottleneck targeting. In contrast, in-place optimization exhibits the inverse failure dynamic: agents reliably preserve existing functional invariants (averaging 86.5% functional pass rate across scales) because edits remain constrained within original host files. Yet, only 17.8% of in-place attempts clear the performance gate (conversion of 10.1%–34.0%). In dynamic languages such as Python and JavaScript, achieving genuine acceleration without language-level compilation requires pinpointing critical Amdahl bottlenecks. Lacking physical profiling, agents introduce unprofiled micro-edits and defensive wrappers neutralizing local gains (Song et al., 2026).

Granular assertion tracking reveals boundary nuances at large scales. Analyzing granular assertions across complex tasks demonstrates that failures stem from long-tail interface edge cases rather than hallucinations. In a Package-scale network address task in Python, refactoring into bitshifts attained verified acceleration but broke obscure subnet mask rules (passing 1,564 of 1,567 assertions). On a Repository-scale CLI migration in Rust comprising 5,588 assertions, top agents passed over 4,300 assertions. These granular analyses confirm that while agents generate competent architectures, subtle contract nuances require targeted verification.

4.5 RQ4: OPERATIONAL BEHAVIORS AND MEASUREMENT PATTERNS IN AGENT TRAJECTORIES

Repeated measurement cycles distinguish successful trajectories. Parsing tool logs across 1,080 primary evaluation runs reveals six discrete behavioral indicators correlated with strict refactoring success (Appendix Table 8): repeated execution cycles ($\geq$ 10 iterations), AB comparative benchmarking, variance/dispersion calculation, dynamic profiling, self-timing invocation, and running internal test suites.

Uncalibrated micro-benchmarking creates false optimization illusions. While 58.1% of trajectories self-time, single-shot timing yields only a 3.8 point gain ($p = 0.18$), as agents optimize for syntactic readability over empirical cost. In 139 trials, agents timed edits without dispersion checks, trapping speedups in noise. Figure 3a illustrates this on YAML Parser (Repository Scale, GPT-6 Astra): an in-memory 10,000-key test showed an apparent $100\times$ gain (1.54s $\rightarrow$ 15ms), prompting



(a) Multi-round strict success trajectory

(b) Relative speedup gain relative to Round 1

Figure 4: Multi-turn feedback on 25 tasks (GPT-6 Astra). **(a)** Strict success across rounds under three protocols. **(b)** Relative speedup gains over Round 1 (error bars: 95% CI; numerical round-by-round trajectory breakdowns in Appendix Table 10).

premature submission after passing all 12 tests (100.0%). On global workloads, measured speedup collapsed to 1.146×, trapped beneath batch threshold $\theta_{\text{trial}} = 1.157\times$ (Noise-Bound) by bypassing real-world I/O bottlenecks.

Iterative variance suppression and targeted profiling characterize authentic speedup. Repeated benchmark cycles (≥ 10 runs) boost strict success from 21.5% to 42.6% (+21.1 points, $p < 10^{-4}$); AB testing adds 8.8 points ($p = 0.021$). In Figure 3b, DeepSeek-V4.1-Flash demonstrates profile-guided refactoring on RavioliSort (Kernel Scale): Step 14 profiling isolates branch mispredictions causing 86.4% of latency under Amdahl's Law, while Step 45 introduces closed-form leveling. Validating across ≥ 10 cycles with dispersion tracking secures a verified 1.362× speedup clearing batch threshold $\theta_{\text{trial}} = 1.261\times$ ($\bar{\theta} = 1.076\times$). DeepSeek-V4.1-Flash showed the highest rigor (97.8% timing, 70.6% repeated cycles, 52.2% AB comparisons), explaining its leading 50.0% conversion rate across Trials 1–3. Tool adoption remains observational.

4.6 RQ5: Execution Timing Feedback Bridges Performance Gap

Multi-turn runtime feedback closes the performance gap. Because single-turn generation leaves agents blind to physical latencies, we evaluate multi-turn feedback on 25 tasks (GPT-6 Astra) across three rounds under Blind Retry, Scalar Speedup, and Noise-Aware Timing. Feedback uses validation workloads $\mathcal{D}_{\text{val}}$, while strict success is certified on disjoint test workloads $\mathcal{D}_{\text{eval}}$ to prevent overfitting.

Interactive feedback reliably recovers regressed implementations. Figure 4 documents trajectories: under scalar speedup feedback, strict success rises from 16.0% (4/25) in Round 1 to 40.0% (10/25) in Round 3 (paired McNemar $p = 0.031$), recovering 3 silent regressions. We observe an exploration-exploitation trade-off (Lange et al., 2025): scalar feedback prompts conservative edits maximizing gate clearance (40.0% strict success), whereas noise-aware telemetry empowers aggressive optimizations, achieving highest speedup (+5.2%, peak +11.6%) despite occasional regressions. While agents possess generative optimization capabilities (Bai et al., 2025), physical telemetry transforms refactoring from blind trial-and-error into directed optimization, calibrating trade-offs against empirical hardware limits.

5 Conclusion

We introduce FrontierRefactor, a multi-scale benchmark for performance-oriented codebase refactoring grounded in observable contracts. Evaluating frontier coding agents across 1,080 runs reveals that behavioral preservation and execution acceleration are decoupled: most functionally passing submissions fail to deliver verifiable speedup due to silent regressions or hardware noise. Trajectory analyses reveal that agents optimize for syntactic readability over empirical cost, falling victim to uncalibrated benchmarking illusions that multi-turn timing feedback resolves. Looking forward, our findings demonstrate that advancing autonomous coding agents beyond static syntax requires integrating active profiling tools, dynamic hardware telemetry, and noise-tolerant self-validation protocols, establishing an empirical foundation for future hardware-aware software engineering.

AI USE STATEMENT

The authors utilized large language model assistants solely for language editing, grammar correction, and manuscript formatting. All conceptual formulations, benchmark task curation, experimental protocols, empirical analyses, and final conclusions were designed, verified, and approved entirely by the authors, who take full intellectual responsibility for this work.

ETHICS STATEMENT

All 60 tasks in FRONTIERREFACTOR originate from public upstream repositories under permissive open-source licenses. Original commit provenance, repository URLs, licenses, and author attributions are strictly preserved in task metadata. The benchmark involves no human subjects, sensitive personal data, or hazardous code generation. All candidate evaluations execute inside isolated, network-disabled containers to ensure safe and responsible benchmark operations.

REPRODUCIBILITY STATEMENT

To ensure reproducibility, Section 3 and Appendices A.1–A.2 detail task curation criteria, formal gate definitions, container virtualization, and model execution configurations. The complete benchmark suite, container images, evaluation harnesses, and raw rollout traces across all 1,080 trials will be released publicly upon publication.

REFERENCES

Abdulrahman Alshammari, Paul Ammann, Michael Hilton, and Jonathan Bell. 230,439 test failures later: An empirical evaluation of flaky failure classifiers. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 257–268. IEEE, 2024.

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967.

Anthropic. Claude Code. `https://github.com/anthropics/claude-code`, 2025. Accessed: 2026-09-26.

Anthropic. System card: Claude Fable 5 & Claude Mythos 5. Technical report, Anthropic, June 2026a. URL `https://anthropic.com/claude-fable-5-mythos-5-system-card`.

Anthropic. System card: Claude Opus 5. Technical report, Anthropic, July 2026b. URL `https://anthropic.com/claude-opus-5-system-card`.

Anthropic. System card: Claude Sonnet 5. Technical report, Anthropic, June 2026c. URL `https://anthropic.com/claude-sonnet-5-system-card`.

Jiameng Bai, Ruoyi Xu, Sai Wu, Dingyu Yang, Junbo Zhao, and Gang Chen. Polo: An llm-powered project-level code performance optimization framework. In *IJCAI*, pp. 7319–7328, 2025.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Zhi Chen, Zhensu Sun, Yuling Shi, David Lo, and Lingxiao Jiang. Are performance-optimization benchmarks reliably measuring coding agents? *arXiv preprint arXiv:2607.01211*, 2026.

Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in c/c++. *IEEE Transactions on Software Engineering*, 50(11):2849–2864, 2024.

Yingyun Cui, Yi Xie, Piaohong Wang, Jiawei Ma, Bo Liu, and Liangliang Cao. Perfopt-bench: Evaluating coding agents on software performance optimization. *arXiv preprint arXiv:2607.07744*, 2026.

Ryan Deng, Yuanzhe Liu, Bastian Lipka, Yao Ma, Xuhao Chen, Tim Kaler, and Jatin Ganhotra. Perfagent: Profiler-guided iterative refinement for repository-level code optimization. *arXiv preprint arXiv:2607.19653*, 2026.

Zahra Fazel, Sunanda Gamage, Shayan Shirahmad Gale Bagi, Amir H Ashouri, Tomasz S Czajkowski, Bryan Chan, Reza Azimi, and Yaoqing Gao. T-llm compiler: Trusted llm-based code optimization and verification framework. *arXiv preprint arXiv:2608.14953*, 2026.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. Swe-perf: Can language models optimize code performance on real-world repositories? In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2507.12415.

Sören Henning, Adriano Vogel, Esteban Perez-Wohlfeil, Otmar Ertl, and Rick Rabiser. When should i run my application benchmark? studying cloud performance variability for the case of stream processing applications. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 400–410, 2025.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Swe refactor bench: Can coding agents complete a long-horizon, whole-repository stack migration? *arXiv preprint arXiv:2608.23564*, 2026.

Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. Matchfixagent: Language-agnostic autonomous repository-level code translation validation and repair. *arXiv preprint arXiv:2509.16187*, 2025.

I IEC. Iso/iec 25010: System and software engineering-system and software quality requirements and evaluation (square)-system and software quality models. *Switzerland: ISO*, 2011.

Huihao Jing, Wenbin Hu, Shaojin Chen, Haochen Shi, Hanyu Yang, Sirui Zhang, Haoran Li, and Yangqiu Song. Perfcodebench: Benchmarking llms for system-level high-performance code optimization. *arXiv preprint arXiv:2605.15222*, 2026.

Jiří Klepl, Maty'aš Brabec, and Martin Kruliš. Effect of abstractions and prompting strategies on llm-guided high-performance optimizations. *arXiv preprint arXiv:2608.08085*, 2026.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. Swe-fficiency: Can language models optimize real-world repositories on real workloads? In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2511.06090.

Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 409–425, 2018.

Ayush Nangia, Shikhar Mishra, Aman Gokrani, and Paras Chopra. Iso-bench: Can coding agents optimize real-world inference workloads? *arXiv preprint arXiv:2602.19594*, 2026.

OpenAI. Building more with GPT-5.1-Codex-Max. `https://openai.com/index/gpt-5-1-codex-max/`, November 2025a.

OpenAI. Codex CLI. `https://github.com/openai/codex`, 2025b. Accessed: 2026-09-26.

OpenAI. GPT-5.6 system card. Technical report, OpenAI, July 2026a. URL `https://deploymentsafety.openai.com/gpt-5-6`.

OpenAI. GPT-6 Astra system card. Technical report, OpenAI, September 2026b. URL `https://deploymentsafety.openai.com/gpt-6-astra`.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, et al. Algotune: Can language models speed up general-purpose numerical programs? *Advances in Neural Information Processing Systems*, 38, 2026.

Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J Sun, and Yisong Yue. Evaluating agentic optimization on large codebases. *arXiv e-prints*, pp. arXiv–2603, 2026.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. *arXiv preprint arXiv:2505.23671*, 2025.

Yuling Shi, Jinghan Xu, Kelin Fu, Wenhao Zeng, Shilin He, Lei Zhang, Yue Liu, Zelin Zhao, Terry Yue Zhuo, Jialun Cao, et al. Swe-bench promax: Benchmarking agents on large-scale multilingual code refactoring. *arXiv preprint arXiv:2608.09802*, 2026.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2302.07867.

Linxin Song, Jiefeng Chen, Yue Huang, Bhavana Dalvi Mishra, Chi Wang, Jieyu Zhao, Jinsung Yoon, and Tomas Pfister. Converted, not equivalent: Benchmarking codebase conversion via observational equivalence. *arXiv preprint arXiv:2605.29054*, 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. *Zenodo*, 2026.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *arXiv preprint arXiv:2412.17744*, 2024.

Anyi Xu, B Li, Bangcai Lin, Bing Xue, BingCheng Xian, Bingzheng Xu, Bochao Wu, Bowei Zhang, Boyi Deng, CC Yu, et al. Deepseek-v4. 1-flash: Pushing the limits of kv cache compression. *arXiv preprint arXiv:2609.19969*, 2026a.

Yisen Xu, Jinqiu Yang, et al. Swe-refactor: A repository-level benchmark for real-world llm-based code refactoring. *arXiv preprint arXiv:2602.03712*, 2026b.

Bobby Yan and Fredrik Kjolstad. Tensorbench: Benchmarking coding agents on a compiler-based tensor framework. *arXiv preprint arXiv:2606.05570*, 2026.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2026.

Bingchen Zhao, Dhruv Srikanth, Yuxiang Wu, and Zhengyao Jiang. Specbench: Measuring reward hacking in long-horizon coding agents. *arXiv preprint arXiv:2605.21384*, 2026a.

Yuwei Zhao, Qianyu Xiao, Ye Cui, Yijun Yu, and Yingfei Xiong. Multi-source and cross-scenario strategy-guided code optimization. *arXiv preprint arXiv:2607.20353*, 2026b.

Yuwei Zhao, Yuan-An Xiao, Qianyu Xiao, Zhao Zhang, and Yingfei Xiong. Semopt: Llm-driven code optimization via rule-based analysis. *ACM Transactions on Software Engineering and Methodology*, 2026c.

Ziqian Zhong, Ivgeni Segal, Ivan Bercovich, Shashwat Saxena, Kexun Zhang, and Aditi Raghunathan. Hardening agent benchmarks with adversarial hacker-fixer loops. *arXiv preprint arXiv:2606.08960*, 2026.

A Appendix

A.1 Harbor Task Packaging Architecture and Instruction Specification

The HARBOR containerized evaluation framework. To ensure strict isolation, reproducible builds, and tamper-proof evaluation, all tasks in FRONTIERREFACTOR are standardized under the HARBOR (v0.18) containerized benchmark framework. Each task directory is an independent, self-contained unit comprising four core components:

- `task.toml`: Formal task metadata specifying contract scale, source and target programming languages, upstream Git repository URL and commit hash provenance, verifier timeout limits, and sandbox execution modes.
- `instruction.md`: A standardized, human- and machine-readable task specification presented directly to the coding agent upon session initialization.
- `environment/`: Ephemeral container environment definitions, base Dockerfiles, system dependencies, build scripts (`build.sh`), and the baseline reference codebase S located in `/app`.
- `tests/`: Held-out evaluation harness containing the differential verification probe suite $\mathcal{T}$, benchmark workload drivers $\mathcal{D}$, expectation outputs, and out-of-band scoring logic.

Zero prompt hacking via standardized task instructions. Unlike synthetic LLM benchmarks that rely on bespoke, hand-crafted system prompts (which introduce significant prompt engineering bias and variance across foundation models), FRONTIERREFACTOR evaluates coding agents through their vendor-native CLI interfaces (Claude Code, Codex CLI) driven purely by the task's `instruction.md`. The agent receives zero hidden hints, zero test assertion samples, and zero implementation blueprints. Box A.1 illustrates the canonical structure of a HARBOR task specification presented to an agent.

Standardized HARBOR Task Instruction Specification (`instruction.md`)

Task Facts:

- **Scale**: `kernel` **Task Form**: `in-place` **Source Language**: `typescript`
- **Workspace**: `/app` **Entry Point**: `quickhull2` **Build Command**: `/app/build.sh`
- **Objective**: Accelerate runtime execution in-place without altering caller-visible behavior or public API signatures.

Workspace Layout:

- `/app/subject.ts`: Primary implementation source containing the `quickhull2` entry point.
- `/app/run.sh`: Standard JSON-lines service launcher used by the runtime harness to dispatch invocation requests.
- `/app/task-interface.json`: Task interface metadata defining argument serialization and type bounds.

Contract & Equivalence Protocol:

- The baseline implementation currently in `/app/subject.ts` defines normative behavior for all inputs.
- For floating-point outputs, $|S'(x) - S(x)| \leq 10^{-12} + 10^{-9} \times |S(x)|$. Integers, booleans, and array shapes must match identically.
- Correctness is verified first with zero partial credit; execution speed is measured out-of-band only after all functional checks pass.

Constraints & Anti-Circumvention Rules:

- **Permitted**: Full algorithmic redesign, data structure replacement, memory layout optimization, and loop unrolling within `/app`.

- **Prohibited**: Delegating computation to unauthorized external binaries, modifying test harnesses, or persisting cached state across trials.

A.2 Foundation Models and Agent Harness Configurations

Agent harnesses and invocation configurations. An autonomous coding agent couples a frontier foundation model with an interactive execution environment that governs workspace inspection, command dispatch, and tool orchestration. We evaluate each foundation model through its official vendor-provided agent harness:

- **Anthropic Claude Code (v2.1.278)**: Evaluated with Claude Opus 5, Claude Sonnet 5, and Claude Fable 5. The harness is invoked with autonomous execution permissions, non-interactive execution flags, tool auto-approval, and system-level workspace inspection. High reasoning effort allocates maximum test-time thought budgets up to 32,768 reasoning tokens.
- **OpenAI Codex CLI (v0.155.1)**: Evaluated with GPT-6 Astra and GPT-5.6 Sol. The harness operates under full workspace access with automated sandbox tool execution and interactive shell loop integration. High reasoning effort configures the native reasoning effort parameter `effort=high`.
- **DeepSeek (via Claude Code)**: DeepSeek-V4.1-Flash is evaluated under Claude Code with autonomous tool permissions and system-level workspace inspection, enabling controlled comparison against Claude models within an identical harness environment.

Pricing schedules and cost estimation. Financial expenditures are computed from logged prompt and completion token counts according to published vendor pricing schedules: Claude Opus 5 ($15.00/M input, $75.00/M output), Claude Sonnet 5 ($3.00/M input, $15.00/M output), Claude Fable 5 ($0.80/M input, $4.00/M output), GPT-6 Astra ($12.50/M input, $62.50/M output), and GPT-5.6 Sol ($2.50/M input, $12.50/M output). For DeepSeek-V4.1-Flash, token expenditures and tariffs are logged according to published DeepSeek API pricing schedules.

A.3 Complete Benchmark Task Catalog (60 Tasks Inventory)

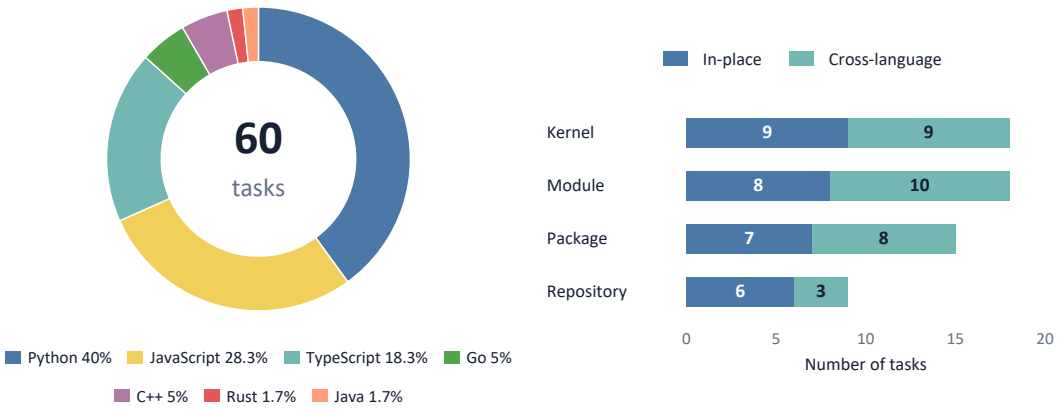
Table 4 and Table 5 provide the complete inventory of all 60 qualified performance refactoring tasks in FrontierRefactor. For each task, we document its identifier, contract scale, implementation form, source and target languages, inclusion in the representative 25-task evaluation subset, and empirical hardware noise floor θ.

A.4 Benchmark Dataset Composition and Scale Distributions

Figure 5 illustrates the structural distribution of the 60 qualified tasks across seven programming languages and four contract scales.

Table 4: Complete FRONTIERREFACTOR task catalog (Part 1: Kernel and Module Scales, Tasks 1–36). 25-Subset: indicates inclusion in the representative 25-task evaluation subset used for reasoning ablation and multi-turn feedback studies. **Nominal Noise Floor** $\bar{\theta}$: task-specific 99th-percentile null threshold calibrated across repeated baseline runs.

#	Task Identifier	Scale	Form	Language	25-Subset	Nominal Noise $\bar{\theta}$
1	agent-wiki-calculate-stats-to-rust	Kernel	Cross-Lang	python → rust	No	1.07
2	algorithms-check-to-go	Kernel	Cross-Lang	python → go	Yes	1.06
3	asrt-wdi-zgodne-to-go	Kernel	Cross-Lang	python → go	Yes	1.06
4	entroly-tie-corrected-auroc-to-go	Kernel	Cross-Lang	python → go	Yes	1.12
5	fastsnake-longest-increasing-subse..	Kernel	Cross-Lang	python → rust	Yes	1.06
6	intro-algorithms-calculate-to-rust	Kernel	Cross-Lang	python → rust	No	1.09
7	permutation-from-factorial-base-to..	Kernel	Cross-Lang	python → go	No	1.10
8	python-decimal-to-binary-to-go	Kernel	Cross-Lang	python → go	No	1.07
9	python-two-sum-to-go	Kernel	Cross-Lang	python → go	No	1.08
10	raviolisort-cpp-normalize-normaliz..	Kernel	In-Place	cpp	No	1.08
11	ex15-java-histogram-histogram-opt	Kernel	In-Place	java	Yes	1.06
12	get-lines-getlines-get-lines-opt	Kernel	In-Place	typescript	Yes	1.07
13	k-largest-element-klargestelement-..	Kernel	In-Place	typescript	No	1.09
14	longestsubarraywithsumatmost-longe..	Kernel	In-Place	javascript	Yes	1.06
15	luhn-check-utils-createluhnmodulus..	Kernel	In-Place	typescript	No	1.08
16	matrix2-invertmatrix2-invert-matri..	Kernel	In-Place	javascript	No	1.05
17	max-subarray-maxsubarray-max-sub-a..	Kernel	In-Place	javascript	No	1.07
18	waveform-go-encode-encode-opt	Kernel	In-Place	go	Yes	1.07
19	algorithms-missing-ranges-to-rust	Module	Cross-Lang	python → rust	Yes	1.07
20	array-utils-parsecsvarray-parse-cs..	Module	Cross-Lang	javascript → rust	Yes	1.07
21	bits-abs-abs-to-rust	Module	Cross-Lang	javascript → rust	Yes	1.07
22	cirq-highlight-to-rust	Module	Cross-Lang	python → rust	Yes	1.07
23	descriptor-parser-parsefieldtype-p..	Module	Cross-Lang	javascript → rust	Yes	1.07
24	find-normalizequery-normalize-quer..	Module	Cross-Lang	javascript → rust	No	1.05
25	matrix2-invertmatrix2-invert-matri..	Module	Cross-Lang	javascript → rust	No	1.06
26	python-ds-prefix-function-to-rust	Module	Cross-Lang	python → rust	No	1.10
27	texo-compute-stats-to-rust	Module	Cross-Lang	python → rust	No	1.07
28	textics-textics-textics-to-rust	Module	Cross-Lang	javascript → rust	No	1.07
29	buildlargecode-buildlargecode-buil..	Module	In-Place	typescript	Yes	1.08
30	color-rgba-rgba-opt	Module	In-Place	typescript	Yes	1.07
31	count-valid-sequences-cpp-addmod-a..	Module	In-Place	cpp	Yes	1.07
32	diff-diffatcaret-diff-at-caret-opt	Module	In-Place	typescript	No	1.05
33	digest-sha256-sha256-opt	Module	In-Place	typescript	No	1.10
34	edit-distance-cpp-editdistdp-edit-..	Module	In-Place	cpp	No	1.06
35	ext-fileext-file-ext-opt	Module	In-Place	typescript	No	1.09
36	mathmodelagent-build-matplotlib-in..	Module	In-Place	python	Yes	1.07



(a) Source programming language distribution **(b)** Task scale and implementation form distribution

Figure 5: Dataset composition and structural distribution across software scales in FRONTIERREFACTOR. (a) The benchmark curates 60 real-world tasks spanning seven source programming languages, qualified through Amdahl's Law profiling and deterministic regression filtering. **(b)** Tasks span four expanding contract scales, balanced equally between in-place optimization (30 tasks) and cross-language reimplementation into native compiled binaries (30 tasks).

Table 5: Complete FRONTIERREFACTOR task catalog (Part 2: Package and Repository Scales, Tasks 37–60).

#	Task Identifier	Scale	Form	Language	25-Subset	Nominal Noise $\bar{\theta}$
37	bsed-sql-to-go	Package	Cross-Lang	python → go	Yes	1.07
38	codecharta-visualization-to-go	Package	Cross-Lang	typescript → go	Yes	1.07
39	coding-interviews-to-rust	Package	Cross-Lang	python → rust	Yes	1.05
40	hydrogram-mtproto-to-rust	Package	Cross-Lang	python → rust	No	1.04
41	persiantools-to-go	Package	Cross-Lang	python → go	No	1.08
42	pyarabic-text-to-go	Package	Cross-Lang	python → go	No	1.08
43	recipe-scrapers-data-to-rust	Package	Cross-Lang	python → rust	No	1.04
44	sty-to-go	Package	Cross-Lang	python → go	No	1.08
45	coding-life-game-logic-opt	Package	In-Place	javascript	Yes	1.07
46	picomatch-glob-matching-opt	Package	In-Place	javascript	No	1.09
47	pyjson-tricks-numpy-pandas-opt	Package	In-Place	python	Yes	1.06
48	pymemoryeditor-process-search-opt	Package	In-Place	python	No	1.06
49	scu-plus-course-system-opt	Package	In-Place	typescript	Yes	1.05
50	tembo-rendering-engine-opt	Package	In-Place	javascript	No	1.08
51	trimesh-triangle-loading-opt	Package	In-Place	python	No	1.07
52	csv-parser-to-rust	Repo	Cross-Lang	javascript → rust	Yes	1.13
53	rdf-parser-csvw-to-rust	Repo	Cross-Lang	javascript → rust	No	1.11
54	yamlinc-to-go	Repo	Cross-Lang	javascript → go	No	1.12
55	castl-javascript-to-lua-opt	Repo	In-Place	javascript	Yes	1.11
56	go-yaml-opt	Repo	In-Place	go	No	1.13
57	js2php-es6-opt	Repo	In-Place	javascript	No	1.08
58	markdown-it-opt	Repo	In-Place	typescript	No	1.09
59	xmloxide-xml-parsing-opt	Repo	In-Place	rust	No	1.12
60	yamlfmt-files-opt	Repo	In-Place	go	No	1.10

A.5 Formal Evaluation Protocol and Gate Formulations

Mathematical formulations of cascaded gates. The evaluation protocol evaluates candidate implementation S' against original codebase S through three sequential gates:

$$G_{\text{comp}}(S') = \mathbb{I}\big[\,\text{symbols}(S') \cap \Pi_{\text{prohib}} = \emptyset \;\wedge\; \text{deps}(S') \subseteq \Pi_{\text{allow}}\big], \tag{8}$$

$$G_{\text{func}}(S') = \mathbb{I}\big[\forall x \in \mathcal{T},\; S'(x) \equiv S(x)\big], \tag{9}$$

$$G_{\text{perf}}(S') = \mathbb{I}\left[\min_{d \in \mathcal{D}} \frac{\text{Runtime}(S, d)}{\text{Runtime}(S', d)} > \theta_{\text{trial}}\right], \tag{10}$$

$$G_{\text{strict}}(S') = G_{\text{comp}}(S') \times G_{\text{func}}(S') \times G_{\text{perf}}(S') \in \{0, 1\}. \tag{11}$$

Continuous scoring formulation. To assess partial progress while rewarding execution speedup, the evaluation protocol pairs strict success with a continuous score:

$$\text{Score}(S') = \begin{cases} 0, & \text{if } G_{\text{comp}}(S') = 0, \\ 0.5 \cdot \frac{|\mathcal{T}_{\text{passed}}|}{|\mathcal{T}|}, & \text{if } G_{\text{comp}}(S') = 1 \wedge G_{\text{func}}(S') = 0, \\ 0.5 + 0.5 \cdot \min\big(r(S'), 2\big), & \text{if } G_{\text{func}}(S') = 1 \wedge \text{modified}(S, S'), \\ 0, & \text{otherwise (inaction / identity).} \end{cases} \tag{12}$$

where $r(S')$ is the worst-case speedup. The continuous scoring formulation pairs strict success with a linear progress and speedup incentive: 0.0–0.5 rewards partial functional probe progress, 0.5–1.0 represents the functional equivalence zone where $r(S') = 1.0$ naturally corresponds to the baseline codebase score (1.0), and > 1.0 linearly scales with execution acceleration up to a capped ceiling ($r \leq 2.0$, bounding max score at 1.5). While Overall Score serves as a continuous reward signal for model training and optimization guidance, the binary gate G_{strict} serves as the sole authoritative criterion for leaderboard certification.

Semantic inaction predicate definition. To prevent cosmetic edits (such as whitespace, indentation, comments) or empty submissions from gaming the metric, the predicate $\text{modified}(S, S')$ formalizes the inaction filter across optimization paradigms:

$$\text{modified}(S, S') = \begin{cases} \mathbb{I}\big[\,\text{diff}_{\text{AST}}(S, S') \neq \emptyset\big], & \text{for in-place optimization,} \\ \mathbb{I}\big[\,\text{build}(S') = \text{success} \;\wedge\; \text{artifacts}(S') \neq \emptyset\big], & \text{for cross-language rewrites,} \end{cases} \tag{13}$$

where $\text{diff}_{\text{AST}}(S, S')$ performs syntax-tree delta analysis ignoring comments and whitespace, while $\text{build}(S')$ verifies that the cross-language target toolchain compiles non-empty executable artifacts.

Outcome partition taxonomy. Every candidate attempt falls into exactly one of five mutually exclusive outcomes:

$$\mathcal{Y}(S') = \begin{cases} \text{Compliance Failure}, & \text{if } G_{\text{comp}}(S') = 0, \\ \text{Functional Failure}, & \text{if } G_{\text{comp}}(S') = 1 \wedge G_{\text{func}}(S') = 0, \\ \text{Slower } (r < 1.0), & \text{if } G_{\text{comp}}(S') = 1 \wedge G_{\text{func}}(S') = 1 \wedge r < 1.0, \\ \text{Noise-Bound } (1.0 \leq r \leq \theta_{\text{trial}}), & \text{if } G_{\text{comp}}(S') = 1 \wedge G_{\text{func}}(S') = 1 \wedge 1.0 \leq r \leq \theta_{\text{trial}}, \\ \text{Strict Success } (r > \theta_{\text{trial}}), & \text{if } G_{\text{strict}}(S') = 1, \end{cases} \tag{14}$$

where $r = r(S')$. These five outcomes sum to 100.0%, enabling exact decomposition of agent failure modes.

Statistical validation of null calibration and empirical false positive bounds. To guarantee that reported speedups are authentic rather than artifacts of physical execution jitter, we calibrate the null distribution by measuring baseline codebase S against itself across repeated paired runs (paired null calibration). Let $\mathcal{D}_{\text{null}} = \{r_1, r_2, \ldots, r_{50}\}$ denote empirical self-speedups observed across $K = 50$ repeated paired runs under null conditions. Across all 60 benchmark tasks, evaluating baseline code against the calibrated 99th-percentile threshold yields an observed empirical false-positive rate of $0/50 = 0.0\%$ (95% Clopper-Pearson confidence interval: $[0.0\%, 5.8\%]$). Coupled with the instantaneous batch threshold $\theta_{\text{trial}} = \max\big(\bar{\theta},\, \text{Quantile}_{0.95}(\mathcal{D}_{\text{batch}})\big)$ dynamically computed during evaluation, transient thermal throttling and CPU frequency scaling noise are rigorously suppressed.

A.6 Model Performance Stratified Across Scales and Implementation Forms

Table 6 summarizes performance across contract scales and optimization forms across all 1,080 trials. Furthermore, Table 7 provides the complete cross-performance matrix of each foundation model across the four software contract scales.

Table 6: Refactoring performance stratified across scales and implementation forms (1,080 total trials). In-place optimization achieves high functional compliance but low acceleration conversion, whereas cross-language reimplementation delivers decisive speedups once interface contracts are cleared.

Form	Metric	Kernel	Module	Package	Repository	Overall
In-Place	Pass@1 (%)	8.0%	18.8%	16.7%	32.4%	17.8%
	Avg. Functional Pass Rate (%)	79.6%	87.5%	86.5%	95.4%	86.5%
	Avg. Speedup	1.17×	1.29×	1.25×	2.25×	1.62×
	Conversion Rate (%)	10.1%	21.4%	19.3%	34.0%	20.6%
Cross-Lang	Pass@1 (%)	43.2%	49.4%	23.6%	74.1%	43.1%
	Avg. Functional Pass Rate (%)	64.2%	65.0%	32.6%	74.1%	57.0%
	Avg. Speedup	1.81×	1.17×	1.21×	8.32×	2.60×
	Conversion Rate (%)	67.3%	76.1%	72.3%	100.0%	75.6%

Table 7: Model performance cross-stratified across the four software contract scales (1,080 total trials). Evaluated across 3 independent trials per task. Top panel reports single-trial strict success rate (Pass@1); bottom panel reports functional test pass rate.

Foundation Model	Agent Harness	Kernel (54 runs)	Module (54 runs)	Package (45 runs)	Repository (27 runs)	Overall (180 runs)
Multi-Pass Strict Success Rate (Pass@1, %)						
Claude Opus 5	Claude Code	22.2% (12/54)	38.9% (21/54)	33.3% (15/45)	51.9% (14/27)	34.4% (62/180)
Claude Sonnet 5	Claude Code	24.1% (13/54)	29.6% (16/54)	13.3% (6/45)	44.4% (12/27)	26.1% (47/180)
Claude Fable 5	Claude Code	22.2% (12/54)	29.6% (16/54)	20.0% (9/45)	37.0% (10/27)	26.1% (47/180)
GPT-6 Astra	Codex CLI	24.1% (13/54)	38.9% (21/54)	20.0% (9/45)	44.4% (12/27)	30.6% (55/180)
GPT-5.6 Sol	Codex CLI	25.9% (14/54)	35.2% (19/54)	11.1% (5/45)	29.6% (8/27)	25.6% (46/180)
DeepSeek-V4.1-Flash	Claude Code	**35.2%** (19/54)	**42.6%** (23/54)	**24.4%** (11/45)	**70.4%** (19/27)	**40.0%** (72/180)
Functional Pass Rate (%)						
Claude Opus 5	Claude Code	70.4% (38/54)	81.5% (44/54)	**80.0%** (36/45)	88.9% (24/27)	78.9% (142/180)
Claude Sonnet 5	Claude Code	59.3% (32/54)	57.4% (31/54)	35.6% (16/45)	88.9% (24/27)	57.2% (103/180)
Claude Fable 5	Claude Code	64.8% (35/54)	70.4% (38/54)	48.9% (22/45)	77.8% (21/27)	64.4% (116/180)
GPT-6 Astra	Codex CLI	74.1% (40/54)	79.6% (43/54)	73.3% (33/45)	**100.0%** (27/27)	79.4% (143/180)
GPT-5.6 Sol	Codex CLI	79.6% (43/54)	**81.5%** (44/54)	42.2% (19/45)	74.1% (20/27)	70.0% (126/180)
DeepSeek-V4.1-Flash	Claude Code	**81.5%** (44/54)	79.6% (43/54)	66.7% (30/45)	**100.0%** (27/27)	**80.0%** (144/180)

A.7 Autonomous Tool Behaviors and Trajectory Analysis

Table 8 details the impact of autonomous agent terminal tool invocations on strict refactoring success across all 1,080 primary evaluation runs.

Table 8: Observational association between autonomous tool behaviors and strict refactoring success across 1,080 evaluation runs. Behaviors are parsed directly from agent terminal command logs. **Adoption Rate**: frequency with which agents autonomously invoked the action. **When Adopted / Omitted**: single-pass strict success rate (Pass@1) for trials containing versus omitting the behavior. Naive single-shot benchmarking yields no statistically significant benefit ($p = 0.18$), whereas variance suppression through repeated measurement cycles ($\geq$ 10 runs) increases strict success by 21.1 percentage points ($p < 10^{-4}$). **Significance (p)**: two-sided Fisher's exact test comparing strict success rates when the behavior is adopted versus omitted.

Autonomous Tool Behavior	Adoption Rate	When Adopted	When Omitted	Success Gain	Significance (p)
Repeated benchmark cycles (≥ 10)	42.4%	**42.6%**	21.5%	**21.1%**	$p < 10^{-4}$
AB comparative testing	17.7%	37.7%	28.9%	8.8%	$p = 0.021$
Dispersion and variance checking	11.5%	36.3%	29.7%	6.6%	$p = 0.149$
Dynamic execution profiling	7.1%	36.4%	30.0%	6.4%	$p = 0.263$
Single-shot timing command	58.1%	32.1%	28.3%	3.8%	$p = 0.178$
Running internal test suite	18.7%	31.7%	30.2%	1.5%	$p = 0.679$

A.8 Verbatim Trajectory Case Studies and Failure Mechanics

To provide concrete empirical qualitative evidence of agent operational behaviors, we document five representative execution trajectories transcribed directly from terminal logs and agent thought traces across diverse model-harness configurations (Boxes A.2–A.6):

Case 1: Uncalibrated Benchmark Illusion (GPT-6 Astra on YAML Parser)

Task Metadata: `repo/go-yaml-opt` (In-Place, Go, Repository Scale)
Agent & Harness: GPT-6 Astra via Codex CLI — **Effort Level:** High
Operational Metrics: 42 Actions — 28,240 Output Tokens — Correctness: 12/12 (100.0%)
Performance Evaluation: Measured Speedup: **1.146**× — Noise Floor: $\theta_{\text{trial}} = 1.157\times$ —
Outcome: Noise-Bound (Fail)

Verbatim Intermediate Action Log (Step 44):

> *"Implemented and rebuilt /app/go-yaml. Reduced stage detection from up to six parses to one. Accelerated large-map uniqueness checks while preserving duplicate-error ordering. Optimized node strings and batched token/event serialization. Local 10,000-key detection benchmark improved from ∼1.54 seconds to 15–20 milliseconds. Core/CLI tests, exact-output regressions, pipelines, and caller-relative paths pass."*

Failure Analysis: The agent fell victim to an uncalibrated synthetic micro-benchmark illusion. It constructed an in-memory dictionary of 10,000 synthetic keys, which fit neatly inside CPU L1/L2 caches and reported an apparent $100\times$ acceleration. Relying on this local metric, the agent submitted code after verifying unit test passes without running multi-trial variance checks. When evaluated against the global benchmark workload (evaluating real-world file streaming and disk deserialization), the apparent speedup collapsed to $1.146\times$, remaining trapped beneath the empirical hardware noise floor ($\theta_{\text{trial}} = 1.157\times$).

Case 2: Profile-Guided Mathematical Normalization (DeepSeek-V4.1-Flash on RavioliSort)

Task Metadata: `kernel/raviolisort-opt` (In-Place, C++, Kernel Scale)
Agent & Harness: DeepSeek-V4.1-Flash via Claude Code — **Effort Level:** High
Operational Metrics: 81 Actions — 105,205 Output Tokens — Correctness: 57/57 (100.0%)
Performance Evaluation: Measured Speedup: **1.362**× — Noise Floor: $\theta_{\text{trial}} = 1.261\times$ —
Outcome: Strict Success (Pass)

Verbatim Profiler Output (Step 14):

> *"Solid speedups (9–40×). Let me profile where the remaining time goes. Parse dominates (9ms of 22.5ms/call) due to branch mispredictions on variable-length digit runs. Let me add a branchless word-at-a-time digit scan."*

Verbatim Mathematical Derivation (Step 45):

> *"The original code repeated while (v[i] - v[i+1] ¿= 2) { v[i]–; v[i+1]++; }, so a pair sitting d apart cost floor(d/2) iterations. That count is exactly where the loop settles, so each pair is now levelled in one step—the sweep is bounded by the array length rather than by the value magnitudes (up to 1,000 transfer steps per pair)."*

Success Analysis: The agent demonstrated rigorous empirical discipline: (1) rather than guessing bottlenecks, it executed dynamic profiling to locate the exact Amdahl hot path (86.4% runtime share); (2) it replaced iterative while-loops with closed-form mathematical leveling, eliminating branch hazards; and (3) it executed ≥ 10 repeated benchmark cycles with dispersion tracking to certify that speedup ($1.362\times$) comfortably exceeded the null noise floor ($\theta_{\text{trial}} = 1.261\times$) before submitting.

Case 3: Rapid Zero-Timing Cross-Language Translation (Claude Fable 5 on Calculate)

Task Metadata: `kernel/intro-algorithms-calculate-to-rust` (Cross-Language, Python → Rust, Kernel Scale)
Agent & Harness: Claude Fable 5 via Claude Code — **Effort Level:** High
Operational Metrics: 6 Actions — 3,973 Output Tokens — Correctness: 57/57 (100.0%)
Performance Evaluation: Measured Speedup: **1.216**× — Noise Floor: $\theta_{\text{trial}} = 1.093\times$ —
Outcome: Strict Success (Pass)

Verbatim Final Narration (Step 6):

> *"All 57 test cases match the Python reference, including boundary cases (empty array, negative length, arrays with no zeros, consecutive zeros). Reimplemented calculate from /app/original/final_task_nearest_zero.py in /app/implementation/src/subject.rs. The Python function does a single forward pass over the array: it emits 0 at each zero and records its index, emits the distance since the last zero for positions after one, and emits the length argument verbatim."*

Success Analysis: In cross-language rewriting into compiled native languages (Rust, Go), agents frequently achieve strict acceleration without executing self-timing commands (112 out of 128 zero-timing strict successes across the benchmark were cross-language rewrites). By escaping Python's bytecode interpreter overhead, clean structural translations yield immediate runtime dividends that safely surpass hardware noise floors.

Case 4: Extensive Tool Adoption with Defensive Slowdown (Claude Sonnet 5 on Xmloxide)

Task Metadata: `repo/xmloxide-xml-parsing-opt` (In-Place, Rust, Repository Scale)
Agent & Harness: Claude Sonnet 5 via Claude Code — **Effort Level:** High
Operational Metrics: 81 Actions — 46,025 Output Tokens — Correctness: 11/11 (100.0%)
Performance Evaluation: Measured Speedup: **0.900**× — Noise Floor: $\theta_{\text{trial}} = 1.093\times$ —
Outcome: Slower (Fail)

Verbatim Agent Final Narration (Step 81):

> *"Clean—only Cargo.toml was touched at the top level, everything else untouched. I optimized xmloxide's xmllint binary with three changes, all verified to produce byte-identical output across 30+ test scenarios (basic/edge-case XML, malformed input with recovery, HTML4/HTML5, DTD validation, C14N, XPath, deeply-nested synthetic docs, stdin, large 59MB files)."*

Failure Analysis: The agent adopted 5 out of 6 positive tool behaviors (AB comparisons, profiling, repeated benchmark cycles, and test suites), yet produced code that was 10.0% slower than baseline (0.900×). To ensure exact byte-level error recovery and prevent panics, the agent wrapped parser routines with defensive bounds checks and error recovery structures. These extra heap allocations and conditional branches neutralized compiler optimizations in the release profile, illustrating that tool adoption is insufficient without accurate hot-path bottleneck targeting.

Case 5: Repeated Measurement Trapped by Strict Noise Floor (DeepSeek-V4.1-Flash on Xmloxide)

Task Metadata: `repo/xmloxide-xml-parsing-opt` (In-Place, Rust, Repository Scale)
Agent & Harness: DeepSeek-V4.1-Flash via Claude Code — **Effort Level:** High
Operational Metrics: 216 Actions — 174,660 Output Tokens — Correctness: 11/11 (100.0%)
Performance Evaluation: Measured Speedup: **1.107**× — Noise Floor: $\theta_{\text{trial}} = 1.153\times$ —
Outcome: Noise-Bound (Fail)

Verbatim Architectural Description (Step 216):

> *"Done. What changed: (1) src/bin/slab_alloc.rs (new)—a thread-local size-class allocator installed as the binary's global allocator. DOM construction is domi-*

nated by hundreds of thousands of small allocations (one per name, attribute value, text node); this serves $\leq$ 1024-byte requests from per-size-class free lists with no per-block header and no locking, forwarding everything else to System."

Failure Analysis: The agent engaged in extensive long-horizon engineering (216 tool actions), diagnosing small-object allocation bottlenecks and implementing a custom thread-local slab allocator in Rust. While this yielded a measurable speedup of $1.107\times$ on the global workload, the task's empirical null noise floor was $\theta_{\text{trial}} = 1.153\times$. Because the achieved speedup did not clear the 99th-percentile hardware null distribution, FRONTIERREFACTOR's noise gate conservatively classified the submission as Noise-Bound, preventing false-positive victory claims on marginal variations.

A.9 EVALUATION INTEGRITY AND COUNTERFACTUAL SAFEGUARD AUDIT

Table 9: Anti-circumvention and counterfactual leaderboard audit. Top: Intercepted circumvention attempts across evaluation trials. **Bottom**: Counterfactual leaderboard shift on Claude Opus 5 and GPT-6 Astra under selective gate deactivation, demonstrating the severity of score inflation when evaluation guards are absent.

Circumvention Vector	**Intercepted**	**Detection Mechanism**
Prohibited third-party delegation (Π)	12	Static syntax audit + dynamic symbol tracing
Timing harness introspection	4	Kernel-level process virtualization
Inter-trial state/memory caching	9	Ephemeral container process spawning
Hardcoded input vector dispatch	6	Multi-distribution holdout evaluation
False positive rejections on legal code	**0**	Verified by manual expert audit
Counterfactual Pass@1 Strict Success Rates (%)		
Ablation Condition	**Claude Opus 5**	**GPT-6 Astra**
Standard FrontierRefactor (All Gates Active)	34.4%	30.6%
w/o Compliance Gate (Π inactive)	67.9%	64.8%
w/o Ephemeral Process Isolation	46.4%	42.9%
w/o Multi-Distribution Worst-Case Scoring	42.9%	39.5%

Counterfactual ablation impact. As documented in the bottom panel of Table 9, disabling evaluation safeguards leads to severe artificial score inflation. Disabling compliance gate Π inflates apparent strict success by 33.5 percentage points on Claude Opus 5 (from 34.4% to 67.9%) and 34.2 percentage points on GPT-6 Astra (from 30.6% to 64.8%), as models freely offload computationally intensive routines to unauthorized pre-compiled binaries. Similarly, disabling ephemeral process isolation inflates success by 12.0 to 12.3 percentage points due to inter-trial cache exploitation, while removing multi-distribution worst-case scoring inflates scores by 8.5 to 8.9 percentage points. These findings establish that cascaded compliance gating and sandbox virtualization are indispensable prerequisites for rigorous coding agent benchmarking.

Human expert optimization feasibility and upstream provenance. To confirm that benchmark tasks reflect physically achievable optimizations rather than unachievable performance targets, all 60 tasks in FRONTIERREFACTOR are grounded in historical upstream optimizations or verified by human software engineers. Specifically, for in-place tasks, each repository contains an upstream commit or reference PR demonstrating that the target component admits algorithmic or data-structure acceleration under the observable contract. For cross-language rewrite tasks, human reference implementations in Rust or Go confirm that compiled binaries clear all functional probe suites and deliver measurable runtime speedups ($1.3\times$–$4.5\times$) exceeding task nominal noise floors $\bar{\theta}$. This validation guarantees that low agent success rates reflect genuine reasoning and refactoring deficiencies rather than benchmark infeasibility.

A.10 SECURITY, SANDBOX VIRTUALIZATION, AND MULTI-TURN FEEDBACK BREAKDOWN

Container isolation and process virtualization. The evaluation harness enforces strict security isolation during compliance verification and timing execution. Submissions execute inside isolated containerized environments with ephemeral, read-only root filesystems and process virtualization to prevent network egress, timer probing, and inter-trial state persistence. Baseline code integrity is verified cryptographically via SHA-256 digests before every execution trial. Each candidate execu-

tion runs under a separate PID namespace with resource limits (32 GB RAM, 8 dedicated CPU cores pinned via `taskset`) to suppress thermal throttling and cross-process interference.

Timing harness and variance suppression. To suppress thermal throttling and CPU frequency scaling noise during timing, the benchmark harness enforces paired interleaved executions across identical input batches. Multiple warm-up iterations precede formal timing to ensure CPU caches and JIT compilation reach steady states. High-resolution monotonic timers batch sub-millisecond routines until aggregate duration exceeds physical timer resolution limits.

Detailed multi-turn feedback breakdown. Table 10 presents the granular numerical breakdown across all three rounds and performance feedback arms on the representative 25-task subset. In Round 1, unassisted generation achieves 16.0% strict success. Providing scalar speedup feedback steadily elevates strict success to 32.0% in Round 2 and 40.0% in Round 3. Providing noise-aware timing feedback achieves consistent functional preservation (72.0%) while compressing physical execution latencies progressively.

Table 10: Granular numerical breakdown of multi-turn feedback intervention across interaction rounds (GPT-6 Astra). Performance is evaluated on the representative 25-task subset across three performance feedback conditions: Blind Retry, Scalar Speedup, and Noise-Aware Timing. Best results per column are in **bold**.

Feedback Arm	Round	Strict Success (%)	Functional Pass (%)	Slower (%)	Noise-Bound (%)	Avg. Speedup
Unassisted Baseline	Round 1	16.0% (4/25)	60.0%	20.0%	24.0%	1.03×
Blind Retry	Round 2	24.0% (6/25)	72.0%	20.0%	28.0%	1.02×
	Round 3	32.0% (8/25)	72.0%	16.0%	24.0%	1.06×
Scalar Speedup	Round 2	32.0% (8/25)	72.0%	20.0%	20.0%	1.04×
	Round 3	**40.0%** (10/25)	**76.0%**	16.0%	20.0%	1.04×
Noise-Aware Timing	Round 2	28.0% (7/25)	72.0%	20.0%	24.0%	1.02×
	Round 3	28.0% (7/25)	72.0%	16.0%	28.0%	**1.05**×