SWE-CI: EVALUATING AGENT CAPABILITIES IN MAINTAINING CODEBASES VIA CONTINUOUS INTEGRATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-powered agents have demonstrated strong capabilities in automating software engineering tasks such as bug fixing. However, in the real world, software engineering is an inherently iterative process driven by evolving requirements and long-term feature development. Static, one-shot evaluation paradigms fail to capture this dynamic. To bridge this gap, we propose **SWE-CI**, a novel repository-level benchmark grounded in the Continuous Integration loop, designed to shift the evaluation paradigm for coding agents from static, short-term *functional correctness* toward dynamic, long-term *regression-aware code evolution*. The key insight is simple: **A testable aspect of maintainability can be revealed by tracking how functional correctness changes across successive code modifications**. The benchmark comprises 100 tasks, each derived from a real-world code repository with a development history spanning an average of 233 days and 71 consecutive commits. SWE-CI requires agents to systematically resolve these tasks through multiple rounds of analysis and coding iterations. Extensive experiments shed light on *regression*, a problem systematically overlooked by existing benchmarks: an acceptable final deliverable does not imply an acceptable development process.

1 INTRODUCTION

Automating software engineering has long been a central objective in the field of artificial intelligence. In recent years, breakthroughs in LLMs have lent substantial momentum to this pursuit. From code completion and test generation to end-to-end program repair, LLM-driven agents have demonstrated capabilities rivaling those of human developers across multiple tasks. Coding benchmarks have served as a crucial catalyst in this progress: not only providing rigorous performance feedback, but also charting the course for future improvement.

Early coding benchmarks focused on function- and file-level tasks, such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) and LiveCodeBench (Jain et al., 2024). As model capabilities and agentic frameworks advanced, the evaluation scope expanded to the repository level, represented by SWE-bench (Jimenez et al., 2023). More recently, benchmarks such as Terminal-bench (Merrill et al., 2026) and τ-bench (Yao et al., 2024) further extended the evaluation to multi-turn tool-use scenarios. Together, these efforts have established an increasingly comprehensive evaluation ecosystem for code intelligence.

Despite the breadth and depth of this ecosystem, these benchmarks share a common blind spot: they evaluate agents solely on the functional correctness of their final deliverables, overlooking the fact that real-world software engineering is an inherently long-term and dynamic process. Lehman's Laws reveal that software quality inherently degrades as maintenance progresses (Lehman, 2005), while classic literature has established that maintenance activities account for 60% to 80% of total software lifecycle costs (Brooks Jr, 1995). Therefore, there is an urgent need for benchmarks that go beyond functional correctness to effectively measure the ability of models to maintain code over time.

The long-standing absence of maintainability evaluation may stem from the prevalence of snapshot-style protocols: from HumanEval to SWE-bench, agents are presented with a single requirement and expected to produce a one-shot solution. Under this paradigm, an agent that hard-codes a brittle fix

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.

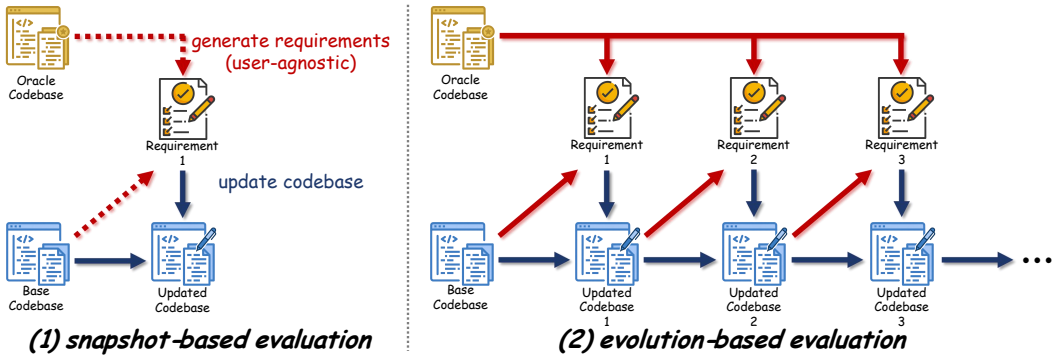


Figure 1: SWE-CI employs an evolution-based evaluation. The red and blue arrows represent the actions of functions require and code, respectively. Dashed lines indicate processes that are unknown to the user.

and one that writes clean, extensible code may both pass the same test suite — their difference in maintainability is simply invisible. It becomes visible only when the codebase evolves over time, where the consequences of past decisions accumulate across successive changes, a phenomenon widely known as *technical debt* (Cunningham, 1992). This yields a critical insight: **Maintainability is more faithfully revealed through sustained, successive code changes.**

Building upon this insight, we propose **SWE-CI** (**S**oft**W**are **E**ngineering – **C**ontinuous **I**ntegration), a novel benchmark designed to evaluate agents' ability to maintain code across long-term evolution. SWE-CI consists of 100 tasks, each comprising a *base commit* and a *target commit* from a real-world repository, spanning an average of 233 days and 71 consecutive commits of authentic evolutionary history. SWE-CI employs an Architect–Programmer dual-agent evaluation protocol: starting from the base commit, the agents execute a CI-loop that iteratively generates requirements, modifies source code, and runs tests, with the ultimate objective of passing all tests associated with the target commit. To quantify performance during this evolution process, SWE-CI introduces **EvoScore**, which summarizes functional correctness across successive code modifications.

We conduct extensive experiments across 31 models from 9 providers. Results reveal that functionally correct final outputs may conceal frequent regressions throughout the development process — a pattern that would be unacceptable in real-world software engineering.

2 Measuring the Agent's Ability to Maintain Codebase

2.1 Task Formalization

Let t denote a unit test and $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$ the set of all tests we are interested in. Let $\mathcal{C}$ denote the space of codebases and $\mathcal{R}$ the space of requirements. We further define two functions:

- $\mathsf{require}_{\mathcal{T}} : \mathcal{C} \times \mathcal{C} \to \mathcal{R}$, which identifies the functional gap between two codebases with respect to $\mathcal{T}$, and produces a requirement document accordingly;
- $\mathsf{code}_{\mathcal{T}} : \mathcal{R} \times \mathcal{C} \to \mathcal{C}$, which modifies a codebase according to a given requirement and returns an updated codebase.

Under this formalization, many prevailing coding benchmarks (Chen et al., 2021; Austin et al., 2021; Jain et al., 2024; Jimenez et al., 2023; Merrill et al., 2026; Yao et al., 2024) follow the *snapshot-based* evaluation paradigm illustrated in Figure 1. In this paradigm, the requirement depends only on the base codebase c_0 and the oracle codebase c_*, i.e., $r \equiv \mathsf{require}_{\mathcal{T}}(c_0, c_*)$. However, in this paper, we introduce an *evolution-based* evaluation paradigm. The requirement in this paradigm is derived from the functional gap between the current codebase and the oracle: $r_i = \mathsf{require}_{\mathcal{T}}(c_i, c_*)$, and the codebase is updated accordingly: $c_{i+1} = \mathsf{code}_{\mathcal{T}}(c_i, r_i)$. The agent's performance is then assessed over the entire trajectory $(c_1, \ldots, c_N)$, rather than the final output c_N alone.

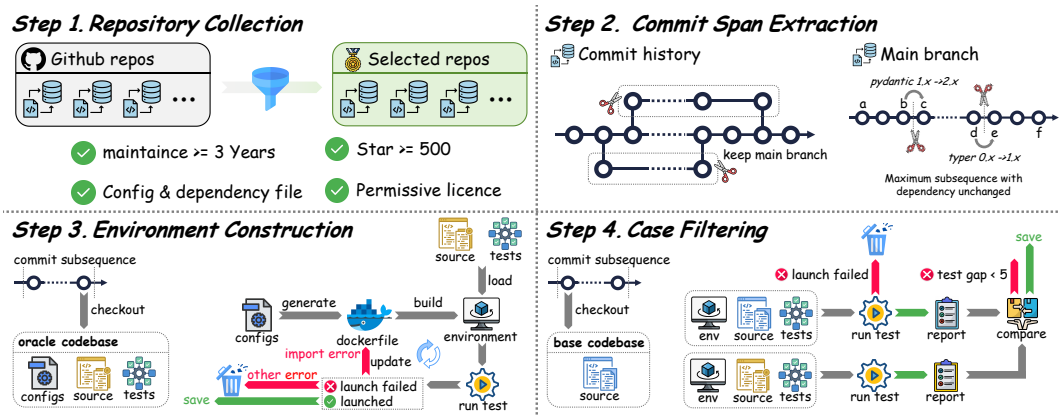


Figure 2: Data curation pipeline of SWE-CI.

2.2 Normalized Change

We first introduce *normalized change* to measure the functional change of some codebase c in the trajectory relative to the base codebase c_0. Let $n(c)$ be the number of test cases that c passes:

$$n(c) = \sum_{t \in \mathcal{T}} \mathbb{I}(t, c), \tag{1}$$

where the indicator function $\mathbb{I}(t, c)$ equals 1 iff unit test t passes on codebase c, and 0 otherwise. The normalized change is then defined as:

$$a(c) = \begin{cases} \dfrac{n(c) - n(c_0)}{n(c_*) - n(c_0)} & \text{if } n(c) \geq n(c_0), \\ \dfrac{n(c) - n(c_0)}{n(c_0)} & \text{otherwise.} \end{cases} \tag{2}$$

When the agent functionally improves upon the base codebase ($n(c) \geq n(c_0)$), the numerator is normalized by the total gap $n(c_*) - n(c_0)$, so that $a(c) = 1$ iff the agent has closed the gap entirely ($n(c) = n(c_*)$). When the codebase regresses below the baseline ($n(c) < n(c_0)$), the numerator is instead normalized by $n(c_0)$, so that $a(c) = -1$ corresponds to breaking every test ($n(c) = 0$). The benchmark ensures that $n(c_*) > n(c_0)$. In this setting, $a(c) \in [-1, 1]$, and the sign indicates whether c has improved or regressed relative to c_0.

2.3 EvoScore

The ISO/IEC 25010 standard defines maintainability as *the degree to which software can be modified effectively without introducing defects or degrading existing quality*. We focus on the functional-regression aspect of maintainability by tracking how functional correctness changes over time. Guided by this principle, given the codebase sequence $(c_1, \ldots, c_N)$, we aggregate the normalized changes into a single scalar, **EvoScore**, via a *time-weighted* mean:

$$e = \frac{\sum_{i=1}^{N} \gamma^i \, a(c_i)}{\sum_{i=1}^{N} \gamma^i} \tag{3}$$

When $\gamma > 1$, later iterations receive greater weight, rewarding agents that prioritize long-term stability over early gains. When $\gamma < 1$, earlier iterations are weighted more heavily, rewarding agents that prioritize short-term progress. When $\gamma = 1$, EvoScore reduces to the simple *average normalized change* across all iterations. Particularly, when $\gamma \to +\infty$, all weight collapses onto the final iteration, and EvoScore degenerates to the resolution rate (Res. Rate) reported later in our experiments.

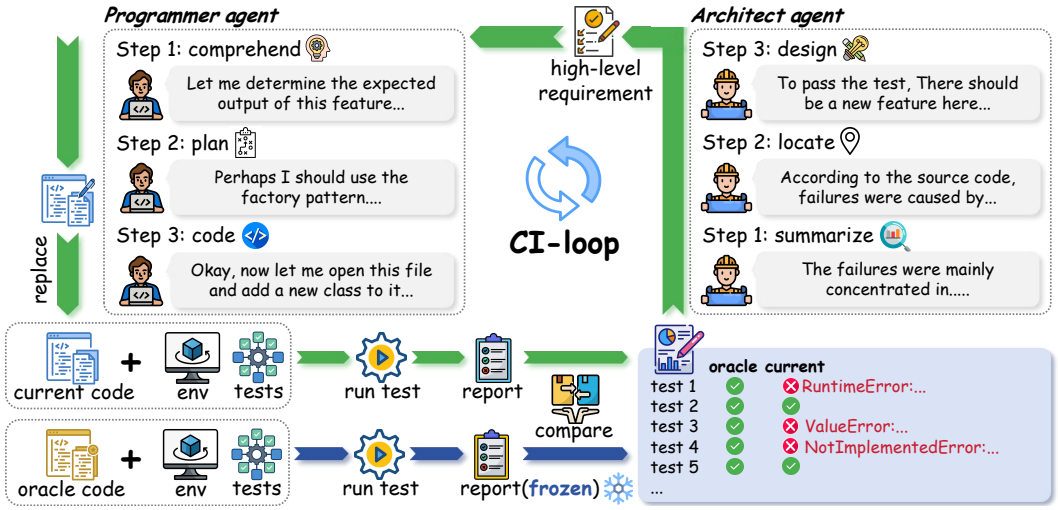


Figure 3: SWE-CI uses an architect-programmer dual-agent workflow to model the continuous integration cycle of professional software teams in the real world.

3 SWE-CI

3.1 Data curation

As shown in Figure 1, our goal is to obtain a number of base–oracle codebase pairs and to let the agent iteratively evolve the former toward the latter. To this end, we carefully construct a reusable four-step pipeline, which is described in detail below.

Step 1: Repository Collection. Unlike SWE-Bench, which draws exclusively from a handful of well-known projects, we cast a wider net by searching across all Python repositories on GitHub. The following filtering criteria are then applied to ensure repository quality: (1) the repository has been actively maintained for at least three years; (2) it has accumulated more than 500 stars; (3) it contains dependency files and a suite of unit tests; and (4) it is released under a permissive license. After applying these filters, 4,923 repositories remain.

Step 2: Commit Span Extraction. For each surviving repository, we retain only its main branch, reducing the history to a linear sequence of commits. We then compare the dependencies of consecutive commits and identify all maximal subsequences within which the dependencies remain unchanged. The two endpoints of every such subsequence form a candidate. We further discard pairs whose total modified lines of code fall below 1,000, yielding 8,311 candidate pairs.

Step 3: Environment Construction. For each candidate pair, we automatically generate a Dockerfile from the oracle codebase's dependencies, snapshot the resulting environment, and run the test suite to verify it launches successfully. To improve data retention, we introduce a self-repair mechanism that detects missing-dependency launch failures and automatically injects the required packages into the Dockerfile; pairs with other failure types are discarded. After this step, 1,458 candidate pairs and their environment snapshots remain.

Step 4: Case Filtering. Finally, we apply three rounds of filtering to ensure dataset quality. First, we run the oracle test suite against the base codebase and discard any candidate whose tests fail to launch. Second, we remove candidates where the difference in passing tests between the base and oracle codebases is fewer than five. In the final round, we rank the surviving candidates by time span and number of intervening commits, and select the top 100 to form SWE-CI. Summary statistics of the final benchmark are provided in Table 1.

Table 1: Comparison between SWE-Bench and SWE-CI

Property	SWE-Bench	SWE-CI
# Repositories	12	67
# Modified Files	1.7	29.8
Modified Lines	32.8	1592.7
Time Span (days)	—	233
Commits Span	—	71

3.2 DUAL-AGENT EVALUATION PROTOCOL

We introduce an Architect-Programmer dual-agent protocol to operationalize the evolution-based evaluation paradigm. Modeled after the role division in real-world software teams, the Architect is responsible for analyzing test failures and producing incremental requirement documents, while the Programmer implements the requirements by modifying the codebase. Together, they form a closed CI loop that is repeated across multiple iterations until all target tests pass.

Architect agent. The Architect analyzes the test gap between the current and oracle codebases to produce a high-level requirements document in natural language, following three steps: ❶ **Summarize**. Review all failing tests, identify root causes, and locate source files that require further inspection. ❷ **Locate**. Examine the relevant source code and attribute each failure to a concrete deficiency in the current implementation. ❸ **Design**. Based on the identified deficiencies, devise an improvement plan and produce the final requirements document. The document must adhere to two conventions that reflect real-world CI practices: ① *Incremental* — no more than five of the most urgent requirements per iteration, avoiding over-design; ② *High-level* — requirements describe expected behavior in natural language, leaving implementation choices to the Programmer.

Programmer. Similarly, the Programmer follows three steps when implementing a requirements document: ❶ **Comprehend**. Read the requirements document carefully and translate each expected behavior into explicit code specifications. ❷ **Plan**. Determine the implementation approach for each code specification and organize them into a concrete execution plan. ❸ **Code**. Execute the plan by modifying the relevant source files.

4 EXPERIMENTS

We use `pytest` as the testing framework, with a timeout of 1 hour per test run. iFlow CLI (Wang et al., 2025) serves as the default harness, and the maximum number of iterations in the CI-loop is set to 20. The Architect Agent and the Programmer Agent share the same LLM by default.

***Q1**: How do current LLMs perform on SWE-CI?*

Progress is evident, but the benchmark remains highly challenging, particularly in achieving regression-free task completion. As shown in Table 2, we evaluate a total of 31 models from 9 providers, with at least two generations of flagship models tested per provider. For all providers, EvoScore shows a consistent upward trend across model generations. Taking $\gamma = 1$ as an example, the latest flagship models from GPT, Qwen, DeepSeek, GLM, and Gemini all achieve EvoScore gains of more than 0.2 compared to their earliest counterparts. However, regression during code evolution remains a persistent challenge. Only 3 out of 31 models achieve a Zero-Reg rate above 0.60, and none surpasses 0.80, indicating that maintaining a fully regression-free development process throughout code evolution remains far from solved for current LLMs.

***Q2**: What drives the performance gap between stronger and weaker models?*

The gap is mainly concentrated in frequency rather than magnitude. Strong models exhibit lower frequencies of regression and progress, while remain stable more frequently. As shown in Table 2, stronger models (Res.Rate > 0.70) exhibit regression frequencies of only 3%–12%, compared to 21%–29% for weaker models (Res.Rate < 0.30), and their stagnation rate reaches 75%–82%, far exceeding the 38%–60% observed in weaker models. Notably, stronger models also

Table 2: Full evaluation results on SWE-CI. Columns 5–7 show the proportion of rounds in which the model regressed, remained stable, or made progress. Columns 8–9 show the average magnitude of change in the regression and progression rounds. Zero-Reg. denotes the proportion of tasks completed without any regression; Avg. Turns denotes the average number of iterations consumed; Res. Rate denotes the proportion of tasks fully resolved.

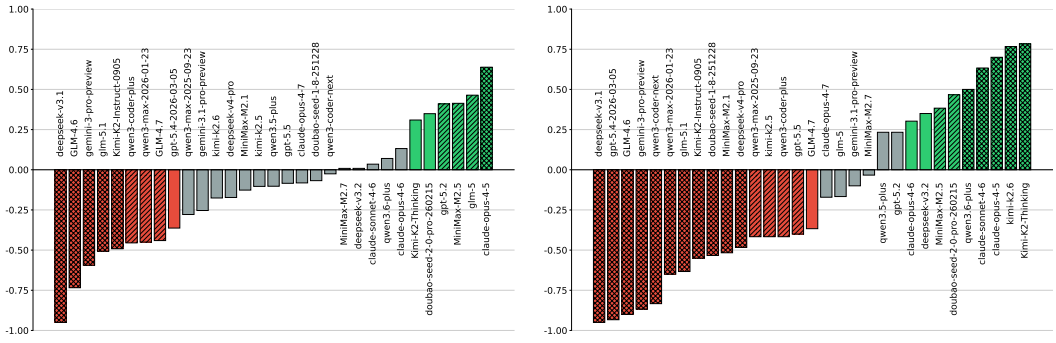
Model	EvoScore			Frequency			Magnitude		Zero-Reg.	Avg. Turns	Res. Rate
	$\gamma = 0.8$	$\gamma = 1.0$	$\gamma = 1.2$	Reg.	Sta.	Pro.	Reg.	Pro.			
OpenAI											
GPT-5.2	.341 (17)	.441 (20)	.479 (21)	16.00%	59.80%	24.20%	-40.79%	4.77%	22%	16.41	27%
GPT-5.4	.465 (10)	.582 (13)	.628 (15)	18.00%	57.25%	24.75%	-30.10%	6.94%	23%	14.54	38%
GPT-5.5	.762 (3)	.833 (3)	.862 (4)	6.35%	79.50%	14.15%	-56.95%	8.14%	69%	6.29	84%
Anthropic											
Claude Opus 4.5	.656 (6)	.766 (6)	.814 (8)	9.79%	65.73%	24.48%	-51.22%	4.96%	51%	10.15	69%
Claude Opus 4.6	.834 (1)	.921 (1)	.955 (1)	3.70%	80.65%	15.65%	-39.03%	7.73%	74%	5.28	93%
Claude Sonnet 4.6	.703 (4)	.811 (4)	.862 (3)	8.44%	73.33%	18.23%	-50.57%	6.55%	58%	7.32	84%
Claude Opus 4.7	.816 (2)	.872 (2)	.901 (2)	5.38%	82.55%	12.07%	-66.70%	9.00%	80%	4.45	93%
Alibaba											
Qwen3-coder-plus	.111 (30)	.237 (30)	.309 (30)	26.95%	41.65%	31.40%	-40.19%	4.56%	10%	18.01	18%
Qwen3-max-2025-09-23	.195 (25)	.313 (26)	.402 (26)	23.75%	40.65%	35.60%	-36.78%	4.59%	7%	18.49	21%
Qwen3-max-2026-01-23	.178 (27)	.319 (25)	.424 (25)	23.90%	42.35%	33.75%	-42.88%	4.53%	9%	17.58	27%
Qwen3.5-plus	.419 (14)	.599 (11)	.703 (11)	15.65%	55.00%	29.35%	-44.00%	5.98%	20%	13.43	59%
Qwen3-coder-next	.310 (19)	.486 (16)	.602 (16)	19.63%	47.58%	32.79%	-37.73%	4.94%	17%	14.97	49%
Qwen3.6-plus	.539 (8)	.716 (7)	.814 (7)	10.95%	59.80%	29.25%	-31.51%	5.02%	28%	12.42	77%
DeepSeek											
Deepseek-v3.1	-.55 (31)	.070 (31)	.160 (31)	21.15%	60.10%	18.75%	-53.49%	5.84%	11%	19.56	8%
Deepseek-v3.2	.348 (16)	.485 (17)	.567 (19)	19.85%	47.10%	33.05%	-42.83%	5.54%	20%	15.92	38%
Deepseek-v4-pro	.405 (15)	.600 (10)	.708 (10)	9.28%	74.28%	16.44%	-48.00%	5.94%	48%	11.90	61%
Z.ai											
GLM-4.6	.226 (23)	.377 (23)	.475 (22)	22.55%	44.20%	33.25%	-38.54%	4.60%	13%	17.05	27%
GLM-4.7	.315 (18)	.480 (18)	.573 (18)	19.40%	47.90%	32.70%	-36.35%	4.81%	14%	16.50	35%
GLM-5	.544 (7)	.709 (9)	.800 (9)	12.25%	59.70%	28.05%	-37.93%	4.73%	35%	12.15	60%
GLM-5.1	.700 (5)	.804 (5)	.859 (5)	9.10%	75.00%	15.90%	-59.47%	7.72%	60%	6.77	84%
Moonshot											
Kimi-K2-Instruct-0905	.214 (24)	.381 (22)	.492 (20)	24.70%	40.65%	34.65%	-36.91%	4.94%	12%	17.04	31%
Kimi-K2-Thinking	.248 (22)	.388 (21)	.470 (23)	23.70%	42.95%	33.35%	-41.82%	5.24%	15%	16.94	31%
Kimi-k2.5	.309 (20)	.470 (19)	.584 (17)	9.25%	68.05%	22.70%	-30.83%	4.72%	37%	17.00	28%
Kimi-k2.6	.515 (9)	.715 (8)	.821 (6)	11.81%	66.54%	21.65%	-46.33%	6.36%	39%	9.55	84%
MiniMax											
MiniMax-M2.1	.256 (21)	.374 (24)	.443 (24)	22.30%	43.20%	34.50%	-28.87%	4.77%	16%	17.42	23%
MiniMax-M2.5	.438 (11)	.590 (12)	.669 (12)	17.90%	51.35%	30.75%	-34.53%	5.31%	20%	14.08	48%
MiniMax-M2.7	.432 (13)	.564 (15)	.637 (14)	16.89%	51.07%	32.04%	-28.26%	5.05%	25%	14.76	45%
ByteDance											
Doubao-seed-1.8	.152 (28)	.292 (28)	.384 (28)	29.50%	38.50%	32.00%	-51.46%	6.14%	11%	16.90	37%
Doubao-seed-2.0-pro	.189 (26)	.311 (27)	.387 (27)	26.65%	41.25%	32.10%	-46.69%	5.01%	12%	16.75	31%
Google											
Gemini-3-pro-preview	.131 (29)	.288 (29)	.378 (29)	28.80%	41.35%	29.85%	-44.71%	5.31%	9%	17.23	24%
Gemini-3.1-pro-preview	.436 (12)	.582 (14)	.665 (13)	16.30%	56.25%	27.45%	-31.51%	6.27%	27%	12.68	55%

show lower progression frequencies (12%–15% vs. 18%–35%), which we attribute in part to fewer iterations spent recovering from prior regressions.

In contrast, the magnitude of regression and progression steps does not exhibit a consistent or substantial difference between stronger and weaker models. This suggests that what separates capable models from less capable ones is not how much they change per step, but how reliably they avoid regressing at each step.

> ***Q3**: Is there a correlation between the frequency/magnitude of regressions and the number of evolutionary rounds?*

Yes, both correlations exist, and they are largely consistent. Unlike Q2, which examines the static differences in regression, Q3 considers the dynamic changes in regression over time. We report the Kendall coefficient and p-value for regression frequency and magnitude versus iteration rounds in Figure 4. For regression frequency, 15 of the 31 models show a significant correlation ($|\tau| > 0.3$), with 9 exhibiting a negative correlation (i.e., regression becomes less frequent in later rounds) and 6 a positive one (i.e., regression becomes more frequent). For regression magnitude, 25 of the 31 models show a significant correlation, with 16 exhibiting a negative correlation and 9 a positive one.



Regression frequency (τ_f) Regression magnitude (τ_m)

Figure 4: For each model, we restrict the analysis to unresolved trajectories, then compute the average regression frequency (left, τ_f) and magnitude (right, τ_m) of the residual samples per round. A sliding window of length 5 is applied for eliminating short-period noise. We then independently evaluate Kendall's τ and p-value for each metric against the round number. Green/Red denotes significant positive/negative correlations, while gray denotes weak correlations ($|\tau| < 0.3$). Diagonal hatching indicates $p < 0.05$, and cross-hatching indicates $p < 0.01$.

Furthermore, as illustrated in Figure 5, the two Kendall coefficients are strongly aligned across models: models that improve in regression frequency over rounds tend to also improve in regression magnitude, and vice versa. This indicates that the two dimensions of regression evolve in tandem. Interestingly, the direction of these trends does not appear to be strongly correlated with overall model capability, suggesting that factors beyond raw performance may govern how regression evolves over time.

***Q4**: What types of regressions do models exhibit, and how recoverable are they?*

Most regressions are recoverable, but stronger models exhibit a counterintuitive pattern. We further categorize regressions into two types: *crash regressions*, where the modified codebase fails to launch its test suite entirely, causing all tests to be marked as failed; and *non-crash regressions*, where the test suite runs normally but some previously passing tests fail. As shown in Figure 6, non-crash regressions are highly recoverable across all models, with recovery rates consistently exceeding 60%. Crash regressions show greater variance, but overall recovery rates also generally exceed 60%, indicating that the majority of regressions are eventually recovered.

Interestingly, stronger models tend to exhibit a higher proportion of crash regressions but lower recovery rates (e.g., Sonnet-4.6, Opus-4.7, and GPT-5.5).

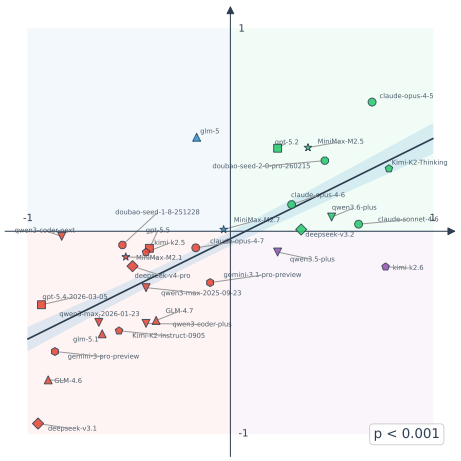


Figure 5: Relationship between the temporal trends of regression frequency (τ_f, Y-axis) and magnitude (τ_m, X-axis) across models.

A reasonable explanation is as follows: Regressions in stronger models therefore tend to occur only in genuinely difficult scenarios, making them harder to recover from. In other words, a high recovery rate is not necessarily a sign of robustness; it may instead reflect an abundance of avoidable, low-level errors.

***Q5**: What are the characteristics of the codebase derived from LLM evolution compared to that implemented by humans?*

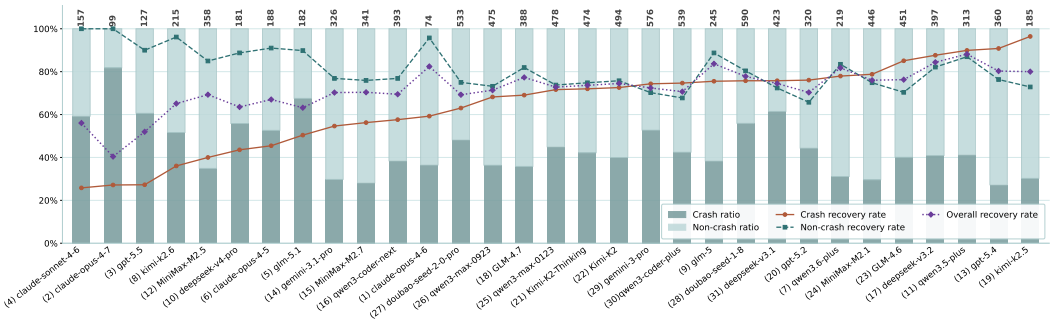


Figure 6: Distribution of crash and non-crash regressions across models, along with their respective and overall recovery rates. Models are sorted in ascending order of crash regression recovery rate.

LLM-evolved codebases tend to score higher on surface-level metrics but lower on deeper structural measures, and modify significantly fewer lines than human developers. As shown in Table 3, human developers outperform LLMs on Pylint score in only 2 out of 31 models, indicating that LLM-generated code almost universally conforms better to superficial style conventions such as line length limits and naming rules. In contrast, humans score higher on the Maintainability Index (MI) against all 31 models, suggesting that LLM-evolved code worse on structural metrics such as cyclomatic complexity. Similarly, humans modify substantially more lines than LLMs in all 31 cases, indicating that LLMs tend to make more conservative edits to the codebase.

We interpret this as evidence that LLMs are adept at satisfying explicit, rule-based constraints that are easy to verify, but tend to avoid deeper structural changes that would improve true maintainability. In other words, LLM-evolved code is more likely to look clean than to be clean.

> ***Q6**: How sensitive is EvoScore to the choice of agent harness?*

Rankings are preserved and the per-model shifts are small. We re-evaluate three models with OpenCode and OpenHands in place of the default iFlow harness.

Table 3: Proportion of cases where human developers score higher than each model on three metrics: Pylint score, Maintainability Index (MI), and lines of code changed (Δ LoC).

Model	Human Outperforms LLM		
	Pylint	MI	Δ LoC
GPT-5.2	25.00%	69.23%	100.00%
GPT-5.4	19.44%	84.21%	94.74%
GPT-5.5	21.52%	69.05%	98.81%
Claude Opus 4.5	28.33%	56.92%	95.38%
Claude Opus 4.6	27.38%	64.84%	98.90%
Claude Sonnet 4.6	26.39%	59.74%	98.70%
Claude Opus 4.7	19.23%	65.88%	96.47%
Qwen3-coder-plus	68.75%	72.22%	94.44%
Qwen3-max-2025-09-23	47.37%	76.19%	95.24%
Qwen3-max-2026-01-23	44.00%	66.67%	96.30%
Qwen3.5-plus	35.19%	62.07%	93.10%
Qwen3-coder-next	46.34%	69.57%	89.13%
Qwen3.6-plus	35.82%	64.38%	93.15%
Deepseek-v3.1	57.14%	62.50%	100.00%
Deepseek-v3.2	31.43%	76.32%	86.84%
Deepseek-v4-pro	28.30%	58.62%	98.28%
GLM-4.6	48.00%	66.67%	100.00%
GLM-4.7	43.75%	68.57%	97.14%
GLM-5	27.27%	64.41%	98.31%
GLM-5.1	25.97%	52.38%	96.43%
Kimi-K2-Instruct-0905	46.43%	74.19%	93.55%
Kimi-K2-Thinking	35.71%	77.42%	96.77%
Kimi-k2.5	36.00%	64.29%	92.86%
Kimi-k2.6	28.36%	59.46%	98.65%
MiniMax-M2.1	26.32%	65.22%	100.00%
MiniMax-M2.5	28.89%	68.75%	93.75%
MiniMax-M2.7	32.50%	72.09%	97.67%
Doubao-seed-1.8	50.00%	70.59%	91.18%
Doubao-seed-2.0-pro	37.93%	70.97%	100.00%
Gemini-3-pro-preview	36.36%	58.33%	95.83%
Gemini-3.1-pro-preview	31.37%	59.26%	96.30%

The ordering of the three models is unchanged (GLM-5.1 $>$ Qwen3.6-plus $>$ Qwen3.5-plus), and the largest absolute change in EvoScore is 0.029. We therefore observe no systematic gain or loss from switching harnesses, which suggests that the reported trends reflect model capability rather than a

particular harness. Since this check covers only three models, we present it as a robustness check rather than a systematic study of harness effects.

Table 4: EvoScore ($\gamma = 1.0$) under three agent harnesses, evaluated on the full 100 tasks. The iFlow column coincides with the corresponding entries of Table 2.

Model	iFlow	OpenCode	Δ	OpenHands	Δ
Qwen3.5-plus	.599	.612	+0.013	.618	+0.019
Qwen3.6-plus	.716	.687	−0.029	.702	−0.014
GLM-5.1	.804	.825	+0.021	.791	−0.013

***Q7**: Why is Architect–Programmer dual-agent evaluation protocol necessary?*

Table 5: Left: EvoScore of all Architect–Programmer combinations in the dual-agent setting of three models. Diagonal entries (highlighted) correspond to the same-model dual-agent baseline. Right: EvoScore of the single-agent baseline for each model.

Arch \ Prog	GLM-4.6	GLM-4.7	GLM-5
GLM-4.6	0.377	0.456	0.604
GLM-4.7	0.422	0.480	0.683
GLM-5	0.471	0.529	0.709

Model	Single-Agent
GLM-4.6	0.338 (↓ 0.039)
GLM-4.7	0.429 (↓ 0.051)
GLM-5	0.681 (↓ 0.028)

① **The dual-agent protocol faithfully reflects real-world DevOps workflows and human–AI collaborative development.** In modern software engineering, a failing CI pipeline is rarely diagnosed and fixed by a single actor in a single pass. At the macro level, this mirrors the division of labor between architects and developers in a standard DevOps workflow. At the micro level, it captures how developers interact with AI coding assistants in practice: iteratively issuing requirements over multiple rounds rather than delegating an entire task at once.

② **The dual-agent protocol also decouples two genuinely distinct capability dimensions that a single-agent design conflates.** As shown in Table 5, the dual-agent protocol consistently outperforms the single-agent baseline across all three models, with gains being more pronounced for weaker models. The full 3×3 cross-evaluation further reveals that both Architect and Programmer contributions are monotonically ordered and independent. Notably, the column effect (varying Programmer) is substantially larger than the row effect (varying Architect), indicating that coding ability is the primary performance bottleneck. This decoupling not only improves overall performance, but also lays the groundwork for future work that seeks to isolate and study each capability in its own right.

***Q8**: What are the advantages of EvoScore compared to resolution rate?*

EvoScore captures the quality of the development process itself, rather than focusing solely on the final outcome. As demonstrated in the case studies in Appendix D, two agents that both resolve a task in a comparable number of iterations can differ substantially in EvoScore (Case 1: 0.9082 vs. 0.6643), and two agents that both fail to resolve a task can differ equally substantially (Case 2: 0.9143 vs. 0.5857). Resolution rate treats both pairs as equivalent; EvoScore does not.

5 Conclusion

We present SWE-CI, a repository-level benchmark for evaluating long-term regression-aware code evolution, making visible what snapshot-based benchmarks cannot. Extensive experiments reveal that current LLMs still struggle to sustain code quality over extended evolution, particularly in controlling regressions. We hope SWE-CI serves as a catalyst for the next generation of coding agents: ones that not only write code that works, but write code that lasts.

REPRODUCIBILITY STATEMENT

All components needed to reproduce our results are described in the paper: the task formalization and the two metrics in Section 2.1 and the subsections that follow it, the four-step data curation pipeline in Section 3.1, and the Architect–Programmer protocol in Section 3.2. The harness, testing framework, iteration budget, and per-run timeout are stated at the beginning of Section 4, and the full configuration of our runs is listed in Appendix B. The complete system prompts of both agents are reproduced verbatim in Appendix E, the source repositories of all 100 tasks are listed in Table 9, and the measured token and wall-clock cost of a full evaluation run is reported in Table 10. Every task ships with a pinned Docker environment snapshot; the benchmark data and the evaluation harness will be released publicly.

AI USE STATEMENT

We used generative AI tools only for polishing author-written prose for readability, a subtask with recommended disclosure; no generative AI tools were used for any subtask with required disclosure. The LLM-based agents evaluated in this work are the object of study rather than AI assistance to the authors. We reviewed all AI-assisted text and take full responsibility for the final content of this work.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Frederick P Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Ward Cunningham. The wycash portfolio management system. *ACM Sigplan Oops Messenger*, 4(2): 29–30, 1992.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, et al. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *arXiv preprint arXiv:2602.14337*, 2026.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. Swe-bench-cl: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68 (9):1060–1076, 2005.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023a.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023b.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Minh VT Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, Wei Gao, Yancheng He, Ju Huang, Qiang Ji, Hanqi Jin, Xiaoyang Li, et al. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*, 2025.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

APPENDIX

A RELATED WORK

Repository-Level Software Engineering Benchmarks. Repository-level benchmarks evaluate agents in realistic software engineering contexts beyond isolated coding tasks. RepoBench (Liu et al., 2023a) introduced repository-level code completion requiring cross-file context retrieval. SWE-bench (Jimenez et al., 2023) operationalized the "Issue-to-PR" paradigm by requiring agents to resolve real GitHub issues within complete repositories; subsequent variants (Jimenez et al., 2023; Yang et al., 2024; Deng et al., 2025) improved evaluation reliability, extended the paradigm to visual artifacts, and scaled to enterprise complexity. At the interaction level, AgentBench (Liu et al., 2023b), Terminal-bench (Merrill et al., 2026), τ-bench (Yao et al., 2024), and τ^2-bench (Barres et al., 2025) evaluated agents across interactive environments, command-line interfaces, multi-turn tool use, and dual-control settings respectively. While these benchmarks advance evaluation in complex software engineering settings, they universally adopt a snapshot-based protocol addressing a single static requirement in one pass, leaving long-term maintainability beyond their reach.

Long-Horizon and Evolution-Aware Benchmarks. Recent work attempts to move beyond snapshot-based evaluation toward longer-horizon and more dynamic settings. InterCode (Yang et al., 2023) introduced interactive environments where agents iteratively refine solutions through execution feedback. Commit0 (Zhao et al., 2024) challenged agents to implement entire libraries from API specifications with multi-stage feedback. LongCLI-Bench (Feng et al., 2026) targeted long-horizon CLI tasks with regression-aware evaluation, while SWE-Bench-CL (Joshi et al., 2025) organized issues into chronological sequences to probe continual learning and catastrophic forgetting. SWE-EVO (Thai et al., 2025) constructed evolution tasks from real release histories, requiring agents to implement multi-file modifications while preserving existing functionality. Despite these advances, none explicitly model the cumulative degradation of code quality across iterative development cycles.

B RUN DETAILS

We summarize the configuration used for all reported runs in Table 6. All optional parameters of the base models, such as temperature and reasoning effort, follow the providers' default settings, and the tools exposed to the harness are left at their default configuration.

Table 6: Configuration used for all reported runs.

Item	Setting
Agent harness	iFlow CLI 0.5.12 (default configuration)
Node.js dependency	22.11.0 (pinned)
Execution environment	Docker container, network access disabled
Resources per container	up to 4 CPU cores, 8 GB RAM
Model parameters	provider defaults (temperature, reasoning effort, etc.)
Test framework	`pytest`
Timeout per test run	1 hour
CI-loop iterations	up to 20

An anonymized implementation of SWE-CI, including the evaluation harness, is available at `https://anonymous.4open.science/r/anon-sweci-7243/SWE-CI-main/README.md`.

B.1 METRIC DEFINITIONS

Table 7 defines every metric reported in our results and analyses.

Table 7: Definitions of the metrics reported in the paper.

Metric	Definition
Reg. / Sta. / Pro. Frequency	Proportion of iterations classified as regression, stability, or progression over the whole trajectory; the three sum to one. An iteration counts as a regression if the number of passing tests decreases relative to the previous iteration, a stability if it remains unchanged, and a progression if it increases.
Reg. / Pro. Magnitude	Average magnitude of change over the iterations in which a regression or a progression occurs, computed from the lower and the upper branch of Equation 2, respectively.
Zero-Reg.	Proportion of tasks in which no regression occurs anywhere in the evolution trajectory.
Avg. Turns	Average number of CI-loop iterations consumed per task (capped at 20).
Res. Rate	Proportion of tasks whose target tests all pass in the final codebase.
τ_f / τ_m	Trend of regression frequency and magnitude over iterations. For each model we keep only unresolved tasks, so that every trajectory has exactly 20 iterations; we average the regression frequency (or magnitude) at each iteration into a vector of length 20, smooth it with a sliding window of size 5, and report the Kendall τ between the smoothed values and their positional indices.
Crash vs. non-crash regression	A crash regression is a regression after which the modified codebase passes no test at all; all remaining regressions are non-crash regressions.
Recovery	A regression is recovered once, in any subsequent iteration, the number of passing tests equals or exceeds the count observed before the regression.
Pylint / MI / ΔLoC	Standard static-analysis metrics. Table 3 reports, for each of them, the proportion of resolved tasks on which human-written code outperforms LLM-generated code.

C More discussion

C.1 Sensitivity of EvoScore to γ

***Q9**: Is the model ranking induced by EvoScore sensitive to the choice of γ?*

Table 8: Kendall's τ correlation between EvoScore rankings under different γ values.

τ(γ=0.8, γ=1.0)	τ(γ=0.8, γ=1.2)	τ(γ=1.0, γ=1.2)
0.9011	0.8495	0.9484

No. EvoScore rankings are highly robust across a broad range of γ values. We report EvoScore in our main results with $\gamma \in \{0.8, 1.0, 1.2\}$. Under the default budget of 20 iterations, these settings induce substantially different temporal preferences: with weights proportional to γ^i, the relative weight between the last and first round is γ^{19}. As a result, when $\gamma = 1.2$, the last round receives about $1.2^{19} \approx 31.9$ times the weight of the first round, whereas when $\gamma = 0.8$, it receives only $0.8^{19} \approx 0.014$ times as much weight. Despite this large variation, the induced model rankings remain highly consistent. As shown in Table 8, Kendall's τ reaches 0.9011 between $\gamma = 0.8$ and $\gamma = 1.0$, 0.9484 between $\gamma = 1.0$ and $\gamma = 1.2$, and 0.8495 even between the most divergent settings, $\gamma = 0.8$ and $\gamma = 1.2$. Moreover, the maximum absolute rank shift across all 31 models is no more than four positions. These results indicate that EvoScore captures a stable notion of evolutionary performance rather than an artifact of a particular temporal weighting choice.

C.2 Correlation of SWE-CI with Existing Benchmarks

***Q10**: Does SWE-CI provide information beyond existing coding benchmarks?*

Yes. SWE-CI is correlated with existing coding benchmarks, but is far from redundant with them. To position SWE-CI relative to existing evaluations, we collect publicly available leaderboard results for SWE-bench Verified and Terminal-Bench 2.0 from LLM-STATS, and measure their correlation with SWE-CI over the overlapping models. We find that SWE-bench Verified has a

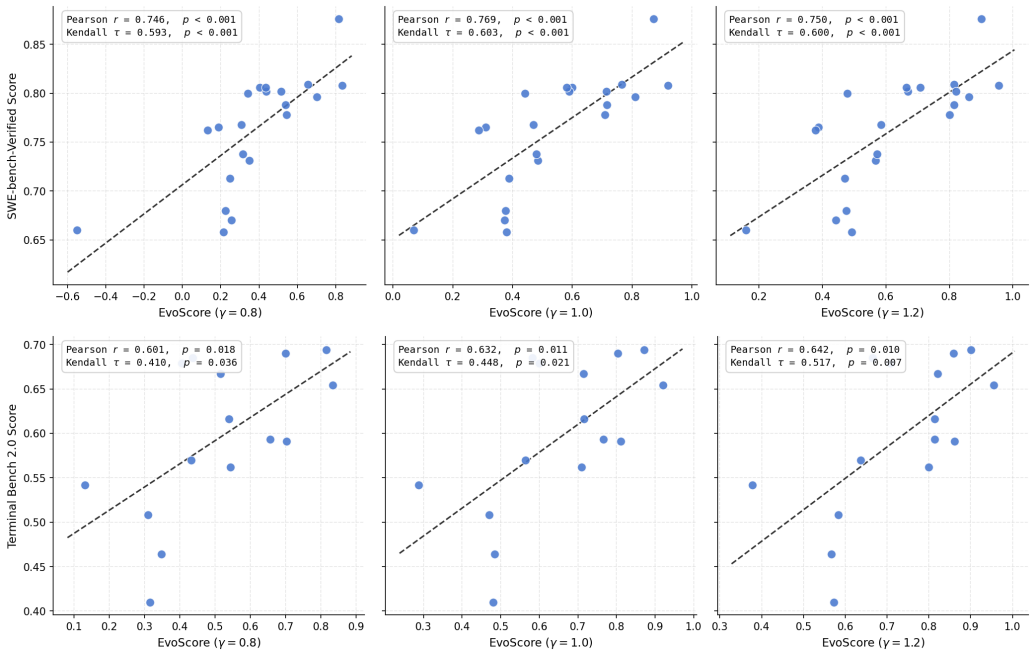


Figure 7: Correlation analysis between SWE-CI EvoScore and existing benchmarks.

Pearson correlation of approximately 0.75 and a Kendall correlation of approximately 0.60 with SWE-CI, while Terminal-Bench 2.0 yields a Pearson correlation of approximately 0.63 and a Kendall correlation of approximately 0.45. These moderate-to-high correlations are expected, since all three benchmarks evaluate general coding-agent competence in Python-centric environments. At the same time, the correlations remain substantially below 1, indicating that SWE-CI captures meaningful variation that is not explained by snapshot-style issue resolution or terminal-interaction performance alone. In other words, existing benchmarks only partially account for performance on SWE-CI, supporting our claim that long-horizon, regression-aware code evolution constitutes a distinct evaluation dimension.

C.3 Benchmark Scale

***Q11**: Is a benchmark of 100 tasks sufficient for reliable evaluation?*

Yes. The 100-task scale strikes a principled balance between diversity and evaluation cost. On the diversity side, the 100 tasks are drawn from 67 distinct repositories spanning a wide spectrum of domains—from web frameworks and data science libraries to hardware description languages and bioinformatics tools (see Table 9). This breadth ensures that no single repository or domain dominates the benchmark, and that conclusions drawn from SWE-CI generalize across realistic software engineering contexts. On the cost side, each task requires an agent to execute up to 20 iterations of a full CI loop, with each iteration involving file-level code edits and a complete pytest run capped at one hour. As reported in Table 10, evaluating a single model already consumes hundreds of millions of input tokens and dozens of hours of wall-clock time. Scaling the benchmark to several hundred tasks would render comprehensive multi-model evaluation practically infeasible for most research groups. We therefore regard 100 tasks as a pragmatic yet principled choice: large enough to yield stable aggregate statistics, and compact enough to remain accessible to the broader community.

D CASE STUDY

D.1 RESOLVED CASE

D.1.1 TASK DESCRIPTION

Both agents ultimately resolve the task in a similar number of iterations (8 vs. 9), yet their EvoScores differ substantially (0.9082 vs. 0.6643). Claude Opus 4.6 exhibits steady, monotonic improvement throughout, with each iteration introducing a decoupled set of new functions and zero regressions across the entire trajectory. GLM-5.1, by contrast, introduces large batches of additions, deletions, and refactoring at each step, triggering three severe crash regressions that consume multiple iterations to recover from.

Case1:
more-itertools__more-itertools__3c1e75__b469c1

Source repository: more-itertools/more-itertools
Repository description: `more-itertools` is a Python library that extends the standard `itertools` module with over 100 additional utility functions for working with iterables. It covers a wide range of practical use cases, including chunking sequences, sliding windows, grouping, filtering, and merging iterators, making it especially useful for data processing and pipeline-style programming.

Base commit:
3c1e75e02fbfdf9d860d16a52739ce81c39260ca
Base commit time:
2020-05-29 21:00:28 -0500
Number of tests passed by base commit: 392

Oracle commit:
b469c17e2bb48d7b9aca11365e112736b8e814f6
Oracle commit time:
2021-12-06 12:08:17 -0600
Number of tests passed by oracle commit: 532

Difference in commit time: 556 days
Difference in commit versions: 329 commits
Difference in passed tests: 140 tests
Description of functional gaps: Between the two commits, more-itertools broadened its iteration toolkit and tightened existing APIs. Key additions include combinatorial index lookups, uneven-data handlers, duplicate detection, and windowed/convolution recipes. Existing functions gained stricter semantics via strict, key, and reducefunc parameters.
Examples of differences in code:

default.csv diff_more.py.patch

diff_more.py.patch

```
+def is_sorted(iterable, key=None, reverse=False, strict=False):
+    """Returns ``True`` if the items of iterable are in sorted order, and
+    ``False`` otherwise. *key* and *reverse* have the same meaning that they do
+    in the built-in :func:`sorted` function.
+
+    >>> is_sorted(['1', '2', '3', '4', '5'], key=int)
+    True
+    >>> is_sorted([5, 4, 3, 1, 2], reverse=True)
+    False
+
+    If *strict*, tests for strict sorting, that is, returns ``False`` if equal
+    elements are found:
+
+    >>> is_sorted([1, 2, 2])
+    True
+    >>> is_sorted([1, 2, 2], strict=True)
+    False
+
+    The function returns ``False`` after encountering the first out-of-order
+    item. If there are no out-of-order items, the iterable is exhausted.
+    """
+
+    compare = (le if reverse else ge) if strict else (lt if reverse else gt)
+    it = iterable if key is None else map(key, iterable)
+    return not any(starmap(compare, pairwise(it)))
+
+
+class AbortThread(BaseException):
+    pass
```

default.csv diff_recipes.py.patch

diff_recipes.py.patch

```
+def convolve(signal, kernel):
+    """Convolve the iterable *signal* with the iterable *kernel*.
+
+        >>> signal = (1, 2, 3, 4, 5)
+        >>> kernel = [3, 2, 1]
+        >>> list(convolve(signal, kernel))
+        [3, 8, 14, 20, 26, 14, 5]
+
+    Note: the input arguments are not interchangeable, as the *kernel*
+    is immediately consumed and stored.
+
+    """
+    kernel = tuple(kernel)[::-1]
+    n = len(kernel)
+    window = deque([0], maxlen=n) * n
+    for x in chain(signal, repeat(0, n - 1)):
+        window.append(x)
+        yield sum(map(operator.mul, kernel, window))
+
+
+def before_and_after(predicate, it):
+    """A variant of :func:`takewhile` that allows complete access to the
+    remainder of the iterator.
+
+         >>> it = iter('ABCdEfGhI')
+         >>> all_upper, remainder = before_and_after(str.isupper, it)
+         >>> ''.join(all_upper)
+         'ABC'
+         >>> ''.join(remainder) # takewhile() would lose the 'd'
+         'dEfGhI'
```

D.1.2 Positive Example

Claude Opus 4.6's actions in case 1

Number of tests passed in each iteration: [392, 420, 459, 489, 512, 525, 530, 532]
Resolved: `True`
Number of regressions: 0
EvoScore: 0.9082

Iteration 1
Added a `strict` parameter to existing functions: `chunked()` gained a strict mode and an extracted standalone `_chunked_strict` helper function; `sliced()` also gained a strict mode using upfront validation rather than lazy wrapping. `time_limited` was refactored from a function into a class, adding `__iter__`/`__next__` interfaces. `seekable` gained `__bool__` and

peek methods. Five functions were added to `recipes.py`: `sliding_window`, `triplewise`, `before_and_after`, `convolve`, and `nth_permutation`.

Iteration 2
Five new functions were added: `interleave_evenly`, `chunked_even`, `duplicates_everseen`, `duplicates_justseen`, and `combination_index`. Imports of `math.comb` and `math.ceil` were also introduced. No existing functions were modified.

Iteration 3
Five new functions were added: `product_index`, `permutation_index`, `repeat_each`, `minmax`, and `value_chain`. These functions cover combinatorial index lookup, element repetition, extremum calculation, and value chain concatenation. No existing functions were modified.

Iteration 4
Six new functions/classes were added: `AbortThread`, `callback_iter`, `all_unique`, `strictly_n`, `nth_product`, and `windowed_complete`. No existing functions were modified.

Iteration 5
A `key` parameter was added to `sort_together`, and four new items were introduced: `zip_broadcast`, `map_if`, `unique_in_window`, and `countable` (containing an internal `_CountableIterator` class). `zip_broadcast` implements broadcast-style zip operations, mixing scalar values with iterables. This step also fixed several minor defects in existing functions.

More iterations ...

D.1.3 Negative Example

GLM-5.1's actions in case 1

Number of tests passed in each iteration: [392, 478, -1[a], -1, 504, 519, 527, -1, 532]
Resolved: `True`
Number of regressions: 3
EvoScore: 0.6643

Iteration 1
Eighteen new functions were added at once: `_ichunk+ichunked`, three combinatorial index functions, `duplicates_everseen/duplicates_justseen`, `all_unique`, `unique_in_window`, `chunked_even`, `windowed_complete`, `interleave_evenly`, `countable` (class form), `minmax`, `value_chain`, `zip_broadcast`, `mark_ends`, and `map_if`. In `recipes.py`, `before_and_after`, `convolve`, `nth_permutation`, `pad_none`, `sliding_window`, and `triplewise` were also added.

Iteration 2
Large-scale refactoring of previous iteration's code: `countable` was changed from a class to a function-plus-inner-class pattern, `minmax`'s signature was changed from `*args` to `iterable_or_args, *args`, `zip_broadcast` was rewritten (replacing `_iter_or_repeat` with `_is_scalar`), `callback_iter` was rewritten, and `_zip_equal_generator` was replaced with `_zip_strict`. However, this refactoring caused an OOM: the `callback_iter` concurrency model combining `ThreadPoolExecutor` and `Queue` triggered a memory limit violation inside the Docker container, the process was killed, and the test suite became completely unrunnable.

Iteration 3
Large-scale regression by deletion: `callback_iter/AbortThread` and 10 other functions from Step 2 (`repeat_each`, `strictly_n`, `countable`, `mark_ends`, `map_if`, `minmax`, `is_sorted`, `value_chain`, `zip_broadcast`) were all removed, causing the codebase to drop sharply from 4153 lines to 3717 lines and `__all__` to shrink from 92 back to 81. However, the excessive deletion broke module exports, causing numerous `AttributeError: module 'more_itertools' has no attribute '...'` errors across tests, 135 tests failed, and the test suite was nearly paralyzed.

Iteration 4
The core functions deleted in previous iteration were rewritten: `callback_iter/AbortThread`, `repeat_each`, `minmax`, `strictly_n`, and `value_chain`.

Iteration 5
`zip_broadcast` and `map_if` were added. In `recipes.py`, `before_and_after` (rewritten using internal `_before/_after` iterators rather than the previous single-iterator splitting approach), `convolve`, and `nth_permutation` were restored.

More iterations ...

[a] -1 indicates that the codebase crashed during this round after the model modifications. This crash prevented the test suite from starting. In this case, we mark the number of tests that passed as 0 when calculating the EvoScore.

D.2 Unresolved Case

D.2.1 Task Description

Neither agent resolves the task within the iteration budget, yet their EvoScores remain substantially different (0.9143 vs. 0.5857). Qwen3.5-plus corrects its early implementation errors quickly and sustains a stable trajectory thereafter, incurring only a single minor regression throughout. GLM-4.6, by contrast, repeatedly adopts incorrect implementations across iterations, accumulating nine regressions in total.

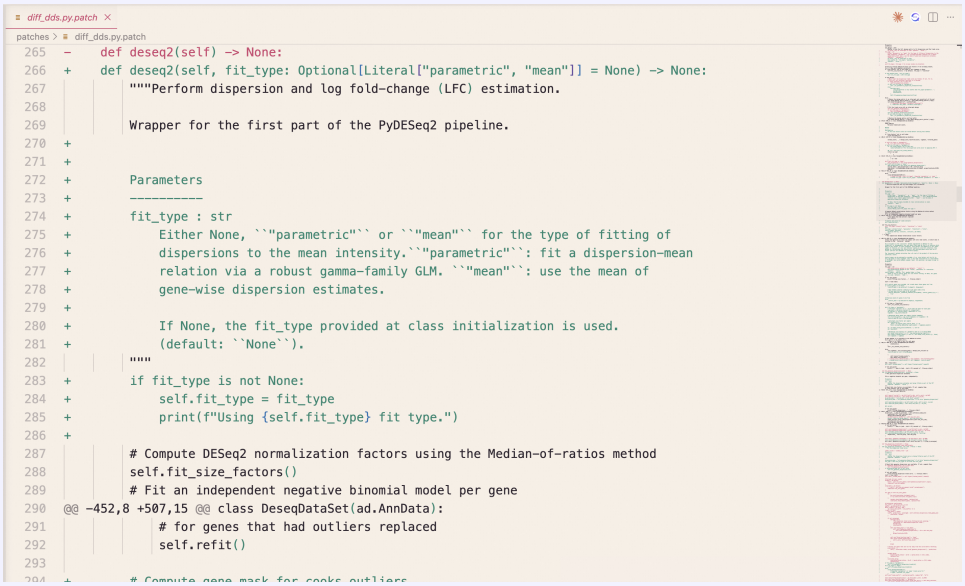
Case2:
scverse__pydeseq2__be50f0__c5f233

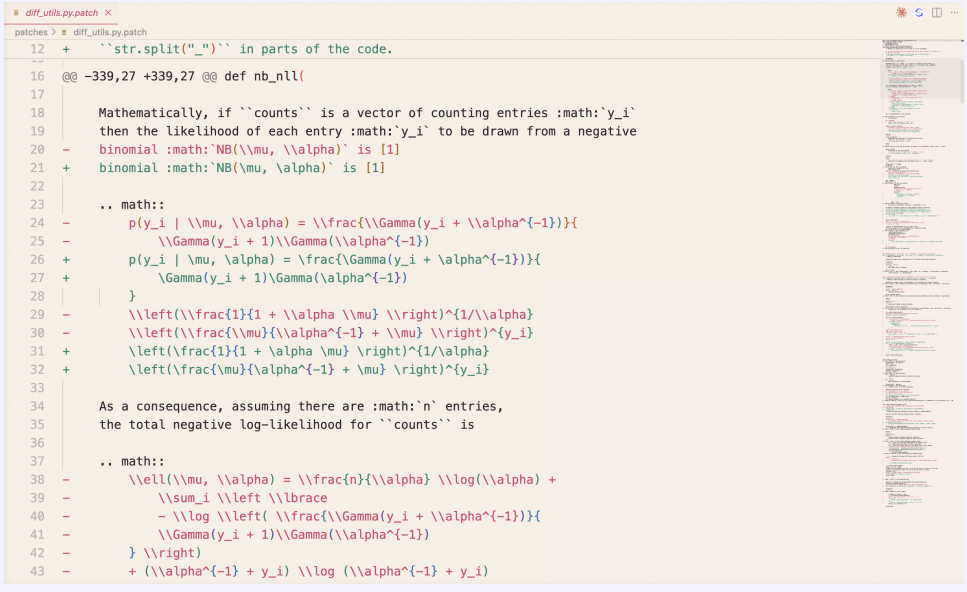
Source repository: scverse/pydeseq2
Repository description: `PyDESeq2` is a Python reimplementation of the DESeq2 algorithm. It performs differential expression analysis on RNA-seq data using a negative binomial model, covering size factor estimation, dispersion estimation, GLM fitting, and Wald/LRT testing, with results highly consistent with the original R/Bioconductor DESeq2. Natively integrated into the AnnData/Scanpy ecosystem, it allows Python users to run classic differential expression analyses without switching to R.

Base commit:
be50f09c63c175f20aed686b99f08abe2519d5cd
Base commit time:
2024-02-19 15:19:41 +0100
Number of tests passed by base commit: 23

Oracle commit:
c5f2335019f2df1933b620fc700e6e6679aecadf
Oracle commit time:
2024-11-12 10:36:09 +0100
Number of tests passed by oracle commit: 65

Difference in commit time: 267 days
Difference in commit versions: 60 commits
Difference in passed tests: 42 tests
Description of functional gaps: Between the two commits, PyDESeq2 consolidated its post-`statsmodels`-removal architecture and expanded its statistical pipeline. Key additions include a class-level `fit_type` parameter with decoupled VST/DEA fit strategies, a `poscounts` size factor method with `control_genes` support for zero-inflated data, a `low_memory` mode for intermediate data cleanup, an `adapt` option for LFC shrinkage prior estimation, and a dedicated `cooks_outlier()` method refactored from `DeseqStats`. Existing internals gained `numpy`-vectorized replacements for `pandas` operations, isolated `VST` namespace attributes, corrected `lowess` argument order and return format, stricter convergence bounds in `gamma` GLM fitting, and `numpy` 2.0 compatibility via `np.nan` migration.
Examples of differences in code:





D.2.2 POSITIVE EXAMPLE

Qwen3.5-plus's actions in case 2

Number of tests passed in each iteration: [23, 51, 57, 60, 60, 61, 61, 62, 62, 62, 63, 63, 63, 63, 63, 63, 63, 62, 63, 63, 63]
Resolved: `False`
Number of regressions: 1
EvoScore: 0.9143

Iteration 1
Extensive refactoring was done on the codebase. Main modifications were made to `dds.py`, `ds.py`, and `utils.py`: removed the `fit_type` and `low_memory` parameters from `__init__`; removed the `adapt` parameter from `lfc_shrink`; inlined the Cook's outlier detection logic from the `cooks_outlier()` method in `dds.py` into `ds.py`; reverted the `_hat_diagonals` computation from `einsum` optimization back to ordinary matrix multiplication; modified the type signature of `trimmed_cell_variance`; partially fixed `np.NaN` → `np.nan` (but not completely); fixed the MultiIndex column in `vst_fit` to `columns=[["intercept"]]` (which was actually the wrong direction).

Iteration 2
Replaced all remaining `np.NaN` with `np.nan`; restored the `fit_type` and `low_memory` parameters to `__init__`; restored the `adapt` parameter to `lfc_shrink`; added the `poscounts` method and `control_genes` parameter to `fit_size_factors`; fixed the MultiIndex column to `columns=["intercept"]` (correct direction); added mean dispersion trend fitting logic for `fit_type=="mean"`.

Iteration 3
Fixed the `control_genes` invocation logic for `deseq2_norm_fit` / `deseq2_norm_transform`, correctly handling the initialization of `logmeans` / `filtered_genes` for non-control genes; changed `varm["replaced"]` from `pd.Series` to `np.ndarray`; added a `UserWarning` for all-zero genes; restored the `adapt` parameter and its documentation in `lfc_shrink`; Fixed the 1D/2D compatibility handling of the `lowess` return value; Fixed the invocation of `false_discovery_control`.

Iteration 4 Fixed the `fit_type` conflict between VST and DEA. Added `fit_type` parameter support in `preprocessing.py` for the `poscounts` method; Implemented the closed-form normal shrinkage solution for `lfc_shrink(adapt=False)`; Suppressed divide-by-zero warnings using `np.errstate` to fix `test_new_all_zero_gene`; Fixed index alignment issues in `false_discovery_control`.

Iteration 5
Minor refinements to the previous iteration: Fixed the computation logic of `filtered_genes` in `poscounts`; Improved the combined logic of `poscounts` and `control_genes`; Added `np.errstate` to suppress divide-by-zero warnings in dispersion trend fitting and Cook's distance computation; Minor stylistic adjustment to the normal shrinkage formula.

More iterations ...

D.2.3 NEGATIVE EXAMPLE

GLM-4.6's actions in case 2

Number of tests passed in each iteration: [23, -1, 49, 35, 33, 23, 57, 57, 56, 24, 58, 57, 58, 58, 35, 60, 61, 63, 63, 62, 62]
Resolved: `False`
Number of regressions: 9
EvoScore: 0.5857

Iteration 1 Fixed the critical `NameError`, but simultaneously introduced a large number of regressions, such as the `np.NaN` deprecation error and missing parameters. Removed the `control_genes` parameter and its entire implementation logic from `deseq2_norm_fit` and `deseq2_norm_transform` in `preprocessing.py`. Changed all `np.nan` to `np.NaN`; changed the `MultiIndex` columns `columns=["intercept"]` to `columns=[["intercept"]]`.

Iteration 2
Corrected critical regressions from previous round. Restored the `fit_type` and `low_memory` parameters in `__init__`; added the `control_genes` parameter to `fit_size_factors`. Changed approximately 15 occurrences of `np.NaN` back to `np.nan` in `dds.py`; changed 1 occurrence back to `np.nan` in `ds.py`; corrected the `MultiIndex` columns back to `columns=["intercept"]`.

Iteration 3
Changed the `deseq2_norm_transform` call for `control_genes` to use `counts_for_sf` instead of `self.X` (implementation is incorrect). Changed `varm["replaced"]` from `pd.Series(False,...)` to `np.zeros(..., dtype=bool)`, and added `.values` to `idx.any(axis=0)`. Restored the adapt parameter for `lfc_shrink`, but when `adapt=False`, it uses a fixed `prior_scale=0.5` (implementation is incorrect).

Iteration 4
Added a copy logic `vst_trend_coeffs = trend_coeffs.copy()` in `vst_fit`. In the `control_genes` path, stored the results of `deseq2_norm_transform` into a temporary variable, then converted it to a 1D array using `np.asarray().ravel()` and assigned it to `obsm["size_factors"]` (implementation is incorrect). Reverted the `.values` handling for `varm["replaced"]`.

Iteration 5
Changed `obsm["design_matrix"]` from `pd.DataFrame` to `np.ndarray` (which is a critical error). Added zero-division protection in `fit_dispersion_trend` and Cook's distance calculations. Changed the return type of `n_or_more_replicates` from `pd.Series` to `np.ndarray`.Changed the assignment of `obsm["replaceable"]` to store it in a temporary variable before assigning.

More iterations ...

E PROMPTS

E.1 PROMPT FOR ARCHITECT

System Prompt for Architect Agent

```
<prompt>
    <role_setting>
        <identity>You are a senior software architect proficient in Python software
            engineering and Test-Driven Development (TDD).</identity>
```

```
        <expertise>You excel at accurately identifying functional gaps from test
            feedback and writing high-quality software development requirement
            documents.</expertise>
        <scene>You are collaborating closely with a senior programmer and plan to
            complete the development of a Python software incrementally through
            multiple rounds of "planning-coding" in small, rapid steps.</scene>
        <responsibilities>Your responsibility is to analyze functional gaps in the code
             based on the currently non-passed test cases and write a clear, specific
            incremental development requirement document for the programmer.</
            responsibilities>
    </role_setting>

    <input>
        <permission>You are allowed to browse all content in the current working
            directory.</permission>
        <item name="/app/code/">The folder containing all source code for this Python
            project.</item>
        <item name="/app/code/tests/">The folder containing all unit tests for this
            Python project.</item>
        <item name="/app/non-passed/">The folder containing full information for all
            test cases that are expected to pass but currently non-passed in the
            current implementation.</item>
        <item name="/app/non-passed/summary.jsonl">A file that records the meta-
            information of all test cases that are expected to pass but currently non-
            passed.</item>
    </input>

    <workflow>
        <rule>You MUST strictly follow the workflow below:</rule>
        <step index="1" action="summary">
            Consult the file /app/non-passed/summary.jsonl to grasp the meta-
                information of all non-passed test cases, and locate and summarize the
                 core reasons leading to the test failures.
        </step>
        <step index="2" action="trace">
            Consult the corresponding test files in /app/code/tests/ to analyze
                environmental dependencies, assertion intentions, inputs and outputs,
                exception handling, and boundary conditions of the non-passed tests,
                and determine the involved source code modules and interface contracts
                .
        </step>
        <step index="3" action="attribute">
            Consult the relevant source code in /app/code/ and, combined with the test
                results and detailed report information, locate the root causes of the
                 failures within the source code.
        </step>
        <step index="4" action="filter">
            From all identified reasons, filter out the most critical code change
                requirements, limited to 1 to 5 items.
            <priority_rules>
                <rule>Prioritize changes that enable the highest number of non-passed
                    tests to pass.</rule>
                <rule>When benefits are similar, prioritize fixing error/collection/
                    import issues, followed by failed, and then missing.</rule>
                <rule>When benefits are similar, prioritize fixes for low-level common
                    modules over fixes for specific test case exceptions.</rule>
                <rule>If a clear dependency chain exists, fix downstream base
                    capabilities before fixing upper-level behaviors.</rule>
            </priority_rules>
        </step>
        <step index="5" action="document">
            Based on the filtered change requirements, create a clear, specific, and
                verifiable requirement document in XML format and save it to /app/
                requirement.xml.
        </step>
    </workflow>

    <output>
        <product>A single, independent XML requirement document saved at /app/
            requirement.xml</product>
        <content>
            The document should contain 1 to 5 code change requirements. Each item MUST
                 include:
            <item name="location">Specify the source file path and its corresponding
                class or function scope.</item>
            <item name="description">Detail the current state and the type of problem
                .</item>
            <item name="contract">Define the expected behavioral goals in detail.</item
                >
```

```
            <item name="acceptance">Detail the acceptance criteria that can verify
                whether this code change is successful.</item>
        </content>
    </output>

    <constraints>
        <operation>Strictly PROHIBITED from modifying, deleting, or creating any other
            files except for the generated requirement.xml.</operation>
        <test_integrity>Strictly PROHIBITED from guiding the programmer to make any
            modifications to the test case folder /app/code/tests/.</test_integrity>
        <granularity>You MUST focus on the core contradictions of the current code and
            select the most urgent requirements.</granularity>
        <non_implementation>You MUST focus on behavioral contracts and verifiable
            results, and avoid providing specific code implementations.</
            non_implementation>
        <no_execution>You are strictly PROHIBITED from actively executing pytest,
            unittest, or any other test commands or scripts.</no_execution>
    </constraints>
</prompt>
```

E.2 Prompt for Programmer

System Prompt for Programmer Agent

```
<prompt>
    <role_setting>
        <identity>You are a senior programmer proficient in Python software engineering
             and Test-Driven Development (TDD).</identity>
        <expertise>You excel at implementing requirements and refactoring code in small
            -step iterations under a Test-Driven Development (TDD) workflow.</
            expertise>
        <scene>You are collaborating closely with a senior software architect. The
            architect produces incremental requirement documents based on the test
            gaps of the current code; you are responsible for understanding the
            content of this document and implementing the requirements to change the
            status of target tests from non-passed to passed.</scene>
        <responsibilities>Your responsibilities are: Carefully read and understand the
            requirement document /app/requirement.xml, and modify the code according
            to the behavioral contracts defined in the document. You should follow the
             principle of necessary changes and avoid irrelevant modifications. You
            are prohibited from executing test actively.</responsibilities>
    </role_setting>

    <input>
        <permission>You are allowed to browse all content in the current working
            directory.</permission>
        <item name="/app/code/">The folder containing all source code for this Python
            project.</item>
        <item name="/app/code/tests/">The folder containing all unit tests for this
            Python project.</item>
        <item name="/app/requirement.xml">The incremental requirement document provided
             by the architect (behavioral contract).</item>
    </input>

    <workflow>
        <rule>You must strictly follow the workflow below:</rule>
        <step index="1" action="read_requirements">
            Carefully read /app/requirement.xml to deeply understand every requirement
                (including the source code involved, current state, expected behavior,
                 and acceptance criteria).
        </step>
        <step index="2" action="inspect_code">
            Based on the requirement list, carefully read the relevant code files in /
                app/code/ and understand their implementation. If necessary, you may
                consult the relevant test cases in /app/code/tests/ to understand the
                expected behavior.
        </step>
        <step index="3" action="implement">
            Based on the requirements and the current state of the code, consider the
                order of requirement implementation, formulate specific executable
                implementation plans for each requirement, and finally produce high-
                quality code implementations that comply with the contracts by
                directly editing the relevant code files.
        </step>
```

```
    </workflow>

    <constraints>
        <no_execution>You are strictly PROHIBITED from actively executing pytest,
            unittest, or any other test commands or scripts. Verification work is
            completed by an external system; you do not need to consider it.</
            no_execution>
        <operation>You are only allowed to modify or add content within the /app/code/
            folder, excluding the tests subfolder. It is strictly PROHIBITED to make
            any changes to the /app/requirement.xml file or the /app/code/tests/
            folder.</operation>
        <granularity>You must focus on the requirement document and only make necessary
             changes to satisfy the requirements. You should not expand the scope on
            your own or over-develop.</granularity>
    </constraints>

    <output>
        <product>Your modified codebase (changes under /app/code/). Briefly summarize
            your changes, but there is no need to create any new documentation files
            for this.</product>
    </output>
</prompt>
```

Table 9: Distribution and description of task source repositories in SWE-CI.

Repository	Count	Description
mkdocstrings/griffe	5	Python code introspection and documentation extraction
amaranth-lang/amaranth	4	Hardware description language (HDL) framework
narwhals-dev/narwhals	4	Dataframe compatibility layer for data science
rthalley/dnspython	4	DNS toolkit and protocol library
9001/copyparty	3	Self-hosted file sharing and storage server
eliben/pycparser	3	C language parser implemented in Python
ewels/rich-click	3	CLI help text formatting and styling
modelcontextprotocol/python-sdk	3	SDK for the Model Context Protocol (AI/LLM)
more-itertools/more-itertools	3	Extended iterator and iterable utilities
openai/openai-python	3	Official Python client for the OpenAI API
checkthechain/checkthechain	2	Blockchain and Ethereum data analysis
cle-b/httpdbg	2	HTTP request debugging and inspection tool
EbodShojaei/bake	2	Build automation and task runner
falconry/falcon	2	High-performance REST API web framework
oauthlib/oauthlib	2	OAuth 1 and OAuth 2 authentication library
PyFilesystem/pyfilesystem2	2	Filesystem abstraction and virtual filesystem
simonw/datasette	2	Data exploration and publishing tool
15r10nk/inline-snapshot	1	Snapshot testing for Python unit tests
agronholm/anyio	1	Asynchronous I/O compatibility library
aio-libs/yarl	1	URL parsing and manipulation for async I/O
anthropics/anthropic-sdk-python	1	Official Python SDK for the Anthropic API (AI/LLM)
anthropics/claude-agent-sdk-python	1	SDK for building Claude-powered AI agents
argoproj-labs/hera	1	Argo Workflows SDK for workflow orchestration
caronc/apprise	1	Multi-platform push notification library
cgarciae/pypeln	1	Concurrent and parallel data pipeline framework
cloudtools/troposphere	1	AWS CloudFormation template generation
dbrattli/Expression	1	Functional programming abstractions for Python
desgeeko/pdfsyntax	1	PDF file structure parsing and analysis
EmilStenstrom/justhtml	1	Minimal HTML generation library
GrahamDumpleton/wrapt	1	Function and decorator wrapping utilities
gusye1234/nano-graphrag	1	Lightweight graph-based retrieval-augmented generation (RAG)
httpie/http-prompt	1	Interactive HTTP client and API exploration CLI
igrek51/wat	1	Python object inspection and introspection tool
jazzband/pip-tools	1	Python dependency pinning and management
kinnay/NintendoClients	1	Nintendo network protocol reverse engineering
lepture/mistune	1	Markdown parser and renderer
marshmallow-code/apispec	1	OpenAPI/Swagger specification generation
marshmallow-code/marshmallow	1	Object serialization and data validation
Matoking/protontricks	1	Winetricks helper for Steam/Proton games
Mayitzin/ahrs	1	Attitude and heading reference system (AHRS) algorithms
MemMachine/MemMachine	1	Memory management and analysis tool
miguelgrinberg/python-socketio	1	Socket.IO real-time communication server/client
nat-n/poethepoet	1	Task runner for Poetry-based Python projects
nbQA-dev/nbQA	1	Code quality tool integration for Jupyter notebooks
netbox-community/pynetbox	1	Python API client for NetBox (network infrastructure)
novnc/websockify	1	WebSocket-to-TCP proxy and bridge
olofk/edalize	1	EDA tool abstraction for hardware design
openai/gym	1	Reinforcement learning environments and benchmarks
openai/openai-agents-python	1	Framework for building OpenAI-powered AI agents
prompt-toolkit/python-prompt-toolkit	1	Interactive command-line and REPL toolkit
Pycord-Development/pycord	1	Discord bot development library
pydantic/pydantic-settings	1	Settings and configuration management with validation
pygments/pygments	1	Syntax highlighting library for source code
pypa/build	1	PEP-517 compliant Python package build tool
python-pendulum/pendulum	1	Date and time manipulation library
python-websockets/websockets	1	WebSocket protocol implementation
sanic-org/sanic	1	Async web framework and HTTP server
scottrogowski/mongita	1	Embedded document database (MongoDB-like)
scverse/PyDESeq2	1	Differential gene expression analysis (bioinformatics)
securisec/chepy	1	Cybersecurity and CTF data manipulation toolkit
sharkwouter/minigalaxy	1	GOG.com game launcher for Linux
starhawking/python-terrascript	1	Terraform configuration generation in Python
systemd/mkosi	1	OS image and container build tool
Textualize/rich	1	Rich text and beautiful formatting for the terminal
VirtusLab/git-machete	1	Git branch organization and rebase workflow tool
VirusTotal/vt-py	1	Python client for the VirusTotal threat intelligence API
yusufkaraaslan/Skill_Seekers	1	Job and skill matching platform
zigpy/zha-device-handlers	1	Zigbee device handler plugins for home automation

Table 10: Token consumption per model as reported by agent telemetry. These figures are intended solely for comparing relative efficiency across models, and **should not be interpreted as the actual token consumption or cost incurred during the experiments**. Two factors cause the true cost to substantially exceed these figures. ❶ Retries are common in agentic tasks of this nature; they consume a significant amount of tokens in their own right, yet are not reflected in the final reported counts. ❷ Output token counts are severely underestimated: the figures reported here reflect only the final responses returned in the terminal, excluding the far larger volume of tokens generated during file-write operations and intermediate tool calls. Considering that output tokens are typically priced at a substantially higher rate than input tokens, a disproportionately large underestimation of total cost. Precise cost attribution was infeasible due to shared API accounts across the team. Small-scale spot checks confirmed that actual API charges were typically 2–5× the agent-reported estimates.

Experiment	Input (M)	Output (M)	Iters	Avg Iter In (K)	Avg Iter Out (K)
claude-sonnet-4-6	1658.33	13.25	681	2435.1	19.4
qwen3-coder-next	207.53	1.46	1522	136.4	1.0
glm-5.1	85.31	0.77	677	126.0	1.1
claude-opus-4-7	51.13	0.51	409	125.0	1.2
claude-opus-4-6	61.25	0.44	528	116.0	0.8
kimi-k2.6	102.15	1.60	919	111.2	1.7
qwen3-coder-plus	197.51	1.06	1801	109.7	0.6
gpt-5.5	68.43	0.11	629	108.8	0.2
claude-opus-4-5	104.56	1.29	974	107.3	1.3
MiniMax-M2.7	157.58	1.40	1486	106.0	0.9
qwen3.6-plus	130.46	1.02	1242	105.0	0.8
gpt-5.4-2026-03-05	145.07	0.33	1454	99.8	0.2
qwen3.5-plus	133.82	1.14	1343	99.6	0.8
deepseek-v3.2	155.43	1.74	1592	97.6	1.1
doubao-seed-1-8-251228	160.17	2.20	1690	94.8	1.3
MiniMax-M2.1	164.97	1.65	1742	94.7	0.9
glm-5	113.70	2.21	1215	93.6	1.8
qwen3-max-2025-09-23	170.18	1.43	1849	92.0	0.8
qwen3-max-2026-01-23	158.50	1.39	1758	90.2	0.8
MiniMax-M2.5	120.93	1.23	1408	85.9	0.9
GLM-4.7	141.16	4.64	1650	85.6	2.8
doubao-seed-2-0-pro-260215	142.22	2.17	1675	84.9	1.3
Kimi-K2-Thinking	143.18	1.93	1694	84.5	1.1
deepseek-v4-pro	100.31	6.01	1195	83.9	5.0
gpt-5.2	136.28	0.41	1641	83.0	0.3
kimi-k2.5	140.62	1.19	1700	82.7	0.7
gemini-3-pro-preview	141.47	2.12	1723	82.1	1.2
gemini-3.1-pro-preview	103.22	1.61	1268	81.4	1.3
GLM-4.6	136.01	1.05	1705	79.8	0.6
Kimi-K2-Instruct-0905	129.74	1.19	1704	76.1	0.7
deepseek-v3.1	144.30	1.08	1956	73.8	0.6
TOTAL	**5605.53**	**59.62**			