SWE-CYCLE: BENCHMARKING CODE AGENTS ACROSS THE COMPLETE ISSUE RESOLUTION CYCLE

Anonymous authors
Paper under double-blind review

ABSTRACT

As autonomous code agents move toward end-to-end software development, evaluating their practical autonomy becomes critical. Current benchmarks hide friction by testing agents in pre-configured environments, and their static evaluation pipelines frequently fail when parsing fully autonomous trajectories. SWE-Cycle addresses these limitations with 489 rigorously filtered instances. It evaluates agents across three isolated tasks, including environment reconstruction, code implementation, and verification test generation, as well as an end-to-end FullCycle task that integrates all three. In FullCycle, one agent starts from a bare repository and carries its own environment, code, tests, and session state across the issue-resolution process. To reliably assess these complex execution paths, we developed SWE-Judge. By combining static code review with dynamic testing, this execution-capable evaluator reliably assesses functional correctness and corrects systematic misjudgments introduced by rigid script-based evaluation pipelines. We evaluate code agents powered by seven state-of-the-art LLMs across these four tasks. The results reveal a sharp drop in solve rates when transitioning from isolated tasks to FullCycle execution, exposing critical bottlenecks in handling cross-phase dependencies and maintaining code quality. Together, SWE-Cycle and SWE-Judge provide a comprehensive framework for accurately measuring the end-to-end capabilities of autonomous software agents.[1]

1 INTRODUCTION

The continuous enhancement of large language models (LLMs) in tool invocation and complex reasoning (Anthropic, 2025a; GLM-5 Team, 2026), alongside the rapid evolution of agentic frameworks (OpenCode Contributors, 2026; Anthropic, 2025b; OpenAI, 2025), has driven significant progress in autonomous code agents. Modern code agents have transitioned from completing isolated algorithmic functions to maintaining entire engineering-level projects. By navigating large legacy codebases and synthesizing cross-file edits, they can now execute continuous development and integrate into real-world software pipelines (Liu et al., 2025b). As these code agents evolve into autonomous developers, their expanding capabilities require robust evaluation paradigms that accurately capture their practical autonomy.

Existing benchmarks evaluate code agents through three largely separate paradigms: issue-resolution benchmarks test patches in existing repositories (Jimenez et al., 2024; Chowdhury et al., 2024; Deng et al., 2025; Zan et al., 2025); greenfield benchmarks cover system design, implementation, and testing (Golnari et al., 2026; Zeng et al., 2025); and task-specific benchmarks isolate environment reconstruction (Eliseeva et al., 2025) or test generation (Wang et al., 2025). Although execution-based metrics verify functional correctness, these settings do not capture autonomous execution across the complete issue resolution lifecycle and often rely on brittle, static evaluation pipelines.

Existing benchmarks fail to reflect an agent's true autonomy in real-world software maintenance. They structurally fragment the development pipeline and bypass the complexity of legacy codebases (Zeng et al., 2025; Golnari et al., 2026; Ding & Others, 2026). Successfully resolving a real-world issue requires a unified progression: reconstructing the execution environment, implementing the fix, and generating verification tests. When these interrelated phases are artificially isolated (Eliseeva et al.,

[1]Code is available at `https://anonymous.4open.science/r/SWE-Cycle-462E/`.

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.

2025; Wang et al., 2025), evaluations shield code agents from cascading errors and the friction of full-process integration, such as dependency resolution and version control navigation (Cemri et al., 2025). Consequently, current benchmarks project an illusion of capability: high performance on pre-configured tasks severely overestimates true autonomy, masking the reality that agents frequently fail completely when forced to onboard codebases, implement fixes, and verify their work without intervention.

Moreover, fixed execution pipelines based on predefined unit tests and rigid parsers can introduce measurement errors in open-ended workflows. Predefined tests may contain misaligned checks or underspecified assertions (Chowdhury et al., 2024; Zhu et al., 2025), while strict parsers can misclassify correct outputs because of formatting or toolchain variation. Binary scripts can therefore reject valid alternatives, miss uncovered defects, or conflate infrastructure failure with solution failure. FullCycle further lacks a fixed interface for all solver-generated artifacts, motivating an evaluator that can inspect the submitted state and adapt its execution procedure.

To address these limitations, we present **SWE-Cycle**, a benchmark evaluating code agents across the complete issue resolution lifecycle. We construct this benchmark from SWE-bench Verified, Pro, and Multilingual, rigorously filtering them to retain 489 high-quality instances. Each instance covers environment reconstruction, code implementation, and verification test generation under two settings: the Isolated setting evaluates each phase independently with the necessary inputs, and the FullCycle setting requires the agent to complete all three autonomously from only a bare repository and an issue description.

To overcome the structural failures of rigid execution protocols, we further propose **SWE-Judge**, a hybrid evaluation agent integrating static code review with dynamic execution. Moving beyond the binary pass/fail metrics of predefined scripts, SWE-Judge employs task-specific validation protocols to accommodate diverse valid implementations, uncover structural defects, and capture fine-grained partial correctness. We systematically evaluate code agents powered by seven state-of-the-art LLMs, reporting comprehensive results across both the isolated tasks and the FullCycle task. Our empirical analysis reveals that even within the isolated tasks, traditional deterministic script evaluation produces severe misjudgments and false signals. In contrast, SWE-Judge provides a reliable assessment protocol across all four SWE-Cycle tasks and is strongly validated against human annotations. Ultimately, SWE-Cycle and SWE-Judge establish a unified benchmark and evaluation framework for assessing code agents across the complete issue resolution lifecycle.

In summary, our key contributions are as follows:

- We introduce **SWE-Cycle**, the first benchmark evaluating agents across the complete issue resolution lifecycle. Curated via a rigorous filtering pipeline to retain 489 high-quality instances, it supports both an Isolated setting that evaluates each phase independently with the necessary inputs and a FullCycle setting that requires agents to operate autonomously from a bare repository. This paired design diagnoses how capabilities change when one agent must preserve and coordinate its own artifacts across phases.
- We propose **SWE-Judge**, a reference-assisted evaluation protocol for these open-ended artifacts. It combines static review, dynamic execution, and partial scoring to accommodate valid alternatives, identify structural defects, and distinguish partial correctness from binary failure.
- Our systematic evaluation of seven LLMs across both the isolated tasks and the FullCycle task establishes a comprehensive capability profile. It explicitly measures model proficiency in environment reconstruction, code implementation, and verification test generation to provide a holistic view of their true autonomous potential.

2 Related Work

Code Agent Benchmarks. SWE-bench (Jimenez et al., 2024) has established the standard for evaluating code agents on real-world GitHub issues. This benchmark has spurred the rapid development of specialized agent architectures (Yang et al., 2024; Wang et al., 2024). Subsequent variants refine the evaluation along multiple dimensions: human-verified instance quality (Chowdhury et al., 2024), longer-horizon tasks (Deng et al., 2025), multilingual coverage (Zan et al., 2025), continuous updates (Gu et al., 2025), heterogeneous comprehensive tasks (Xu et al., 2025), and large-scale training data synthesis (Yang et al., 2025). Despite these diverse advances, all SWE-bench variants

Table 1: Comparison between SWE-Cycle and current software engineering benchmarks.

Benchmark	Scenario	Task			End-to-End	Evaluation Protocol	
		Env.	Impl.	TestGen.		Judger	Partial Scoring
SWE-bench & Variants	Issue	✗	✓	✗	✗	UnitTest	✗
EnvBench	Env Setup	✓	✗	✗	✗	UnitTest	✗
TestEval	Existing Code	✗	✗	✓	✗	UnitTest	✗
DevBench	Greenfield	✗	✓	✓	✗	UnitTest+LLM	✗
PRDBench	Greenfield	✗	✓	✗	✓	Agent	✓
NL2Repo	Greenfield	✗	✗	✗	✓	UnitTest	✗
SWE-Cycle (Ours)	Issue	✓	✓	✓	✓	Agent	✓

inherit the same structural limitation: they supply pre-built Docker environments and evaluate agents against fixed gold test suites. This design entirely excludes environment reconstruction and test generation from the evaluation scope.

Alternative benchmarks address orthogonal aspects of software engineering, evaluating agents on greenfield project creation (Zeng et al., 2025), feature development (Li et al., 2025; Zhou et al., 2026), or automated code review (Zhang et al., 2026). However, greenfield development differs fundamentally from maintaining legacy systems; real-world engineering predominantly involves navigating complex existing codebases rather than starting from empty directories. Meanwhile, EnvBench (Eliseeva et al., 2025) and TestEval (Wang et al., 2025) isolate environment reconstruction and test generation, respectively. DevBench (Golnari et al., 2026), PRDBench (Fu et al., 2025), and NL2Repo (Ding & Others, 2026) instead target greenfield or repository-generation settings with different notions of end-to-end completion and evaluation. Consequently, no existing benchmark captures environment reconstruction, issue-driven implementation, and verification test generation within a single repository-maintenance trajectory.

Prior systems use execution feedback or explicit testing stages to improve code generation and multi-phase development (Le et al., 2022; Qian et al., 2024). SWE-Cycle instead studies these stages as an evaluation protocol for repository maintenance, preserving one agent's artifacts and state throughout a complete issue-resolution trajectory.

Automated Evaluation Approaches. Existing code agent benchmarks predominantly rely on unit test execution, which suffers from well-documented limitations: flawed gold tests, binary pass/fail scoring that discards partial correctness, and inapplicability when agents must generate their own verification code (Chowdhury et al., 2024). LLM-as-a-judge offers an alternative approach that assesses output quality on continuous scales without requiring gold references (Zheng et al., 2024; Gu et al., 2024). However, systematic studies reveal significant reliability concerns, including position bias that causes agreement fluctuations of up to 14% (Raina et al., 2025) and the fundamental limitations of execution-free judges when verifying runtime behavior (Liu et al., 2025a). To address these shortcomings, Agent-as-a-Judge represents an emerging paradigm that augments LLM judges with agentic capabilities, including planning, tool use, and iterative verification (You et al., 2026). Recent works demonstrate that such agentic evaluators, especially when fine-tuned, substantially outperform pure LLM-as-a-judge methods in both human alignment and cost efficiency (Zhuge et al., 2025; Fu et al., 2025). SWE-Cycle adopts this paradigm through SWE-Judge, an execution-capable evaluation agent that combines static code review with dynamic test execution to overcome the limitations of execution-free assessment.

Comparison with Existing Benchmarks. Table 1 summarizes the structural distinctions between current benchmarks. SWE-Cycle uniquely enforces the full issue resolution cycle while evaluating outputs through an execution-aware, fine-grained judge.

3 METHODOLOGY

To evaluate code agents across the software development lifecycle, we construct SWE-Cycle, a benchmark targeting the full issue resolution process. As detailed in Figure 1, the framework consists of three core components: dataset filtering for observable quality risks, task formulation separating isolated capabilities from end-to-end execution, and hybrid evaluation via SWE-Judge. By combining static analysis with dynamic execution, this architecture overcomes the brittleness of predefined scripts, ensuring robust evaluation of autonomous workflows.

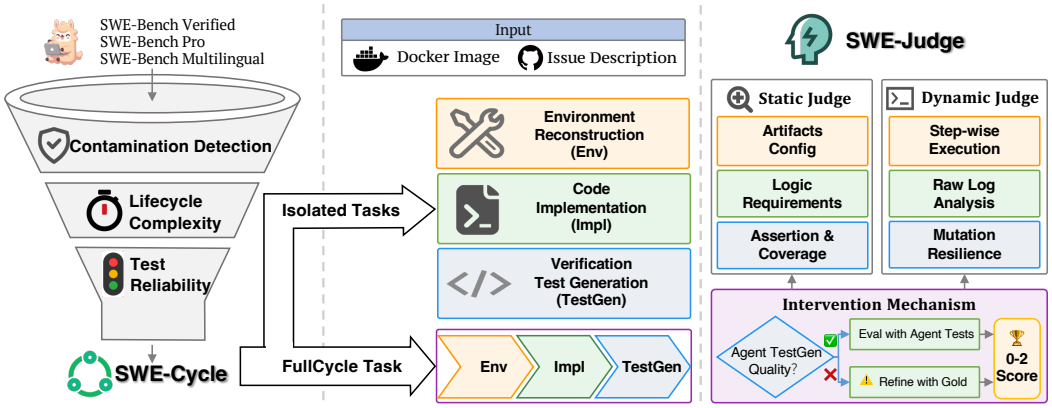


Figure 1: **Overview of the SWE-Cycle framework.** *Left:* High-quality instances are curated through a rigorous filtering pipeline. *Center:* Agents execute environment reconstruction, code implementation, and test generation in either isolated tasks or the FullCycle task. *Right:* SWE-Judge evaluates outputs via hybrid static-dynamic analysis and a test intervention mechanism to yield a robust 0-2 score.

3.1 Dataset Curation

To evaluate code agents on issue resolution tasks, we source 1,531 initial instances from three established datasets: SWE-bench Verified (Chowdhury et al., 2024), SWE-bench Pro (Deng et al., 2025), and SWE-bench Multilingual (Zan et al., 2025). However, the raw instances across these benchmarks suffer from data contamination, trivial task complexity, and invalid tests. To ensure the high quality required for assessing the complete issue resolution lifecycle, we design a three-stage filtering pipeline that resolves these flaws, distilling the initial pool into 489 rigorous instances.

Contamination Detection. Recent evidence confirms severe memorization in existing benchmarks Liang et al. (2025). To minimize the risk of training-data leakage, we prompt a recent LLM to solve each task without providing the issue description. Successful resolution under this condition is treated as a high-probability signal of contamination, and the corresponding instance is excluded. This step removes 128 instances.

Lifecycle Complexity Filtering. To ensure that SWE-Cycle better reflects real-world software development rather than instantaneous, isolated edits, we require each pull request to contain at least one code review comment and its resolution cycle to span at least one day. After this step, the data pool is reduced to 523 instances.

Test Reliability Filtering. Because our experiments compare against script-based scores, we validate the gold tests to reduce threats to the rigor of our conclusions from known test-suite defects and dependency rot (OpenAI, 2026b; Yu et al., 2025). For each instance, `FAIL_TO_PASS` tests must fail and `PASS_TO_PASS` tests must pass in the buggy state, while both must pass in the fixed state. This check removes 34 instances with corrupted test behavior.

Ultimately, SWE-Cycle yields 489 instances: 225 from SWE-bench Verified, 203 from SWE-bench Pro, and 61 from SWE-bench Multilingual. The detailed results of each filtering stage are summarized in Appendix A.

3.2 Task Formulation

In SWE-Cycle, we decompose the issue resolution lifecycle into three isolated tasks:

- **Environment Reconstruction (Env)** task evaluates the environment configuration capabilities of code agents. Given only the source code, code agents must independently build the execution environment and resolve all dependencies within a Docker container.
- **Code Implementation (Impl)** task assesses issue-driven development capabilities. Given an issue description and a preconfigured codebase, agents must modify the repository to implement the requested changes.

- **Verification Test Generation (TestGen)** task measures the capability to design effective unit tests. Given an issue description and a patched codebase, agents must generate tests targeting the specified issue.

We also integrate these phases into an end-to-end task, **FullCycle**, to simulate a realistic developer workflow. Given an issue description and a raw codebase, the agent must handle environment setup, code implementation, and test generation within a single autonomous session. This paired design enables a controlled comparison of performance between the Isolated and FullCycle settings, revealing how capabilities change when one agent must preserve and coordinate its own artifacts across phases without step-by-step guidance.

3.3 SWE-JUDGE

We introduce SWE-Judge to score the aforementioned tasks by combining static code review with dynamic execution. SWE-Judge applies task-specific criteria to yield independent Static and Dynamic scores, each ranging from 0 to 2. Static review assesses whether the submitted artifacts encode a complete and reproducible solution: it examines environment configurations and setup scripts, implementation logic and change scope, and the quality and coverage of test assertions. Dynamic execution evaluates behavior observed in the current container by activating and validating the environment, running functional and regression tests, and checking whether generated tests distinguish the buggy and fixed states. To support a more accurate task-specific assessment, SWE-Judge is reference-assisted and may consult the issue description, gold patch, and gold tests associated with each benchmark instance. The evaluation pipeline for each task is outlined below. Detailed scoring rubrics are available in Appendix B.

Env Task Evaluation. Environment failures can occur even when configuration artifacts appear complete. SWE-Judge therefore combines static checks for missing dependencies and version conflicts with three-stage dynamic validation: environment and toolchain activation; project import/build and test collection; and successful execution of benchmark-specified target and regression tests in the gold-patched repository. This protocol assesses runtime reproducibility while localizing setup failures.

Impl Task Evaluation. Script-based evaluation can miss logical errors outside fixed test coverage. SWE-Judge extracts the root cause, expected behavior, and critical edge cases from the issue and gold patch, then statically assesses the submitted changes against these requirements. During dynamic validation, it runs the benchmark-specified target and regression tests and interprets their raw logs rather than relying only on rigid parser outputs. This combination evaluates both implementation logic and observed behavior.

TestGen Task Evaluation. Our script baseline follows the conventional two-state protocol: submitted tests must fail on the buggy repository and pass after the gold patch is applied. This binary condition is easily exploited and ignores test quality metrics like assertion accuracy and scenario coverage. To address this, SWE-Judge supplements dynamic execution with static analysis: it evaluates assertion quality against the gold tests and maps scenario coverage to verify critical execution paths. This dual verification ensures the generated tests genuinely isolate the target bug rather than exploit evaluation loopholes.

FullCycle Task Evaluation. Because FullCycle produces solver-defined artifact locations and interfaces, it has no directly equivalent fixed-script baseline. SWE-Judge applies the preceding task-specific evaluation protocols sequentially to Env, TestGen, and Impl, assigning zero to subsequent Dynamic scores when environment setup fails. When the submitted tests are insufficient for reliable evaluation, SWE-Judge may refine them using gold tests. In practice, SWE-Judge uses improved tests in 43.8% of evaluations and writes an additional custom evaluation script in 34.6% of evaluation trajectories when assessing Code Dynamic. The detailed pipeline and scoring criteria are provided in Appendix B, and Appendix C.1 presents a representative intervention case.

4 EXPERIMENTS

In this section, we evaluate code agents powered by seven state-of-the-art LLMs on SWE-Cycle to quantify their capabilities across the full issue-resolution lifecycle and validate the reliability of

SWE-Judge. Specifically, our experiments, including an ablation study of SWE-Judge, are structured around four core research questions:

- **RQ1:** How reliable is the evaluation produced by SWE-Judge?
- **RQ2:** How do evaluated code agents perform across the four SWE-Cycle tasks?
- **RQ3:** How does SWE-Judge overcome the limitations of traditional script-based evaluations?
- **RQ4:** How do agent behaviors differ when resolving an issue end-to-end versus step-by-step?

4.1 Experimental Setup

We evaluate code agents powered by seven state-of-the-art LLMs spanning both proprietary and open-weight families: GPT-5.4 (OpenAI, 2026a), Claude-Sonnet-4.6 (Anthropic, 2025a), Qwen-3.5 (Qwen, 2026), GLM-5.1 (GLM-5 Team, 2026), Kimi-K2.5 (Kimi, 2025), MiniMax-M2.7 (MiniMax, 2026), and Claude-Fable-5 (Anthropic, 2026). We use the same open-source OpenCode[2] harness for all evaluated models, holding the system prompt, tool interface, container protocol, and execution strategy constant to support fair comparison and reproducibility. Each task executes within an isolated Docker container. To ensure a fair comparison while accounting for FullCycle's broader scope, each isolated task receives the required context and a 90-minute budget, whereas FullCycle starts from a bare repository and receives a single extended 180-minute budget for all three phases. Generated artifacts are evaluated even after solver timeout.

To balance assessment depth with computational cost, we use Claude-Opus-4.5 (Anthropic, 2025) as the SWE-Judge backbone. Appendix D reports a judge-model robustness ablation that re-evaluates identical agent outputs with GPT-5.4 and checks both score stability and same-family bias. The evaluation agent runs in the same container to inherit the solving agent's environment and artifacts. To support reproducibility, the complete SWE-Cycle and SWE-Judge codebase, experiment configurations, judge prompts, evaluation trajectories, and representative execution logs are available via SWE-Cycle's anonymous repository.

Metrics. We measure agent performance in SWE-Cycle using four metrics. *Static* assesses structural correctness or static analysis results without execution. *Dynamic* (*Dyn.*) measures functional correctness through actual execution. Both are initially scored as 0, 1, or 2, then normalized to a 0–1 scale. *Score* averages the *Static* and *Dynamic* results; for the FullCycle setting, this is macro-averaged across all three phases. Finally, *Solve* denotes the perfect resolution rate, representing the fraction of instances where *Score* equals 1. All metrics are reported as percentages.

Script baselines. For Env and Impl, we use the native task-specific evaluation pipelines provided by the source benchmarks. These pipelines typically execute benchmark-specified FAIL_TO_PASS and, where applicable, PASS_TO_PASS tests, while their final binary outcomes may also depend on additional harness logic, artifact checks, patch application, execution wrappers, and output parsing. For TestGen, we follow the standard two-state protocol: submitted tests must fail on the buggy repository and pass after the gold patch is applied. FullCycle has no directly equivalent fixed-script baseline.

4.2 Reliability of SWE-Judge (RQ1)

To audit SWE-Judge, we conduct stratified human checks across all four tasks: 20% of the Script–Judge disagreement cases and 5% of the agreement cases for the isolated tasks, and one submission for each of the 489 FullCycle issues, spanning all seven evaluated models. Three professional annotators who are independent of the author team conduct the audit. They first annotate a pilot set independently and proceed to full-scale annotation after achieving 97.9% raw agreement. Across the final annotation dimensions, the minimum Fleiss' κ is 0.88 and the minimum ordinal Krippendorff's α is 0.92, indicating high inter-annotator reliability.

Table 2: Human–SWE-Judge alignment across four tasks. N is the number of sampled instances.

Task	N	Alignment%
Env	143	99.3
Impl	113	95.6
TestGen	201	99.5
FullCycle	489	96.9

[2] https://github.com/anomalyco/opencode/releases/tag/v1.4.6

Table 3: Leaderboard on isolated tasks. Best results are in **bold**, and second best are underlined.

Model	Env				Impl				TestGen			
	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve
Claude-Sonnet-4.6	**90.59**	**84.97**	**87.78**	**78.12**	68.00	57.16	62.58	40.08	91.00	85.79	88.39	67.28
GLM-5.1	87.73	76.48	82.11	73.01	65.95	53.68	59.82	37.83	85.89	82.82	84.36	60.53
GPT-5.4	87.83	74.44	81.13	71.78	69.12	51.84	60.48	39.67	74.44	76.38	75.41	42.13
Kimi-K2.5	81.09	68.80	74.95	61.34	64.38	51.06	57.72	34.46	80.79	82.44	81.61	52.07
MiniMax-M2.7	56.13	60.94	58.54	45.40	52.86	43.46	48.16	30.88	52.66	55.11	53.89	33.33
Qwen-3.5	82.62	75.05	78.83	66.46	56.24	43.35	49.80	28.83	72.90	76.89	74.90	46.01
Claude-Fable-5	86.32	76.86	81.59	74.32	**88.90**	**83.29**	**86.25**	**80.29**	**93.69**	**85.98**	**89.84**	**82.07**

Table 4: Leaderboard on FullCycle task. Best results are in **bold**, and second best are underlined.

Model	Env		Impl		TestGen		Score	Solve
	Static	Dyn.	Static	Dyn.	Static	Dyn.		
Claude-Sonnet-4.6	84.56	**97.55**	61.35	**89.67**	68.61	81.39	80.52	12.27
GLM-5.1	87.93	96.73	61.76	89.37	70.35	**82.82**	81.49	13.50
GPT-5.4	87.22	93.15	62.88	88.96	65.95	81.49	79.94	10.84
Kimi-K2.5	65.88	89.81	58.91	80.90	56.22	71.35	70.51	2.15
MiniMax-M2.7	46.01	55.83	31.70	43.35	26.38	35.79	39.84	4.29
Qwen-3.5	85.07	94.89	56.24	80.06	56.34	74.03	74.44	6.75
Claude-Fable-5	**91.44**	87.95	**85.20**	81.29	**80.34**	75.16	**81.92**	**60.58**

As shown in Table 2, SWE-Judge aligns with human judgment in over 95% of cases across all tasks. This confirms that SWE-Judge can reliably evaluate both isolated tasks and open-ended FullCycle resolutions. Appendix E details the human-audit protocol and results.

4.3 Main Results (RQ2)

Isolated Tasks. Table 3 reports performance on the three isolated tasks. Across these tasks, TestGen reaches the highest solve rate, 82.1%, while Impl remains the primary bottleneck for most models. Notably, Claude-Fable-5 is strong on Impl and TestGen but weaker on Env because it does not consistently validate repository-native execution paths (Appendix F). Static and Dynamic scores differ in both magnitude and direction across models and tasks, showing that the two evaluations capture complementary failure modes. Dynamic execution measures observed behavior, whereas Static review identifies deficiencies in artifact completeness, structural quality, and reproducibility that successful runtime outcomes may not expose.

FullCycle Task. Table 4 presents results for the end-to-end FullCycle task, revealing key findings about complete issue resolution, cross-phase execution performance, and static artifact quality.

- FullCycle Solve falls below isolated-task Solve for all seven models and varies widely across models. Claude-Fable-5 leads in Score (81.92%) and Solve (60.58%). Three other models also achieve average Scores around 80%, yet their Solve rates remain below 14%. This reveals that strong performance on individual phases or high average scores across lifecycle dimensions does not ensure complete autonomous issue resolution, as reliable autonomy requires an agent to maintain the joint correctness of its environment, implementation, and verification tests throughout the same issue-resolution trajectory.
- Env and Impl Dynamic scores improve over isolated tasks for most models. This is because generating and running tests provides feedback for repairing code and environment defects, consistent with RQ4's phase-removal results. TestGen Dynamic, however, declines for five of seven models because FullCycle TestGen relies on the agent's own issue understanding and implementation rather than the reference-patched codebase used in isolated TestGen.
- Static scores decline for most models in FullCycle relative to the corresponding isolated tasks despite these Dynamic gains. Agents often generate submissions that hack the corresponding tests to pass runtime execution, but static analysis identifies the resulting implementation flaws. Claude-Fable-5 nevertheless maintains high static code and artifact quality in FullCycle.

To further analyze this FullCycle gap, we categorize unsuccessful instances in Figure 2. Compound errors dominate: simultaneous losses in implementation and test generation, or across all three phases, are more common than isolated single-phase failures. This pattern shows that lifecycle failures commonly span phase boundaries rather than arising from a single independent task. The co-occurrence is consistent with cross-phase propagation but does not prove it, because common issue difficulty and evaluator gating can also correlate phase outcomes. The phase-removal study in RQ4 provides complementary intervention evidence.

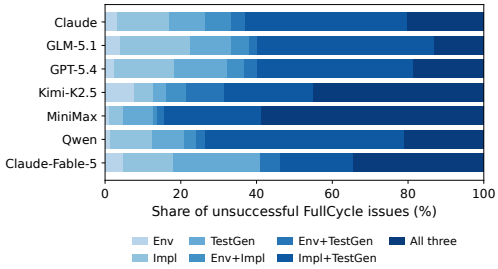


Figure 2: Distribution of failure categories across models in the FullCycle task.

Taken together, these findings show that FullCycle measures an integrated lifecycle capability beyond the sum of isolated-task scores: coordinating interdependent phases while maintaining the joint correctness of solver-generated environments, code, tests, and state throughout an issue-resolution trajectory. The sharp decline in Solve therefore exposes cross-phase dependencies and code-quality bottlenecks that isolated evaluation obscures. Appendices F and G report per-dataset results and efficiency metrics, respectively.

4.4 Effectiveness of SWE-Judge (RQ3)

SWE-Judge vs. Script Evaluation. We compare SWE-Judge with the task-specific binary script baselines defined in Section 4.1 and use human adjudication to analyze their disagreements.

Within the sampled Script–Judge disagreement cases, human adjudication supports SWE-Judge in 98.6% of cases, including 100% in the audited TestGen and Env disagreements; scripts are supported in 0.5% (Table 5). The script failures fall into excessive strictness (36.0%), evaluation breakdown (32.8%), excessive leniency (27.0%), and other cases (4.2%). Appendix H presents the detailed definitions and cases, evidence for Static–Dynamic complementarity, and representative SWE-Judge overrating and underrating cases.

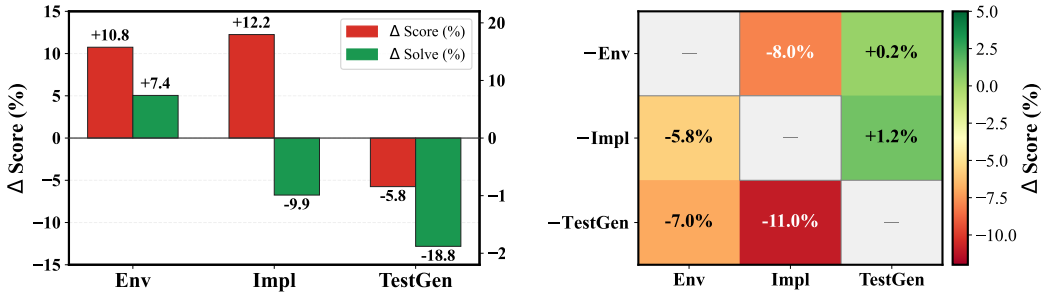
Adaptive Evaluation Workflow. We analyze SWE-Judge trajectories across valid FullCycle evaluations. Table 6 reports average action counts and coverage. SWE-Judge anchors on code diffs and reference comparisons, then selectively runs tests and build checks. It writes an additional custom evaluation script in 34.6% of trajectories and uses fault injection in 4.8%. As described in Section 3.3, improved tests are used for Code Dynamic in 43.8% of evaluations. Appendix C presents representative workflows. Appendix I separately studies reference-patch visibility.

Table 5: Human adjudication of 371 disagreements between SWE-Judge and script-based metrics. Percentages denote how often humans agreed with each method.

Task	N	SWE-Judge	Script	Neither
Impl	93	94.6%	2.2%	3.2%
TestGen	173	100.0%	0.0%	0.0%
Env	105	100.0%	0.0%	0.0%
Overall	371	98.6%	0.5%	0.8%

4.5 End-to-End vs. Isolated Evaluation (RQ4)

To quantify how end-to-end integration reshapes model behavior, we evaluate 489 instances paired across FullCycle and isolated tasks. Figure 3a visualizes the performance shift by plotting the absolute difference (Δ = FullCycle − Isolated) in average score and overall solve rate across the three tasks. The results show a straightforward trend: integration improves upstream performance but degrades downstream metrics. Upstream Env benefits for most models because subsequent runtime execution exposes configuration defects that models then return to fix. Midstream Impl gains average dynamic functionality through this iterative write-run-fix loop, but the continuous patching compromises static structural quality, dropping the solve rate. Downstream TestGen degrades across all metrics. This occurs because models often hack the verification step by writing trivial tests that simply pass their own implementations, aiming to terminate the task as quickly as possible rather than rigorously testing the code.



(a) (b)

Figure 3: End-to-end integration effects. (a) Per-dimension score and solve rate change (Δ = FullCycle − Isolated) across three tasks. (b) Score degradation of remaining phases when one phase is removed from the FullCycle task.

To measure the inter-dependency of these tasks, Figure 3b maps a phase-removal experiment. We systematically remove a single phase from the FullCycle setup and record the resulting score changes across the remaining active phases. The results show that the stages in FullCycle are inter-dependent rather than independently composable. Most notably, removing the downstream TestGen phase reduces Env and Impl scores by 7.0% and 11.0%, respectively, demonstrating that verification provides feedback that helps agents improve earlier artifacts. This finding highlights verification capability as an important driver of end-to-end performance and shows that FullCycle captures an orchestration capability that isolated tasks do not measure. Appendix J provides the ablation setup and per-dataset results.

Table 6: SWE-Judge action distribution across FullCycle evaluations.

Action	# Avg	Coverage
Code Review (git diff)	9.7	100%
Reference Comparison (Read)	7.9	100%
Env & File Inspection	16.5	99.9%
Test Execution	1.6	45.7%
Build verification	0.4	36.1%
Adaptive eval scripting	—	34.6%
Fault injection	—	4.8%
Total steps	39.5	—

5 Conclusions & Limitations

SWE-Cycle evaluates code agents across the full issue resolution lifecycle, addressing challenges in measuring their end-to-end autonomy. Alongside this benchmark, SWE-Judge integrates static and dynamic analysis to provide comprehensive and reliable assessments. Human validation and case studies show that SWE-Judge corrects systematic failure modes in traditional script-based evaluation, enabling accurate and scalable assessment across complex software engineering tasks. Using this framework, we uncover a critical blind spot in current LLM evaluation: optimizing localized accuracy in isolated tasks does not yield full-lifecycle autonomy. During end-to-end integration, cross-phase feedback boosts dynamic correctness while accumulating structural debt, which ultimately degrades downstream test generation. Together, SWE-Cycle and SWE-Judge provide a full-lifecycle standard for evaluating code agents, shifting attention from isolated code generation toward global planning and long-term maintainability.

Limitations. SWE-Cycle is currently instantiated primarily with SWE-bench-style GitHub issues, but its central contribution is the FullCycle evaluation formulation, which can be readily extended to other repository-level benchmarks to assess agents' end-to-end issue-resolution autonomy across more diverse software-engineering scenarios. We use the same OpenCode harness for all evaluated models to support fairness and reproducibility. SWE-Judge employs detailed, task-specific rubrics to reduce evaluation variability, and our judge-model ablation shows limited sensitivity to the choice of evaluator backbone.

AI USE STATEMENT

Generative AI systems were used to solve and evaluate benchmark instances as disclosed components of the experimental methodology: the evaluated language models serve as code-agent backbones, and the SWE-Judge backbone supports reference-assisted assessment. Separately, we used generative AI tools to assist with English-language editing and LaTeX formatting and debugging during manuscript preparation. We did not use these authoring tools to determine the research questions, experimental design, benchmark inclusion decisions, reported measurements, or scientific conclusions. All AI-assisted edits were reviewed by the authors, and all technical claims, citations, numerical results, and rendered artifacts were checked against the underlying sources and experimental records. The authors take responsibility for the final content of the work, including all text and artifacts produced with AI assistance.

ETHICS STATEMENT

SWE-Cycle is intended to improve the measurement of code-agent capabilities rather than to establish that such systems are ready for unsupervised deployment. Reporting isolated-task performance without the corresponding FullCycle limitations could overstate practical autonomy, while generated patches or tests may introduce functional, security, licensing, or maintenance risks if used without human review. The benchmark is constructed from issues and repositories in existing public software-engineering benchmarks; downstream users should preserve source attribution and license obligations and should execute untrusted agent-generated artifacts only in appropriately isolated environments. The human audit was conducted by professional annotators who were independent of the author team, worked no more than eight hours per day, and were compensated in compliance with local labor regulations. The released materials are intended for research evaluation and should not be interpreted as a substitute for security review, license review, or accountable human oversight in real software systems.

REPRODUCIBILITY STATEMENT

The paper and anonymous supplementary repository support reproducibility by documenting and releasing the public-data filtering pipeline, task interfaces, evaluation rubrics, execution budgets, prompts, configurations, analysis scripts, trajectories, and execution logs used in this study.

REFERENCES

Anthropic. Claude 4.6 sonnet system card. Technical report, Anthropic, 2025a. URL https://www-cdn.anthropic.com/78073f739564e986ff3e28522761a7a0b4484f84.pdf.

Anthropic. Claude code, 2025b. URL https://github.com/anthropics/claude-code.

Anthropic. Introducing claude opus 4.5. Anthropic Blog, 2025. URL https://www.anthropic.com/news/claude-opus-4-5.

Anthropic. Claude fable 5 and claude mythos 5. Anthropic announcement, June 2026. URL https://www.anthropic.com/news/claude-fable-5-mythos-5.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, et al. Why do multi-agent LLM systems fail?, 2025. URL https://arxiv.org/abs/2503.13657.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified. OpenAI Blog, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

DeepSeek-AI and Others. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Jingzhe Ding and Others. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents, 2026. URL https://arxiv.org/abs/2512.12730.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. EnvBench: A benchmark for automated environment setup, 2025. URL https://arxiv.org/abs/2503.14443.

Lingyue Fu, Bolun Zhang, Hao Guan, Yaoming Zhu, Lin Qiu, Weiwen Liu, Xuezhi Cao, Xunliang Cai, Weinan Zhang, and Yong Yu. Automatically benchmarking llm code agents through agent-driven annotation and evaluation, 2025.

GLM-5 Team. Glm-5: From vibe coding to agentic engineering. arXiv preprint arXiv:2602.15763, 2026.

Pareesa Ameneh Golnari, Adarsh Kumarappan, Wen Wen, Xiaoyu Liu, Gabriel Ryan, Yuting Sun, Shengyu Fu, and Elsie Nallipogu. Devbench: A realistic, developer-informed benchmark for code generation models, 2026. URL https://arxiv.org/abs/2601.11895.

Alex Gu, Naman Jain Liu, Nikhil Thakur, Wen-Ding Shi, Dídac Suris, Sanjay Jain, Naomi Saphra, Celine Lee Xia, Graham Neubig, and Aditi Raghunathan. SWE-bench goes live!, 2025. URL https://arxiv.org/abs/2505.23419. NeurIPS 2025 Datasets and Benchmarks Track.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. A survey on LLM-as-a-judge, 2024. URL https://arxiv.org/abs/2411.15594.

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

Kimi. Kimi k2.5: Scaling reinforcement learning with llms. Kimi Blog, 2025. URL https://www.kimi.com/blog/kimi-k2-5.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In Advances in Neural Information Processing Systems, 2022.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. In Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL), pp. 17160–17176, 2025.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The swe-bench illusion: When state-of-the-art llms remember instead of reason, 2025. URL https://arxiv.org/abs/2506.12286.

Xichang Liu et al. CodeJudgeBench: Benchmarking LLM-as-a-judge for coding tasks, 2025a. URL https://arxiv.org/abs/2507.10535.

Zhengyu Liu et al. AI-augmented CI/CD pipelines: From code commit to production with autonomous decisions, 2025b. URL https://arxiv.org/abs/2508.11867.

MiniMax. Minimax 2.7. MiniMax Blog, 2026. URL https://www.minimaxi.com/models/text/m27.

OpenAI. Introducing codex, 2025. URL https://openai.com/index/introducing-codex/.

OpenAI. Introducing GPT-5.4. OpenAI Blog, 2026a. URL https://openai.com/index/introducing-gpt-5-4/.

OpenAI. Why swe-bench verified no longer measures frontier coding capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/, February 2026b.

OpenCode Contributors. Opencode, 2026. URL https://github.com/opencode-ai/opencode.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024.

Qwen. Qwen3.5: Towards native multimodal agents. Qwen Blog, 2026. URL https://qwen.ai/blog?id=qwen3.5.

Vibhu Raina et al. Judging the judges: A systematic study of position bias in LLM-as-a-judge. In Proceedings of the International Joint Conference on Natural Language Processing (IJCNLP), 2025. URL https://arxiv.org/abs/2406.07791.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation, 2025. URL https://arxiv.org/abs/2406.04531.

Xingyao Wang, Boxuan Chen, Parker Adler, Zijian Cheng, Kexun Hu, Jieyu Li, Yuqi Li, Ziniu Liu, Yufan Lu, Jiasheng Ning, et al. OpenHands: An open platform for AI software developers as generalist agents, 2024. URL https://arxiv.org/abs/2407.16741.

Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, Kepeng Lei, Yifan Yao, Xinping Lei, Wenqiang Zhu, Zongxian Feng, Han Li, Junqi Xiong, Dailin Li, Zuchen Gao, Kun Wu, Wen Xiang, Ziqi Zhan, Yuanxing Zhang, Wuxuan Gong, Ziyuan Gao, Guanxiang Wang, Yirong Xue, Mengtong Li, Mengfei Xie, Xiaojiang Zhang, Jinghui Wang, Wenhao Zhuang, Zheng Lin, Huiming Wang, Zhaoxiang Zhang, Yuqun Zhang, Haotian Zhang, Bin Chen, and Jiaheng Liu. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models, 2025. URL https://arxiv.org/abs/2511.05459.

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

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798. NeurIPS 2025 Datasets and Benchmarks Track Spotlight.

Runyang You, Hongru Cai, Caiqi Zhang, et al. A survey on agent-as-a-judge, 2026. URL https://arxiv.org/abs/2601.05111.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. Utboost: Rigorous evaluation of coding agents on swe-bench. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL), 2025. URL https://arxiv.org/abs/2506.09289.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Zhengran Zeng, Yixin Li, Rui Xie, Wei Ye, and Shikun Zhang. Benchmarking and studying the LLM-based agent system in end-to-end software development, 2025. URL https://arxiv.org/abs/2511.04064.

Yuntong Zhang, Zhiyuan Pan, Imam Nur Bani Yusuf, Haifeng Ruan, Ridwan Shariffdeen, and Abhik Roychoudhury. Code review agent benchmark. arXiv preprint arXiv:2603.23448, 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, et al. Judging LLM-as-a-judge with MT-bench and chatbot arena, 2024. URL https://arxiv.org/abs/2306.05685.

Q Zhou, J Zhang, H Wang, R Hao, J Wang, M Han, Y Yang, S Wu, F Pan, L Fan, D Tu, and Z Zhang. Featurebench: Benchmarking agentic coding for complex feature development. arXiv preprint arXiv:2602.10975, 2026.

Yuxuan Zhu, Tian Jin, Yada Pruksachatkun, Aston Zhang, Shayne Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Richard Weiss, et al. Establishing best practices for building rigorous agentic benchmarks. arXiv preprint arXiv:2507.02825, 2025.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. In Proceedings of the 42nd International Conference on Machine Learning (ICML), 2025. URL https://arxiv.org/abs/2410.10934.

A Dataset Construction

A.1 Detailed Filtering Statistics

Table 7 reports the number of retained instances at each filtering stage. For contamination detection, we use DeepSeek-V3.2 (DeepSeek-AI & Others, 2025) to solve each task without the issue description. We consider successful resolution under this condition a high-probability contamination signal and exclude the corresponding instance.

Table 7: The SWE-Cycle filtering pipeline and the number of retained instances at each step.

Targeted Problem	**Verified**	**Pro**	**Multi**	Total Retained
Initial Source Pool	500	731	300	1,531
Contamination Detection	499	607	297	1,403
Lifecycle Complexity Filtering	239	209	75	523
Test Reliability Filtering	225	203	61	**489**

The contamination filter removes 1/500 Verified instances (0.2%), 124/731 Pro instances (17.0%), and 3/300 Multilingual instances (1.0%). We consider tasks that also satisfy the review-comment and elapsed-time criteria more representative of complex real-world issue-resolution work, where changes undergo review and unfold over nontrivial development cycles.

A.2 Language Distribution and Environment Setup

The final 489 instances span 9 programming languages: Python (68.9%), Go (18.0%), C (3.3%), Ruby (2.7%), JavaScript (2.5%), Rust (1.4%), TypeScript (1.2%), Java (1.2%), and PHP (0.8%). For Impl and TestGen tasks, each instance includes a pre-built Docker image with pinned runtimes and dependencies to ensure reproducible evaluation. In contrast, the Env task starts from a minimal base image containing only the OS and language runtime; the agent must independently resolve all project-level dependencies, build configurations, and toolchain setup.

B Scoring Rubric

This appendix details the scoring criteria used by SWE-Judge for each task type in SWE-Cycle. All task types except FullCycle use a two-dimensional rubric (Static Analysis + Dynamic Execution), each scored 0–2, yielding a maximum of 4 points. FullCycle uses three dimensions (Environment, Code, Test), each with a static and dynamic sub-dimension (0–2 each), yielding a maximum of 12 points. The final metric is the **score ratio** = total score / maximum achievable score. Detailed eval prompts are in GitHub repo.

B.1 Information Boundaries and Script Baselines

The solving agent and evaluator have different information. In FullCycle, the solver receives only the issue description and bare repository; it does not receive the gold patch, gold tests, judge rationale, or intervention output. To support a more accurate task-specific assessment, SWE-Judge may consult the issue description, gold patch, gold tests, and submitted artifacts. We therefore describe solving as open-ended and evaluation as reference-assisted.

The traditional baselines used in the disagreement analysis are task-specific binary evaluation pipelines. For Env, tests run on the gold-patched repository inside the submitted environment; Verified and Multilingual require all specified FAIL_TO_PASS tests to pass, while Pro also checks specified PASS_TO_PASS tests. Impl applies the submitted patch in a preconfigured environment and runs the specified target and regression tests. For both tasks, the final binary outcome may additionally depend on benchmark-specific harness logic, artifact checks, patch application, execution wrappers, and output parsing. TestGen follows the conventional two-state protocol: submitted tests must fail on the buggy repository and pass after the gold patch is applied. These baselines do not

assess configuration persistence, assertion semantics, or scenario coverage, and infrastructure/parser failures are recorded as failures. FullCycle has no directly equivalent fixed-script baseline because all three artifact interfaces are solver-generated.

B.2 ENV TASK

This task evaluates whether the agent correctly configured the required project dependencies and test runtime environment. Dynamic validation proceeds progressively through environment and toolchain activation, project import/build and test collection, and execution of the benchmark-specified target and regression tests in the gold-patched repository.

Table 8: Scoring rubric for the Environment task (max 4 points).

Dimension	Score	Criteria
Static Analysis	0	No environment configured: core dependencies completely absent; `setup.sh` missing or fundamentally broken.
	1	Environment created, core dependencies basically installed, but obvious shortcomings: missing dependencies, version incompatibility risk, `setup.sh` not reproducible.
	2	Environment complete: all necessary dependencies correctly installed, `setup.sh` clear and reproducible, no compatibility issues.
Dynamic Execution	0	Cannot activate the environment or required toolchain; or no specified test passes because import/build, collection, or execution fails.
	1	The environment activates and tests execute, but only some FAIL_TO_PASS tests pass or PASS_TO_PASS has regressions.
	2	Activation, import/build, and test collection succeed; all FAIL_TO_PASS tests pass and PASS_TO_PASS has no new failures.

B.3 IMPL TASK

This task evaluates whether the agent's code patch correctly and completely resolves the reported bug.

Table 9: Scoring rubric for the Development task (max 4 points).

Dimension	Score	Criteria
Static Code Analysis	0	Patch is empty or unrelated to the problem; or contains a fundamental logic error.
	1	Fix direction is correct but has obvious shortcomings: incomplete logic, or change scope exceeds what is necessary.
	2	Fix logic is correct and complete, change scope is reasonable, targets the root cause with no significant side effects.
Dynamic Execution	0	Tests cannot run; or FAIL_TO_PASS tests have failures.
	1	All FAIL_TO_PASS tests pass, but PASS_TO_PASS has new failures (regression).
	2	All FAIL_TO_PASS tests pass, PASS_TO_PASS has no new failures.

B.4 TESTGEN TASK

This task evaluates the quality, coverage, and effectiveness of the test cases written by the agent. The dynamic dimension uses a two-phase protocol: Phase 1 reverts the code fix (expect tests to **fail**); Phase 2 restores the fix (expect tests to **pass**).

Table 10: Scoring rubric for the TestCase task (max 4 points).

Dimension	Score	Criteria
Static Analysis	0	`eval.sh` missing; contains git operations; all meaningless assertions (`assert True`); content unrelated to the bug.
	1	Direction correct but quality insufficient: imprecise assertions, single scenario, low alignment with gold test scenarios; or tests merely mirror the code diff without additional insight.
	2	Good design: targets bug core behavior, meaningful assertions, covers main scenarios with good alignment to gold test coverage; demonstrates understanding beyond the diff (e.g., edge cases, boundary conditions).
Two-Phase Dynamic	0	Phase 1 passes on buggy code (cannot detect bug); or `eval.sh` cannot execute.
	1	Phase 1 fails but with imprecise cause (import error / setup failure / permanently-false assertion); or Phase 2 still fails.
	2	Phase 1 fails precisely (points to the bug), Phase 2 passes.

B.5 FULLCYCLE TASK

This task evaluates the agent's end-to-end performance. The scoring criteria aggregate the individual rubrics from the Environment, TestCase, and Development tasks, with execution dependencies following Algorithm 1. For ENV Dynamic, the same progressive structure is applied to the submitted FullCycle state: environment activation, project import/build and test collection, and execution of the relevant project tests.

Algorithm 1 SWE-Judge evaluation pipeline for the FullCycle task

Require: Agent submissions $A_{\text{env}}, A_{\text{test}}, A_{\text{impl}}$; gold tests T_{gold}
Ensure: Scores for each task $(S_{\text{env}}, S_{\text{test}}, S_{\text{impl}})$, including dynamic and static scores

$S^{\text{stat}}_{\text{env}}, S^{\text{stat}}_{\text{test}}, S^{\text{stat}}_{\text{impl}} \leftarrow \text{EVALSTATIC}(A_{\text{env}}, A_{\text{test}}, A_{\text{impl}})$ ▷ Static scores are always preserved
$S^{\text{dyn}}_{\text{env}} \leftarrow \text{EVALENVDYNAMIC}(A_{\text{env}})$
$S_{\text{env}} \leftarrow (S^{\text{stat}}_{\text{env}}, S^{\text{dyn}}_{\text{env}})$
if $S^{\text{dyn}}_{\text{env}} = 0$ **then** ▷ Halt: Upstream failure blocks execution
    return $\big(S_{\text{env}}, (S^{\text{stat}}_{\text{test}}, 0), (S^{\text{stat}}_{\text{impl}}, 0)\big)$
end if
$S^{\text{dyn}}_{\text{test}} \leftarrow \text{EVALTESTDYNAMIC}(A_{\text{test}})$
$S_{\text{test}} \leftarrow (S^{\text{stat}}_{\text{test}}, S^{\text{dyn}}_{\text{test}})$
if $S^{\text{stat}}_{\text{test}} < 2$ **then** ▷ Intervention: Refine agent tests
    $T_{\text{exec}} \leftarrow \text{REFINETESTS}(A_{\text{test}}, T_{\text{gold}})$
else
    $T_{\text{exec}} \leftarrow A_{\text{test}}$
end if
$S^{\text{dyn}}_{\text{impl}} \leftarrow \text{EVALIMPLDYNAMIC}(A_{\text{impl}}, T_{\text{exec}})$
$S_{\text{impl}} \leftarrow (S^{\text{stat}}_{\text{impl}}, S^{\text{dyn}}_{\text{impl}})$
return $(S_{\text{env}}, S_{\text{test}}, S_{\text{impl}})$

CODE_DYNAMIC uses the submitted tests directly when TEST_STATIC$=2$ (Algorithm 1); otherwise, to ensure an accurate evaluation of CODE_DYNAMIC, the evaluator may refine them using the gold tests. We refer to the resulting tests as improved tests. TEST_STATIC is capped below 2 when assertions are ineffective, coverage is below $\lceil 0.7N \rceil$ of the N core reference scenarios, or a critical boundary, error, or security path is omitted. Improved tests affect only CODE_DYNAMIC and cannot raise the submitted TEST scores. In practice, improved tests are used for CODE_DYNAMIC in 43.8% of evaluations, and SWE-Judge writes an additional custom evaluation script in 34.6% of evaluation trajectories.

Detailed eval prompts are shown in GitHub repo.

Table 11: Scoring rubric for the FullCycle task (max 12 points).

Dim.	Sub-dim.	Score	Criteria
ENV (0–4)	Static	0	No environment created; core dependencies absent; `setup.sh` has fundamental errors.
		1	Environment created, basically installed, but obvious gaps (missing deps / version issues / no editable install).
		2	Complete and excellent: all deps correctly installed, `setup.sh` clear and reproducible.
	Dynamic	0	Layer 1 fails: cannot activate environment / toolchain missing.
		1	Layer 1 succeeds, but Layer 2 or Layer 3 fails (import/build/collection or test-execution errors).
		2	Layers 1–3 all pass.
TEST (0–4)	Static	0	No test code; only meaningless assertions; `eval.sh` contains git operations.
		1	Direction correct, quality insufficient: covers only one scenario, imprecise assertions, low alignment with gold scenarios.
		2	Good: targets bug core behavior, meaningful assertions, covers main scenarios, good alignment with gold test coverage.
	Dynamic	0	Phase 1 passes on buggy code (cannot detect bug); or no test code exists.
		1	Phase 1 fails but with imprecise cause (ImportError/setup failure); or Phase 2 still fails.
		2	Phase 1 fails precisely (points to bug), Phase 2 passes.
CODE (0–4)	Static	0	No functional code changes; fundamental logic error; completely unrelated to the issue.
		1	Direction correct but incomplete: misses key edge cases, overly broad scope, or partial fix.
		2	Complete and precise: targets root cause, changes minimal, edge cases handled, highly aligned with gold direction.
	Dynamic	0	Environment failure prevents running; or all functional tests fail.
		1	Some functional tests pass (core logic OK, missing edge/secondary scenarios).
		2	All (or almost all) tests pass.

C SWE-JUDGE WORKFLOW CASE STUDIES

We select three FullCycle evaluation workflows from Table 6 that each illustrate a distinct capability: adaptive eval scripting, fault injection, and build verification with multi-dimensional scoring.

C.1 CASE 1: ADAPTIVE EVAL SCRIPTING

Instance. `NodeBB/NodeBB#8168c6c4` (FullCycle, Claude-Sonnet-4.6). The issue requires implementing profile image cleanup: when users remove cover photos or avatars, the corresponding files on disk must be deleted.

Workflow Summary. SWE-Judge executes 6 steps across 47 tool calls:

1. **Instruction and gold patch review.** SWE-Judge reads the issue description and gold patch to establish the expected behavior: file deletion via `rimraf` with glob patterns for accumulated profile images.
2. **Agent patch macro-review.** SWE-Judge reviews the agent's diff across 5 modified files (`src/groups/cover.js`, `src/socket.io/user/picture.js`, `src/user/delete.js`, `src/user/picture.js`). It identifies a critical divergence: the agent uses `getLocalCoverPath`/`getLocalAvatarPath` to delete only the current file, while the gold patch uses glob patterns to delete all accumulated files.
3. **ENV evaluation.** Static: `setup.sh` runs `npm install` correctly. Dynamic: Node.js v18.20.8 available, packages import successfully, 359 tests collected. Score: 4/4.
4. **TEST evaluation.** Static: Agent covers 3 of 4 key scenarios (missing account deletion cleanup test). Dynamic: Phase 1 fails with `TypeError: User.getLocalCoverPath is not a function` (imprecise failure). Score: 2/4.
5. **Adaptive eval scripting (triggered by TEST_STATIC < 2).** The agent's test suite lacks coverage for account deletion cleanup. SWE-Judge writes `eval_improved.sh` and `test/improved-image-cleanup.js`, a custom test that creates dummy profile images, calls the account deletion function, and verifies that 0 files remain afterward. The first execution discovers 3 orphaned files. After debugging a path configuration issue and re-executing, the test confirms the agent's implementation leaves orphaned files during account deletion.
6. **CODE evaluation using custom test results.** The custom test output directly informs CODE_DYNAMIC: 3/4 tests pass (group cover, user cover, user avatar succeed; account deletion cleanup fails). Score: 2/4.

Final Scores. ENV: 4, TEST: 2, CODE: 2. Total: 8/12 (0.667).

This case shows that SWE-Judge writes its own verification scripts when existing coverage is insufficient (34.6% of FullCycle evaluations), exposing gaps that the agent's own tests miss.

C.2 CASE 2: FAULT INJECTION

Instance. `internetarchive/openlibrary` (FullCycle, Qwen-3.5). The issue requires adding a Solr boolean clause limit configuration (`-Dsolr.max.booleanClauses=30000`) to `docker-compose.yml` and a corresponding `FILTER_BOOK_LIMIT` constant in `bookshelves.py`.

Workflow Summary.

1. **Initial review.** SWE-Judge reads the instruction, gold patch, and agent patch. The agent correctly implements both required changes: adding `-Dsolr.max.booleanClauses=30000` to `SOLR_OPTS` and defining `FILTER_BOOK_LIMIT = 30_000`.
2. **Agent test execution (Phase 2).** SWE-Judge runs the agent's test suite (`eval.sh`) on the fixed code. Both tests pass: `test_filter_book_limit_constant_exists` and `test_solr_opts_has_boolean_clauses_limit`.

3. **Fault injection (Phase 1).** SWE-Judge reverts the agent's changes to simulate the buggy state:

```
git show base_commit:docker-compose.yml > \
    /tmp/docker-compose-buggy.yml
cp /tmp/docker-compose-buggy.yml docker-compose.yml
git show base_commit:openlibrary/core/bookshelves.py > \
    /tmp/bookshelves-buggy.py
cp /tmp/bookshelves-buggy.py openlibrary/core/bookshelves.py
```

   SWE-Judge then re-runs the agent's tests against this reverted codebase. Both tests now fail: `FILTER_BOOK_LIMIT` is not found in `bookshelves.py`, and `-Dsolr.max.booleanClauses` is absent from `SOLR_OPTS`.

4. **Verdict.** The tests correctly discriminate between buggy and fixed states. SWE-Judge confirms the agent's tests are not trivial or overfitted: they verify specific code content rather than relying on indirect signals. TEST_DYNAMIC: 2/2.

5. **ENV evaluation.** Static: Agent uses venv instead of the requested conda environment, deviating from the instruction. Dynamic: Python 3.11.1 available, but core package import fails (`ModuleNotFoundError: web`). Score: 2/4.

Final Scores. ENV: 2, TEST: 3, CODE: 4. Total: 9/12 (0.75).

Fault injection verifies that the agent's tests genuinely detect the bug rather than passing for spurious reasons. SWE-Judge uses this technique in 4.8% of evaluations, typically when the tests appear suspiciously simple or when configuration changes could easily produce false positives.

C.3 Case 3: Build Verification and Multi-Dimensional Scoring

Instance. `flipt-io/flipt#292fdac` (FullCycle, Claude-Sonnet-4.6). The issue requires implementing an optional configuration versioning feature for the Flipt feature flag server (Go).

Workflow Summary.

1. **Code review via git diff.** SWE-Judge examines the agent's changes: adding a `Version` field to the configuration struct, implementing validation logic, updating the schema, and creating test data files.
2. **Reference comparison.** SWE-Judge reads the gold patch and performs a structural comparison. The agent's implementation aligns closely with the gold patch, using cleaner error handling patterns in some cases.
3. **Build verification.** SWE-Judge runs `go build ./...` to confirm compilation succeeds, then uses a non-matching test pattern to verify test collection without execution.
4. **Test execution with fault injection.** SWE-Judge reverts the code to the buggy state and runs the agent's tests. Tests fail with `cfg.Version undefined` (compilation error). SWE-Judge notes this is a weaker detection mechanism (compile-time rather than assertion-based) but still validates that the tests cannot pass without the fix.
5. **Multi-dimensional scoring.**
   - ENV: Static 2/2 (complete setup), Dynamic 2/2 (Go toolchain available, packages import, tests collect). Score: 4/4.
   - TEST: Static 2/2 (comprehensive test coverage aligned with the gold patch), Dynamic 1/2 (Phase 1 failure is imprecise: compilation error rather than assertion failure). Score: 3/4.
   - CODE: Static 2/2 (correct implementation matching the gold patch), Dynamic 2/2 (all target tests pass on fixed code). Score: 4/4.

Final Scores. ENV: 4, TEST: 3, CODE: 4. Total: 11/12 (0.917).

Build verification (used in 36.1% of FullCycle evaluations, mostly compiled languages) serves as a gate: a failed build immediately invalidates dynamic scores. The multi-dimensional scoring here separates a correct implementation (CODE: 4/4) from an imprecise test design (TEST: 3/4), a distinction that binary pass/fail cannot express.

D EVAL MODEL ROBUSTNESS

We test whether SWE-Judge's scores depend on the choice of judge model by having Claude-Opus-4.5 and GPT-5.4 independently evaluate the same agent outputs. We sample 414 evaluation trials spanning multiple solver models and all four task types. Each trial is scored by both evaluator models, with Solve rate as the primary metric.

Table 12: Solve rate comparison between two eval models across task types. Diff = Claude-Opus-4.5 − GPT-5.4.

Task Type	Claude-Opus-4.5	GPT-5.4	Diff
Impl	0.4192	0.4034	+0.016
TestGen	0.3985	0.3718	+0.027
Env	0.6032	0.5929	+0.010
FullCycle	0.0630	0.0419	+0.021

The solve rate difference between Claude-Opus-4.5 and GPT-5.4 is within 3% across all four task types (largest: +2.7% on TestGen). The relative ranking of coding models is preserved under both eval models.

Same-family bias check. Within this ablation sample, Claude-Sonnet-4.6 is the only solver from the same model family as the default evaluator, Claude-Opus-4.5. We compare its per-trial score difference between Claude-Opus-4.5 and GPT-5.4 against the corresponding average across the remaining solver models. Across all four task types, Claude-Sonnet-4.6's difference is equal to or lower than the average for the remaining models (Impl: -0.001 vs. +0.006; Env: +0.003 vs. +0.008; FullCycle: +0.143 vs. +0.194), except TestGen where the difference is negligible (+0.025 vs. +0.024). Claude-Opus-4.5 does not exhibit same-family scoring bias.

E HUMAN ANNOTATION FOR SWE-JUDGE RELIABILITY

We conduct 946 human inspections across all four tasks to audit SWE-Judge. For the isolated tasks, the audit samples 20% of the Script–Judge disagreement cases and 5% of the agreement cases. For FullCycle, it covers one submission for each of the 489 issues and includes submissions from all seven evaluated models.

Scope. For isolated tasks (Impl, TestGen, Env), we annotate 457 cases. For FullCycle, the audit covers all 489 issues, with one submission selected per issue across the seven evaluated models.

Annotation Criteria. Annotators adjudicate the factual correctness of SWE-Judge's verdict by cross-referencing multiple evidence sources: the issue description, gold patch, agent's submitted patch, and execution logs. For isolated tasks, annotators determine whether SWE-Judge or the script evaluator is correct. For FullCycle, annotators independently assign scores on 6 dimensions (each 0–2) using the same rubric as SWE-Judge and flag cases of overrating, underrating, or hallucination.

Quality Control. During the pilot annotation, the three annotators achieve a raw agreement rate of 97.9%. They then proceed to full-scale annotation using the shared criteria above. Across the evaluated dimensions in the final annotation, the minimum Fleiss' κ is 0.88 and the minimum ordinal Krippendorff's α is 0.92, indicating high inter-annotator reliability. We report raw agreement separately from the chance-corrected categorical and ordinal reliability coefficients because these statistics capture different aspects of annotator consistency.

E.1 HUMAN ANNOTATORS

Our annotation team consisted of three full-time professional annotators who were not members of the author team. Each held at least a bachelor's degree in a computer-related discipline (e.g., information security or software engineering) and had more than two years of Python development experience. They maintained an average annotation rate of one task per hour, with daily working hours capped at eight. All annotators were compensated in strict compliance with local labor regulations.

E.2 Isolated Tasks Detailed Results

Table 13 reports the per-task alignment between human annotations and SWE-Judge for isolated tasks.

Table 13: Per-task alignment on isolated tasks. Alignment = annotator confirms SWE-Judge is correct.

Task	N	Alignment%
Impl	113	95.6%
TestGen	201	99.5%
Env	143	99.3%
Overall	**457**	**98.5%**

For isolated tasks, our annotation samples 20% of cases where SWE-Judge and the script evaluator disagree (371 instances), plus a 5% random sample of agreement cases (86 instances). Among the agreement sample, 84/86 (97.7%) are confirmed correct by human annotators.

E.3 FullCycle Human Support

Table 14 reports case-level human support for SWE-Judge on the 489 audited FullCycle submissions. The 96.9% rate in the main text corresponds to 474 supported cases; the remaining 15 cases contain at least one judgment that human reviewers do not fully support.

Table 14: Case-level human support for SWE-Judge on FullCycle.

Outcome	N	%
Human supports SWE-Judge	474	96.9
Not fully supported	15	3.1
Total	**489**	**100.0**

F Per-Dataset Results

Tables 15–17 decompose the aggregate Isolated results (Table 3) by benchmark source. Static, Dynamic, and Score are percentages normalized to $[0, 100]$; Solve is the fraction of evaluated instances achieving a perfect score.

Table 15: **Per-dataset CodeImpl performance.** Best results are in **bold**, and second best are underlined.

Model	Verified				Multi.				Pro			
	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve
Claude-Sonnet-4.6	74.00	70.00	72.00	53.3	60.50	55.50	58.00	34.4	63.50	43.50	53.50	28.1
GLM-5.1	71.50	66.00	68.75	48.0	60.50	51.50	56.00	29.5	61.50	40.50	51.00	30.5
GPT-5.4	75.50	64.50	70.00	52.0	58.00	50.00	54.00	31.1	65.50	38.00	51.75	29.1
Kimi-K2.5	70.81	65.16	67.99	48.0	56.03	49.14	52.59	24.1	59.54	35.57	47.55	22.2
MiniMax-M2.7	68.50	61.50	65.00	47.1	43.50	37.00	40.25	24.6	38.00	25.50	31.75	16.3
Qwen-3.5	55.50	49.00	52.25	36.4	61.50	51.50	56.50	32.8	55.50	34.50	45.00	19.2
Claude-Fable-5	**94.67**	**89.85**	**92.55**	**90.4**	**82.00**	**76.00**	**79.00**	**72.0**	**84.05**	**77.61**	**80.83**	**70.6**

Performance degrades consistently from Verified to Pro across all models. The Multilingual subset has an uneven difficulty profile: Env scores are often higher than Verified (smaller repositories, simpler dependency chains), while CodeImpl scores drop sharply (unfamiliar language semantics and toolchains). TestGen on Multilingual tracks Verified for top-tier models but diverges for weaker ones, due to the difficulty of generating discriminative tests in non-Python ecosystems.

Table 16: **Per-dataset TestGen performance.** Best results are in **bold**, and second best are underlined.

Model	Verified				Multi.				Pro			
	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve
Claude-Sonnet-4.6	92.50	**96.50**	**94.50**	**85.8**	84.50	86.00	85.25	70.5	**91.00**	73.50	82.25	45.8
GLM-5.1	88.00	93.50	90.75	77.3	83.50	**90.00**	**86.75**	68.9	84.50	68.50	76.50	39.4
GPT-5.4	69.00	78.50	73.75	40.9	73.00	81.00	77.00	50.8	81.00	72.50	76.75	40.9
Kimi-K2.5	82.00	92.89	87.44	63.6	74.56	81.58	78.07	54.4	81.19	71.04	76.11	38.6
MiniMax-M2.7	71.00	79.50	75.25	51.6	31.00	32.00	31.50	21.3	38.50	35.00	36.75	16.7
Qwen-3.5	73.50	86.50	80.00	56.9	69.00	81.00	75.00	42.6	73.50	65.00	69.25	35.0
Claude-Fable-5	**97.09**	87.30	92.20	85.2	**92.22**	81.11	86.67	**73.3**	90.12	**85.80**	**87.96**	**80.9**

Table 17: **Per-dataset Env performance.** Best results are in **bold**, and second best are underlined.

Model	Verified				Multi.				Pro			
	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve	Static	Dyn.	Score	Solve
Claude-Sonnet-4.6	**96.00**	**89.00**	**92.50**	**85.3**	95.00	**98.50**	**96.75**	**95.1**	**83.50**	**76.50**	**80.00**	**66.0**
GLM-5.1	93.00	86.00	89.50	83.6	95.00	**98.50**	**96.75**	**95.1**	79.50	59.50	69.50	55.2
GPT-5.4	94.50	87.00	90.75	84.4	96.50	87.50	92.00	88.5	77.50	56.50	67.00	56.2
Kimi-K2.5	77.56	68.67	73.11	59.1	96.49	95.61	96.05	91.2	80.67	61.08	70.88	55.2
MiniMax-M2.7	77.00	68.50	72.75	63.1	72.00	90.00	81.00	72.1	28.00	43.50	35.75	18.2
Qwen-3.5	94.00	87.50	90.75	84.4	87.50	94.50	91.00	83.6	68.50	55.50	62.00	43.3
Claude-Fable-5	90.00	86.79	88.39	85.0	**97.22**	94.44	95.83	94.4	78.75	60.00	69.38	55.8

F.1 Claude-Fable-5 Environment Case Analysis

Claude-Fable-5 has the second-highest isolated Env solve rate, but this result trails its own Impl and TestGen performance. Its non-perfect Env cases are concentrated in SWE-bench Pro, particularly in the Ansible, OpenLibrary, and qutebrowser project families. Trajectory inspection indicates that the recurring limitation is not dependency identification itself, but environment closure: validating that the submitted setup works through the repository-native wrapper, interpreter, and toolchain path used during evaluation.

Sphinx: the target tests pass through the wrong entry point. In `sphinx-9320`, Claude-Fable-5 inspects `tox.ini` and `setup.py`, creates a Python 3.9 environment, and pins Jinja2, MarkupSafe, and several `sphinxcontrib-*` packages. All ten target tests pass when invoked directly with `pytest`. However, the agent does not install or validate `tox`, whereas the repository-native evaluation command is `tox -current-env -epy39 -v - tests/test_quickstart.py`; evaluation therefore terminates with `tox: command not found`. A second Sphinx trajectory, `sphinx-9673`, similarly resolves substantial version incompatibilities and runs more than 1,700 tests, but never closes the loop through the `tox` entry point. These submissions are plausible under the agent-selected command but are not reproducible through the execution interface used by the repository.

Ansible: the native wrapper crosses interpreter boundaries. In a representative Ansible trajectory, Claude-Fable-5 creates a conda `testbed`, installs the required pytest plugins, addresses `PYTEST_ADDOPTS`, and obtains 39 passing tests inside that environment. The repository-native `ansible-test` wrapper, however, invokes `/usr/bin/python` rather than the conda interpreter. Missing plugin metadata on the system Python causes argument parsing to fail before any target test executes. The agent validates direct `python -m pytest` execution but not the interpreter and plugin path selected by `ansible-test`.

Successful closure as a contrast. The limitation is not universal. In a successful Django case, Claude-Fable-5 traces an import failure to `sys.path`, persists the correction through a `.pth` file, and verifies 91 tests through the intended execution path. Similar successful trajectories configure the native Go and Rust toolchains for Flipt and Nushell. These contrasts show that Claude-Fable-5

generally reasons well about dependencies, but does not consistently complete the final validation loop when projects combine test wrappers, multiple interpreters, or historical toolchain boundaries. Consistent with this interpretation, non-perfect partial-credit runs are longer rather than prematurely terminated: their median trajectory contains 41 steps, compared with 13 for full-score runs.

Confidence Intervals. We report 95% confidence intervals for all core metrics. For binary metrics (Solve), we use Wilson score intervals. For continuous scores (Score), we use bootstrap percentile intervals with 10,000 resamples.

Table 18: **Per-dataset CodeImpl performance with 95% CI.** Score uses bootstrap CI; Solve uses Wilson CI.

Model	Verified		Multi.		Pro	
	Score	Solve	Score	Solve	Score	Solve
Claude-Sonnet-4.6	72.00±4.50	53.3±6.5	58.00±9.75	34.4±11.6	53.50±4.75	28.1±6.1
GLM-5.1	68.75±4.50	48.0±6.5	56.00±9.00	29.5±11.2	51.00±5.00	30.5±6.3
GPT-5.4	70.00±4.50	52.0±6.5	54.00±9.75	31.1±11.3	51.75±4.75	29.1±6.2
Kimi-K2.5	67.99±4.64	48.0±6.6	52.59±9.27	24.1±11.2	47.55±4.70	22.2±5.9
MiniMax-M2.7	65.00±5.00	47.1±6.5	40.25±10.25	24.6±10.6	31.75±5.00	16.3±5.1
Qwen-3.5	52.25±5.50	36.4±6.2	56.50±9.50	32.8±11.5	45.00±4.50	19.2±5.4
Claude-Fable-5	92.55±3.22	90.4±4.1	79.00±9.75	72.0±12.1	80.83±4.98	70.6±6.9

Table 19: **Per-dataset TestGen performance with 95% CI.** Score uses bootstrap CI; Solve uses Wilson CI.

Model	Verified		Multi.		Pro	
	Score	Solve	Score	Solve	Score	Solve
Claude-Sonnet-4.6	94.50±2.00	85.8±4.6	85.25±7.00	70.5±11.2	82.25±2.75	45.8±6.8
GLM-5.1	90.75±2.75	77.3±5.4	86.75±6.25	68.9±11.3	76.50±3.50	39.4±6.7
GPT-5.4	73.75±3.75	40.9±6.4	77.00±7.50	50.8±12.2	76.75±3.25	40.9±6.7
Kimi-K2.5	87.44±2.61	63.6±6.2	78.07±7.68	54.4±12.3	76.11±3.22	38.6±6.7
MiniMax-M2.7	75.25±4.25	51.6±6.5	31.50±10.75	21.3±10.1	36.75±5.50	16.7±5.1
Qwen-3.5	80.00±4.00	56.9±6.4	75.00±7.75	42.6±12.0	69.25±4.25	35.0±6.5
Claude-Fable-5	92.20±2.78	85.2±5.1	86.67±7.50	73.3±12.5	87.96±4.48	80.9±6.0

Table 20: **Per-dataset Env performance with 95% CI.** Score uses bootstrap CI; Solve uses Wilson CI.

Model	Verified		Multi.		Pro	
	Score	Solve	Score	Solve	Score	Solve
Claude-Sonnet-4.6	92.50±2.75	85.3±4.6	96.75±3.75	95.1±5.9	80.00±4.25	66.0±6.5
GLM-5.1	89.50±3.50	83.6±4.8	96.75±3.75	95.1±5.9	69.50±5.00	55.2±6.8
GPT-5.4	90.75±3.00	84.4±4.7	92.00±5.25	88.5±8.1	67.00±5.00	56.2±6.8
Kimi-K2.5	73.11±4.89	59.1±6.2	96.05±3.95	91.2±7.0	70.88±4.90	55.2±7.2
MiniMax-M2.7	72.75±5.25	63.1±6.3	81.00±8.25	72.1±11.0	35.75±5.00	18.2±5.3
Qwen-3.5	90.75±3.00	84.4±4.7	91.00±5.00	83.6±9.2	62.00±5.25	43.3±6.8
Claude-Fable-5	88.39±4.73	85.0±5.9	95.83±7.43	94.4±9.0	69.38±6.80	55.8±8.7

G Efficiency Analysis

Table 22 reports solver output tokens, agent execution time, and verifier wall time for all seven evaluated models across all task types. The token column measures solver rather than SWE-Judge usage. FullCycle requires substantially more solver tokens and time than isolated tasks. Median evaluation time ranges from 2.6 to 5.6 minutes for isolated tasks and from 6.0 to 16.0 minutes for FullCycle.

Table 21: **Per-dataset FullCycle performance with 95% CI.** Score uses bootstrap CI; Solve uses Wilson CI.

Model	Verified		Multi.		Pro	
	Score	Solve	Score	Solve	Score	Solve
Claude-Sonnet-4.6	83.50±2.00	20.4±5.2	82.50±3.17	8.2±7.1	76.58±1.92	4.4±2.9
GLM-5.1	85.50±1.67	22.2±5.4	82.75±3.42	8.2±7.1	76.67±3.08	5.4±3.2
GPT-5.4	83.92±1.67	15.1±4.7	79.00±4.75	9.8±7.6	75.83±2.42	6.4±3.4
Kimi-K2.5	68.42±2.47	0.4±0.7	76.13±5.30	10.2±7.6	71.24±2.40	1.6±1.9
MiniMax-M2.7	49.00±5.17	8.4±3.7	43.00±8.75	1.6±4.2	28.58±4.83	0.5±1.3
Qwen-3.5	77.83±2.58	12.4±4.3	74.42±5.50	3.3±5.1	70.67±2.42	1.5±1.9
Claude-Fable-5	87.71±3.48	67.7±6.1	72.22±9.65	58.3±12.1	78.35±4.25	53.3±6.9

Table 22: **Efficiency metrics across task types.** OutTok = median solver output tokens (K); Agent = median agent execution time (min); Eval = median verifier execution time (min). All values are medians computed over all instances across three datasets.

Model	Impl			TestGen			Env			FullCycle		
	OutTok	Agent	Eval	OutTok	Agent	Eval	OutTok	Agent	Eval	OutTok	Agent	Eval
Claude-Sonnet-4.6	5.4K	5.7	3.6	7.3K	8.4	3.9	3.1K	8.2	2.9	11.0K	11.7	8.0
GLM-5.1	3.0K	11.6	3.1	3.0K	12.1	4.0	1.5K	12.7	2.6	5.9K	13.8	7.7
GPT-5.4	2.8K	4.2	3.2	2.6K	4.9	4.0	4.4K	20.1	2.6	6.0K	10.9	8.6
Qwen-3.5	8.9K	23.5	3.1	6.7K	18.8	3.9	2.6K	21.9	2.7	12.4K	29.5	7.6
Kimi-K2.5	8.5K	10.8	4.0	7.5K	10.4	5.0	4.0K	29.5	3.5	10.6K	27.8	16.0
MiniMax-M2.7	4.2K	12.3	3.2	5.9K	16.1	3.8	2.0K	14.1	2.8	1.9K	5.7	6.0
Claude-Fable-5	9.7K	8.4	5.6	11.9K	9.4	5.5	3.4K	11.9	3.5	18.0K	20.2	7.7

G.1 SWE-Judge Output-Side Cost Estimate

Historical trajectories do not retain API-reported input, output, or cache-token fields. We therefore estimate setting-specific output usage from retained judge transcripts using four characters per token, imputing unavailable transcripts with their task-type mean. Table 23 applies these historical per-evaluation means to the current seven-model design (489 instances $\times$ 7 models = 3,423 evaluations per setting). We use Claude-Opus-4.5's published output price of $25 per million tokens (Anthropic, 2025). These are planning estimates rather than complete API bills: they exclude input and cache tokens, retries, discounts, and some tool-call payloads, and the Claude-Fable-5 evaluations are extrapolated from the historical setting-level means.

Table 23: Estimated SWE-Judge output-side cost for the current seven-model design. "Per eval" is one model–instance–task evaluation; subtotals extrapolate historical setting-level means to 3,423 evaluations.

Setting	Mean output tokens	Cost/eval	Estimated subtotal
Impl	3,572.57	$0.089	$305.72
TestGen	5,144.80	$0.129	$440.27
Env	7,812.67	$0.195	$668.57
FullCycle	11,409.68	$0.285	$976.38
Overall	6,984.93	$0.175	$2,390.94

Under these assumptions, the current seven-model design requires an estimated 95.64 million judge output tokens, corresponding to $2.39K in output-side cost. This extrapolation is useful for budgeting but should not be interpreted as measured spend. Future runs should record exact input, output, cache, and retry usage together with the pricing date.

H Evaluator Analysis and Failure Cases

H.1 Disagreement Annotation Protocol

To categorize Script–SWE-Judge disagreements, the audit team labels each case using the protocol below. For each disagreement instance, annotators receive a review package containing: the issue description, the agent's submitted patch, the gold patch, SWE-Judge's scoring with reasoning, the script evaluator's binary verdict with execution logs, and LLM-generated auxiliary analysis highlighting potential discrepancies.

Each annotation follows a structured protocol:

1. Read the issue description to understand the problem context.
2. Examine the gold patch to establish the correct solution approach.
3. Review the agent's submission to understand what the agent implemented.
4. Read SWE-Judge's scoring and reasoning.
5. Cross-reference with execution logs and LLM auxiliary analysis when static review is insufficient.
6. Assign a failure category from the predefined taxonomy and record which evaluator is correct.

Human Verification Results. We conduct human deep annotation on 371 disagreement instances (a 20% sample) plus 86 agreement instances (a 5% random sample of cases where SWE-Judge and the script concur). For disagreement cases, human annotators support SWE-Judge in 98.6% (366/371), the script in 0.5% (2/371), and neither in 0.8% (3/371). For agreement cases, 84/86 (97.7%) are confirmed by human review.

H.2 Complementarity of Static and Dynamic Evidence

Static and Dynamic evaluation target complementary properties. Static review assesses whether submitted artifacts are complete, persistent, and semantically aligned with the issue, whereas Dynamic execution tests behavior observed in the current container. Representative cases show why both signals are needed: static review can still assess a submitted configuration when corrupted workspace state blocks execution (Appendix H.7), while dynamic execution reveals whether a plausible artifact actually runs. Conversely, a runtime state transition alone can be misleading when a test fails because of an import error rather than the target bug (Appendix H.8).

H.3 Representative SWE-Judge Errors

The audit also identifies errors in both directions. In the OpenLibrary GPT-5.4 Full-Cycle trial `instance_internetarchive__openli__SwQYDUt`, SWE-Judge assigned ENV_DYNAMIC= 1 even though its rationale stated that the first activation layer failed because `/opt/miniconda3` was unavailable; human review reduced the overall score from 9/12 to 8/12. Conversely, in the Qwen-3.5 Teleport trial `pXtcr85`, SWE-Judge penalized a valid alternative implementation; human review increased the overall score from 10/12 to 11/12. Both corrections are one-point adjustments and leave the strict pass/fail outcome unchanged. Given their low frequency and limited magnitude, these small variations do not materially change the aggregate experimental results; the above-95% human support across audited cases continues to support the overall reliability of SWE-Judge, while motivating instance-level auditing for partial-credit boundaries and alternative implementations.

H.4 Disagreement Categories

Table 24 summarizes the categorization results.

H.5 Representative Case Studies

We present representative cases organized by the three failure categories.

Table 24: Distribution of Script evaluation error categories.

Category	%
Excessive strictness	36.0
Evaluation breakdown	32.8
Excessive leniency	27.0
Others	4.2

H.6 Excessive Strictness

Scripts demand exact alignment with the gold patch and reject functionally equivalent alternatives. Two manifestations appear: alternative implementations receiving zero credit, and partial fixes losing all information through binary scoring.

Case 1: Alternative Implementation in Valkey. In `valkey-io/valkey#1499`, the gold patch modifies the command table to fix a permission checking issue. The agent instead uses `executing_client->cmd` to check the actual command being executed, with null-safety handling. All FAIL_TO_PASS tests pass and no PASS_TO_PASS regressions occur. The script assigns 0 because its tests are coupled to the specific implementation path of the gold patch. SWE-Judge performs static analysis, confirms the semantic equivalence of both approaches, and assigns 1.0.

Case 2: Constant Exporting in Teleport. In a Teleport issue requiring namespace configuration constants, the gold patch inlines string literals across multiple files. The agent exports `NamespaceEnv` and `ReleaseNameEnv` as package-level constants and updates all references. This is a cleaner refactoring that produces identical behavior. All 13 FAIL_TO_PASS tests pass, but the script assigns 0 because the modified file set differs from the expected set. SWE-Judge recognizes the functional equivalence and awards full marks.

Case 3: Partial Fix in PHPSpreadsheet. In a PHPSpreadsheet task, the agent correctly adds a `__toString()` method to the `StructuredReference` class, fixing the immediate string conversion error. However, it misses additional changes for cross-worksheet table and structured reference handling that the gold patch includes. The script assigns 0, indistinguishable from a completely wrong submission. SWE-Judge assigns 0.25, recognizing that the core direction is correct but coverage is incomplete. This proportional credit separates near-miss attempts from zero-effort submissions.

Case 4: Qutebrowser Path Resolution. The agent creates `FilePathCategory` with proper path resolution for `file://`, tilde, and absolute paths, integrates it into the URL model, and updates documentation. The implementation handles all major scenarios but misses minor edge cases in the gold patch. SWE-Judge assigns 0.75, reflecting a nearly complete solution. The script's binary 0 fails to capture this meaningful progress.

H.7 Evaluation Breakdown

Nearly a third of disagreements occur because the evaluation pipeline itself fails, independent of solution quality. Parser incompatibilities and infrastructure rot are the primary causes.

Case 5: Gradle Output Parsing in Apache Lucene. Multiple Apache Lucene environment tasks exhibit this pattern. The agent correctly configures JDK 21, the Gradle wrapper, and all build dependencies. SWE-Judge independently confirms via JUnit XML that all 108 tests pass with 0 failures. However, the SWE-bench evaluation parser expects Maven-style output format and cannot parse Gradle's `BUILD SUCCESSFUL` format, reporting a zero score. This is a failure of the evaluation tool, not the agent.

Case 6: Maven Daemon Timeout in Google Gson. In a Google Gson environment task, the agent's setup correctly installs Java and Maven. All 10 tests pass when executed with standard

Maven. However, the evaluation script uses `mvnd` (Maven Daemon), which times out during cold start. SWE-Judge identifies this as an infrastructure artifact: the original script evaluation failures stem from `mvnd` daemon issues (timeout/crashes), not actual test failures. The agent receives full marks from SWE-Judge.

Case 7: Node.js Workspace Corruption. After `yarn install`, the `node_modules` state file is missing or corrupted in approximately 178 cases, causing all subsequent commands to fail with "Couldn't find the node_modules state file." The agent's code is never evaluated because the test framework collapses before reaching any relevant assertion. SWE-Judge evaluates the agent's configuration through static analysis and awards credit based on the quality of the submitted patch, independent of whether the test infrastructure executed successfully.

These cases illustrate a structural limitation: script evaluation conflates "the framework crashed" with "the solution is wrong." As dependencies deprecate and runtime versions drift, this conflation worsens over time.

H.8 Excessive Leniency

Script evaluation can conflate superficial execution success with semantic correctness.

Case 8: Trivial State Transition in TestGen. In an Ansible test task, the agent writes tests that import `set_multipart_encoding` at module level. This function exists only after the fix. On buggy code, the test fails with `ImportError` before any test logic executes. The script's dynamic state transition protocol checks only whether Phase 1 (buggy code) produces a non-zero exit code and Phase 2 (fixed code) passes. Both conditions are met, so the script awards full marks. SWE-Judge recognizes the failure mechanism: "When `code_patch` is reverted, module import fails with `AttributeError` before any tests can run to detect actual bug behavior." The test provides zero discriminative power because any pre-fix version would fail regardless of the specific bug.

Case 9: Regression Escape in Django. In `django/django#13590`, the agent correctly fixes `namedtuple` support in `Values()` by unpacking values. However, the patch unconditionally unpacks all types, breaking regular list/tuple construction. The FAIL_TO_PASS test passes, but 5 PASS_TO_PASS tests fail with `TypeError`. The script monitors only the target test scope and awards full marks. SWE-Judge runs the complete test suite and identifies the regression: the gold patch uses `hasattr(type_, '_make')` to detect namedtuples and only unpacks for those types. SWE-Judge assigns 0.25.

Case 10: Incomplete Test Coverage in coreutils. In `uutils/coreutils#6575` (TestGen), the agent tests non-UTF-8 filename handling but covers only CRC mode, missing the SHA256 mode test present in the gold patch. The script's pass/fail check accepts this single-scenario test as fully correct. SWE-Judge evaluates coverage depth against the gold patch and assigns 0.25, recognizing that the test provides insufficient coverage to serve as a reliable regression test.

I Ablation: Reference-Guided vs. Blind Evaluation

Does reference access cause the evaluator to penalize valid alternative implementations? We compare **Gold eval**, which may inspect the issue, submitted artifacts, gold patch, and gold tests, with **Blind eval**, which receives the issue, repository state, and submitted artifacts but neither the gold patch nor gold tests.

I.1 Score Comparison

Table 25 reports the dimension-level comparison across 8,678 paired trials.

In Isolated evaluation, Blind eval inflates Impl scores by +4.4 pp and TestGen by +2.3 pp. In FullCycle, the inflation concentrates in CODE (+18.4 pp), where assessing correctness without a reference is hardest. ENV scores remain stable (-0.7 pp) because environment correctness is largely verifiable through execution. The inflation is driven almost entirely by static sub-scores ($\Delta_S = +0.17$) while

Table 25: Gold vs. Blind evaluation comparison across all task categories. Isolated: n=5,771; FullCycle: n=2,907. All scores normalized to percentages. Diff = Blind − Gold (percentage points).

Setting	Category	Gold (%)	Blind (%)	Diff (pp)
Isolated	Impl	64.0	68.4	+4.4
	TestGen	64.1	66.4	+2.3
FullCycle	ENV	83.2	82.5	−0.7
	CODE	64.3	82.7	**+18.4**
	TEST	62.9	68.1	+5.3

dynamic sub-scores remain unchanged ($\Delta_D = +0.01$): execution-based verification is objective regardless of reference availability.

I.2 Illustrative Case: Gold Eval Favors a Correct Alternative

We present a case where Gold eval is more lenient than Blind eval toward an alternative implementation.

`django__django-16877` (Impl, Claude 4.6) — Script=0, Gold=4/4, Blind=1/4

> *Gold (static=2):* "Implementation is functionally identical to gold.patch—correctly implements `escapeseq` filter with equivalent logic."
> *Blind (static=1):* "Fix direction correct, but agent left unresolved merge conflicts in test file."

Gold eval confirms semantic equivalence with the reference and correctly identifies the merge conflict markers as irrelevant to functional correctness. Blind eval, lacking this anchor, is misled by the cosmetic issue and penalizes a correct submission. Reference access here protects the alternative implementation by providing a semantic equivalence check.

I.3 Difficulty Stratification

Table 26 stratifies trials by Gold score to examine where the Gold–Blind gap concentrates.

Table 26: Difficulty stratification: Blind–Gold score difference by Gold score bin (n=5,771 Isolated trials).

Gold Score Bin	n	Blind Mean	Diff
$= 0$ (clearly wrong)	1,017	0.024	+0.024
$(0, 0.25]$	1,033	0.289	+0.039
$(0.25, 0.5]$	388	0.599	+0.099
$(0.5, 0.75]$ (partial)	1,033	0.886	**+0.136**
$(0.75, 1.0]$ (correct)	2,300	0.978	−0.022

The pattern forms an inverted-U: inflation peaks at the $(0.5, 0.75]$ bin (+0.136) and reverses for near-perfect submissions (−0.022). If Gold eval penalized correct alternatives, the highest-scoring bin would show Gold > Blind. Instead, the slight negative difference confirms Gold eval does not under-score correct submissions. The only divergence direction is Blind eval over-scoring partial fixes in the ambiguous middle range.

J Phase-Removal Analysis

To probe cross-phase dependence, we run a **single-phase-removal ablation**: within FullCycle, we omit one phase (Env, Code, or Test) and measure the remaining scores against paired ordinary

FullCycle runs. We evaluate Claude-Sonnet-4.6 and GLM-5.1 across Verified, Pro, and Multilingual, yielding up to 978 model–issue pairs per condition.

All conditions retain the issue descriptions, base repositories, 180-minute and token limits, and SWE-Judge procedure. In −Env, environment instructions are removed and the preconfigured Development image is supplied; the agent modifies code and writes tests without gold code or tests. In −Code, implementation is prohibited and the repository remains at the buggy commit without a reference patch; the agent configures the environment and writes tests. In −Test, test generation is removed without supplying gold tests; the agent configures the environment and implements the fix. The removed phase is excluded from scoring. Because each removal also changes available artifacts, context, and the effective budget of remaining phases, this experiment measures pipeline sensitivity rather than a controlled causal effect of session integration.

J.1 Phase Coupling Analysis

Table 27 reports the score change (Δ) of each remaining dimension, decomposed into Static (S) and Dynamic (D) sub-scores.

Table 27: Phase-removal analysis. Each cell reports the score change (Δ) of the column dimension when the row phase is removed from the three-stage FullCycle setting. Scores are on a 0–4 scale. $N \approx 485$ per model per condition.

		ΔEnv			**ΔCode**			**ΔTest**		
Removed	**Model**	**S**	**D**	**Tot**	**S**	**D**	**Tot**	**S**	**D**	**Tot**
−Env	Claude		—		−0.25	−0.08	−0.34	−0.06	+0.02	−0.05
	GLM		—		−0.28	−0.02	−0.31	−0.03	+0.09	+0.06
	Avg.		—		−0.27	−0.05	**−0.32**	−0.04	+0.06	**+0.01**
−Code	Claude	−0.10	−0.21	−0.32		—		−0.03	−0.03	−0.07
	GLM	−0.03	−0.10	−0.13		—		+0.04	+0.12	+0.16
	Avg.	−0.07	−0.16	**−0.23**		—		+0.00	+0.05	**+0.05**
−Test	Claude	−0.21	−0.13	−0.34	−0.32	−0.15	−0.47		—	
	GLM	−0.14	−0.09	−0.23	−0.30	−0.11	−0.41		—	
	Avg.	−0.17	−0.11	**−0.28**	−0.31	−0.13	**−0.44**		—	

Three patterns emerge. First, Code is sensitive to both surrounding phases: removing Env changes Code by -0.32 (8.0% of the 0–4 scale), and removing Test changes it by -0.44 (11.0%). The former difference is concentrated in Static (-0.27 versus -0.05 Dynamic), whereas the latter affects both Static (-0.31) and Dynamic (-0.13). Second, Env is sensitive to Code removal ($\Delta = -0.23$, or 5.8%), primarily through Dynamic loss (-0.16). These patterns are consistent with downstream execution and testing providing feedback for earlier artifacts. Third, average Test changes are small but heterogeneous: the cross-model average is $+0.01$ after removing Env and $+0.05$ after removing Code, with Claude declining and GLM improving. Thus, the experiment does not support a universal claim that every removal degrades every remaining phase.

J.2 Per-Dataset Results

The Test-removal change in Code is largest on Pro (-0.49 to -0.57) and smaller on Verified (-0.27 to -0.42). GLM's Test score increases when Code is removed on Verified and Pro (both $+0.21$), whereas several Multilingual cells decline. These per-source differences are descriptive and do not by themselves identify instance difficulty, attention allocation, or environment simplicity as causes.

Table 28: Per-dataset phase-removal results. Each value is the total-score change (Δ) when a phase is removed.

Dataset	Model	Removed	N	ΔEnv	ΔCode	ΔTest
Verified	Claude	Env	224	—	-0.37	-0.13
	Claude	Code	225	-0.30	—	$+0.04$
	Claude	Test	225	-0.52	-0.42	—
	GLM	Env	225	—	-0.28	$+0.02$
	GLM	Code	224	-0.22	—	$+0.21$
	GLM	Test	225	-0.31	-0.27	—
Pro	Claude	Env	200	—	-0.36	$+0.06$
	Claude	Code	200	-0.40	—	-0.19
	Claude	Test	202	-0.23	-0.49	—
	GLM	Env	197	—	-0.34	$+0.17$
	GLM	Code	195	-0.02	—	$+0.21$
	GLM	Test	196	-0.10	-0.57	—
Multilingual	Claude	Env	61	—	-0.15	-0.10
	Claude	Code	61	-0.11	—	-0.08
	Claude	Test	59	-0.02	-0.58	—
	GLM	Env	61	—	-0.33	-0.18
	GLM	Code	61	-0.18	—	-0.20
	GLM	Test	59	-0.32	-0.42	—