SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios

Anonymous ACL submission

Abstract

Real-world software engineering requires developers to interpret high-level requirements, coordinate changes across many files, and evolve codebases over multiple iterations while preserving functionality. Yet current benchmarks for AI coding agents evaluate only isolated, single-issue tasks such as fixing one bug or adding a small feature. We introduce SWE-EVO, a benchmark that closes this gap by targeting long-horizon software evolution. Constructed from release notes of seven mature open-source Python projects, SWE-EVO comprises 48 tasks requiring multi-step modifications spanning an average of 21 files, validated against test suites averaging 874 tests per instance. Experiments with two agent frameworks and 18 state-of-the-art models reveal a striking capability gap: GPT-5.4 with OpenHands achieves only 25% on SWE-EVO versus 72.80% achieved by GPT-5.2 on SWE-Bench Verified, showing that current agents struggle with sustained, multi-file reasoning. We also propose *Fix Rate*, a fine-grained metric that captures partial progress on these complex, long-horizon tasks.

1 Introduction

Large language models (LLMs) have achieved remarkable progress in automating software engineering (SE) tasks, including code generation (Wei et al., 2023; Chen et al., 2021a; Wang et al., 2023; Li et al., 2022; Zhuo et al., 2024; Bui et al., 2023; Manh et al., 2023; To et al., 2023), bug fixing (Jimenez et al., 2023; Xia et al., 2024), and test synthesis (Chen et al., 2022; Wang et al., 2024b; Jain et al., 2024). These advancements have facilitated the emergence of AI-powered coding agents capable of assisting or automating key aspects of the software development lifecycle (Zhang et al., 2023; Fan et al., 2023; Gao et al., 2025; He et al., 2025).

Building on these capabilities, multi-agent systems have rapidly emerged to address long-horizon challenges in software engineering. By coordinating specialized agents for tasks such as repository navigation, bug localization, patch generation, and verification, these frameworks have begun to outperform single-agent architectures in scalability and performance. Recent systems (Yang et al., 2024a; Wang et al., 2024d; Phan et al., 2024; Nguyen et al., 2025b) demonstrate this trend, enabling autonomous handling of complex workflows in real-world repositories. Industry adoption reflects this momentum: over 90% of engineering teams now integrate generative AI into SE workflows, up from 61% in 2024 (DORA Research Program, 2025).

To evaluate these agents rigorously, benchmarks have become essential. Early efforts like HumanEval (Chen et al., 2021b) focused on function-level code completion (Chen et al., 2021a), while SWE-Bench (Jimenez et al., 2023) marked a shift by curating real-world GitHub issues, tasking agents with generating verifiable patches for isolated problems (Jimenez et al., 2023). SWE-Bench has gained prominence as a de facto standard for assessing multi-agent capabilities in practical coding scenarios.

While recent models achieve up to around 75% on SWE-Bench-Verified (e.g., `GPT-5.2` (OpenAI, 2025e)) and around 40% on the full leaderboard (e.g., OpenCSG Starship at 39.67% (Jimenez et al., 2024)), the benchmark is showing signs of saturation with diminishing gains on isolated tasks. More importantly, SWE-Bench focuses on discrete issue resolution and does not capture the core challenge of software engineering: the continuous evolution of existing systems (Kaur and Singh, 2015; Singh et al., 2019). In practice, up to 80% of effort is spent maintaining and evolving legacy code, requiring coordinated changes across modules, versions, and specifications (Kaur and Singh, 2015; Singh et al., 2019).

This gap between benchmark tasks and real-world evolution scenarios motivates our central research

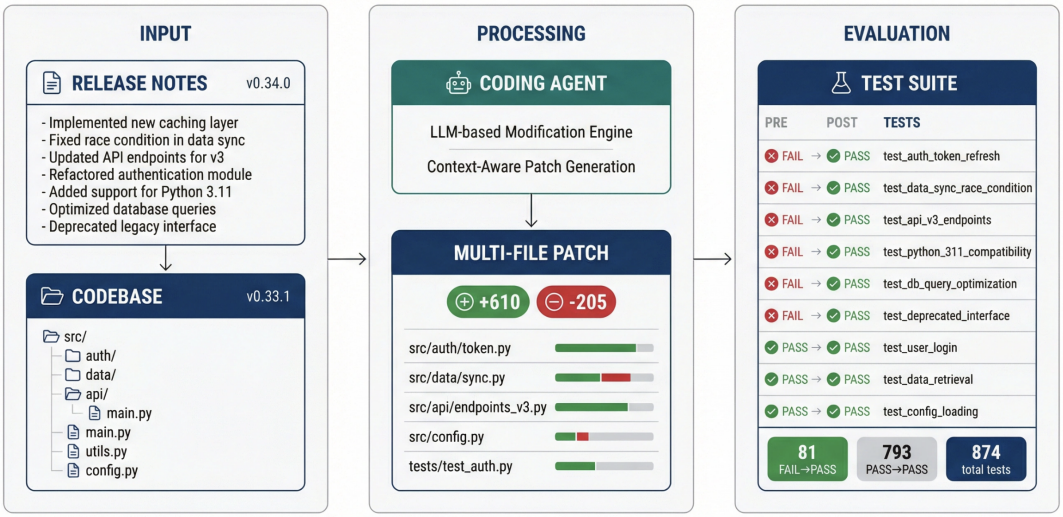


Figure 1: Overview of the SWE-EVO task pipeline. **Input**: the agent receives release notes specifying multiple changes (bug fixes, new features, maintenance) alongside the full codebase at the start version. **Processing**: a coding agent generates a multi-file patch spanning dozens of files. **Evaluation**: the patch is validated against a comprehensive test suite comprising FAIL→PASS tests (verifying the required changes) and PASS→PASS tests (ensuring no regressions).

question: *Given an existing codebase, can multi-agent LLM systems autonomously evolve the system in response to dynamic input requirements, demonstrating sustained planning, adaptability, and innovation across long-horizon tasks?*

To address this gap, we introduce SWE-EVO, a benchmark for autonomous software evolution rather than single-issue repair. SWE-EVO leverages release notes, commit histories, and versioned snapshots from mature open-source Python projects (e.g., `scikit-learn`, `pydantic`) to construct realistic tasks where agents must interpret SRS, plan multi-step modifications, and iteratively evolve codebases across versions. In total, SWE-EVO comprises 48 evolution tasks across 7 repositories (Table 1 and Figure 3), each representing a multi-change evolution scenario. We describe the benchmark construction in Section 3 and report experimental results in Section 4.

Figure 1 illustrates the SWE-EVO task pipeline: given release notes and a codebase at a starting version, the agent must generate a multi-file patch validated by a comprehensive test suite. Software evolution (see Figure 6 in Appendix B) requires alignment with high-level Software Requirement Specifications (SRS), holistic program understanding, and coordinated multi-step changes. While SWE-Bench focuses on isolated issues, real-world evolution involves codebase-scale modifications across multiple files and subsystems. Figure 2 highlights the differences between SWE-Bench and our benchmark, SWE-EVO.

Our evaluation with two agent frameworks (OpenHands and SWE-agent) and 18 state-of-the-art models reveals a significant capability gap: even the best-performing model (`gpt-5.4`) resolves only 25% of SWE-EVO tasks compared to 72.80% achieved by `gpt-5.2` on SWE-Bench Verified. Model performance exhibits intuitive scaling, where larger models consistently outperform smaller variants, and the relative ranking aligns with SWE-Bench results, validating SWE-EVO as a meaningful benchmark. Trajectory-level failure analysis shows that stronger models primarily fail on instruction following (misinterpreting nuanced release notes), while weaker models struggle with tool use and syntax errors, indicating that SWE-EVO's difficulty stems from semantic reasoning rather than interface competence.

2 Related Work

2.1 Code Generation and SE Benchmarks

Code generation evaluation has evolved from function-level tasks (Chen et al., 2021a; Austin et al., 2021; Hendrycks et al., 2021) to repository-level challenges. Early benchmarks like HumanEval, MBPP, and APPS focused on single-file, synthetic scenarios and are now largely saturated (Zhou et al., 2023), while more recent efforts such as CodeMMLU (Nguyen et al., 2025a)

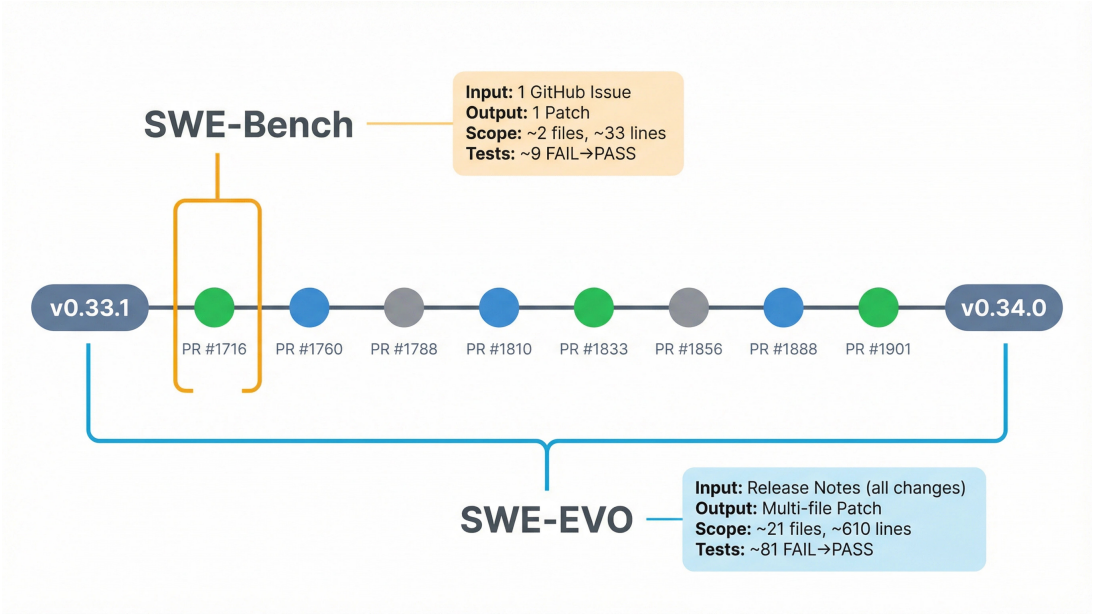


Figure 2: Comparison of SWE-Bench and SWE-EVO. SWE-Bench tasks agents with resolving a single GitHub issue to produce one patch. SWE-EVO requires agents to interpret release notes and implement comprehensive changes that resolve multiple PRs, develop new features, and fix multiple bugs to evolve the codebase to a new version.

and CodeWiki (Hoang et al., 2025) expand to code understanding and documentation. SWE-bench (Jimenez et al., 2024) marked a shift toward practical SE evaluation by requiring models to generate verifiable patches for real GitHub issues, with curated subsets (OpenAI, 2024b) and extensions spanning multiple languages (Zan et al., 2025), multimodal inputs (Yang et al., 2024b), live collection (Zhang et al., 2025b; Badertdinov et al., 2025), and enterprise-level complexity (Scale AI, 2025). Despite these advances, all existing benchmarks focus on isolated issue resolution rather than multi-step software evolution spanning multiple commits and long-horizon planning.

2.2 Software Engineering Agents

Interactive bug-fixing approaches (Xia and Zhang, 2023, 2024; Chen et al., 2023; Yuan et al., 2024) and autonomous agents like Devin AI (Cognition, 2024) laid the groundwork for dedicated agent scaffolds. SWE-agent (Yang et al., 2024a) highlighted the importance of agent–computer interfaces, OpenHands (Wang et al., 2024d) introduced a multi-agent platform evaluated across 15 benchmarks, AutoCodeRover (Zhang et al., 2024) combined LLMs with AST-based search, and Agentless (Xia et al., 2024) demonstrated competitive performance with a simple localization-repair pipeline. Multi-agent architectures have also proven effective: AgentCoder (Huang et al., 2023) introduced role specialization, AgileCoder (Nguyen et al., 2025b) incorporated agile principles, and CodeAct (Wang et al., 2024c) unified action spaces through executable Python. Recent LLMs have been post-trained on real-world SE data to improve issue resolution (Wei et al., 2025; Luo et al., 2025; DeepSeek AI, 2025; Chen et al., 2025; Kimi Team, 2025a; Carbonneaux et al., 2025), complemented by synthetic data efforts (Pham et al., 2025). A parallel line explores self-evolving agents and tool creation (Robeyns et al., 2025; Zhang et al., 2025a; Wang et al., 2025; Cai et al., 2024; Wang et al., 2024a; Qian et al., 2024; Wang et al., 2024e; Qiu et al., 2025) (see Appendix C).

2.3 Context Engineering for Long-Horizon Agents

Context management is a fundamental challenge for long-running agent systems, particularly relevant to SWE-EVO where tasks require sustained multi-turn interaction across large codebases. Recent surveys (Mei et al., 2025; Hua et al., 2025) organize this field around context retrieval, processing, and management, tracing its evolution from prompt engineering through RAG to the current era treating context as a first-class engineering concern. Key techniques include compression (Ge et al., 2024), retrieval methods such as Self-RAG (Asai

et al., 2024), RAPTOR (Sarthi et al., 2024), and GraphRAG (Gutierrez et al., 2024), and memory architectures like MemGPT (Packer et al., 2023) and hierarchical storage (Zhong et al., 2024) that enable agents to transcend single-session limitations. At the frontier, Meta Context Engineering (Ye et al., 2026) treats context assembly as an optimization problem, achieving 89.1% on SWE-bench Verified versus 70.7% for hand-engineered baselines. These advances are directly relevant to SWE-EVO, where agents must maintain coherent reasoning across specifications spanning thousands of words and patches touching dozens of files.

3 SWE-EVO Dataset

SWE-EVO is a benchmark constructed from release notes and version histories of popular open-source repositories, capturing real software evolution scenarios where coding agents must interpret high-level Software Requirement Specifications (SRS), plan and implement multi-step modifications across versions, and ensure that the evolved system passes validation tests aligned with those specifications.

3.1 Benchmark Construction

The construction of SWE-EVO consists of three major phases: (1) repository selection and data scraping, (2) candidate selection and filtering, and (3) execution-based filtering. We describe each phase in turn.

Stage I: Repository Selection and Data Scraping. To maximize reproducibility and comparability, we inherit repositories and execution environments from SWE-bench and it keeps our benchmark "plug-and-play" for existing SWE agents. Concretely, we begin by collecting samples from SWE-bench (Jimenez et al., 2024) and from SWE-gym (Pan et al., 2024) as our seed pool of task instances. Both resources already give us: a real repository snapshot, an executable environment, and a set of tests tied to a human-authored change.

Stage II: Candidate Selection and Filtering. Unlike SWE-bench, which frame tasks as resolving a single issue, we target evolving a codebase between two release versions. We therefore create candidate instances by selecting only those samples whose base commit corresponds exactly to a version tag of the repository (i.e., a release snapshot). For each such candidate, we define the problem statement as the release-note/SRS delta between that version and its next tagged version; the agent's job is to implement the specified changes across the codebase to satisfy this SRS.

Stage III: Execution-Based Filtering. Following SWE-Bench, we validate each candidate by applying the instance's test patch content and logging test outcomes *before* and *after* the remaining patch content is applied. We retain only instances that exhibit at least one `FAIL_TO_PASS` test (i.e., a test that fails pre-patch and passes post-patch), ensuring a measurable behavioral change attributable to the required evolution. We additionally discard candidates that trigger installation or runtime errors under the benchmark environment. The resulting set comprises evolution tasks with verifiable behavioral deltas and stable execution characteristics.

After these filtering stages, SWE-EVO comprises 48 high-quality task instances. Figure 3 reports their distribution across repositories, while Table 1 summarizes key instance characteristics. Compared to SWE-Bench, SWE-EVO features markedly longer issue descriptions, broader patch scope, and heavier test suites (see Figure 7 in Appendix D), demanding long-horizon reasoning, extensive context understanding, and strong memory capabilities from agents.

3.2 Task Formulation

Having described how SWE-EVO instances are constructed, we now formalize the task definition and evaluation metrics.

Model input. A model is provided with (i) a release-note item that specifies the intended change (bug fix or feature refinement), and (ii) a complete codebase at the *pre-release* commit. The model must produce edits to the codebase that implement the described change. In practice, we represent a solution as a patch files that specifies which lines to modify across files. For clarity, we refer to the patch generated by the model as the `model patch`, while the ground-truth patch extracted from the start-version to end-version change inside the instance is referred to as the `gold patch`.

Evaluation Metrics. Consistent with the evaluation setup of SWE-bench (Jimenez et al., 2024), we use the **Resolved Rate (%)** as the primary metric, representing the proportion of task instances successfully solved by the agent. In addition, we report the **Patch Apply Rate (%)**, which measures

Table 1: Average and maximum numbers characterizing different attributes of a SWE-EVO task instance. Statistics are micro-averages calculated without grouping by repository.

		Mean	Max
Issue Text	Length (Words)	2390.5	22344
Codebase	# Files (non-test)	363	1046
	# Lines (non-test)	78K	272K
Gold Patch	# Lines edited	610.5	4113
	# Files edited	20.9	105
	# Func. edited	51.0	379
Tests	# Fail to Pass	81.4	2774
	# Total	874.0	8552

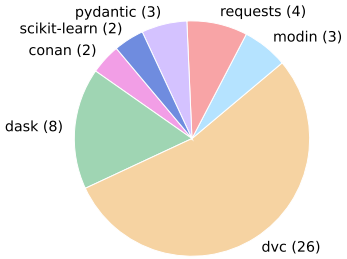


Figure 3: Distribution of SWE-EVO tasks (in parenthesis) across 7 open source GitHub repositories that each contains the source code for a popular, widely downloaded PyPI package.

the percentage of generated patches that are syntactically valid and can be applied to the codebase without errors. Beyond the hard-score **Resolved Rate (%)**, we introduce a soft-score, **Fix Rate (%)**, to provide a more fine-grained assessment of SWE-EVO while remaining consistent with Resolved Rate.

Resolved Rate. Following SWE-bench, this metric focuses on two categories in the test outcomes:

- `FAIL_TO_PASS` tests that were initially failing (`FAILED` or `ERROR`) but pass (`PASSED` after applying the gold patch.
- `PASS_TO_PASS` tests pass both before and after the gold patch and serve as regression checks to ensure unrelated functionality is preserved.

The **Resolved Rate** of an instance is equal 1 if *all* tests in both `FAIL_TO_PASS` and `PASS_TO_PASS` are `PASSED`; otherwise, it is 0. Hence, this metric imposes a strict binary criterion: an instance is counted as resolved only if every relevant test succeeds.

Fix Rate: A Soft Metric. While Resolved Rate provides a clear pass/fail signal, it may hide meaningful partial progress, especially when instances contain thousands of tests (Table 1, Figure 7). In such cases, models may fix many failing tests without fully resolving the instance. To capture this progress, we introduce **Fix Rate**, a soft metric that measures the fraction of `FAIL_TO_PASS` tests successfully fixed.

Fix Rate also enforces a regression constraint: if any `PASS_TO_PASS` test fails after applying the patch, the instance receives a score of 0. This rewards partial fixes while penalizing regressions. Let F_i denote the `FAIL_TO_PASS` tests and P_i the `PASS_TO_PASS` tests for instance i. The **Fix Rate** is defined in Equation 1 as:

$$\text{Fix Rate}(i) = \begin{cases} \frac{\#\{t \in F_i \mid t \text{ passes}\}}{|F_i|}, & \text{if all } t \in P_i \text{ pass;} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

The overall **Fix Rate (%)** is the average across all N instances:

$$\text{Fix Rate} = 100 \times \frac{1}{N} \sum_{i=1}^{N} \text{Fix Rate}(i). \quad (2)$$

Fix Rate lies in $[0, 1]$ and is consistent with Resolved Rate: an instance is resolved when its Fix Rate equals 1. Aggregated across instances, this metric captures partial progress that binary Resolved Rate cannot reflect, while remaining aligned with it (Table 5).

3.3 Features of SWE-EVO

SWE-EVO differs from SWE-Bench along several key dimensions. First, it presents substantially richer supervision: longer specifications, broader patches (more lines, files, and functions edited), and heavier test suites (see Appendix D). Second, unlike SWE-Bench where each task maps to a single pull request, SWE-EVO instances may aggregate multiple PRs, yielding a wide range of difficulty levels (see Appendix F). Finally, evaluation is robust: at least one `FAIL_TO_PASS` test validates each gold patch (81% have two or more), with an additional mean of 793 `PASS_TO_PASS` regression tests per instance.

4 Experiments

4.1 Experiment Setup

We report the experimental results on SWE-EVO. The following section outlines the experimental

scaffold, evaluation metrics, and configurations to ensure reproducibility. Our assessment covers a diverse range of models, including frontier proprietary systems and open-weight models.

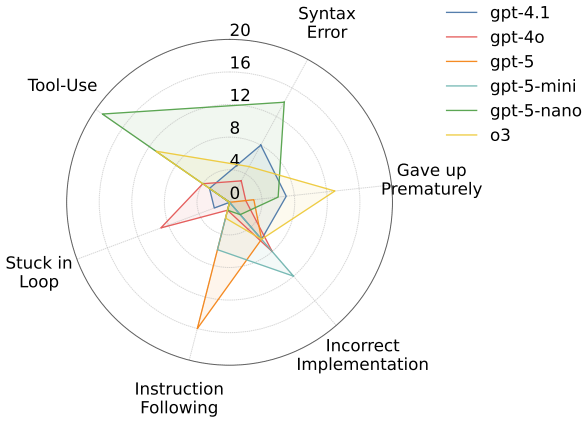
Agents and Model Selection. We evaluate SWE-EVO using two established coding agent frameworks: **OpenHands** (Wang et al., 2024d) (with CodeActAgent, max 100 iterations) and **SWE-agent** (Yang et al., 2024a) (max 100 LLM calls). We test 18 state-of-the-art LLMs spanning five providers: OpenAI (10 models including GPT-5 series, o3, GPT-4.1, GPT-4o, and GPT-oss-120b), DeepSeek (3 models), Zhipu AI (3 models), Qwen (1 model), and Moonshot AI (2 models). The full list of models with references is provided in Table 3 in the Appendix.

For all reasoning models, we use a medium reasoning effort setting, which balances inference cost and accuracy.

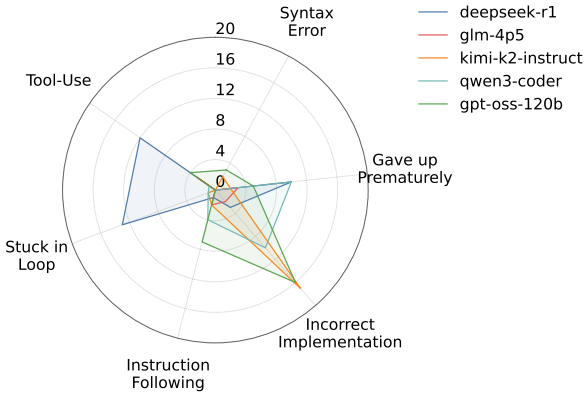
4.2 Performance on SWE-EVO

In SWE-EVO, each task is defined by an original release-note entry that we use as the problem statement. These release notes often reference one or more upstream pull requests or GitHub issues whose contents are typically only available online. However, to prevent models from accessing code from future versions or pull requests, we block internet access during evaluation. Therefore, we evaluate agents under our default setting, *release-note + PR/issue context*, where in addition to the raw release note we provide the content of the pull requests and issues referenced in that release note. For more details on the context format, see Appendix J.1, and for concrete examples of full problem statements, see Appendix J (Boxes J.2 and J.2). We report the performance of both OpenHands and SWE-agent scaffolds across all models on SWE-EVO under this setting in Table 2. For comparison, we also include each model's performance on SWE-bench Verified.

A detailed quantitative comparison of task complexity between SWE-EVO and SWE-Bench (specification length, patch scope, and test-suite size) is provided in Appendix D (Figure 7). Overall, SWE-EVO is substantially more challenging than SWE-bench Verified. Even the strongest model `gpt-5.4` resolves only around **25%** of instances on SWE-EVO, compared to 72.80% resolved by `gpt-5.2` on SWE-bench Verified. Beyond this difficulty, performance on SWE-EVO exhibits



(a) GPT-series models.



(b) Open-source models.

Figure 4: Failure mode distribution for SWE-agent with several model trajectories of unresolved instances. Each instance is automatically labeled using `gpt-5-mini` (OpenAI, 2025a) with categories from Table 6.

a clean and intuitive scaling trend: larger models reliably outperform their smaller variants (e.g., `gpt-5` > `gpt-5-mini` > `gpt-5-nano` or `gpt-5.4` > `gpt-5.2` > `gpt-5`), and the ranking of proprietary models closely mirrors their relative performance on SWE-bench Verified. However, we also observe an unusual exception for `glm-5` and `glm-4p7`. While both models achieve very strong results on SWE-agent, even surpassing `gpt-5.4`, their performance drops sharply on OpenHands, falling below even `glm-4p5`. This suggests that the gap is more likely due to model-specific scaffold sensitivity than benchmark instability. One possible explanation is that these models are more adapted to agent trajectories and prompting styles similar to SWE-agent, but less robust under the CodeActAgent setting used in OpenHands. Understanding this discrepancy is an interesting direction for future work. Taken together, these results indicate that SWE-EVO is a more demanding and

Table 2: Results on SWE-EVO with *release note + PR/issue context*: Resolved and Apply rates for OpenHands and SWE-agent. †Results reported on swebench.com (Jimenez et al., 2024). "—" indicates that the result is not reported on the SWE-bench website.

	Model	SWE-EVO (OpenHands)		SWE-EVO (SWE-agent)		SWE-bench Verified (bash only)
		% Resolved	% Apply	% Resolved	% Apply	% Resolved †
OpenAI	gpt-5.4	**25.00**	97.92	**25.00**	97.92	—
	gpt-5.2	18.75	100	22.92	97.92	72.80
	gpt-5-08-07	18.75	100	20.83	100	65.00
	gpt-5-mini-08-07	10.42	97.92	10.42	100	59.80
	gpt-5-nano-08-07	4.17	85.42	4.17	100	34.80
	o3-2025-04-16	4.17	93.75	6.25	100	58.40
	gpt-4.1-2025-04-14	2.08	87.50	10.42	97.92	39.58
	gpt-4o-2024-11-20	6.25	97.92	6.25	100	21.62
	gpt-oss-120b	2.08	100	6.25	100	26.00
DeepSeek	deepseek-v3p2	20.83	95.83	23.40	95.83	70.00
	deepseek-v3p1	16.67	97.92	10.42	100	—
	deepseek-r1-0528	10.42	100	8.33	100	57.60
Zhipu AI	glm-5	8.33	97.92	37.50	100	72.80
	glm-4p7	4.17	100	39.58	97.92	—
	glm-4p5	16.67	97.92	16.67	100	54.20
Qwen	qwen3-coder-480b-a35b	14.58	97.92	14.58	97.92	55.40
Moonshot AI	kimi-k2p5	22.92	97.92	25.00	97.92	70.80
	kimi-k2-instruct	16.67	100	18.75	100	43.80

more realistic benchmark for evaluating end-to-end software-evolution capabilities.

We also evaluate a more challenging *release-note only* setting, where agents receive no PR/issue context. As shown in Table 4 (in Appendix E), performance drops modestly but trends remain consistent, confirming that SWE-EVO poses substantial challenges regardless of specification detail.

Fine-Grained Analysis with Fix Rate. Beyond the binary *Resolved Rate*, we examine our soft metric *Fix Rate* (Section 3.2), which captures partial progress on large test suites. As shown in Table 5 (Appendix G), models indistinguishable under Resolved Rate can differ meaningfully: e.g., under OpenHands, both `gpt-4.1` and `gpt-oss-120b` resolve only 2.08% of SWE-EVO, but their Fix Rates are 4.65% vs. 2.08%, revealing that `gpt-4.1` repairs more failing tests per instance.

4.3 Analysis of Agent Behaviour

Beyond aggregate metrics, we investigate *why* agents fail on SWE-EVO by analyzing their execution trajectories. This qualitative analysis reveals distinct failure patterns across different model families.

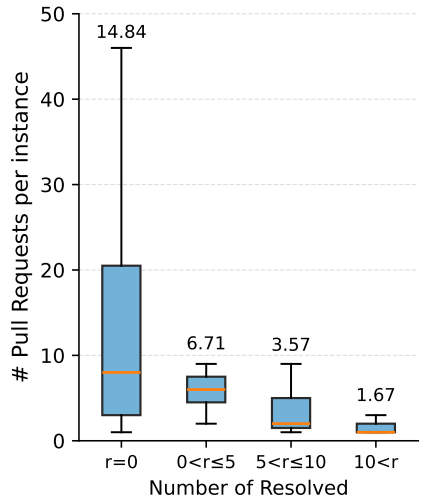
4.3.1 Trajectory Failure Modes Analysis

We perform an LLM-as-a-judge analysis of failure modes for SWE-agent trajectories on SWE-EVO, following the methodology proposed in prior work on SWE-agent (Yang et al., 2024a).

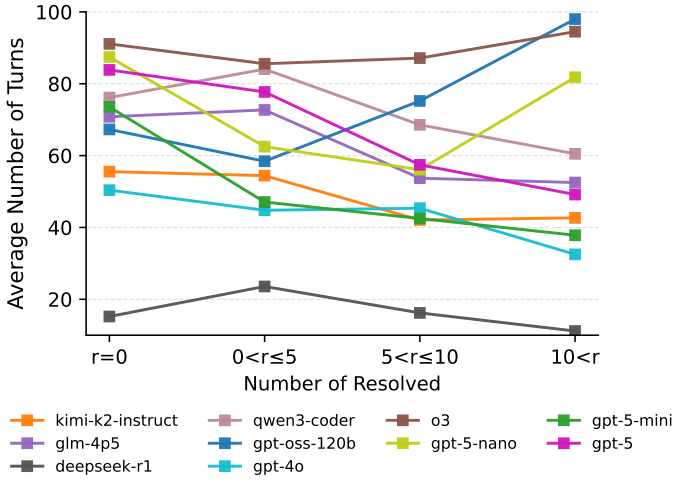
Method. We first run SWE-agent with each model on all SWE-EVO instances and restrict attention to unresolved cases. For every failing trajectory, we extract the last 20 turns, which we found to provide a good balance between capturing the decisive part of the interaction and avoiding unnecessary noise from early exploration. The judge is prompted to first briefly justify its reasoning and then assign exactly one primary failure label from table 6 in Appendix H to the trajectory. This automated labeling allows us to systematically characterize how different models fail on SWE-EVO.

Categories. The failure taxonomy is summarized in Table 6 in Appendix H. It covers both low-level and high-level error modes.

Results. Figure 4 reveals distinct failure patterns across model families. For the GPT series, `gpt-5` rarely fails due to syntax or tool issues; instead, over 60% of failures come from



(a) Average number of pull requests per difficulty group, showing strong correlation between PR count and empirical difficulty.



(b) Average number of turns by model across difficulty groups, illustrating the effort of each model.

Figure 5: Difficulty analysis of SWE-EVO instances: (a) instances associated with more pull requests are significantly harder; (b) stronger models scale turn usage with difficulty, while others show limited adaptivity.

Instruction Following, suggesting difficulty interpreting complex release notes. Smaller variants (`gpt-5-mini`, `gpt-5-nano`) show increasing *Incorrect Implementation*, *Tool-Use*, and *Syntax Error* failures, while older models (`o3`, `gpt-4.1`, `gpt-4o`) exhibit more looping and early-termination issues. In contrast, open and hybrid models follow different patterns: `kimi-k2-instruct`, `qwen3-coder`, and `gpt-oss-120b` fail mainly due to *Incorrect Implementation*, indicating weaker semantic reasoning but stable tool use, whereas `deepseek-r1` often gets stuck in loops or fails in execution, and `glm-4p5` shows more evenly distributed errors across categories.

4.3.2 Difficulty Analysis

A key property of SWE-EVO is that the difficulty of each instance varies widely depending on the number of pull requests. We provide a full distributional analysis of PR counts across instances and repositories in Appendix F (Figure 8). We group instances by their empirical difficulty: for each instance, we count how many times it is successfully resolved across all model-scaffold combinations, yielding a resolution count r. We then group instances into four difficulty groups: $r = 0$, $0 < r \leq 5$, $5 < r \leq 10$ and $r > 10$, containing roughly 64%, 15%, 15%, and 6% of all instances respectively.

Figure 5a shows a clear monotonic trend: easier instances tend to be associated with fewer PRs, while harder ones correspond to substantially more PRs (average 14.84, 6.71, 3.57, and 1.67 across the four groups). Figure 5b reports the average number of turns taken by each model. Stronger models such as `gpt-5` invest more turns on difficult instances and terminate earlier on easier ones. In contrast, `o3` consistently runs at a high turn count regardless of difficulty, and `deepseek-r1` displays notably fewer turns even on hard instances.

5 Conclusion

We introduce SWE-EVO, a benchmark for evaluating coding agents on realistic software evolution rather than isolated bug fixing. SWE-EVO requires interpreting release notes, performing multi-file changes, and preserving functionality under comprehensive tests, making it substantially more challenging than SWE-Bench. Experiments with OpenHands and SWE-agent across 18 models reveal a large capability gap: the best model (`gpt-5.4`) solves only 25% of tasks, compared to 72.80% by `gpt-5.2` on SWE-Bench Verified. Failure analysis shows that stronger models struggle with instruction following, while weaker ones exhibit tool-use and syntax errors. We expect SWE-EVO to complement existing benchmarks with a focus on long-horizon software evolution.

6 Limitations

Our work has several limitations that suggest directions for future research. First, SWE-EVO currently covers only Python projects; prior work has shown that agent performance varies substantially across languages (Badertdinov et al., 2025), and extending to multilingual settings (Zan et al., 2025) would strengthen the generality of our findings. Second, we rely on release notes as the sole specification format, which does not capture all real-world evolution scenarios such as security patches, dependency updates, or internal refactors driven by design documents rather than user-facing changelogs. Third, the benchmark comprises 48 curated task instances: while this ensures high quality with execution-validated behavioral deltas, it limits statistical power for fine-grained comparisons (e.g., per-repository or per-difficulty analysis), and scaling up through automated pipelines (Zhang et al., 2025b; Badertdinov et al., 2025) would improve robustness. Fourth, like all benchmarks derived from public repositories, SWE-EVO is susceptible to data contamination; although version-to-version transitions are less likely to appear verbatim in training data than isolated issue-patch pairs, future iterations should explore temporal separation or evolving benchmark versions to further mitigate this risk. Finally, our Fix Rate metric treats all tests equally and does not account for test complexity, code quality, or maintainability, leaving room for more nuanced evaluation measures.

References

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *International Conference on Learning Representations*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. 2023. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *International Conference on Representation Learning*.

Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, and 1 others. 2025. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*.

Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, and 1 others. 2025. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Cognition. 2024. Devin ai. https://cognition.ai/blog/introducing-devin.

DeepSeek-AI. 2025. Deepseek-r1-0528 release. https://api-docs.deepseek.com/news/news250528. Accessed 2026-03-12.

DeepSeek AI. 2025. DeepSeek V3.1. https://api-docs.deepseek.com/news/news250821.

DeepSeek-AI. 2025a. Deepseek-v3.1 release. https://api-docs.deepseek.com/news/news250821. Accessed 2026-03-12.

DeepSeek-AI. 2025b. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

DORA Research Program. 2025. State of ai-assisted software development: 2025 dora report. https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report. DevOps Research and Assessment (DORA).

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE.

Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. 2025. The current challenges of software engineering in the era of large language models. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30.

Tao Ge, Jing Hu, Xun Wang, Si-Qing Chen, and Furu Wei. 2024. In-context autoencoder for context compression in a large language model. *International Conference on Learning Representations*.

GLM-4.5 Team. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Bernal Jimenez Gutierrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. 2024. Hipporag: Neurobiologically inspired long-term memory for large language models. *Advances in Neural Information Processing Systems*, 37.

Junda He, Christoph Treude, and David Lo. 2025. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi DQ Bui. 2025. Codewiki: Evaluating ai's ability to generate holistic documentation for large-scale codebases. *arXiv preprint arXiv:2510.24428*.

Qishuo Hua, Lyumanshan Ye, Dayuan Fu, Yang Xiao, Xiaojie Cai, Yunze Wu, Jifan Lin, Junfei Wang, and Pengfei Liu. 2025. Context engineering 2.0: The context of context engineering. *arXiv preprint arXiv:2510.26493*.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2024. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752*.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Uttamjit Kaur and Gagandeep Singh. 2015. A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, 118(1):6–11.

Kimi Team. 2025a. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Kimi Team. 2025b. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Kimi Team. 2026. Kimi k2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, and 1 others. 2025. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl.

Dung Nguyen Manh, Nam Le Hai, Anh TV Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi DQ Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. *arXiv preprint arXiv:2305.06156*.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, and 1 others. 2025. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334*.

Dung Manh Nguyen, Thang Chau Phan, Nam Le Hai, Tien-Thong Doan, Nam V Nguyen, Quang Pham, and Nghi DQ Bui. 2025a. Codemmlu: A multi-task benchmark for assessing code understanding & reasoning capabilities of codellms. In *The Thirteenth International Conference on Learning Representations*.

Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. 2025b. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 156–167. IEEE.

OpenAI. 2024a. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/. Accessed 2026-03-12.

OpenAI. 2024b. Swe-bench verified. https://openai.com/index/introducing-swe-bench-verified/.

OpenAI. 2025a. Gpt-5 system card. Technical report, OpenAI.

OpenAI. 2025b. Gpt-5 system card. https://openai.com/index/gpt-5-system-card/. Accessed 2026-03-12.

OpenAI. 2025c. gpt-oss-120b & gpt-oss-20b model card. https://cdn.openai.com/pdf/419b6906-9da6-406c-a19d-1bb078ac7637/oai_gpt-oss_model_card.pdf. Accessed 2026-03-12.

OpenAI. 2025d. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/. Accessed 2026-03-12.

OpenAI. 2025e. Introducing gpt-5.2. https://openai.com/index/introducing-gpt-5-2/. Accessed 2026-03-12.

OpenAI. 2025f. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/. Accessed 2026-03-12.

OpenAI. 2026. Introducing gpt-5.4. https://openai.com/index/introducing-gpt-5-4/. Accessed 2026-03-12.

Charles Packer and 1 others. 2023. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *Preprint*, arXiv:2412.21139.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. 2025. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*.

Huy Nhat Phan, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *CoRR*, abs/2409.16299.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2024. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. *Preprint*, arXiv:2305.14318.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. 2025. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *Preprint*, arXiv:2505.20286.

Qwen Team. 2025. Qwen3-coder. https://github.com/QwenLM/Qwen3-Coder. Accessed 2026-03-12.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. 2025. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. 2024. Raptor: Recursive abstractive processing for tree-organized retrieval. *International Conference on Learning Representations*.

Scale AI. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Chamkaur Singh, Neeraj Sharma, and Narender Kumar. 2019. Analysis of software maintenance cost affecting factors and estimation models. *Int. J. Sci. Technol. Res*, 8(9):276–281.

Hung Quoc To, Minh Huynh Nguyen, and Nghi DQ Bui. 2023. Functional overlap reranking for neural code generation. *arXiv preprint arXiv:2311.03366*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024b. Testeval: Benchmarking large language models for test case generation. *arXiv preprint arXiv:2406.04531*.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. 2025. Huxley-gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. *Preprint*, arXiv:2510.21614.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024c. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024d. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024e. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *Preprint*, arXiv:2401.12869.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*.

Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024a. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, and 1 others. 2024b. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. 2026. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

Z.ai. 2025. Glm-4.7: Advancing the coding capability. https://z.ai/blog/glm-4.7. Accessed 2026-03-12.

Z.ai Team. 2026. Glm-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, and 1 others. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025a. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025b. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, Vienna, Austria. ACM.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. *AAAI Conference on Artificial Intelligence*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Evaluated Models

We evaluate SWE-EVO on a diverse set of recent LLMs from multiple providers, including both closed- and open-weight models. Table 3 lists all evaluated models.

B Software Evolution Conceptual Model

Software evolution is an iterative cycle where, starting from an existing system, engineers identify required changes, analyze their impact, and carry out an evolution process that yields a new system aligned with updated requirements. This cycle repeats as the new system becomes the starting point for subsequent iterations. SWE-EVO captures this process by tasking agents with evolving codebases between consecutive release versions.

Table 3: Full list of LLMs evaluated on SWE-EVO, grouped by provider.

Provider	Model	Type	Reference
OpenAI	gpt-5.4	Closed-weight	(OpenAI, 2026)
	gpt-5.2	Closed-weight	(OpenAI, 2025e)
	gpt-5-08-07	Closed-weight	(OpenAI, 2025b)
	gpt-5-mini-08-07	Closed-weight	(OpenAI, 2025b)
	gpt-5-nano-08-07	Closed-weight	(OpenAI, 2025b)
	o3-2025-04-16	Closed-weight	(OpenAI, 2025f)
	gpt-4.1-2025-04-14	Closed-weight	(OpenAI, 2025d)
	gpt-4o-2024-11-20	Closed-weight	(OpenAI, 2024a)
	gpt-oss-120b	Open-weight	(OpenAI, 2025c)
DeepSeek	deepseek-v3p2	Open-weight	(DeepSeek-AI, 2025b)
	deepseek-v3p1	Open-weight	(DeepSeek-AI, 2025a)
	deepseek-r1-0528	Open-weight	(DeepSeek-AI, 2025)
Zhipu AI	glm-5	Open-weight	(Z.ai Team, 2026)
	glm-4p7	Open-weight	(Z.ai, 2025)
	glm-4p5	Open-weight	(GLM-4.5 Team, 2025)
Qwen	qwen3-coder-480b-a35b	Open-weight	(Qwen Team, 2025; Yang et al., 2025)
Moonshot AI	kimi-k2p5	Open-weight	(Kimi Team, 2026)
	kimi-k2-instruct	Open-weight	(Kimi Team, 2025b)

C Self-Evolving Agents and Tool Creation

Due to the huge design space for software agents, building an optimal agent scaffold can be extremely challenging and costly. As a result, several self-improving and self-evolving software agents have emerged recently. The Self-Improving Coding Agent (SICA) (Robeyns et al., 2025) introduced mechanisms for agents to learn from their own experiences. Darwin-Gödel Machine (DGM) (Zhang et al., 2025a) proposed open-ended evolution of self-improving agents through iterative refinement. Huxley-Gödel Machine (HGM) (Wang et al., 2025) further advanced this paradigm by approximating optimal self-improving machines for human-level coding. However, such self-improving agents typically require costly offline training on known benchmarks and may not generalize well across different LLMs, benchmarks, and issue types.

Beyond the software engineering domain, prior work has explored using LLMs to create tools for general reasoning or embodied tasks. Large Language Models as Tool Makers (LATM) (Cai et al., 2024) demonstrated that LLMs can autonomously create reusable tools to solve complex tasks more efficiently. Voyager (Wang et al., 2024a) introduced an open-ended embodied agent that continuously acquires new skills through code generation in Minecraft. CREATOR (Qian et al., 2024) proposed disentangling abstract and concrete reasoning through tool creation. TroVE (Wang et al., 2024e) focused on inducing verifiable and efficient toolboxes for programmatic tasks. Alita (Qiu et al., 2025) presented a generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. While these works demonstrate the potential of self-evolving agents and tool creation, they do not specifically target real-world software engineering problems that require long-horizon evolution across multiple commits and versions.

D SWE-EVO vs. SWE-Bench Comparison

Compared to SWE-Bench, as shown in figure 7, SWE-EVO presents substantially richer supervision and stricter verification signals. The natural-language specification is markedly longer, the gold patches modify more lines, span more files, and touch more functions, and the associated test suites include many more originally failing tests alongside larger total counts. These quantities exhibit a pronounced long tail, indicating that SWE-EVO captures broader, multi-edit changes and stronger regression risk. Overall, SWE-EVO requires deeper semantic understanding, broader reasoning, and stronger generalization across complex multi-file edits than SWE-Bench, making it a more challenging and realistic benchmark for evaluating software capabilities.

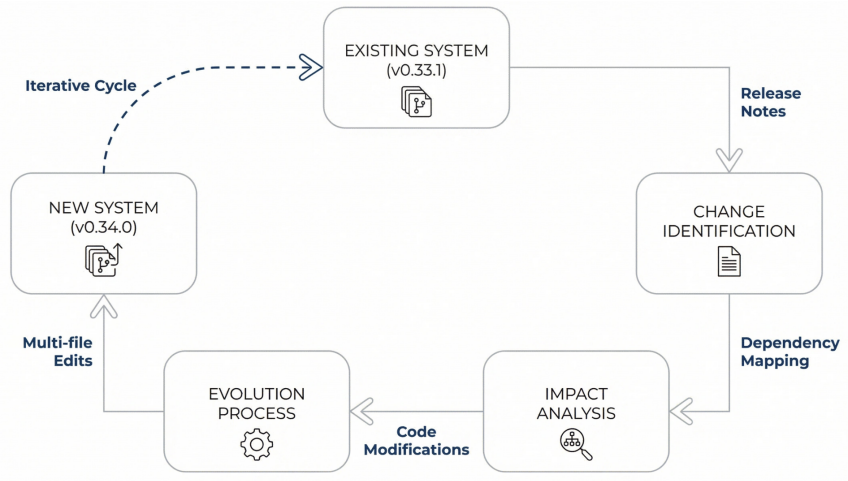


Figure 6: Conceptual model of software evolution, depicting the iterative cycle from an existing system to a new system through change identification, impact analysis, and the evolution process.

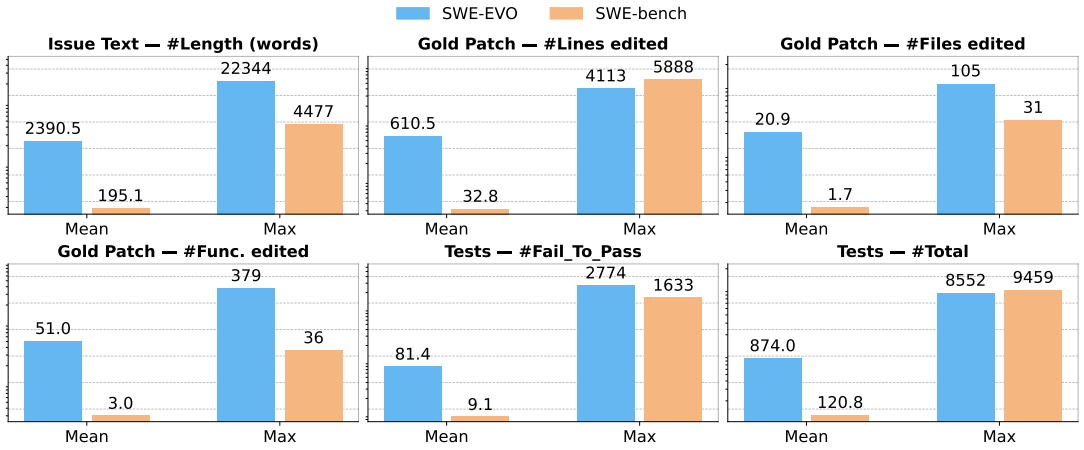


Figure 7: SWE-EVO is markedly more demanding than SWE-Bench, with longer issue descriptions, broader patch scope (more lines/files/functions edited), and heavier test suites (`FAIL_TO_PASS` and total) in both mean and max. These properties require agents capable of long-context reasoning, coordinated multi-file edits, and regression-safe fixes.

E Context Comparison Results

We further compare performance under *release-note only* and *release-note + PR/issue context* settings. As shown in Table 4, adding PR/issue context generally improves resolved rates across most models, while preserving the overall ranking and relative trends. This suggests that additional context provides useful signals.

F Pull Request Distribution Analysis

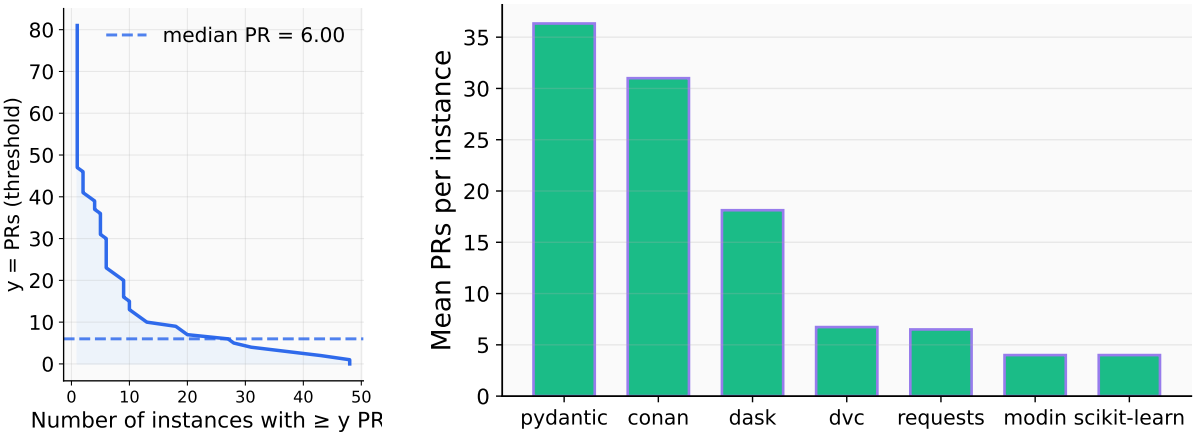
Unlike SWE-Bench, where each task instance corresponds to a single pull request (PR), an instance in SWE-EVO may be associated with multiple pull requests that collectively implement or refine a release-note change. We hypothesize that instances associated with a larger number of pull requests reflect more complex, multi-step development efforts, as each pull request typically represents a distinct feature enhancement or bug fix. As shown in Figure 8, the number of pull requests per instance varies widely, indicating that SWE-EVO spans a broad range of difficulty levels.

G Fix Rate Results

Beyond the binary *Resolved Rate*, we also examine our soft metric *Fix Rate*, which captures partial progress on large test suites (Section 3.2). As shown in Table 5, models that appear similar under Resolved Rate can differ meaningfully once partial improvements are considered. For example, under OpenHands, both `gpt-4.1` and `gpt-oss-120b` resolve only (2.08%) of SWE-EVO, but their Fix

Table 4: Results on SWE-EVO under two settings: **release-note only** and **release-note + PR/issue context**. Reported metric is **Resolved Rate (%)** on **OpenHands** and **SWE-agent**.

	release-note only		release-note + PR/issue context	
Model	OpenHands	SWE-agent	OpenHands	SWE-agent
gpt-5-08-07	**14.58**	**16.67**	**18.75**	**20.83**
gpt-5-mini-08-07	8.33	10.42	10.42	10.42
o3-2025-04-16	4.17	6.25	4.17	6.25
gpt-4.1-2025-04-14	2.08	8.33	2.08	10.42
gpt-4o-2024-11-20	8.33	4.17	6.25	6.25
gpt-oss-120b	0.00	0.00	2.08	6.25
deepseek-r1-0528	10.42	6.25	10.42	8.33
glm-4p5	6.25	12.50	16.67	16.67
qwen3-coder-480b-a35b	**14.58**	12.50	14.58	14.58
kimi-k2-instruct	8.33	12.50	16.67	18.75



(a) Complementary cumulative distribution of PR counts, showing how many instances contain at least y linked pull requests.

(b) Mean PRs per repository, highlighting that certain codebases require substantially more upstream context to resolve.

Figure 8: Pull Request statistics across SWE-EVO instances.

Rates differ (4.65% vs. 2.08%), indicating that `gpt-4.1` consistently fixes more failing tests per instance. More broadly, we observe a consistent gap between Fix Rate and Resolved Rate across all models, suggesting that many trajectories make partial but incomplete progress. This finer granularity is especially important on SWE-EVO, where each task involves many tests, and binary outcomes would otherwise obscure systematic differences in model capability.

H Failure Mode Descriptions

To better understand how coding agents fail on SWE-EVO, we adopt a coarse but interpretable taxonomy of failure modes, inspired by prior analysis of SWE-agent (Yang et al., 2024a) trajectories on SWE-bench Lite. For each unresolved instance, we extract the SWE-agent trajectory and ask a judge model to assign exactly one primary failure label from a fixed set of categories.

The taxonomy is designed to balance coverage and reliability: it separates low-level execution issues (e.g., syntax errors or broken tool calls) from higher-level semantic problems (e.g., incorrect logic or misinterpreted requirements), and includes behavioural modes such as looping or prematurely giving up. When multiple issues are present, the judge is instructed to choose the most fundamental cause that best explains why the instance ultimately remains unresolved.

Syntax Error captures patches that break parsing or formatting, preventing tests from running. **Incorrect Implementation** denotes patches that touch the right area of the code but do not correctly imple-

Table 5: Comparison of **Resolved Rate (%)** and **Fix Rate (%)** across models on **OpenHands**, **SWE-Agent**, and their **Average**.

	Model	OpenHands		SWE-Agent		Average	
		Resolved (%)	Fix (%)	Resolved (%)	Fix (%)	Resolved (%)	Fix (%)
OpenAI	gpt-5.4	**25.00**	**33.89**	**25.00**	**33.98**	**25.00**	**33.96**
	gpt-5.2	18.75	23.43	22.92	30.21	20.84	26.82
	gpt-5-08-07	18.75	27.64	20.83	31.46	19.79	29.55
	gpt-5-mini-08-07	10.42	17.48	10.42	17.48	10.42	17.48
	gpt-5-nano-08-07	4.17	5.99	4.17	5.26	4.17	5.63
	o3-2025-04-16	4.17	6.47	4.17	13.72	4.17	10.10
	gpt-4.1-2025-04-14	2.08	4.65	10.42	14.79	6.25	9.72
	gpt-4o-2024-11-20	6.25	7.77	6.25	10.15	6.25	8.96
	gpt-oss-120b	2.08	2.08	6.25	7.88	4.17	4.98
DeepSeek	deepseek-v3p2	20.83	26.89	23.40	31.41	22.12	29.15
	deepseek-v3p1	16.67	21.83	10.42	15.51	13.55	18.67
	deepseek-r1-0528	10.42	14.31	8.33	9.89	9.38	12.10
Zhipu AI	glm-5	8.33	9.67	37.50	44.27	22.92	27.45
	glm-4p7	4.17	5.19	39.58	45.87	21.88	25.53
	glm-4p5	16.67	23.74	16.67	26.55	16.67	25.15
Qwen	qwen3-coder-480b-a35b	14.58	19.56	14.58	23.74	14.58	21.65
Moonshot AI	kimi-k2p5	22.92	27.75	**25.00**	32.91	23.96	30.33
	kimi-k2-instruct	16.67	22.42	18.75	24.03	17.71	23.23

ment the required behaviour. **Instruction Following** indicates that the agent misinterprets or ignores the release note, effectively solving the wrong task. **Tool-Use** covers errors in invoking the SWE-agent tools (e.g., failing to run tests, misusing edit/apply commands, or opening the wrong paths). **Stuck in Loop** reflects trajectories where the agent repeatedly reads files or reruns tests without making meaningful progress. **Gave Up Prematurely** corresponds to agents that stop early or declare failure while viable next steps remain. Finally, **Other** collects rare or ambiguous patterns that do not cleanly fit into the above category. Table 6 summarizes the categories and the criteria used by the judge model when annotating trajectories.

I Dataset Fields

Table 7 describes the SWE-EVO dataset fields and outlines how they are obtained throughout the curation process.

J Problem Statement

For each SWE-EVO instance, the problem statement is defined by the official release notes text that describes the changes between the *start* version and the *end* version of this instance, as published by the project maintainers. This release note is then presented to the agent as the sole natural-language specification of how the codebase should change between the two versions.

J.1 Pull Request, Issue Context

Release notes frequently refer to specific pull requests or issues (e.g., "see #1234" or "thanks to PR #5678"). To make these references actionable for agent, we augment the problem statement with the context of the linked artifacts. For each referenced pull request or issue, we fetch its body from github. These texts are then concatenated into a compact "PR / Issue Context" block that is presented to the agent alongside the release note. We then append these texts directly below the release note in a structured format, without rewriting or summarizing them. In general, the final problem statement presented to the agent has the form: *release-note* followed by a sequence of sections `\n### PR xxx:\n` or `\n### Issue yyy:\n` with their corresponding content for every referenced pull request and issue. so that the agent sees the original release note followed by the full text of any referenced pull requests and issues.

J.2 Example

To make this concrete, we present two example problem statements (Boxes J.2 and J.2), corresponding to the instances `iterative__dvc_0.33.1_0.34.0`

Table 6: Descriptions of failure mode categories.

Category	**Description**
Syntax Error	The patch or edits introduced syntax or formatting mistakes (e.g., missing imports, bad indentation, invalid JSON/YAML) that prevented tests or execution from succeeding.
Incorrect Implementation	The agent made a change to a reasonable area but their solution didn't correctly address the issue.
Instruction Following	The agent misread or deviated from the stated requirements (issue/release note), implemented something else, or ignored an explicit instruction.
Tool-Use	Progress was blocked by incorrect or missing tool usage (e.g., failing to run tests, bad edit/apply commands, wrong path in `open`/`ls`), so the solution could not be completed.
Stuck in Loop	The agent repeated similar actions (reading, scrolling, retrying edits/tests) without making substantive progress toward a fix.
Gave Up Prematurely	The agent stopped or concluded early after encountering difficulty, without exhausting reasonable next steps.
Other	There was some other problem that prevented the agent from resolving this issue.

in the `iterative/dvc` repository and `dask__dask_2023.6.1_2023.7.0` in the `dask/dask` repository.

Table 7: Required fields for a typical issue-solving task instance. Fields marked with * are newly added compared to SWE-bench.

Field	Type	Description
repo	str	Git repository identifier for the task instance, e.g., the GitHub owner/name.
base_commit	str	The commit on which the pull request is based, representing the repository state before the issue is resolved.
start_version	str	Git tag or version identifier of the starting release for this task instance.
end_version	str	Git tag or version identifier of the target release after the change has been applied.
end_version_commit	str	Git commit hash corresponding to the end_version tag.
patch	str	Gold patch proposed by the pull request, in .diff format.
test_patch	str	Modifications to the test suite proposed by the pull request that are typically used to check whether the issue has been resolved.
problem_statement	str	Issue description text, typically describing the bug or requested feature, used as the task problem statement.
FAIL_TO_PASS	List[str]	Test cases that are expected to successfully transition from failing to passing and are used to evaluate the correctness of the patch.
PASS_TO_PASS	List[str]	Test cases that are already passing prior to applying the gold patch; a correct patch should not introduce regression failures in these tests.
*image	str	Instance-level Docker image that provides an execution environment.
*test_cmds	List[str]	Command(s) used to run the test suite, as identified by the verify agent in REPOLAUNCH, enabling detailed logging of each test item's status (e.g., via pytest -rA).
*log_parser	str	Type of log parser required for the instance—by default, pytest.

Problem Statement: iterative__dvc_0.33.1_0.34.0

1) [‘dvc metrics show‘ now nicely formats multiline metrics files like tsv/htsv, csv/hcsv, json](https://github.com/iterative/dvc/issues/1716); Kudos @mroutis :medal_sports:
2) [‘dvc remote add‘ no longer silently overwrites existing sections](https://github.com/iterative/dvc/issues/1760);
3) [Use a workaround to bypass SIP protection on osx, when accessing libSystem](https://github.com/iterative/dvc/issues/1515);
4) [Don’t try to create existing container on azure](https://github.com/iterative/dvc/issues/1811); Kudos @AmitAronovitch :medal_sports:
5) Dvc repository now uses read-only http remote for our images instead of s3;
6) Fix bug in ‘dvc status‘ where an error is raised if cache is not present locally nor on the remote;
7) [Fix progress bar on ‘dvc pull‘]
(https://github.com/iterative/dvc/issues/1807); Kudos @pared :medal_sports:
8) [Automatically detect metrics file type by extension](https://github.com/iterative/dvc/issues/1553); Kudos @cand126 :medal_sports:

Welcome new contributor @AmitAronovitch ! :tada:

Issue 1716:
When displaying TSV metrics output the printout is not readable:
```
../../20181223-TrainSetZero/SanityCheck/Bravo_on_TrainSetZero.metrics:
value_mse deviation_mse data_set
0.421601 0.173461 train
0.67528 0.289545 testing
0.671502 0.297848 validation
```
I think it would be much easier to read if a newline were added, and the remaining text were formatted as though it had been passed via `column -t`:
```
../../20181223-TrainSetZero/SanityCheck/Bravo_on_TrainSetZero.metrics:
value_mse deviation_mse data_set
0.421601 0.173461 train
0.67528 0.289545 testing
0.671502 0.297848 validation
```

Issue 1807:
```
(3.7.0-dvc) →dvc git:(dvctags) ×dvc pull
Preparing to download data from 's3://dvc-share/dvc/'
Preparing to collect status from s3://dvc-share/dvc/
[##############################] 100% Collecting information
[##############################] 100% Analysing status.
(1/3): [##############################] 100% dvc_up.bmp
(2/3): [##############################] 100% dvc.ico
(3/3): [##############################] 100% dvc_left.bmp
(4/3): [##############################] 100% Checkout finished!
```
looks like checkout didn’t reset progress counter

Issue 1553:
E.g. if we see that output has .json suffix, we could safely assume that it is `-type json` without explicitly specifying it.

Issue 1760:
The `dvc remote add` command ignores the existing remote and overwrites it silently.
`→dvc -version`
`0.32.1+7d7ed4`
To reproduce
`dvc remote add s3 s3://bucket/subdir`
`dvc remote add s3 s3://bucket/subdir2`
Expected behavior
The second command `dvc remote add s3 s3://bucket/subdir2` should fail with the Remote with name "s3" already exists message.
Current behavior
Remote URL is silently overwriten:
`> cat .dvc/config`
`[ 'remote "s3"' ]`
`url = s3://bucket/subdir2`

Issue 1811:
Azure SAS connection strings can be used to allow read-only access to specific container, which is useful in the context of CI pipelines and automation.
However, when using such limited-credentials connection string, `dvc pull` abends with the error below.
Some digging reveals that this is because `remote/azure.py` always tries to create the bucket (which already exists). If we check for existence before trying to create - this should work...
(will send PR)
`[######### ] 30% Collecting information`
`Client-Request-ID=19129d6a-5393-11e9-80ab-5800e34ea0d9 Retry policy did not allow for a retry: Server-Timestamp=Sun, 31 Mar 2019 08:58:06 GMT, Server-Request-ID=c72fbeda-501e-0089-3c9f-e78298000000, HTTP status code=403, Exception=This request is not authorized to perform this operation. ErrorCode: AuthorizationFailure <?xml version="1.0" encoding="utf-8"?><Error><Code>AuthorizationFailure</Code><Message>This request is not authorized to perform this operation.`
`RequestId:c72fbeda-501e-0089-3c9f-e78298000000`
`Time:2019-03-31T08:58:06.2059267Z</Message></Error>.`
Error: failed to pull data from the cloud - This request is not authorized to perform this operation.
ErrorCode: AuthorizationFailure
`<?xml version="1.0" encoding="utf-8"?><Error><Code>AuthorizationFailure`
`</Code><Message>This request is not authorized to perform this operation.`
`RequestId:c72fbeda-501e-0089-3c9f-e78298000000`
`Time:2019-03-31T08:58:06.2059267Z</Message></Error>`
Having any troubles? Hit us up at https://dvc.org/support, we are always happy to help!
Please provide information about your setup
DVC version: 0.29.0+220b4b.mod
linux x86_64 py3.6 pip

Issue 1515:
`$ dvc -V`
`0.23.2+bad2ef.mod`
DVC creates hardlinks instead of reflinks in APFS.
File system:

```
$ diskutil info / | grep -i system
File System Personality: APFS
```
See last two lines:
```
$ dvc add Tags.xml -v
Debug: PRAGMA user_version;
Debug: fetched: [(3,)]
Debug: CREATE TABLE IF NOT EXISTS state (inode INTEGER PRIMARY KEY, mtime TEXT
NOT NULL, size TEXT NOT NULL, md5 TEXT NOT NULL, timestamp TEXT NOT NULL)
Debug: CREATE TABLE IF NOT EXISTS state_info (count INTEGER)
Debug: CREATE TABLE IF NOT EXISTS link_state (path TEXT PRIMARY KEY, inode
INTEGER NOT NULL, mtime TEXT NOT NULL)
Debug: INSERT OR IGNORE INTO state_info (count) SELECT 0 WHERE NOT EXISTS
(SELECT * FROM state_info)
Debug: PRAGMA user_version = 3;
Debug: Skipping copying for '/Users/dmitry/src/modules-example/tmp/
test/Tags.xml', since it is not a symlink or a hardlink.
Adding 'Tags.xml' to '.gitignore'.
Saving 'Tags.xml' to cache '.dvc/cache'.
Debug: Path /Users/dmitry/src/modules-example/tmp/test/Tags.xml inode
12888021464
Debug: SELECT * from state WHERE inode=12888021464
Debug: fetched: []
Debug: INSERT INTO state(inode, mtime, size, md5, timestamp) VALUES
(12888021464, "15478541678481879 04", "0", "6d861a1605e60b2f3a977a5a2a4419a2",
"15478541786091269 12")
Debug: File '/Users/dmitry/src/modules-example/tmp/test/.dvc/cache/6d/
861a1605e60b2f3a977a5a2a4419a2', md5 '6d861a1605e60b2f3a977a5a2a4419a2',
actual 'None'
Debug: Cache type 'reflink' is not supported: reflink is not supported
Debug: Created 'hardlink': /Users/dmitry/src/modules-example/tmp/
test/.dvc/cache/6d/861a1605e60b2f3a977a5a2a4419a2 →/Users/dmitry/src/
modules-example/tmp/test/Tags.xml
```

Problem Statement: dask__dask_2023.6.1_2023.7.0

2023.7.0
Released on July 7, 2023
Enhancements

- Catch exceptions when attempting to load CLI entry points (#10380) Jacob Tomlinson

Bug Fixes

- Fix typo in `_clean_ipython_traceback` (#10385) Alexander Clausen
- Ensure that `df` is immutable after `from_pandas` (#10383) Patrick Hoefler
- Warn consistently for `inplace` in `Series.rename` (#10313) Patrick Hoefler

Documentation

- Add clarification about output shape and reshaping in rechunk documentation (#10377) Swayam Patil

Maintenance

- Simplify `astype` implementation (#10393) Patrick Hoefler
- Fix `test_first_and_last` to accommodate deprecated `last` (#10373) James Bourbeau
- Add `level` to `create_merge_tree` (#10391) Patrick Hoefler
- Do not derive from `scipy.stats.chisquare` docstring (#10382) Doug Davis

PR 10385:
Regression from (#10354) https://api.github.com/repos/dask/dask/pull/10354 - exception types of unhandled exceptions in ipython contexts were mangled and always set to type.
To reproduce, run the following in a jupyter notebook cell with dask 2023.6.1:
```
import dask
raise TypeError("wat")
```
Then, observe the websocket connection to jupyter; the message with `msg_type="error"` looks like this:
(Image Link)
Notice `content["ename"]` is set to "type".
With dask 2023.6.0, `content["ename"]` is properly set to "TypeError":
(Image Link)
These values aren't inconsequential - they end up being serialized into the ipynb file, and cause hard to debug issues.
(nb: I would prefer if this kind of issue was not something that can be caused by `import dask`, by opting in to this functionality by configuration or actively registering the hook functions)

- Tests added / passed
- Passes `pre-commit run --all-files`

Thanks to (@matbryan52) for the help debugging this.
PR 10383:

- Tests added / passed

- Passes `pre-commit run --all-files`

We had a similar problem in dask-expr

PR 10393:

- Tests added / passed

- Passes `pre-commit run --all-files`

PR 10391:

- Passes `pre-commit run --all-files`

PR 10373:
This fixes
`FAILED dask/dataframe/tests/test_dataframe.py::test_first_and_last[last] - FutureWarning: last is deprecated and will be removed in a future version. Please create a mask and filter using .loc instead`
which we're currently seeing in the upstream build (xref (#10347) (Workflow Run URL)
Python 3.10 Test Summary
`dask/dataframe/tests/test_multi.py::test_concat_categorical[True-False-True]: [XPASS(strict)] fails on pandas dev:` https://api.github.com/repos/dask/dask/issues/10558
`dask/dataframe/tests/test_multi.py::test_concat_categorical[False-False-True]: [XPASS(strict)] fails on pandas dev:` https://api.github.com/repos/dask/dask/issues/10558
Note that we were already emitting a FutureWarning for `DataFrame.last`, so this is a tests-only PR
cc (@j-bennet) (@phofl)

PR 10382:

- Closes (#10381) Looks like our docs are running into an error on main. See this build from a recent PR https://readthedocs.org/projects/dask/builds/21146384/

x Tests added / passed

- Passes `pre-commit run --all-files`

`scipy.stats.chisquare` recently started using a doi Sphinx directive that exists in their repo but it's not importable. This is breaking dask docs generation. We can just point folks to that docstring instead of deriving from it.
for reference here is the PR in scipy: (scipy/scipy#17682)

PR 10377:

Closes (#10361) It is not documented anywhere that `dask.array.rechunk` expects the output to have the same shape as the input and does not allow reshaping. The validation step is also quite hidden. This has led to some confusion (dask/distributed#7897 (comment)), so I think it would be good to add a remark about this in the documentation.

- Tests added / passed

- Passes `pre-commit run --all-files`

PR 10313:
The previous warning seemed very inconsistent

PR 10380:

- Closes (#10379) If a package registers an entry point for the CLI but the entrypoint is broken or not importable the CLI cannot work at all.

This is a little hard to create an MRE for but I seeing this with dask-ctl when updating textual to a more recent version. The error comes from the following line.
(dask/dask/cli.py)
Line 97 in `/dask/dask/commit/85c99bc20abc382774cfb6e5bf5f2db76ac0937885c99bc`
`command = entry_point.load()`
When this line is called third-party code is loaded, so we have no control over what happens. We could broadly catch exceptions here, print a warning and carry on.

- Tests added / passed

- Passes `pre-commit run --all-files`