Beyond Isolated Tasks: A Framework for Evaluating Coding Agents on Sequential Software Evolution

Anonymous Authors[1]

Abstract

Real software development is cumulative: changes persist, tests accumulate, and every new line of code becomes a dependency for future work. Yet most coding-agent benchmarks evaluate isolated tasks from clean repository states, missing temporal and stateful pressures that arise when agents are used repeatedly on the same codebase. Thus, to capture this dynamic of software engineering, we introduce SWE-STEP, a stateful and temporal evaluation framework. Instead of treating tasks as independent, SWE-STEP evaluates agents across continuous, temporally ordered pull requests (PRs) to measure not just immediate functional correctness, but long-term repository health. We instantiate this framework with SWE-STEP-Full - comprising 168 tasks and 963 PRs across six Python repositories - and test agents in both sequential (conversational coding) & bundled workflows (requirements are provided upfront). Our experiments reveal that stateless, isolated evaluations (e.g., SWE-Bench) overestimate agent capabilities by up to 20 percentage points by masking spillover effects of past mistakes. Furthermore, we find that even when agents pass functional tests, they steadily degrade repository health by introducing higher cognitive complexity & technical debt than human developers.

1. Introduction

LLM-based coding agents are increasingly moving from simple coding tasks (Chen et al., 2021) into software development workflows, including pull request review, debugging, and even repository generation from scratch (Zhao et al., 2025; Wang et al., 2025; Neubig, 2024; Cihan et al., 2025). However, real-world software development is not well characterized as either solving isolated programming tasks or generating repositories from scratch; it is typically an incremental process (Zaidman et al., 2010) of modifying an existing, evolving system, where each change must be made within a codebase already shaped by prior changes. These prior changes typically span heterogeneous engineering activities (bug fixes, feature additions, refactors), a pattern evident from version bump logs in libraries such as NumPy, SciPy, and pandas[1] Thus, the repository state encountered by a developer or agent reflects the accumulated effects of many overlapping changes, each introduced to satisfy a different goal. *This cumulative nature of software evolution means that code must remain not only functionally correct, but also regression-safe, understandable, and extensible over time* (Lehman, 2005; Bennett & Rajlich, 2000; Martin, 2009).

Existing SWE benchmarks have made important progress by grounding evaluation in real GitHub issues. SWE-bench (Jimenez et al., 2024) and SWE-gym (Pan et al., 2025), for example, evaluate agents on repository-level tasks with real issue descriptions and tests. Yet these benchmarks remain largely *stateless*: each task begins from a clean, human-validated repository state. This reset-based design is useful for measuring single-issue repair ability, but it removes key pressures of real development: previous edits persist, test suites grow, regressions must be avoided, context accumulates, and technical debt can compound (Kruchten et al., 2012; Letouzey, 2012). Recent long-horizon benchmarks such as SWE-EVO (Thai et al., 2025) highlight the need to move beyond isolated issue repair. However, two critical aspects remain underexplored: the conversational coding setting (Tang et al., 2026), where developers issue a sequence of requests to the same agent within an evolving repository state; and second, evaluating not only functional correctness but also stepwise regression preservation and repository health over time.

This motivates our central question: *if a coding agent is deployed into a real repository at time t_{start} and asked to implement the sequence of changes that historically occurred until t_{end}, can it continue solving tasks while pre-*

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

[1]Release notes: NumPy https://numpy.org/doc/2.1/release/1.24.3-notes.html; SciPy https://docs.scipy.org/doc/scipy/release/1.16.0-notes.html; pandas https://pandas.pydata.org/pandas-docs/stable/whatsnew/v2.0.1.html.

serving previous functionality and maintaining code health? This framing also reflects the repository-specific nature of software evolution: the difficulty of sustained code modification depends on the structure of the codebase, the density and coverage of its tests and the clarity of task specifications. Consequently, agent performance need not transfer uniformly across repositories - i.e, although larger LLMs generally achieve strongest overall results, the performance gap to lower-cost models varies.

To answer this question, we evaluate agents over historical development windows. Within each window, we retain temporally adjacent PRs so that each request is evaluated on the repository state on which it was developed; skipping intermediate PRs can omit symbols, APIs, refactors, or tests that later changes depend on, making the resulting PR ill-defined. Crucially, these windows are not intended to define a single semantically unified task. Rather, they define bounded repository-evolution trajectories that include heterogeneous changes, which may contain both dependency-linked and weakly coupled changes. This distinction is useful: dependency-linked PR chains test whether agents can handle cumulative code-level coupling, while weakly coupled chains isolate other stateful-development pressures such as cascading tests, prior agent edits, context interference, and repository-state drift. Studying both lets us disentangle degradation caused by direct inter-PR dependencies from degradation caused by statefulness itself.

We operationalise this idea through **SWE-STEP** (**S**tateful and **T**emporal **E**valuation **P**rocess), a framework for constructing and evaluating coding-agent tasks from git history. Given a repository, an agent, a temporal window, and an evaluation setting, SWE-STEP extracts adjacent PR chains, builds per-PR task specifications, validates executability through test-driven filtering, and evaluates agents under persistent repository state. Each per-PR specification contains two components: a human-written PR or task summary, and a list of functions/classes to add or modify. Within this framework, we study two evaluation settings - the *conversational coding* setting, where agent receives one PR-style specification at a time, while repository state, prior edits, conversation history, and regression tests persist across the chain. The other is *Aggregated Task Specification* (ATS) setting, where same per-PR specifications are provided upfront as a bundled spec. The *ATS* setting allows us to study how agent's planning impacts its software evolution capabilities.

A reliable coding agent evaluation must also guard against benchmark exploitation. Prior work shows that agents can pass tests without solving the task, e.g., by hardcoding expected outputs, modifying tests, overloading operators, or copying future fixes from repository history (Zhong et al., 2025; deb, 2026; ber, 2026). SWE-STEP therefore makes execution protocol part of the framework: we lock test files, remove future git history, restrict internet access, and provide standardized harness-mediated test feedback. These controls ensure that success reflects implementation ability rather than evaluation gaming.

Our contributions and empirical findings are:

- We introduce **SWE-STEP**, a stateful and temporal evaluation framework that constructs coding-agent tasks from temporally-ordered, pull-request chains and measures functional correctness and repository-health evolution.
- The framework supports two settings: conversational coding (sequential PR-style requests) and ATS coding (bundled upfront specification) and includes an anti-gaming execution protocol with locked tests, pruned git history, restricted execution, and standardized harness-mediated feedback to ensure measured success reflects implementation ability rather than benchmark exploitation.
- We release **SWE-STEP-Full**, a dataset of 168 tasks spanning 963 PRs across six popular Python repositories, together with stratified subsets **SWE-STEP-Lite** (50 tasks; 276 PRs) and **SWE-STEP-Mini** (20 tasks; 105 PRs) for lower-cost evaluation.
- Across a tiered evaluation matrix spanning three agents (OpenHands, Aider, Codex) and five LLMs in both frontier and cost-effective tiers, we show that isolated PR evaluation (SWE-bench style) substantially overestimates agent capability. PR-level success rate drops by up to 20.0 pp on SWE-STEP-Mini and 19.9 pp on SWE-STEP-Lite when moving from isolated-PR evaluation to stateful conversational coding. Long dependency-linked chains further widens this gap to 40.9 pp.
- Beyond functional correctness, we show that even agent patches that pass all tests can degrade repository health by increasing cognitive complexity and SQALE technical debt relative to human-written code, motivating that evaluation criteria should move beyond pass rates.

2. Related Work

Before detailing our framework, we situate this work against three threads of prior research: SWE agents, code-generation datasets, and repository-scale benchmarks. Software engineering agents have evolved from code-completion systems to autonomous tools capable of multi-step repository editing, including SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Aider (Gauthier, 2023). Existing evaluations, however, primarily measure immediate functional correctness on isolated tasks. Earlier code-generation benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), CodeMMLU (Nguyen et al.), APPS (Hendrycks et al., 2021), CodeFeedback (Zheng et al., 2024), and ConvCodeBench (Han et al., 2025) evaluate standalone function synthesis or algorithmic reasoning, but do not capture repository-level context, persistent state, or long-term soft-

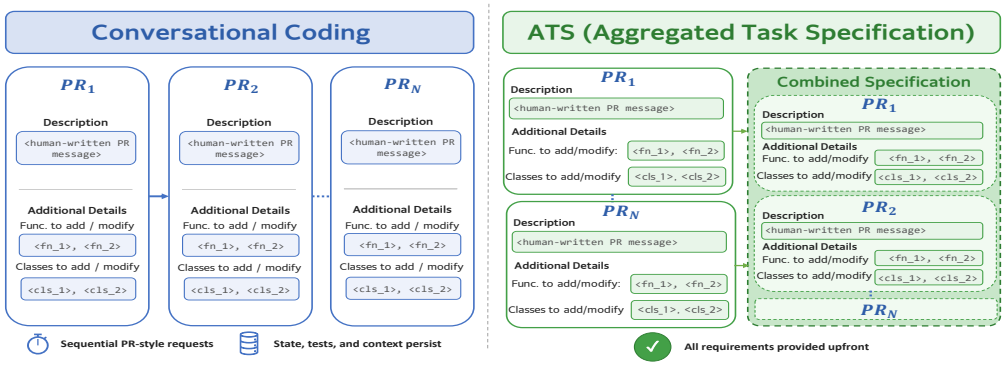


Figure 1. **Input presented to the agent under Conversational coding and ATS settings.**

ware evolution.

Repository-level benchmarks such as SWE-bench (Jimenez et al., 2024), SWE-Gym (Pan et al., 2025), FEA-Bench (Li et al., 2025), COMMIT-0 (Zhao et al., 2024), and CodeAssistBench (Kim et al., 2025) evaluate agents on realistic issues from real codebases, but largely follow a single-issue, reset-based protocol. Recent efforts such as SWE-Evo (Thai et al., 2025), EvoClaw (Deng et al., 2026), and SWE-Bench-CL (Joshi et al., 2025) move toward sequential software evolution, but do not jointly evaluate persistent repository state, cross-PR regression verification, repository-health degradation, verified inter-PR dependencies, and anti-gaming safeguards. SWE-STEP complements these benchmarks by evaluating whether agents can sustain correctness and maintainability across temporally ordered PR-style changes. A detailed comparison is present in the appendix Table 4.

3. Dataset Design and Construction

Having identified the gaps in existing benchmarks, we now formalize SWE-STEP. We start from three desiderata in Section 3.1, describe the data-generation pipeline that operationalizes them in Section 3.2, and instantiate the framework as the SWE-STEP dataset in Section 3.3.

3.1. Dataset Desiderata & Framework Formulation

Following the motivation in Section 1, an evaluation framework should capture three properties: First, *sequential evolution*: tasks should form chains of PRs where code changes accumulate over time. Second, *stateful execution*: agents should operate on the repository state produced by earlier steps, including prior agent edits, growing tests, and accumulated technical debt. Third, *multidimensional evaluation*: success should measure not only functional correctness, but also regression preservation and repository health over time. We operationalize these desiderata below through three design choices: state-complete request chains for agents, defining evaluation settings that control context & memory, and measuring both functional correctness & repository health.

State-complete request chains. Let $(p_1, \ldots, p_T)$ be the historical PR sequence of a repository, and let q_t be the agent-facing request derived from PR p_t, containing the PR message and target functions/classes. Let $D(q_t)$ be the artifacts needed to interpret, implement, and verify q_t: source files, tests, configurations. A request chain $\mathcal{Q} = (q_i, \ldots, q_j)$ is *state-complete* if every request is dependency-closed with respect to earlier requests: for each q_k, all artifacts in $D(q_k)$ are either present in the initial repository state or introduced by correctly implementing some earlier request q_ℓ, $\ell < k$. Contiguous historical windows satisfy this property; non-adjacent chains may omit PRs that later requests implicitly depend on.

Heterogeneous Chains: State-complete chains need not correspond to a single coherent change. This is intentional: real development episodes often contain loosely related changes within the same evolving repository state. Human developers rarely treat each change as fully stateless; they maintain a working mental model of the codebase, task intent, dependencies, and side effects across successive edits (LaToza et al., 2006; Sillito et al., 2006). Our aim is to study the agent under such real deployment settings.

Framework. We define SWE-STEP as a stateful evaluation pipeline $\mathcal{F}(R, A, W, S) \rightarrow (\mathcal{T}, \mathcal{M})$, where R is a git repository, A is a coding agent, $W = [t_{\text{start}}, t_{\text{end}}]$ is an evaluation window, and S is the evaluation setting (described below). The pipeline extracts state-complete PR chains from W, converts each PR into a request specification, executes the agent under setting S, and computes functional and repository-health metrics.

Evaluation settings. The setting S determines how requests, repository state, memory, and tests are exposed to the agent. Refer Figure 1.

Conversational Coding. (Tang et al., 2026) The agent receives one PR-style request at a time, in historical order. Each request contains a task description and a definition specification listing the functions and classes to add or modify. The repository state, prior agent edits, conversation history, and test obligations persist across the chain. Tests cascade: obligations introduced or relevant in earlier PRs remain in scope for later PRs.

Aggeragated Task Specifications (ATS) based Coding. The same per-PR requests are concatenated into a single ATS and provided upfront. The agent receives all requirements at once and produces a final repository state. The accumulated test suite from all PRs is evaluated at the end.

Individual PR Baseline. Each PR is executed independently from its clean historical pre-state. The agent receives the corresponding task description and definition specification, but the environment resets before every PR, removing spillover from earlier agent edits. This setting matches the reset-based evaluation style used in SWE-bench-like benchmarks.

Framework outputs. SWE-STEP outputs a set of generated tasks $\mathcal{T}$ and a metric suite $\mathcal{M}$. Each task $\tau \in \mathcal{T}$ is a tuple $\tau = (R_0, \mathcal{Q}, \mathcal{V}, S_b)$, where R_0 is the initial repository state, $\mathcal{Q} = (q_1, \ldots, q_n)$ is a state-complete request chain, $\mathcal{V} = (v_1, \ldots, v_n)$ is the corresponding verification suite, and S_b is the sandboxed execution protocol. Each request q_i contains a task description and a definition specification (see Figure 2 for visual illustration). Each verifier v_i contains `FAIL_TO_PASS` tests for the requested behavior and `PASS_TO_PASS` tests for regression preservation.

The metric suite $\mathcal{M}$ captures two dimensions. First, *functional correctness*: PR-level and task-level success, computed from `FAIL_TO_PASS` and `PASS_TO_PASS` outcomes. A PR succeeds only if all required behavior is implemented and no required prior behavior regresses. A task succeeds only if all PRs in the chain succeed. Second, *repository health*: static-analysis metrics such as cognitive complexity (Campbell, 2017) and SQALE technical debt (Letouzey, 2012), comparing the agent-produced repository state against the historical human implementation.

3.2. Proposed Data Generation Framework

The core challenge in constructing realistic long-horizon datasets is identifying coherent sequences of development work from git repositories while ensuring each task is properly validated and executable.

Our approach addresses this through an automated pipeline that extracts task chains directly from git commit graphs, validates their executability through rigorous test-driven verification, and constructs comprehensive specifications that mirror the information available to human developers. The pipeline operates in two stages (illustrated in Figure 2): (1) mining PR metadata and dependency relationships from git history to extract a set of PRs and (2) validating task executability through test-driven filtering. This process ensures that generated tasks reflect development patterns-with valid and comprehensive test suites, and specification quality comparable to human-written requirements-while maintaining reproducibility and dataset validity. We now describe each stage in detail.

Phase I: Repository Mining & Metadata Extraction. We select 6 popular repositories (>8k GitHub stars) from SWE-Gym and scrape the most recent 2,000 pull requests from each. For transparency and analysis purposes, we automatically categorize each PR into one of six types using an LLM-based classifier (Table 7): `Feature/Enhancement`, `Bug Fix`, `Maintenance`, `Infrastructure`, `Documentation`, and `Testing` (prompt for LLM classifier in Appendix A.7.1).

For each PR, we perform metadata extraction, capturing attributes detailed in Table 6. Critically, we extract: commit IDs, the PR number and issue descriptions, the patches applied, the list of modified files, the specific functions or classes that were added, modified, or deleted, and the associated unit test cases. Unit tests newly introduced in a given PR are classified as F2P test cases, while pre-existing unit tests in modified files are classified as P2P.

Phase II: Test-Driven Validation and Refinement Raw PR data often contains inconsistencies, incomplete test coverage, or environment-specific issues that compromise dataset validity. We implement a test-driven validation process to ensure each task instance provides a fair evaluation signal. This validation procedure operates in two steps:

1. **Test patch validation**: Apply only the `test_patch` and verify that all P2P tests remain passing while all F2P tests remain failing (confirming tests properly capture missing functionality)
2. **Fix patch validation**: Apply the `fix_patch` containing the implementation and confirm that all P2P tests continue passing while all F2P tests transition to passing

We exclude PRs lacking test cases and PRs involving only infrastructure or documentation changes, as these fall outside our focus on functional code evolution. During validation, we address data quality challenges through a two-stage filtering process: first, we identify and filter test cases exhibiting behavior contrary to their expected state (e.g., F2P tests passing prematurely); second, we re-execute tests to validate behavioral consistency. This ensures a clean test suite that neither unfairly penalizes nor artificially benefits evaluated agents.

Phase III: Multi-Step Task Chain Construction While existing datasets evaluate agents on isolated, single-PR scenarios, real-world development involves sequences of temporally ordered changes. We construct multi-step task instances by identifying sequences of PRs that fall within a temporal range (based on commit timestamps), representing development trajectories between the two points.

For each PR within a task chain, we automatically synthesize the `task description` and `definition description` components (as defined in Section 3.1):

- The `task description` is constructed by aggregating information from the PR description, linked GitHub issues, and any associated documentation changes extracted during Phase I

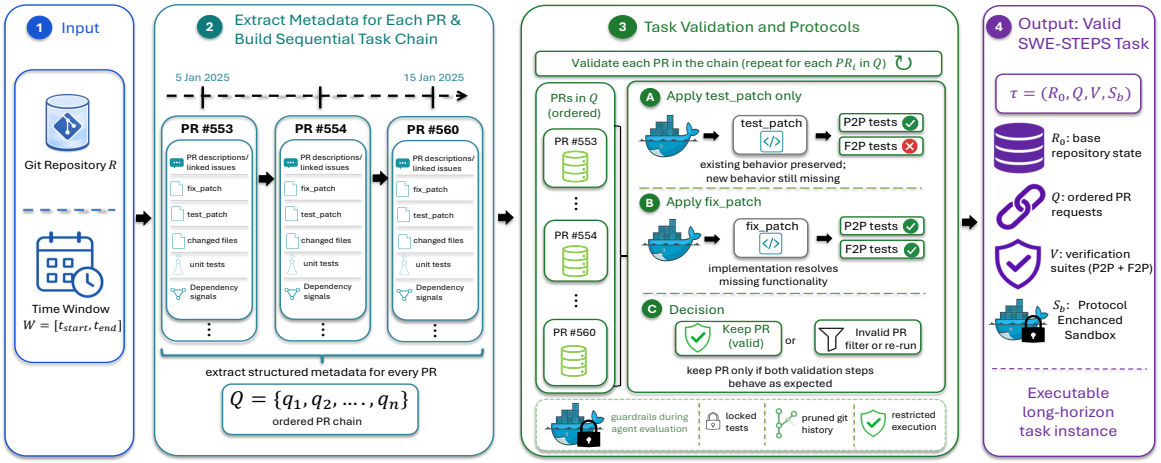


Figure 2. **SWE-STEP Task Extraction Pipeline.**

- The `definition description` is automatically extracted by parsing the `changes` metadata to identify function and class signatures that were added, modified, or deleted in the gold patch

The agent's objective is to implement the chain of changes while satisfying both functional requirements (test passage) and structural constraints (correct naming and interfaces). Figure 2 illustrates this extraction process, showing how we traverse the git commit graph to identify task chains and synthesize comprehensive specifications from metadata.

3.3. SWE-STEP Dataset

As an instantiation of our task generation framework **SWE-STEP**, we construct a dataset comprising **168 tasks** spanning **963 PRs** across 6 diverse Python repositories.
The repositories range from moderately-sized projects (Conan: $1.52M development cost estimate) to large-scale systems (Moto: $129.13M development cost estimate), ensuring diversity in architectural complexity and development effort. Repository complexity is estimated using industry-standard metrics via the SCC tool (Boyter, 2025), including lines of code, number of files, and cyclomatic complexity. Repositories span domains including build tools (Conan), AI versioning (DVC), scientific computing and AI (Haystack), database engines (Sqlglot), scientific/engineering libraries (Pandas), and software development/testing infrastructure (Moto). See appendix for detailed repository metrics.

Dataset Characteristics. Our tasks feature PR chains ranging from *3 to 11 PRs* and detailed problem descriptions corresponding to complex issues and feature requests.

Statistical Analysis and Comparison. Compared with isolated-issue benchmarks, SWE-STEP tasks are substantially larger in both specification and implementation scope. On average, SWE-STEP task descriptions contain 3,656 words, roughly 15–19× longer than SWE-Gym and SWE-Bench. The corresponding gold patches modify 17.1 files and 36.0 functions on average, compared with only 1.7–2.5 files and 3.0–4.1 functions in prior benchmarks. This reflects the central design difference: SWE-Bench and SWE-Gym optimise for breadth of isolated issues. In contrast, SWE-STEP optimises for depth of interdependent development sequences that mirror realistic multi-week engineering efforts. Even the Lite and Mini subsets preserve this profile, with average descriptions above 2.6K words and patches modifying roughly 15 files per task. Detailed statistics are in Appendix Table 5.

Lite and Mini Subsets. Evaluating coding agents on long-horizon tasks is computationally expensive. To facilitate broader dataseting and rapid iteration, we introduce two stratified subsets: **SWE-STEP-Lite** (50 tasks) and **SWE-STEP-Mini** (20 tasks) (Refer figure 6).

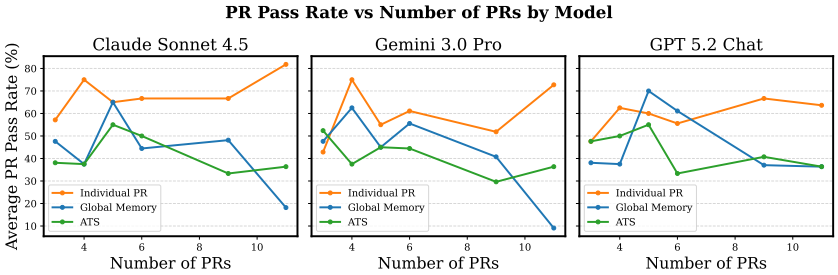
4. Experimental Setup

We use OpenHands (Wang et al., 2025) with CodeAct (Wang et al., 2024) as our primary evaluation agent because its open-source design enables white-box analysis of tool calls, execution traces, and file edits. To assess generality across agents, we additionally evaluate Aider (Gauthier, 2023) in Appendix A.6.4 and the closed-source Codex agent Appendix A.6.5.

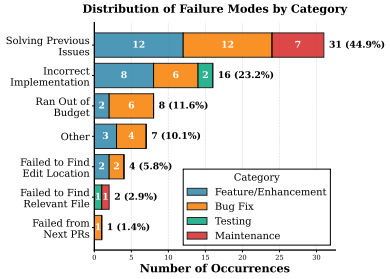
Agent Configuration *Architecture and Tools. Architecture and Tools.* OpenHands is equipped with a `Terminal`, `FileEditor`, and `Task Tracker`. We provide expected function signatures to reduce naming-related false negatives. In the *ATS* setting, a planner decomposes bundled requirements, while `ContextCondenser` manages growing context in both Global and ATS settings.
Models. We evaluate performance across varying model capabilities to balance reasoning depth with statistical coverage: (i) *Mini Split* (Deep Analysis): Evaluated using

Table 1. (OpenHands - Mini dataset) Comparison of models across multiple repositories using mini dataset (total expenditure: $3,827.78). The **PR** & **Task** columns indicate number of completed PRs & tasks. For confidence intervals on computed PR success rate, see App A.6.1.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Tot. PRs	PR Succ.	Total Cost
		PR /25	Task /4	PR /18	Task /5	PR /7	Task /2	PR /15	Task /2	PR /20	Task /5	PR /20	Task /2	Passed /105	Rate	($)
Individual	Gemini 3 Pro	19	1	8	1	5	0	8	0	8	0	11	0	**59**	**56.19**	258.91
	Claude Sonnet 4.5	19	0	9	1	7	2	10	0	11	0	14	0	**70**	**66.67**	289.30
	GPT 5.2 Chat	21	2	7	1	6	1	10	0	7	0	11	0	**62**	**59.04**	80.97
	Gemini 3 Flash	17	1	9	1	7	2	10	0	7	0	8	0	**58**	**55.24**	54.34
	GPT 5.1 Codex Mini	16	0	3	0	6	1	6	0	8	0	6	0	**45**	**42.85**	68.38
Global	Gemini 3 Pro	14	0	6	0	7	2	7	0	9	0	3	0	**46**	**43.81**	330.81
	Claude Sonnet 4.5	20	2	6	1	6	1	6	0	7	0	4	0	**49**	**46.67**	408.55
	GPT 5.2 Chat	17	2	6	1	5	0	10	0	7	0	5	0	**50**	**47.62**	197.41
	Gemini 3 Flash	12	0	7	1	6	1	5	0	5	0	2	0	**37**	**35.23**	90.80
	GPT 5.1 Codex Mini	12	1	2	0	5	0	7	0	5	0	2	0	**33**	**31.43**	86.46
ATS	Gemini 3 Pro	13	0	6	1	5	0	5	0	8	0	6	0	**43**	**40.95**	488.23
	Claude Sonnet 4.5	13	1	4	1	4	0	9	0	8	0	6	0	**44**	**41.90**	838.12
	GPT 5.2 Chat	20	2	6	1	5	1	3	0	7	0	5	0	**46**	**43.80**	294.69
	Gemini 3 Flash	13	0	4	0	4	0	5	0	4	0	2	0	**32**	**30.47**	194.08
	GPT 5.1 Codex Mini	12	1	0	0	4	0	4	0	12	1	3	0	**35**	**33.33**	146.73



(a) **Pass Rate Analysis.** Comparison of Average PR Pass Rates for Large models relative to the total number of PRs on Mini Dataset Experiment. The plots reveal that the *Individual PR* setting consistently maintains higher pass rates.



(b) **Distribution of Failure Modes in the Global Setting.** Analysis of failure categories for Claude Sonnet 4.5 on the Mini dataset, evaluated by Gemini 3.0 Pro.

high-performance models such as Gemini 3 Pro(DeepMind, 2025), Claude Sonnet 4.5 (Anthropic, 2025) using Vertex AI API (Google Cloud), and GPT-5.2 Chat (OpenAI, 2025) using Azure OpenAI (Microsoft) Endpoints. (ii) *Lite Split* (Broad Coverage): Evaluated using cost-effective models - Gemini 3 Flash from Vertex AI API and GPT-5.1 Codex Mini from Azure OpenAI endpoints.

Environment and Execution Protocol *Runtime Environment and Safety.* Experiments are conducted in isolated Docker containers with strict anti-gaming protocols. *Network Isolation* restricts internet access to dependency installation only, disabling browser usage to prevent web-based debugging. *File Immutability* locks test files to read-only mode, preventing the agent from altering assertions and revoking all relevent testcases during agent execution to prevent shortcut solutions by agent. *History Pruning* truncates the git history at the base commit, ensuring the agent cannot "leak" future ground-truth solutions by inspecting logs.
Execution Protocol (Reflection Cycles). We adopt a standardized interaction protocol based on test-driven feedback, following experimental setup of Commit0 (Zhao et al., 2024). For consistency and comparability, we use same setup for all agents. Specifically, execution is structured into *Reflection Cycles*, each comprising at most 40 agent iterations. At the end of each cycle-either when iteration budget is exhausted or when agent explicitly submits-the environment runs formal test suite. If all tests pass, the task is marked as solved; otherwise, the resulting stderr logs are returned to the agent as feedback for the next cycle. To further study this setup, we also include a scaling analysis in Appendix A.6.6, where we measure performance as a function of number of reflection cycles.

Budgets and Evaluation Settings. We instantiate the settings defined in Section 3.1 with specific computational constraints: In *Individual & Global Settings*, the agent is allocated a budget of 3 reflection cycles per PR. In *Global* setting, the repository state persists across the task; in *Individual* setting, it resets to the ground truth after every PR.

Table 2. Coding agents' performance across evaluation settings in the *Mini Dataset*. All agents use **GPT-5.1 Codex Mini**. **Cost** denotes total USD cost incurred. **Succ. Rate** denotes the percentage of successfully completed PRs. Values in red denote the absolute drop in PR success rate, relative to the *Individual* setting.

Agent	Individual		Global		ATS	
	Succ.	Cost	Succ.	Cost	Succ.	Cost
OpenHands	**42.85**	68.38	31.43 ↓11.42	86.46	33.33 ↓9.52	146.73
Aider	**34.29**	3.4	17.14 ↓17.15	5.0	13.33 ↓20.96	10.0
Codex	**57.14**	34.4	51.43 ↓5.71	24.6	44.76 ↓12.38	54.2
Average	**44.76**	1.77	33.33 ↓11.43	1.93	30.47 ↓14.29	3.52

Similarly, in the *ATS Setting*, to ensure comparable total compute, the agent is allocated a pooled budget of $3 \times N$ cycles (where N is number of PRs). More details can be found in Appendix A.4.3

5. Results And Analysis

In this section, we empirically evaluate the impact of long-horizon software evolution on coding agents. Unless explicitly stated, all the analysis is being done on the mini dataset. Our investigation is guided by five key questions:

RQ1 (Performance Inflation): How does introduction of stateful dependencies (Global/ATS settings) impact agent success rates compared to stateless, isolated benchmarks (SWE-bench style)?
RQ2 (Temporal Degradation): How does agent reliability correlate with the accumulation of history, specifically as a function of task chain length and expanding test suites?
RQ3 (Repository Health): Do coding agents prioritize short-term functional correctness at the expense of long-term repository maintainability?
RQ4 (Error Attribution): What are the dominant failure modes? Do agents fail due to immediate implementation errors or inability to manage regressions from previous tasks?
RQ5 (Dependency Sensitivity): Does performance degradation under stateful settings intensify when PR chains exhibit verified inter-PR dependencies, and does this effect scale with chain length?

> ***RQ1:*** *How does introduction of stateful dependencies (Global/ATS) impact agent success rates compared to stateless, isolated benchmarks (SWE-bench style)*

We find isolated PR setting provides significantly inflated scores compared to our realistic settings. As shown in Table 1, comparisons on *Mini* dataset reveal a consistent trend where the standard "Individual PR" setting outperforms "Global" (conversational) and "ATS-based" settings across all evaluated repositories. Notably, Claude Sonnet 4.5 and Gemini 3 Flash exhibit a substantial performance drop when moving from isolated to continuous settings, with issue resolution rates falling by $\approx$ **20 percentage points** (e.g., Claude Sonnet 4.5 drops from 66.67% to 46.67%).

This trend is further corroborated by the *Lite* dataset results in Table 10. When scaling the evaluation, models such as Gemini 3 Flash see a degradation from 56.52% in the Individual setting to just 36.59% in the Global setting. Across all LLMs, we observe a consistent performance decrease ranging between **15% to 25%** when statefulness is introduced. This gap shows that SWE-bench style evaluations—where agents start with a clean, human-validated codebase—mask the difficulties of handling "spillover" effects, where previous buggy code or accumulated technical debt hampers future task resolution. This analysis holds true across coding agents (Table 2).

Takeaway: *Isolated PR-based evals overestimate agent capabilities by upto 20 percentage points*

> ***RQ2:*** *How does agent reliability correlate with the accumulation of history, specifically as a function of task chain length and expanding test suites?*

We analyze the difficulty of our evaluation settings by examining the correlation between performance, task chain length, and test suite size on the mini dataset.
Impact of Chain Length: In Figure 3a, we see a sharp drop in performance as number of PRs increases. Furthermore, across all models, *Individual PR* trendline consistently remains above Global & ATS trajectories, effectively establishing an upper performance bound.
Impact of Test Accumulation: Figure 9, reveals a similar trend regarding test suite size. As the volume of active tests increases, agents struggle to maintain regression stability. *GPT 5.2 Chat*, for example, sees its ATS pass rate plummet from over 60% to $\sim$10% in the largest test bin, whereas the Individual PR setting maintains significantly higher success rates throughout.

Takeaway: *Agents degrade significantly under contextual burden of long PR chains & accumulated regression tests*

> ***RQ3:*** *Do coding agents prioritize short-term functional correctness at the expense of long-term repository maintainability?*

Beyond functional correctness, we evaluated quality of codebase using code quality metrics on the *Mini dataset*. To track evolution, we filtered for task chains exceeding 5 PRs and normalized progress into five bins. We focus our analysis on the *Global* setting, as its conversational nature best mirrors the evolution of a repository. Figure 4 and 10 illustrates the divergence in *Cognitive Complexity* and *SQALE Index* (Technical Debt) between agents and human developed code (Gold Trace). Details about metrices in Appendix A.5
The Hidden Cost of High Performance: We observe a concerning inverse relationship between functional success and code health. For instance, in the `haystack` repository, *Gemini 3.0 Pro* attains a perfect 7/7 PR resolution rate in the

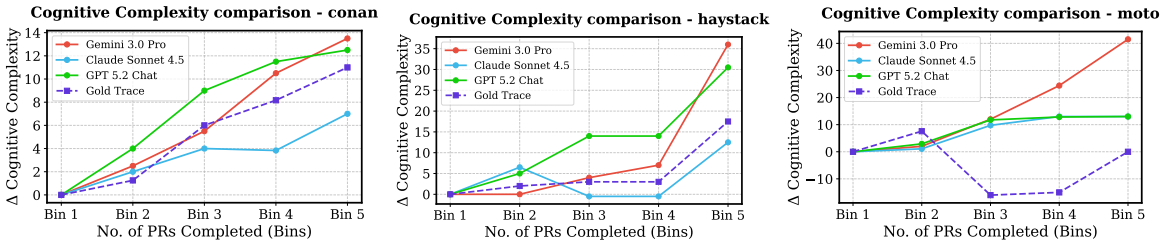


Figure 4. **Comparative Analysis of Code Quality Evolution.** The figure illustrates the progression of Δ Cognitive Complexity across `conan`, `haystack`, and `moto`. Plots reveal that autonomous agents tend to introduce higher complexity compared to the human code.

Global setting (Table 1). Yet, examining Figure 10 (middle column), we see a sharp degradation in code health: the agent's SQALE Index (technical debt) surges abruptly in the final bin, ending significantly higher than human reference, which maintains near-zero debt. Similarly, in `moto` repository (right column), while agents maintain competitive pass rates, *Gemini 3.0 Pro* exhibits a linear increase in cognitive complexity (Bin 1 to Bin 5). In contrast, human trace often shows negative slopes, perhaps indicating that humans actively refactor to reduce debt while solving tasks. Code snippets where agent causes higher complexity are in Appendix A.6.8

Takeaway: *High agent performance often masks degrading repository health.*

RQ4: *What are the dominant failure modes? Do agents fail due to immediate implementation errors or inability to manage regressions from previous tasks?*

We conducted a qualitative error analysis on the *Mini dataset* using execution traces from *Claude Sonnet 4.5*. We focus our analysis on the *Global* setting. To categorize failures, we employed *Gemini 3.0 Pro* as an evaluator, using a prompt detailed in the Appendix. We adopted the error taxonomy from SWE-agent (Yang et al., 2024), but introduced a new category - *"Solving Previous Issues"*, to capture failures specific to our sequential Global and ATS settings.

The Dominance of Contextual Failures: Figure 3b presents the distribution of failure modes. The results show the unique difficulty of stateful evaluation. The dominant failure mode, accounting for **44.9% (20 occurrences)** of all errors, is *Solving Previous Issues*. This failure is pervasive across both *Feature/Enhancement* and *Bug Fix* tasks. Figure 11 gives a walkthrough of one of such errors.

Takeaway: *Context overhead of previous tasks is a bottleneck for coding agents.*

RQ5: *Does stateful degradation increase when PR chains contain verified inter-PR dependencies?*

Dependency extraction. We identify inter-PR dependencies using two signals: *symbol-level reuse*, where classes added, modified, or deleted in PR_i are referenced by later PRs; and *line-level ancestry*, where Git blame shows that PR_j modifies a line introduced or changed by PR_i. These signals are used only for analysis, not for filtering chains. To separate genuine cumulative dependence from temporal co-occurrence, we compare Individual-Global and Individual-ATS performance gaps for chains with and without verified dependencies, stratified by chain length. Table 3 shows that dependency-verified chains exhibit larger stateful degradation, especially for long chains. For Claude Sonnet 4.5, the Individual-Global gap increases from $+2.6$ to $+\mathbf{24.4}$ on short chains and from $+11.1$ to $+\mathbf{40.9}$ on long chains. In contrast, non-dependent chains show smaller or less consistent gaps.

Table 3. Performance gap $(\Delta)_t$ between Individual and stateful settings (Global/ATS) for chains with and without verified inter-PR dependencies, across short (2–5 PRs) and long (6–11 PRs) chains. Bold values indicate dependency-verified results.

Model	Deps	Short Chains		Long Chains	
		Δ(Ind−Glo)	Δ(Ind − ATS)	Δ(Ind−Glo)	Δ(Ind − ATS)
Claude Sonnet 4.5	No	+2.6	+13.1	+11.1	0.0
	Yes	**+24.4**	**+24.4**	**+40.9**	**+32.2**
Gemini 3.0 Pro	No	−4.5	+3.8	0.0	+22.2
	Yes	−6.7	**+4.4**	**+22.9**	**+24.4**
GPT 5.2 Chat	No	+5.5	−1.9	0.0	+44.4
	Yes	+4.4	**+11.1**	**+11.0**	+31.8

Takeaway: *Degradation is jointly driven by temporal accumulation and inter-PR dependencies, with verified dependencies compounding the effect beyond chain length alone.*

6. Conclusion and Limitation

To bridge the gap between isolated coding tasks and the reality of cumulative software development, we introduce an evaluation framework and the SWE-STEP dataset. Our findings reveal that current methods inflate agent success rates and, more critically, that agent-generated code degrades long-term repository health, underscoring the need for evaluations that value maintainability alongside functional correctness. A limitation is use of Python-specific repositories for our analysis, which limits generalizability of results to other languages. See A.2 for a detailed discussion

7. Impact Statement

We collected the data from publicly available Github repositories only for research purposes. All the repositories have licenses that allow free software use. LLMs are used only for classification during the construction of the dataset, so no harmful information can be created in the dataset. The dataset and code for our proposed method will be made publicly available for academic research. However, we should note that the inference results of the task instances from the benchmark may contain code that is harmful to computer systems. Evaluation by docker is recommended, just like in SWE-bench (Jimenez et al., 2024).

References

How we broke top AI agent benchmarks. https://rdi.berkeley.edu/blog/trustworthy-benchmarks-cont/, 2026.

Finding widespread cheating on popular agent benchmarks. https://debugml.github.io/cheating-agents, 2026.

Anthropic. Claude sonnet 4.5 system card. Technical report, Anthropic, 2025. URL https://www.anthropic.com/claude-sonnet-4-5-system-card. Accessed 2026-01-29.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Bennett, K. H. and Rajlich, V. T. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pp. 73–87, 2000.

Boyter, B. scc: v3.5.0, 2025. URL https://github.com/boyter/scc/.

Campbell, G. A. Cognitive complexity: A new way of measuring understandability. White paper, SonarSource, 2017. URL https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

Campbell, G. A. Cognitive complexity: A new way of measuring understandability. Technical report, SonarSource, 2023. URL https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Cihan, U., Haratian, V., İçöz, A., Gül, M. K., Devran, Ö., Bayendur, E. F., Uçar, B. M., and Tüzün, E. Automated code review in practice. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 425–436. IEEE, 2025.

DeepMind, G. Gemini 3 pro: Model card. Technical report, Google DeepMind, 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf. Accessed 2026-01-29.

Deng, G., Chen, Z., Yu, Z., Fan, H., Liu, Y., Yang, Y., Parikh, D., Kannan, R., Cong, L., Wang, M., et al. Evoclaw: Evaluating ai agents on continuous software evolution. *arXiv preprint arXiv:2603.13428*, 2026.

Gauthier, P. Aider: Ai pair programming in your terminal. https://github.com/aider-ai/aider, 2023.

Google Cloud. Model Garden on Vertex AI (Model Catalog). https://cloud.google.com/model-garden. Accessed 2026-01-29.

Han, H., hwang, s.-w., Samdani, R., and He, Y. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments. In Yue, Y., Garg, A., Peng, N., Sha, F., and Yu, R. (eds.), *International Conference on Learning Representations*, volume 2025, pp. 38737–38776, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/6091f2bb355e960600f62566ac0e2862-Paper-Conference.pdf.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Jain, N., Singh, J., Shetty, M., Zheng, L., Sen, K., and Stoica, I. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Joshi, T., Chowdhury, S., and Uysal, F. Swe-bench-cl: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Kim, M., Garg, S., Ray, B., Kumar, V., and Deoras, A. Codeassistbench (cab): Dataset & benchmarking for multi-turn chat-based code assistance. *arXiv preprint arXiv:2507.10646*, 2025.

Kruchten, P., Nord, R. L., and Ozkaya, I. Technical debt: From metaphor to theory and practice. *Ieee software*, 29 (6):18–21, 2012.

LaToza, T. D., Venolia, G., and DeLine, R. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pp. 492–501, 2006.

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

Letouzey, J.-L. The SQALE method for evaluating technical debt. In *Third International Workshop on Managing Technical Debt (MTD)*, pp. 31–36. IEEE, 2012.

Li, W., Zhang, X., Guo, Z., Mao, S., Luo, W., Peng, G., Huang, Y., Wang, H., and Li, S. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In Che, W., Nabende, J., Shutova, E., and Pilehvar, M. T. (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.839. URL https://aclanthology.org/2025.acl-long.839/.

Martin, R. C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

Microsoft. AI Model Catalog | Azure AI Foundry (Microsoft Foundry Models). https://ai.azure.com/catalog/models. Accessed 2026-01-29.

Neubig, G. Open-source coding agents in your github, fixing your issues, 2024. URL https://openhands.dev/blog/open-source-coding-agents-in-your-github-fixing-your-issues.

Nguyen, D. M., Phan, T. C., Le Hai, N., Doan, T.-T., Nguyen, N. V., Pham, Q., and Bui, N. D. Codemmlu: A multi-task benchmark for assessing code understanding & reasoning capabilities of codellms. In *The Thirteenth International Conference on Learning Representations*.

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

OpenAI. Update to gpt-5 system card: Gpt-5.2. Technical report, OpenAI, 2025. URL https://openai.com/index/gpt-5-system-card-update-gpt-5-2/. Accessed 2026-01-29.

Pan, J., Wang, X., Neubig, G., Jaitly, N., Ji, H., Suhr, A., and Zhang, Y. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*, 2025. URL https://arxiv.org/abs/2412.21139. arXiv:2412.21139, accepted at ICML 2025.

Sillito, J., Murphy, G. C., and De Volder, K. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 23–34, 2006.

Tang, N., Chen, C., Fang, Z., Xu, G., Dhakal, M., Shi, Y., McMillan, C., Huang, Y., and Li, T. J.-J. Programming by chat: A large-scale behavioral analysis of 11,579 real-world ai-assisted ide sessions. *arXiv preprint arXiv:2604.00436*, 2026.

Thai, M. V., Le, T., Manh, D. N., Nhat, H. P., and Bui, N. D. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Zaidman, A., Pinzger, M., and van Deursen, A. Software evolution. In *Encyclopedia of Software Engineering*. Taylor & Francis, 2010.

Zhao, W., Jiang, N., Lee, C., Chiu, J. T., Cardie, C., Gallé, M., and Rush, A. M. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

Zhao, W., Jiang, N., Lee, C., Chiu, J. T., Cardie, C., Gallé, M., and Rush, A. M. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., and Yue, X. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 12834–12859, 2024.

Zhong, Z., Raghunathan, A., and Carlini, N. Impossiblebench: Measuring llms' propensity of exploiting test cases, 2025. URL https://arxiv.org/abs/2510.20270.

A. Appendix

In this section, we provide additional results and details that we could not include in the main paper due to space constraints. We have shared our SWE-STEP dataset [2] and the code to generate the data as a zip file. In particular, this appendix contains the following:

[2] https://anonymous.4open.science/r/swe-steps-data-anno-083C/README.md

Table 4. **Comparison of existing SWE evaluation approaches**. *Sequential Evolution*: evaluates agents on a trajectory of tasks with persistent state; *Repository Health Analysis*: tracks static analysis metrics (cognitive complexity and technical debt); *Interdependent Tasks*: tasks are causally linked (our dataset SWE-STEP's 48.7% of tasks have interdependent PRs); *Sequential Regression Verification*: verifies previous functionality against new changes; *Anti-gaming Protocols*: prevents models from gaming evaluation metrics. FEA-Bench (Li et al., 2025) & COMMIT-0 (Zhao et al., 2024) both adapt SWE-Bench style, though FEA deals with `features` instead of `issues`, & COMMIT-0 benchmarks coding agents not LLMs.

Dataset	Repository Level	Execution Environment	Real-World Tasks	Sequential Evolution	Repository Health Analysis	Inter-ependent Tasks	Sequential Regression Verification	Anti-gaming Protocols
CodeFeedback (Zheng et al., 2024)	×	×	✓	×	×	×	×	×
APPS (Hendrycks et al., 2021)	×	✓	✓	×	×	×	×	×
HumanEval (Chen et al., 2021)	×	✓	✓	×	×	×	×	×
MBPP (Austin et al., 2021)	×	✓	✓	×	×	×	×	×
ConvCodeBench (Han et al., 2025)	×	✓	✓	×	×	×	×	×
R2E (Jain et al., 2025)	✓	✓	×	×	×	×	×	×
SWE-Bench (Jimenez et al., 2024)	✓	×	✓	×	×	×	×	×
CodeAssistBench (Kim et al., 2025)	✓	✓	✓	×	×	×	×	×
SWE-Gym (Pan et al., 2025)	✓	✓	✓	×	×	×	×	×
SWE-Bench-CL (Joshi et al., 2025)	✓	✓	✓	✓	×	×	×	×
SWE-EVO (Thai et al., 2025)	✓	✓	✓	✓	×	×	×	×
EvoClaw (Deng et al., 2026)	✓	✓	✓	✓	×	✓	×	×
Ours	✓	✓	✓	✓	✓	✓	✓	✓

A.1. Impact Statement

We collected the data from publicly available GitHub repositories only for research purposes. All the repositories have licenses that allow free software use. LLMs are used only for classification during dataset construction, so that no harmful information is created. The dataset and code for our proposed method will be made publicly available for academic research. However, we should note that the inference results of the task instances from the benchmark may contain code that is harmful to computer systems. Evaluation by docker is recommended, just like in SWE-bench (Jimenez et al., 2024).

A.2. Limitations

Our work is intentionally focused on a well-defined setting: long-horizon coding tasks in Python repositories. While this enables a cleaner and more practical benchmark design, the findings may not fully generalise to other programming languages or software ecosystems. Similarly, our selection of top PyPI repositories emphasises popular and well-maintained projects, which supports dataset quality but may underrepresent smaller, newer, or less actively maintained repositories. We also exclude some early-stage repository changes that did not undergo rigorous code review, prioritizing reliability over broader but noisier coverage. Finally, because repository-level tasks require long-context reasoning across multiple files, experiments with LLM-based agents can be computationally expensive, and the dataset is best viewed as a benchmark for long-horizon repository-level coding rather than a comprehensive measure of all software engineering ability.

A.3. Extended Related Work

In section 2, we situated our work within prior research on software engineering (SWE) agents, code generation datasets, and repository-scale benchmarks. Here we extend this analysis with more details. Table 4 further provides a comparative overview of existing evaluation approaches w.r.t. ours.

Software Engineering Agents and Architectural Evolution. The capabilities of coding assistants have evolved from auto-completion to autonomous agents capable of multi-step reasoning, such as SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Aider (Gauthier, 2023). While current systems demonstrate impressive problem-solving skills on single-issue datasets, the evaluation criteria have primarily centered on functional resolution—passing the immediate test cases. Less attention has been paid to the potential accumulation of technical debt. Code health metrics let us

assess this, i.e., whether agents solve tasks without degrading architectural integrity over time.

Evolution of Code Generation Datasets. Early evaluation frameworks focused primarily on function-level synthesis and template completion. Datasets such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and CodeMMLU (Nguyen et al.) established the foundational capabilities of LLMs by evaluating their ability to generate standalone functions from natural language. While effective for assessing basic syntax and logic, these datasets generally operate without *Repository Level* context. Subsequent datasets like APPS (Hendrycks et al., 2021), CodeFeedback (Zheng et al., 2024), and ConvCodeBench (Han et al., 2025) shifted focus toward complex algorithmic reasoning. However, these tasks remain largely independent; they assess whether a model can solve a specific puzzle in isolation.

Repository-Level Programming Datasets. The field has recently advanced towards repository-scale evaluation of coding agents, recognizing that real-world coding requires navigating complex file structures. SWE-bench (Jimenez et al., 2024) and its successors, including SWE-gym (Pan et al., 2025), FEA-Bench (Li et al., 2025), COMMIT-0 (Zhao et al., 2024), and CodeAssistBench (Kim et al., 2025), represent significant progress by evaluating agents on resolving GitHub issues within real-world repositories. A distinct characteristic of these repository-level datasets, however, is their reliance on a *single-issue, reset-based* model. That is, the environment is reset after every task, here PR, to ensure isolation. This differs from professional workflows where *sequential evolution* of the codebase is inevitable.

Sequential Software Evolution and Regression Verification. A few recent efforts, namely SWE-Evo (Thai et al., 2025), EvoClaw (Deng et al., 2026), and SWE-Bench-CL (Joshi et al., 2025), introduce evaluation across sequences of issues on an evolving codebase. However, they remain limited primarily to *functional correctness*. In contrast, SWE-STEP explicitly models persistent repository state and cross-PR regression verification: once behavior is implemented in an earlier PR, its corresponding tests remain active obligations for later PRs. This differs from isolated per-PR PASS_TO_PASS checks, where regression safety is verified only within a single reset-based task. SWE-STEP also analyzes verified inter-PR dependencies, allowing us to study whether degradation arises from direct PR dependencies or from broader stateful pressures such as accumulated tests, prior agent edits, and repository-state drift.

Repository Health and Anti-Gaming Protocols. Existing SWE benchmarks primarily evaluate whether agents pass the immediate functional tests. They do not account for long-term repository health or incorporate anti-gaming protocols that prevent agents from exploiting evaluation shortcuts. SWE-STEP complements these benchmarks by tracking maintainability-oriented metrics such as cognitive complexity (Campbell, 2017) and technical debt(Letouzey, 2012), and by using safeguards such as locked tests, pruned git history, restricted execution, and standardized harness-mediated feedback. These controls ensure that measured success reflects implementation ability rather than test modification, future-solution leakage, or other benchmark-gaming behavior.

As illustrated in Table 4, our work seeks to complement existing datasets by introducing a long-horizon temporal dimension to agent evaluation. We unify repository-level execution with a *sequential evolution* paradigm, where agents navigate a sequence of sub-tasks with persistent state. This design allows us to enforce *cross-PR Regression Verification*—ensuring previously implemented functionality remains stable under future changes—and to conduct continuous *Repository Health Analysis*.

Table 5. Dataset Comparison (Task wise)

Category	Metric	SWE-Bench	SWE-Gym	Ours (All)	Ours (Lite)	Ours (Mini)
Issue Text	Length by Words	195.1	239.8	3656.0	3161.1	2666.2
Gold Patch	# Files edited	1.7	2.5	17.1	15.7	15.4
	# Func. edited	3.0	4.1	36.	33.4	26.7
Tests	# Fail to Pass	9.0	10.0	15.9	17.5	15.4
	# Total	132.5	760.8	235.1	224.9	199.3

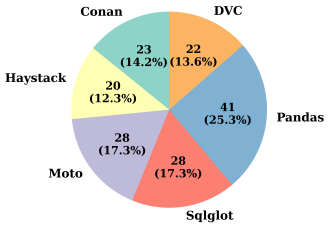


Figure 5. Distribution of tasks across repositories

A.4. Dataset and Execution Environment

A.4.1. Dataset Statistics and Comparison

Table 5 reports task-wise statistics comparing SWE-STEP with SWE-Bench and SWE-Gym. Since each SWE-STEP task aggregates a temporally ordered PR chain rather than a single isolated issue, its specifications and implementation scope are substantially larger. The full split has an average description length of 3,656 words, compared with 195.1 for SWE-Bench and 239.8 for SWE-Gym, and its gold patches modify 17.1 files and 36.0 functions on average. This indicates that SWE-STEP evaluates agents under broader repository context and larger code-change surfaces. The Lite and Mini subsets preserve this long-horizon profile while reducing evaluation cost: both retain average descriptions above 2.6K words and modify more than 15 files per task on average. Overall, these statistics show that SWE-STEP differs from prior benchmarks not merely in scale, but in the type of software-evolution behavior it is designed to evaluate.

A.4.2. Dataset Composition and Attributes

This section details the attributes of the dataset, analyzing repository statistics and the complexity of tasks based on their categorization.

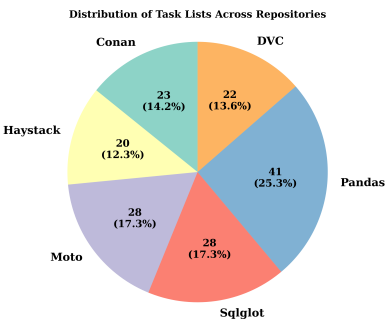
Dataset Distribution and Representativeness. Figures 6 and 7 provide a detailed breakdown of the task composition across the *Standard*, *Lite*, and *Mini* datasets.

Figure 6 illustrates the distribution of tasks across the six evaluated repositories. In the *Standard* dataset (Figure 6a), the tasks are spread across diverse domains, with *Pandas* (25.3%) and *Sqlglot* (17.3%) representing a significant portion of the workload. Crucially, the *Lite* and *Mini* subsets (Figures 6b and 6c) generally preserve this proportional distribution, ensuring that the smaller evaluation sets remain representative of the broad repository diversity found in the full benchmark.

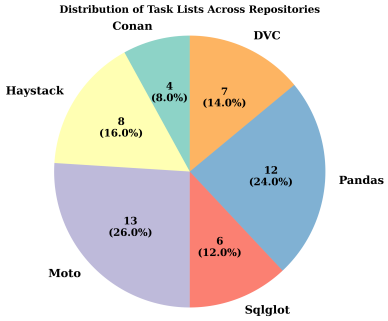
Figure 7 serves as a proxy for task complexity by categorizing the problems by their nature. The analysis reveals that the benchmark is rigorously focused on complex software engineering problems rather than trivial changes. Across all three splits, the vast majority of tasks fall into the **Bug Fix** and **Feature/Enhancement** categories. For instance, in the *Standard* dataset, Bug Fixes (46.6%) and Features (40.1%) combined account for over 86% of the total tasks. This trend holds consistent in the *Lite* (Figure 7b) and *Mini* (Figure 7c) datasets, confirming that the reduced-size benchmarks retain the high complexity profile required to effectively evaluate agent reasoning and coding capabilities.

Repository Structure and Context Attributes. To provide a comprehensive understanding of the experimental setup, we present the structural breakdown of the evaluated repositories and the specific attributes used for agent context. Table 8 outlines the diversity of the selected repositories, which span distinct domains ranging from AI versioning (DVC) and scientific engineering (Pandas) to database engines (Sqlglot). This domain heterogeneity ensures that the agents are evaluated against varied coding paradigms and architectural patterns. Within these repositories, tasks are classified according to the taxonomy defined in Table 7, distinguishing between activities such as routine maintenance, complex feature enhancements, and critical bug fixes. Finally, Table 6 details the granular metadata extracted for each Pull Request. These attributes—including the raw code diffs (`fix_patch`, `test_patch`), modified file lists, and regression testing constraints (F2P)—constitute the full context window provided to the agents during execution.

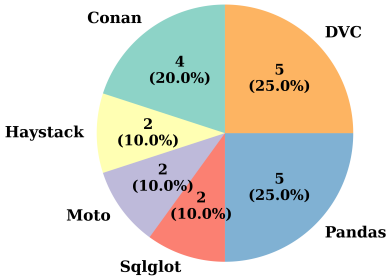
Figure 6. Repository task lists distribution (pie charts)



(a) Standard

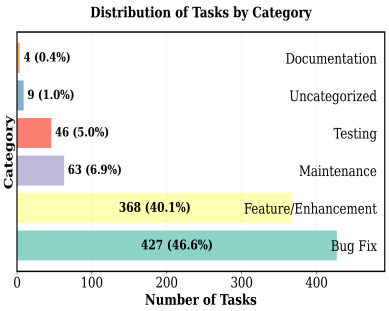


(b) Lite

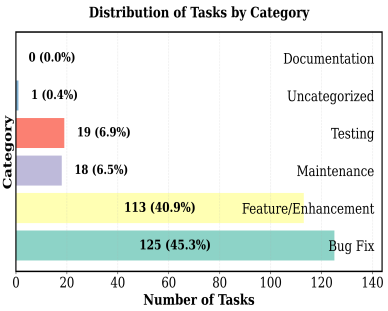


(c) Mini

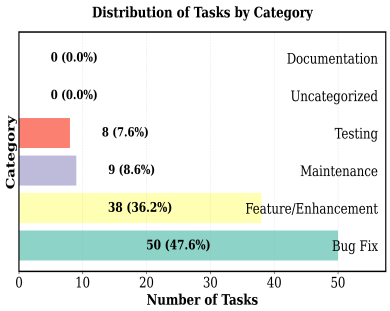
Figure 7. Task distribution (bar charts)



(a) Standard



(b) Lite



(c) Mini

Table 6. Attributes extracted for each Pull Request.

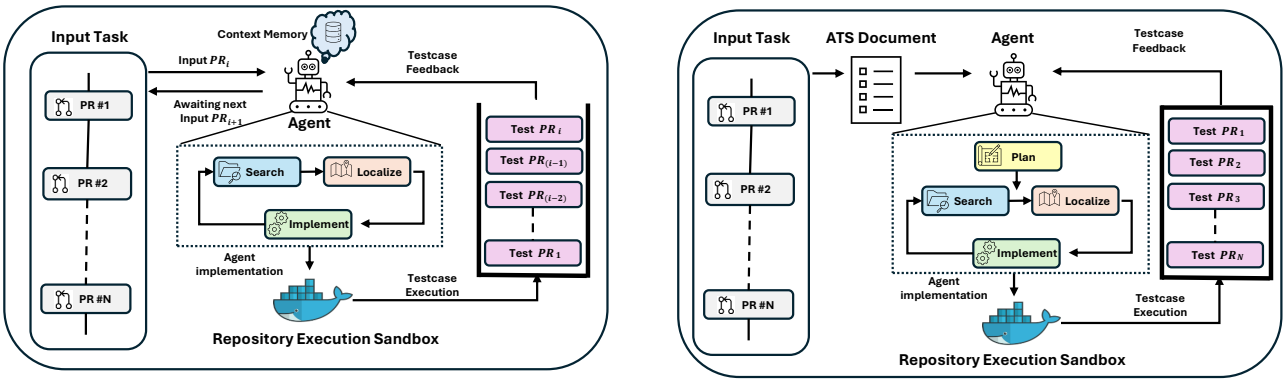
Attribute	Description
`commit_id`	Commit ID of the PR.
`parent_id`	Commit ID of the base repository.
`pr_number`	PR number for merging.
`prs`	Dictionary with PR information.
`changed_files`	List of modified files.
`test_files`	List of modified test files.
`changes`	Added, modified, or deleted items.
`pass_to_pass`	Tests that must remain passing.
`fail_to_pass`	Tests changing from fail to pass.
`fix_patch`	Git diff of source code.
`test_patch`	Git diff of test code.
`all_texts`	Issues, docs, PR messages.

Table 7. Pull Request classification categories.

Category	Description
Feature/ Enhanc.	New features, performance, security.
Bug Fix	Bug fixes and issue resolution.
Maintenance	Refactoring, cleanup, updates.
Infrastructure	Build, CI, configuration changes.
Documentation	Documentation updates.
Testing	Test additions and modifications.

Table 8. Domains of evaluated repositories.

Repository	Domain
Conan	Build Tools
DVC	AI Versioning
Haystack	Scientific, AI
Sqlglot	Database Engines
Pandas	Scientific/Engineering
Moto	Software Dev, Testing



(a) Global Setting *(b)* ATS setting

Figure 8. **Comparison of Agent Execution Configurations.** (a) **Global Memory Setting (Conversational Coding):** The agent processes Pull Requests sequentially ($PR_1 \rightarrow PR_N$), maintaining context across the chain. Verification involves cascading tests ($Test_{PR_i} + Test_{PR_{i-1}} \ldots$) to ensure new changes do not regress previous fixes. (b) **ATS Setting:** Task requirements are aggregated into a unified Aggregated Task Specification (ATS). The agent synthesizes a high-level plan before implementation, validated against the complete accumulated test suite.

A.4.3. Execution Setup and Diagrammatic Representation

To evaluate the adaptability of autonomous agents to different development workflows, we structured the experiments around three distinct execution configurations. Figure 8 illustrates the architectural differences between the sequential, stateful approach and the holistic planning approach. The specific protocols for these configurations are defined as follows:

1. **Global Memory Configuration (Conversational Coding).** As depicted in Figure 8a, the agent processes Pull Requests in a strict sequential order while maintaining shared conversational and working context across the entire task chain. The `task description` and `definition description` are provided iteratively. A key feature of this setting is the *cascading test suite*: obligations introduced in earlier PRs (PR_{i-1}) remain in the active test scope for current and future PRs (PR_i). This simulates a realistic development workflow where regressions must be prevented and previously fixed behaviors must remain stable throughout the lifecycle of the feature chain.

2. **ATS Configuration (ATS-based Coding).** Illustrated in Figure 8b, this setting shifts the agent's focus from iterative execution to architectural planning. We concatenate the requirements from the entire task chain into a unified Aggregated Task Specification (ATS). The agent receives this high-level specification upfront and is prompted to generate a comprehensive plan before writing code. Unlike the iterative approach, the agent must infer the necessary orchestration strategy to satisfy all requirements simultaneously, and the accumulated test suite is evaluated only at the final state.

A.5. Repository Code Quality Evaluation Metrics

A.5.1. Cognitive Complexity

Cognitive Complexity is a metric designed to measure the mental effort required by a developer to understand the control flow of a piece of code. Unlike older metrics that count lines or logical paths, it focuses on how humans perceive code readability by penalizing nesting and non-linear logic[3].

Formula: The calculation follows three basic rules:

1. **Increments:** +1 for each control flow structure (`if`, `for`, `try`, etc.).
2. **Nesting:** +1 for each level of nesting inside a control structure.
3. **Abrupt Flow:** +1 for jumps like `break` or `continue`.

$$\text{Total Score} = \sum(\text{Structural Increments}) + \sum(\text{Nesting Penalties})$$

Python Example:

```
def process_data(data):
    if data:                        # +1
        for item in data:           # +2 (1 for for, 1 for nesting)
            if item < 0:            # +3 (1 for if, 2 for nesting)
                continue            # +1 (abrupt break)
    return data                     # Total Score = 7
```

A.5.2. SQALE Index

The SQALE (Software Quality Assessment based on Lifecycle Expectations) Index represents the total effort required to fix all current "code smells" and violations in a codebase. It is primarily used to quantify Technical Debt in terms of time (person-days)[4]

Formula: The core calculation is based on the Remediation Cost, which is the sum of time required to fix each individual issue.

$$\text{SQALE Index} = \sum_{i=1}^{n} \text{Remediation Cost of Issue}_i$$

The **Technical Debt Ratio** is then calculated to provide a grade (A-E):

$$\text{Debt Ratio} = \frac{\text{Remediation Cost}}{\text{Estimated Development Cost}}$$

Python Concept Example:

```
# Static Analysis Report Simulation:
violations = [
    {"type": "Complexity", "cost_to_fix": 15}, # 15 mins
    {"type": "Vulnerability", "cost_to_fix": 45} # 45 mins
]

sqale_index = sum(v["cost_to_fix"] for v in violations) # Total: 60 mins debt
```

[3] SonarQube, "Cognitive Complexity, Because Testability != Understandability," https://www.sonarsource.com/blog/cognitive-complexity-because-testability-understandability

[4] SonarQube, "Technical Debt and SQALE".

A.6. Performance Analysis

A.6.1. Confidence Interval for Results on Mini Dataset

Model	Comparison	Mean paired gap	95% bootstrap CI	Permutation p
Claude Sonnet 4.5	Individual − Global	+16.38 pp	[+6.05, +26.59]	0.0029
Claude Sonnet 4.5	Individual − ATS	+22.19 pp	[+11.08, +33.09]	0.0009
Gemini 3 Flash	Individual − Global	+15.84 pp	[+7.58, +24.83]	0.0010
Gemini 3 Flash	Individual − ATS	+23.93 pp	[+13.43, +34.10]	0.0002
GPT-5.2 Chat	Individual − Global	+8.81 pp	[+0.28, +17.86]	0.0352
GPT-5.2 Chat	Individual − ATS	+10.84 pp	[−1.11, +21.06]	0.0398
GPT-5.1 Codex Mini	Individual − Global	+8.93 pp	[−0.57, +19.21]	0.0542
GPT-5.1 Codex Mini	Individual − ATS	+9.07 pp	[−2.92, +19.25]	0.0671
Gemini 3 Pro	Individual − Global	+7.27 pp	[−3.53, +18.88]	0.1143
Gemini 3 Pro	Individual − ATS	+10.07 pp	[−5.96, +24.16]	0.1125

Table 9. **Paired task-level evidence that reset-based SWE-bench-style evaluation inflates coding-agent performance relative to our proposed evaluation settings.** Each row compares the Individual PR baseline against one proposed setting, Global or ATS, for the same model on the same SWE-STEP-Mini task chains. The *Comparison* column specifies the paired difference being estimated. The *Mean paired gap* column reports the average task-level difference in PR pass rate, in percentage points, computed as Individual minus the proposed setting; positive values therefore indicate higher measured performance under the reset-based Individual protocol. The *95% bootstrap CI* column reports a task-chain bootstrap confidence interval for this mean gap, obtained by resampling task chains with replacement. The *Permutation p* column reports the one-sided paired sign-flip test for the directional hypothesis that Individual outperforms the proposed setting. Rows whose confidence intervals exclude zero provide the strongest per-model evidence of performance inflation, while positive means with intervals overlapping zero indicate directionally consistent but weaker evidence.

Paired significance analysis. For each model in Table 1, we compare the Individual PR baseline against Global and ATS on the same task chains from SWE-STEP-Mini. The Individual setting corresponds to a SWE-bench-style reset-based protocol, where each PR is evaluated independently from a clean historical pre-state. In contrast, Global and ATS are our proposed evaluation settings, which introduce persistent repository state or aggregated multi-PR specifications. Positive gaps therefore indicate that the reset-based Individual baseline yields higher task-level PR pass rates than the corresponding proposed setting, i.e., that isolated PR evaluation inflates measured performance relative to our more realistic stateful evaluation protocols.

We conduct this paired analysis in Table 9 on SWE-STEP-Mini because long-horizon agent evaluation is computationally expensive. Even on the 20-task Mini split, the OpenHands evaluation matrix in Table 1 costs approximately $3.8K in inference alone across models and settings; scaling the same paired statistical analysis to the full benchmark would be substantially more expensive. This motivates using the Mini split for detailed paired uncertainty analysis, while using the Lite split and additional agents as complementary evidence of the same trend.

Mean paired gaps in Table 9 are reported in percentage points. Confidence intervals are computed by resampling task chains with replacement, rather than resampling individual PRs, so that the paired structure of each task chain and the dependence among PRs within a chain are preserved. We also report a one-sided paired sign-flip permutation test because our hypothesis is directional: the reset-based Individual baseline should overestimate performance relative to our proposed stateful settings. The test therefore asks whether the paired task-level gaps are systematically positive under a null hypothesis of no consistent advantage for either setting. We interpret the bootstrap intervals and permutation tests as complementary evidence: rows whose confidence intervals exclude zero provide the strongest per-model evidence of performance inflation, while rows with positive means but intervals overlapping zero are treated as directionally consistent but not individually conclusive.

A.6.2. OpenHands: General Benchmarks

Analysis of performance on the Lite and Mini datasets, focusing on Pass-to-Pass (P2P) and Fail-to-Pass (F2P) metrics.

Table 10 presents the specific breakdown of the Lite dataset results by repository. It lists the number of completed pull requests and tasks for each setting (Individual, Global, ATS), illustrating how the performance degradation and "spillover" effects vary across different codebase structures and sizes.

Tables 11 and 12 dissect the test-level performance across the *Mini* and *Lite* datasets, respectively. A consistent trend emerges across both scales: models exhibit a strong bias toward regression safety (high P2P scores) at the expense of

task resolution (F2P scores) as statefulness increases. In Table 11, top-performing models like Claude Sonnet 4.5 see F2P rates drop by nearly **30%** when moving from Individual to ATS settings. Table 12 corroborates this on a larger scale, where Gemini Flash's F2P performance nearly halves (52.45% to 27.65%) in the ATS environment. This data suggests that "spillover" effects in long-context interactions cause agents to struggle with fault localization, leading them to preserve the status quo rather than successfully implementing the necessary fixes.

Table 10. (Lite dataset) Performance comparison of models across multiple repositories using the lite dataset (total expenditure: $1,745). The **PR** and **Task** columns indicate the number of completed pull requests and tasks, respectively.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Total	PR Success	Total Cost
		PR /25	Task /4	PR /24	Task /7	PR /45	Task /8	PR /78	Task /13	PR /55	Task /12	PR /49	Task /6	Passsed /276	Rate	($)
Individual	GPT 5.1 Codex Mini	16	0	6	0	31	1	27	0	25	0	15	0	**120**	**43.48**	186.39
	Gemini 3 Flash	17	1	12	1	35	2	46	1	25	0	21	0	**156**	**56.52**	139.52
Global	GPT 5.1 Codex Mini	12	1	4	0	15	0	18	0	17	0	5	0	**71**	**25.72**	255.79
	Gemini 3 Flash	12	0	11	2	23	1	31	1	16	0	8	0	**101**	**36.59**	216.43
ATS	GPT 5.1 Codex Mini	12	1	3	0	11	0	14	0	24	1	5	0	**69**	**25.00**	454.51
	Gemini 3 Flash	13	0	8	1	18	0	22	0	17	0	11	1	**89**	**32.25**	492.36

Table 11. (Mini dataset) Performance comparison of models across multiple repositories using the mini dataset. The **P2P** and **F2P** columns indicate the percentage of Pass to Pass and Fail to Pass tests passed.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Average	
		P2P /230	F2P /34	P2P /348	F2P /42	P2P /80	F2P /64	P2P /954	F2P /31	P2P /650	F2P /50	P2P /812	F2P /18	P2P /3074	F2P /239
Individual	Gemini 3 PRO	99.57	64.71	52.87	59.52	100.00	96.88	63.94	41.94	56.31	42.00	95.07	44.44	72.90	63.18
	Claude Sonnet 4.5	95.22	67.65	71.26	71.43	100.00	100.00	100.00	38.71	78.46	78.00	100.00	61.11	91.83	74.90
	GPT 5.2 Chat	100	52.94	58.62	61.9	100.00	42.19	95.28	35.48	65.08	30.00	99.26	55.56	86.27	44.77
	Gemini 3 Flash	81.74	47.06	70.98	66.67	100.00	100.00	93.61	41.94	57.69	30.00	94.95	38.89	83.08	59.83
	GPT 5.1 Codex Mini	92.17	35.29	51.72	21.43	100.00	98.44	90.15	9.68	78.46	54.00	94.95	16.67	85.00	48.95
Global	Gemini 3 PRO	99.57	38.24	60.06	54.76	100.00	100.00	96.02	22.58	73.23	62.00	99.14	11.11	88.32	58.58
	Claude Sonnet 4.5	100.00	47.06	61.21	40.48	100.00	87.50	100.00	25.81	49.38	32.00	99.14	22.22	84.68	48.95
	GPT 5.2 Chat	100	50	54.89	42.86	100.00	26.56	98.85	51.61	75.23	40.00	99.01	33.33	89.04	39.33
	Gemini 3 Flash	97.39	20.59	62.93	66.67	100.00	93.75	90.04	22.58	87.54	52.00	92.86	22.22	88.00	55.23
	GPT 5.1 Codex Mini	79.57	41.18	50.86	14.29	92.50	64.06	90.15	12.90	63.69	48.00	98.65	27.78	81.62	39.33
ATS	Gemini 3 PRO	80.87	23.53	58.91	54.76	100.00	95.31	92.35	6.45	71.08	56.00	97.78	33.33	84.84	53.56
	Claude Sonnet 4.5	72.17	29.41	49.14	42.86	100.00	26.56	99.79	45.16	82.00	66.00	98.77	33.33	87.96	41.00
	GPT 5.2 Chat	99.57	61.76	91.95	28.57	100.00	25.00	42.66	3.23	84.00	50.00	98.40	27.78	77.46	33.47
	Gemini 3 Flash	80.87	23.53	44.54	40.48	100.00	95.31	94.65	16.13	74.62	44.00	75.49	5.56	78.79	47.70
	GPT 5.1 Codex Mini	98.7	26.47	78.74	11.9	100.00	15.62	99.79	3.23	83.23	68.00	99.14	11.11	93.66	25.52

Table 12. (Lite dataset) Performance comparison of models across multiple repositories using the lite dataset. The **P2P** and **F2P** columns indicate the percentage of Pass to Pass and Fail to Pass tests passed.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Average	
		P2P /230	F2P /34	P2P /467	F2P /63	P2P /1328	F2P /280	P2P /3747	F2P /287	P2P /2834	F2P /173	P2P /1765	F2P /42	P2P /10371	F2P /879
Individual	GPT 5.1 Codex Mini	92.17	35.29	64.03	26.98	95.33	49.64	96.32	13.24	60.09	29.48	97.34	14.29	**84.92**	**29.92**
	Gemini 3 Flash	81.74	47.06	74.95	50.79	99.70	69.29	96.77	57.84	51.73	27.97	88.05	35.71	**82.04**	**52.45**
Global	GPT 5.1 Codex Mini	79.57	41.18	63.38	20.63	78.31	33.57	93.3	4.88	47.85	27.17	96.77	11.9	**77.90**	**21.27**
	Gemini 2.5 Flash	97.39	20.59	72.38	52.38	94.73	50.71	96.93	19.51	79.78	27.75	90.71	19.05	**89.81**	**33.45**
ATS	GPT 5.1 Codex Mini	98.7	26.47	84.15	20.63	86.45	12.5	96.02	3.83	66.94	36.99	89.75	4.76	**85.31**	**15.24**
	Gemini 2.5 Flash	80.87	23.53	47.54	33.33	95.11	38.57	91.22	14.98	61.36	31.79	86.57	19.05	**80.57**	**27.65**

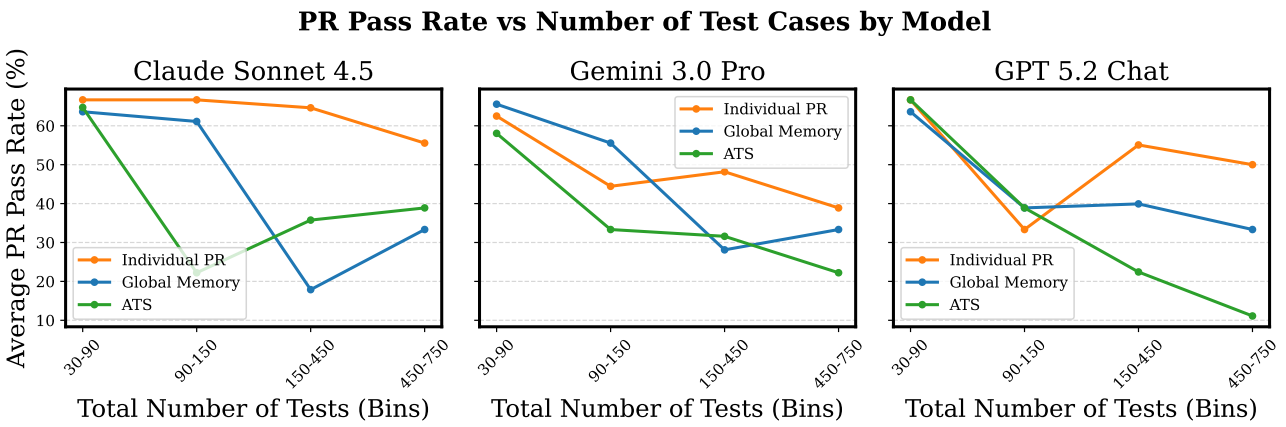


Figure 9. **Pass Rate Analysis.** Comparison of Average PR Pass Rates for Claude, Gemini, and GPT models relative to test suite size. The plots reveal that the *Individual PR* setting consistently maintains higher pass rates compared to *Global Memory* and *ATS* strategies, particularly as the number of tests increases.

Figure 9 shows agent performance with the variation in chain length and the test suite size.

A.6.3. Repository-Specific Analysis: SonarQube

A focused analysis comparing performance on the SonarQube repository against other repositories in the dataset.

Figure 10 visualizes the longitudinal evolution of code health metrics—Δ Cognitive Complexity and Δ SQALE Index (Technical Debt)—across six repositories. The data is normalized into five temporal bins to allow for a direct comparison between the trajectories of autonomous agents (Gemini 3.0 Pro, Claude Sonnet 4.5, GPT 5.2 Chat) and the Human Gold Trace. The plots serve to illustrate the deviation in coding standards: while the human baseline typically reflects a steady, controlled progression (often near-zero or negative slope due to refactoring), agent trajectories frequently diverge, exhibiting either unbounded accumulation of complexity (e.g., `moto`, `sqlglot`) or erratic volatility (e.g., `pandas`), indicative of a struggle to maintain code maintainability over long-context interactions.

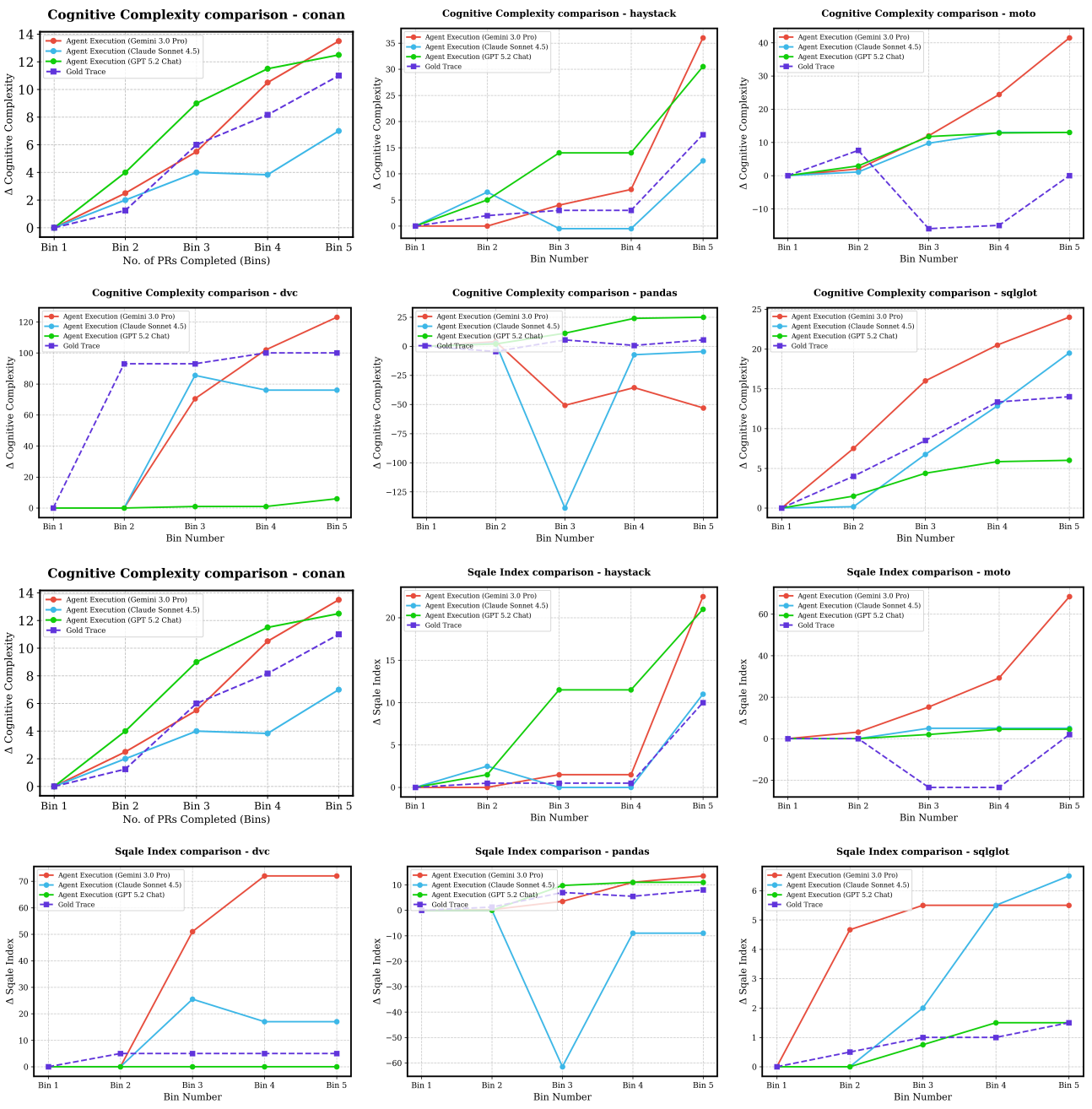


Figure 10. **Code Quality Evolution Across Repository Difficulty Levels.** The figure displays Δ Cognitive Complexity (rows 1–2) and Δ SQALE Index (rows 3–4) across six repositories of varying difficulty. **Rows 1 & 3** (`conan`, `haystack`, `moto`): Performance evaluated across five difficulty bins (Bin 1–5) for three Large Language Model agents (Gemini 3.0 Pro, Claude Sonnet 4.5, GPT 5.2 Chat) versus the human ground truth (*Gold Trace*). The plots reveal that while the Gold Trace maintains near-zero or low fluctuation in complexity and technical debt, autonomous agents introduce significantly higher complexity and debt accumulation, particularly in higher bins. **Rows 2 & 4** (`dvc`, `pandas`, `sqlglot`): High-difficulty scenarios where agents struggled significantly. Rather than consistently accumulating debt, agents often exhibit flatlines (e.g., GPT 5.2 in `dvc`) or erratic negative drops (e.g., `pandas`), indicating failure to generate necessary code modifications compared to the steady progression of the human ground truth.

A.6.4. Aider Performance Benchmarks

To show that our findings are generalizable, we evaluate our framework with the another open source coding agent, Aider (Gauthier, 2023). Contrast to OpenHands, it is a semi-autonomous pair programming tool that operates via a command line

interface. We used the Aider Python SDK to simulate an autonomous workflow similar to the one followed in OpenHands. Table 13 and Table 14 showcase Aider's performance across multiple repositories in the *Individual*, *Global* and *ATS* settings.

The general trend in performance seems to be consistent with our claims that *Indivdual* setting are often riddled with inflated scores that do not truly capture the essense of software engineering.

Since Aider is not equipped with a standalone planning agent or a good context condensation agent like OpenHands, we try to give it a fair chance by introducing a new setting: *Local*. In this setting, the context is automatically refreshed after each PR is complete. This bypasses the need for a context condensation agent, but comes at the cost of missing stateful context. We also supplement the context by providing all the file paths and function declarations to the agent. The performance in *Local* is considerably better than *Global*, necessitating the need for active context management when using AI pair programming agents.

Table 13. **(Aider Agent - Lite dataset)** Performance comparison of models across multiple repositories (total expenditure: $356.5). The **PR** and **Task** columns indicate the number of completed pull requests and tasks, respectively.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Total	PR Success	Total
		PR /25	Task /4	PR /24	Task /7	PR /45	Task /8	PR /78	Task /13	PR /55	Task /12	PR /49	Task /6	Passsed /276	Rate	Cost ($)
Individual	GPT 5.1 Codex Mini	13	0	6	1	22	0	24	0	14	0	6	0	**85**	**30.79**	32.0
	Gemini 2.5 Flash	16	0	7	0	22	0	39	1	19	0	22	0	**125**	**45.29**	41.0
Global	GPT 5.1 Codex Mini	7	0	3	0	7	0	5	0	5	1	1	0	**28**	**10.14**	31.50
	Gemini 2.5 Flash	8	0	2	1	7	0	11	1	4	0	1	0	**33**	**11.96**	126.0
ATS	GPT 5.1 Codex Mini	8	0	3	0	7	0	12	0	4	0	1	0	**35**	**12.68**	29.0
	Gemini 2.5 Flash	6	0	3	0	6	0	12	0	4	0	1	0	**32**	**11.59**	198.0

Table 14. **(Aider Agent - Lite dataset)** Performance comparison of models across multiple repositories. The **P2P** and **F2P** columns indicate the percentage of Pass to Pass and Fail to Pass tests passed.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Average	
		P2P /230	F2P /34	P2P /467	F2P /63	P2P /1328	F2P /280	P2P /3747	F2P /287	P2P /2834	F2P /173	P2P /1765	F2P /42	P2P /10371	F2P /879
Individual	GPT 5.1 Codex Mini	40.87	47.06	32.55	19.05	54.59	43.57	26.31	15.68	15.35	12.14	13.60	16.67	**25.38**	**25.37**
	Gemini 2.5 Flash	78.26	35.29	68.95	41.27	80.95	54.29	91.09	35.89	80.06	52.02	77.90	38.10	**83.25**	**45.39**
Global	GPT 5.1 Codex Mini	30.87	8.82	26.34	14.29	52.03	6.07	12.06	2.44	27.83	19.65	34.45	23.81	**26.36**	**9.10**
	Gemini 2.5 Flash	89.57	26.47	59.74	23.81	55.57	20.71	62.74	17.42	39.59	43.93	60.96	11.90	**55.65**	**24.23**
ATS	GPT 5.1 Codex Mini	99.62	0.00	100.00	0.00	95.18	16.79	99.52	0.0	99.85	0.00	98.52	0.00	**98.91**	**5.35**
	Gemini 2.5 Flash	85.22	29.41	73.88	17.46	48.42	37.85	62.12	13.24	64.64	51.44	53.99	4.76	**58.20**	**29.12**

Table 15. **(Aider Agent - Lite dataset)** Performance comparison of models across multiple repositories. The **P2P** and **F2P** columns indicate the percentage of Pass to Pass and Fail to Pass tests passed.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Average	
		P2P /230	F2P /34	P2P /467	F2P /63	P2P /1328	F2P /280	P2P /3747	F2P /287	P2P /2834	F2P /173	P2P /1765	F2P /42	P2P /10371	F2P /879
Individual	GPT 5.1 Codex Mini	60.00	50.00	43.25	20.63	59.04	38.93	39.20	17.07	10.87	10.98	18.47	16.67	**31.11**	**24.34**
w/Filepaths	Gemini 2.5 Flash	71.74	41.18	71.95	50.79	79.97	53.57	45.29	31.71	46.89	16.18	81.70	54.76	**58.15**	**38.45**
Global	GPT 5.1 Codex Mini	74.35	41.18	14.13	23.81	51.43	23.57	46.52	27.18	5.93	2.31	5.16	14.29	**22.04**	**7.17**
w/Filepaths	Gemini 2.5 Flash	88.70	44.12	58.67	57.14	59.26	45.71	50.76	15.68	60.52	57.80	61.93	19.05	**41.26**	**21.39**
Local	GPT 5.1 Codex Mini	99.62	0.00	100.00	0.00	95.18	16.79	99.52	0.0	99.85	0.00	98.52	0.00	**28.18**	**20.82**
w/Filepaths	Gemini 2.5 Flash	85.22	29.41	73.88	17.46	48.42	37.85	62.12	13.24	64.64	51.44	53.99	4.76	**57.61**	**37.76**

A.6.5. CODEX AGENT PERFORMANCE BENCHMARKS

To further examine whether our observations generalize beyond open-source coding agents, we evaluate our framework with Codex, OpenAI's cloud-based coding agent. Codex is designed as a managed agentic software engineering system capable of working over repository-level tasks such as feature implementation, bug fixing, refactoring, and pull-request generation (OpenAI, 2025). We use Codex with GPT 5.1 Codex Mini as the underlying model and evaluate it under the same benchmark protocol used for the other agents.

Table 16 and Table 17 detail the results when using Codex agent with the Mini Dataset. Similar to the trends observed with OpenHands, we see a performance drop from *Individual* setting to *Global* to *ATS* (**57.14** (*Individual*) → **51.43** (*Global*) → **45.71** (*ATS*)).

Table 16. **(Codex Agent - Mini Dataset)** Performance comparison of models across multiple repositories (total expenditure: $113.1). The **PR** and **Task** columns indicate the number of completed pull requests and tasks, respectively.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Total	PR Success	Total
		PR /25	Task /4	PR /18	Task /5	PR /7	Task /2	PR /15	Task /2	PR /20	Task /5	PR /20	Task /2	Passsed /105	Rate	Cost ($)
Individual	GPT 5.1 Codex Mini	16	1	11	0	5	0	7	0	11	0	10	0	**60**	**57.14**	34.3
Global	GPT 5.1 Codex Mini	17	1	8	0	4	0	7	0	7	0	11	1	**54**	**51.43**	24.6
ATS	GPT 5.1 Codex Mini	17	1	3	0	4	0	4	0	10	0	10	0	**48**	**44.76**	54.2

Table 17. **(Codex Agent - Mini Dataset)** Performance comparison of models across multiple repositories. The **P2P** and **F2P** columns indicate the percentage of Pass to Pass and Fail to Pass tests passed.

Setting	Model	Conan		DVC		Haystack		Moto		Pandas		Sqlglot		Average	
		P2P /230	F2P /34	P2P /348	F2P /42	P2P /80	F2P /64	P2P /954	F2P /31	P2P /650	F2P /50	P2P /812	F2P /18	P2P /3074	F2P /289
Individual	GPT 5.1 Codex Mini	99.13	52.94	77.87	42.85	92.50	98.44	93.82	32.26	84.15	70.00	100.00	38.89	**91.96**	**63.18**
Global	GPT 5.1 Codex Mini	93.35	35.29	65.80	45.24	83.75	7.81	92.24	19.35	85.38	58.0	83.74	55.56	**85.49**	**33.89**
ATS	GPT 5.1 Codex Mini	73.48	38.24	67.24	9.52	88.75	17.19	99.79	0.00	78.46	66	99.26	44.44	**89.20**	**28.87**

Table 18. Effect of increasing the reflection-cycle budget in the sequential/global setting with `gpt-5.1-codex-mini`. Values report the number of PRs passed in each repository.

Repo	#PRs	1-cycle	3-cycle	5-cycle	7-cycle
Conan	10	5	7	7	7
Haystack	7	4	5	7	5
DVC	8	0	0	1	2
Total	**25**	**9**	**12**	**15**	**14**

A.6.6. Scaling with Reflection-Cycle Budget

We study how performance changes as the amount of harness-mediated test feedback increases. In our main setup, agents cannot initiate test runs on demand; instead, the harness executes the test suite after a fixed amount of work and returns standardised feedback. To understand how additional feedback affects performance, we run an ablation in the global setting using `gpt-5.1-codex-mini` on three repositories (`Conan`, `Haystack`, and `DVC`), varying the number of reflection cycles per PR.

Table 18 reports the number of PRs resolved as the reflection-cycle budget increases from 1 to 7. Performance improves from 9 passed PRs in a single cycle to a peak of 15 passed PRs over 5 cycles, then slightly declines to 14 passed PRs over 7 cycles. This suggests that additional rounds of standardised test feedback are beneficial initially, but the gains saturate after a few cycles and may even reverse when the agent is allowed too many reflection opportunities.

A.6.7. Qualitative Analysis of Agent Failure

Figure 11 illustrates a representative failure mode that emerges specifically in the stateful setting: the agent solves a previous issue instead of the current one. In this example, the current task is to fix a bug in `pd.json_normalize` from Pandas PR #62848. However, because the environment retains artifacts from earlier sub-tasks, the agent observes residual test logs associated with a different PR, PR #62729. During reflection, the agent incorrectly interprets these logs as evidence about the current task and redirects its implementation toward the already completed `NamedAgg` functionality. As a result, the agent produces a patch for the wrong behavior, leaving the original `json_normalize` issue unresolved. This example highlights how persistent state and accumulated context can induce goal drift in long-horizon coding agents, even when the immediate task specification is clear.

Failure Mode: Solving Previous Issues

Pandas: PR #62848 (3rd PR in the task)
Fix a bug in `pd.json_normalize` where an `AttributeError` was raised if `max_level` was set to an integer and input data contained non-dict items.

Cycle 1 (Observation)
The agent tries to solve the given task. The agent runs tests. The environment (shared with previous tasks) displays residual logs from a *different* PR #62729 (1st PR).

```
pytest pandas/tests/groupby/aggregate/test_aggregate.py
...
PASSED pandas/tests/groupby/aggregate/test_aggregate.py
(Test case summary:  All tests passed for NamedAgg support)
```

Cycle 2 (Reflection)
The agent hallucinates that the passed tests for the *previous* PR are actually the requirements for the *current* PR.

> *"I see the tests for `test_aggregate.py` are passing. Wait, looking at the logs, it seems I am interacting with the `NamedAgg` functionality (PR #62729). I will rewrite the functionality to match PR #62729."*

Outcome: *Failure.* The agent implemented support for `NamedAgg` (PR #62729) instead of fixing `json_normalize` (PR #62848). The original issue remains unresolved.

Figure 11. Agent conflates current sub-task with artifacts found in env. from a previous sub-task.

A.6.8. Qualitative Analysis of Model-Generated Code Maintainability

The maintainability impact of model-generated code is assessed using *SonarQube*, so the flagged issues are consistent with SonarSource's definition of Cognitive Complexity (Campbell, 2023). The examples below are drawn from tasks in our dataset that SonarQube marked as higher complexity. For each example, we highlight *why* the pattern is problematic and provide a *possible fix*.

Pattern 1: Deeply Nested Control Flow. This pattern increases cognitive complexity because the reader must track multiple levels of branching and iteration before reaching the actual action. Such nesting makes the code harder to follow, test, and modify.

Pattern 1: Deeply Nested Control Flow

Problematic Code

```
if isinstance(tools, list):
   for item in tools:
     if isinstance(item, Toolset):
       for tool in item:
         if hasattr(tool, "warm_up"):
           tool.warm_up()
```

Why it is problematic: The reader must track 5 levels of nesting before reaching the action. Each additional level multiplicatively increases the number of states a reader must hold in working memory, raising the SonarQube cognitive complexity score.

Fixed Code

```
def warm_up_toolset(toolset):
   for tool in toolset:
     if hasattr(tool, "warm_up"):
       tool.warm_up()

for item in tools:
   if isinstance(item, Toolset):
     warm_up_toolset(item)
```

Fix: Extracting nested logic into a helper function reduces the maximum nesting depth and isolates responsibility, directly lowering the cognitive complexity score.

Figure 12. Deeply nested control flow flagged by SonarQube and its refactored equivalent.

Pattern 2: Bare Exception Handling. Using `except:` without specifying an exception class catches *all* exceptions, including unintended ones such as `KeyboardInterrupt` and `SystemExit`. This can silently suppress real failures and make debugging significantly harder.

Pattern 2: Bare Exception Handling

Problematic Code

```
try:
   value = eval(raw_value)
except:
   value = raw_value.strip()
```

Why it is problematic: A bare `except:` clause intercepts all exceptions including `KeyboardInterrupt` and `SystemExit`, masking unrelated failures and making the root cause of bugs harder to trace.

Fixed Code

```
try:
   value = eval(raw_value)
except Exception:
   value = raw_value.strip()
```

Fix: Specifying `Exception` ensures only application-level exceptions are caught, allowing system-level signals to propagate correctly.

Figure 13. Bare exception handling flagged by SonarQube and its corrected equivalent.

A.7. Prompts, Inputs, and Artifacts

A.7.1. Task Generation and Categorization Prompts

The specific prompts used to categorize PRs before execution.

```
You are a software development expert tasked with categorizing Git commits based on
    their associated text content (PR titles, descriptions, etc.).
Commit ID: {commit_id}
Text Content:
{commit_text}
Available Categories:
{chr(10).join(f"- {cat}" for cat in categories)}
Category Descriptions:
- Feature/Enhancement: New features, performance improvements, security enhancements
- Bug Fix: Bug fixes and issue resolution
- Maintenance: Refactoring, code cleanup, dependency updates
- Infrastructure: Build changes, CI changes, configuration changes
- Documentation: Documentation updates
- Testing: Test additions, modifications, test-related changes
Instructions:
1. Analyze the provided text content carefully
2. Determine the primary purpose/type of this commit
3. Select the most appropriate category from the list above
4. Provide a brief explanation for your categorization choice
5. Identify the exact keywords/phrases from the input text that influenced your
    decision
Response Format (JSON only):
Return your response as a valid JSON object with the following structure:
{{
    "category": "Selected Category",
    "explanation": "Brief explanation of why this category was chosen",
    "confidence": "HighMediumLow",
    "reasoning": "Key indicators that led to this categorization",
    "keywords": ["exact", "keywords", "or phrases", "from input text"]
}}
Please categorize this commit now and return only the JSON response.
```

A.7.2. AGENT SYSTEM PROMPTS

The comprehensive additional user prompts provided to the agents along with openhands system prompts for Individual and Sequential PR execution, as well as ATS processing.

```
You are working on a SINGLE pull request in the repository at: {workdir}.

Task:
{task_text}

You are NOT running over a full task list or stacking tests from other PRs.
Focus only on this PR and the currently failing tests.

Hard constraints (you MUST obey these):
- DO NOT run tests yourself for any reason.
  - Never call `pytest`, `python -m pytest`, `tox`, `nox`, `nose`,
    `unittest`, `coverage run`, or any other test runner commands.
- Treat the feedback below as the only ground truth about which tests are failing.
- Never run any test wrapper scripts created by the harness, such as:
  - `combined_PR<PR_NO>_<timestamp>.sh`
  - `run_tests_PR<PR_NO>.sh`
  - any script whose name contains `combined_PR` or `run_tests_PR`.
- Do not execute any shell command that looks like it is running tests.

If there are missing dependencies in environment or you want to install them,
please note that testcase execution uses the following environment path: {env_path}
- Use the provided env only for all installation.

Feedback from the last test run:
{feedback}

Rules:
- Make minimal, deterministic code changes.
- Prefer editing existing files over creating new ones.
- When you edit files, ensure they remain syntactically valid.
- Do NOT add or modify tests unless explicitly instructed.
- DO NOT ADD TIMEOUT FOR MORE THEN 120.0s at max in any case.
```

```
You are working in the repository at: {workdir}.

Task:
{task_text}

Do not look into test files or execute testcases unless specified.

Hard constraints (you MUST obey these):
- DO NOT run tests yourself for any reason.
  - Never call `pytest`, `python -m pytest`, `tox`, `nox`, `nose`,
    `unittest`, `coverage run`, or any other test runner commands.
- Treat the feedback above as ground truth about which tests are failing.
- Never run any test wrapper scripts created by the harness, such as:
  - `combined_PR<PR_NO>_<timestamp>.sh`
  - `run_tests_PR<PR_NO>.sh`
  - any script whose name contains `combined_PR` or `run_tests_PR`.
- Do not execute any shell command that looks like it is running tests.

If there are missing dependencies in environment or you want to install them,
please note that testcase execution uses the following environment path: {env_path}
- Use the provided env only for all installation.
```

```
Feedback from the last test run:
{feedback}

Rules:
- Make minimal, deterministic code changes.
- Prefer editing existing files over creating new ones.
- When you edit files, ensure they remain syntactically valid.
- Do NOT add or modify tests unless explicitly instructed.
- DO NOT ADD TIMEOUT FOR MORE THEN 120.0s at max in any case.
```

```
You are working in the repository at: {workdir}.

You are given following list of code and document change requests on everything
    needed to implement multiple changes together.

{task_text}

PROCESS (MANDATORY)
1) Before making any code changes, delegate to planning_agent to produce PLAN.md.
2) Wait for the plan output and follow it step-by-step.
3) Only after PLAN.md is written, start implementation if requirements.
4) If the plan becomes invalid after new evidence (tests/logs), re-delegate and
    update PLAN.md.

DELEGATION
Use the DelegateTool to call planning_agent with:
- Objective
- ATS/requirements (summarize if large)
- Constraints (time/budget, max iterations)
- Required deliverables
Planning agent must write/update PLAN.md using PlanningFileEditorTool.

Do not look into test files or execute testcases unless specified.

Hard constraints (you MUST obey these):
- DO NOT run tests yourself for any reason.
  - Never call `pytest`, `python -m pytest`, `tox`, `nox`, `nose`,
    `unittest`, `coverage run`, or any other test runner commands.
- Treat the feedback above as ground truth about which tests are failing.
- Never run any test wrapper scripts created by the harness, such as:
  * `combined_PR<PR_NO>_<timestamp>.sh`
  * `run_tests_PR<PR_NO>.sh`
  * any script whose name contains `combined_PR` or `run_tests_PR`.
- Do not execute any shell command that looks like it is running tests.

If there are missing dependencies in environment or you want to install them,
please note that testcase execution uses the following environment path: {env_path}
- Use the provided env only for all installation.

Feedback from the last test run:
{feedback}

Rules:
- Make minimal, deterministic code changes.
- Prefer editing existing files over creating new ones.
- When you edit files, ensure they remain syntactically valid.
- Do NOT add or modify tests unless explicitly instructed.
- DO NOT ADD TIMEOUT FOR MORE THEN 120.0s at max in any case.
```

A.7.3. Sample Inputs: PR Descriptions and Requirements

Examples of the input data provided to the model, including raw Pull Request descriptions and structuredAggregated Task Specification.

This is a task request in which we need to add new code or modify the existing code in the repository or do both.

Task Request

Request

Task Description

- Fix `OpenAIChatGenerator response_format` serialization errors by allowing proper serialization of `response_format` dictionary objects. The issue occurs because `{"type": "json_object"}` is a valid `response_format`, which is a dictionary object, not a class, causing `TypeError: issubclass() arg 1 must be a class`.

Issue Description

No separate issue entry present.

Documentation Changes

No documentation changes present.

No new functions or classes were added in this commit. Some existing functions or classes were modified.
There are several functions or classes that need to be modified, using the definitions below:

Modified Definitions

Definitions

Function: `OpenAIChatGenerator.to_dict`
Docstring: Serialize this component to a dictionary.
Returns:

- The serialized component as a dictionary.

Class: `OpenAIChatGenerator`
Declaration: `class OpenAIChatGenerator:`
Docstring: Completes chats using OpenAI's large language models (LLMs).
It works with the `gpt-4` and `o-series` models and supports streaming responses from OpenAI API. It uses `ChatMessage` format in input and output.
You can customize how the text is generated by passing parameters to the OpenAI API. Use the `**generation_kwargs` argument when you initialize the component or when you run it. Any parameter that works with `openai.ChatCompletion.create` will work here too.
For details on OpenAI API parameters, see OpenAI documentation.

Usage example

```
from haystack.components.generators.chat import OpenAIChatGenerator
from haystack.dataclasses import ChatMessage

messages = [ChatMessage.from_user("What's Natural Language Processing?")]
client = OpenAIChatGenerator()
response = client.run(messages)
print(response)
```

Output

```
{'replies':
    [ChatMessage(_role=<ChatRole.ASSISTANT: 'assistant'>, _content=
    [TextContent(text="Natural Language Processing (NLP) is a branch of
artificial intelligence
        that focuses on enabling computers to understand, interpret, and
generate human language in
        a way that is meaningful and useful.")],
     _name=None,
     _meta={'model': 'gpt-4o-mini', 'index': 0, 'finish_reason': 'stop',
```

```
    'usage': {'prompt_tokens': 15, 'completion_tokens': 36, 'total_tokens':
51}})]
}
```

Please note that in addition to the newly added components mentioned above, you also need to make other code changes to ensure that the new feature can be executed properly.

1. Objective

Implement coordinated enhancements to DVC for database import functionality:

1. **Import-DB Command**: New `dvc import-db` command with `--sql` and `--model` modes for importing data from databases
2. **LFS Pre-fetching Support**: Fix Git-LFS pointer issue by pre-fetching LFS objects during imports
3. **SQLAlchemy Connection Strings**: Add config support for database connections with experimental SQLAlchemy integration
4. **Config Bug Fix**: Fix global config environment variable handling in `global_config_dir()`
5. **Functional Testing**: Add comprehensive tests for database import functionality

These features work together to enable DVC to import and track data from various database platforms using dbt adapters for authentication and SQLAlchemy for direct connections.

2. Context Summary

DVC Architecture Overview

COMMANDS LAYER (`DVC/COMMANDS/`)

- Each command has a class extending `CmdBase`/`CmdBaseNoRepo`
- Commands registered in `dvc/cli/parser.py` via `add_parser()` function
- Commands delegate to repository methods

REPOSITORY LAYER (`DVC/REPO/`)

- `Repo` class imports methods from individual files (e.g., `from dvc.repo.imp import imp`)
- Import logic: `imp()` → `imp_url()` → creates Stage with dependencies
- Locking via `@locked` decorator ensures thread safety

STAGE LAYER (`DVC/STAGE/`)

- `Stage` class represents pipeline stages and data operations
- Properties: `is_import`, `is_repo_import`, `is_versioned_import`, `is_partial_import`
- Stage operations: `update()`, `save()`, `run()`, etc.
- Import helpers: `update_import()`, `sync_import()` in `stage/imports.py`

DEPENDENCY LAYER (`DVC/DEPENDENCY/`)

- `Dependency` extends `Output` class
- `RepoDependency`: Imports from other repos (uses DVCFileSystem)
- `ParamsDependency`: Parameter tracking
- Factory pattern in `__init__.py`: `_get()` creates appropriate dependency type
- Schema defined in `SCHEMA` dict

CONFIG LAYER (`DVC/CONFIG.PY`, `DVC/CONFIG_SCHEMA.PY`)

- Config levels: system, global, repo, local
- Schema validation with voluptuous
- `dirs.py` provides config directory locations

FILESYSTEM LAYER (`DVC/FS/`)

- `GitFileSystem`: Git repository access
- `DVCFileSystem`: DVC repository access
- Version-aware filesystems for imports

Current Import Flow

1. Command: `dvc import <url> <path>` → `CmdImport.run()`
2. Repo: `repo.imp()` → `repo.imp_url()`
3. Stage Creation: `repo.stage.create()` with `deps=[url]`, `outs=[path]`
4. Graph Check: `repo.check_graph()` validates stage
5. Execution: Stage runs, downloads data via dependency
6. Save: Stage dumps to `.dvc` file

Key Issues to Address

ISSUE 1: CONFIG BUG

- `global_config_dir()` in `dvc/dirs.py` incorrectly uses `DVC_SYSTEM_CONFIG_DIR` instead of `DVC_GLOBAL_CONFIG_DIR`

ISSUE 2: LFS POINTERS

- Git-LFS tracked files are read as pointer text instead of actual content
- Need to pre-fetch LFS objects before `GitFileSystem` reads them

A.7.4. SAMPLE OUTPUT: GENERATED PLANS

A truncated qualitative example of a `PLAN.md` file generated by the OpenHands planning agent from the ATS mentioned in subsubsection A.7.3

1. Objective

Implement coordinated enhancements to DVC for database import functionality:

1. **Import-DB Command**: New `dvc import-db` command with `-sql` and `-model` modes for importing data from databases
2. **LFS Pre-fetching Support**: Fix Git-LFS pointer issue by pre-fetching LFS objects during imports
3. **SQLAlchemy Connection Strings**: Add config support for database connections with experimental SQLAlchemy integration
4. **Config Bug Fix**: Fix global config environment variable handling in `global_config_dir()`
5. **Functional Testing**: Add comprehensive tests for database import functionality

These features work together to enable DVC to import and track data from various database platforms using dbt adapters for authentication and SQLAlchemy for direct connections.

2. Context Summary

DVC Architecture Overview

COMMANDS LAYER (`DVC/COMMANDS/`)

- Each command has a class extending `CmdBase`/`CmdBaseNoRepo`
- Commands registered in `dvc/cli/parser.py` via `add_parser()` function
- Commands delegate to repository methods

REPOSITORY LAYER (DVC/REPO/)
- `Repo` class imports methods from individual files (e.g., `from dvc.repo.imp import imp`)
- Import logic: `imp()` → `imp_url()` → creates Stage with dependencies
- Locking via `@locked` decorator ensures thread safety

STAGE LAYER (DVC/STAGE/)
- `Stage` class represents pipeline stages and data operations
- Properties: `is_import`, `is_repo_import`, `is_versioned_import`, `is_partial_import`
- Stage operations: `update()`, `save()`, `run()`, etc.
- Import helpers: `update_import()`, `sync_import()` in `stage/imports.py`

DEPENDENCY LAYER (DVC/DEPENDENCY/)
- `Dependency` extends `Output` class
- `RepoDependency`: Imports from other repos (uses DVCFileSystem)
- `ParamsDependency`: Parameter tracking
- Factory pattern in `__init__.py`: `_get()` creates appropriate dependency type
- Schema defined in `SCHEMA` dict

CONFIG LAYER (DVC/CONFIG.PY, DVC/CONFIG_SCHEMA.PY)
- Config levels: system, global, repo, local
- Schema validation with voluptuous
- `dirs.py` provides config directory locations

FILESYSTEM LAYER (DVC/FS/)
- `GitFileSystem`: Git repository access
- `DVCFileSystem`: DVC repository access
- Version-aware filesystems for imports

Current Import Flow
1. Command: `dvc import <url> <path>` → `CmdImport.run()`
2. Repo: `repo.imp()` → `repo.imp_url()`
3. Stage Creation: `repo.stage.create()` with `deps=[url]`, `outs=[path]`
4. Graph Check: `repo.check_graph()` validates stage
5. Execution: Stage runs, downloads data via dependency
6. Save: Stage dumps to `.dvc` file

Key Issues to Address

ISSUE 1: CONFIG BUG
- `global_config_dir()` in `dvc/dirs.py` incorrectly uses `DVC_SYSTEM_CONFIG_DIR` instead of `DVC_GLOBAL_CONFIG_DIR`

ISSUE 2: LFS POINTERS
- Git-LFS tracked files are read as pointer text instead of actual content
- Need to pre-fetch LFS objects before `GitFileSystem` reads them

...