Test Coverage Analysis of Agentic Pull Requests

Atish Kumar Dipongkor
University of Central Florida
Orlando, FL, USA
atish.kumardipongkor@ucf.edu

Talank Baral
George Mason University
Fairfax, VA, USA
tbaral@gmu.edu

Wing Lam
George Mason University
Fairfax, VA, USA
winglam@gmu.edu

Kevin Moran
University of Central Florida
Orlando, FL, USA
kpmoran@ucf.edu

***Abstract*—AI coding agents increasingly submit complete pull requests (PRs) with minimal human intervention, shifting software development from AI-assisted to *autonomous* workflows. As these agents become more prevalent, ensuring the code they generate is adequately tested, by existing tests or by tests the agents write, is critical to preventing regressions, yet little is known about testing in agentic PRs. To address this gap, we analyze 4882 agent-generated PRs from the AIDev dataset (532 Java and 4350 Python PRs) produced by five coding agents. We study (i) how often agents include test changes and (ii) how well covered are code changes by existing and agent-written tests. Agents include test changes in only 49.6% of PRs that change *code under test* files. Existing tests provide an incomplete safety net: they cover 61.5% of agents' changed executable lines in Java and only 27.0% in Python, where 64.8% of PRs have no changed line executed by any existing test. Agent-written tests improve coverage over existing tests, but only in a minority of PRs: 35.9% of Java and 22.5% of Python *Code + Tests* PRs show a coverage gain. Across both languages, error-handling constructs (e.g., `try` and `catch` blocks) are the most consistently under-tested, with miss rates reaching 86.0% in Java and 81.0% in Python. These findings motivate coverage-aware development practices, coverage feedback loops for coding agents, and evaluation benchmarks that measure test quality to better help agents reliably test their own code.**

***Index Terms*—AI coding agents, test coverage, pull requests, software testing, empirical study**

I. Introduction

The emergence of autonomous AI coding agents marks a paradigm shift in software engineering practices [1]. As agents autonomously carry out software development tasks at scale, ensuring the quality of generated code is becoming a notable bottleneck, which raises fundamental questions about software quality assurance practices, particularly regarding testing.

Testing serves as one of the most important tools for verifying functional correctness and preventing regressions [2]–[6]. However, how well tested the code introduced by autonomous coding agents is remains unknown. Agent-introduced changes can be exercised by two sources of tests: the project's existing tests, or tests the agent writes alongside its changes. Neither source has been evaluated in practice: benchmarks like SWE-bench [7] assess agents' ability to resolve issues by passing existing tests, but do not measure whether those tests exercise the code agents introduce, nor whether agents create adequate tests for their new functionality. Prior research on AI-based test generation does assess test adequacy, demonstrating that Large Language Models (LLMs) [8] can produce syntactically valid tests [9], [10] and achieve reasonable code coverage when guided by appropriate prompting strategies [11], [12]. E.g., Meta's TestGen-LLM [13] and benchmarks, such as SWT-Bench [14], have shown promise in augmenting existing tests with LLM-generated tests that improve coverage or reproduce bugs. However, these studies evaluate settings where AI is explicitly instructed to generate tests. Such controlled evaluations do not reveal how well agents test their own changes when operating autonomously.

Recent empirical studies of agentic coding reveal concerning patterns: agents often prioritize functional code over comprehensive testing, and nearly one-third of merged agent-generated pull requests (PRs) require subsequent bug fixes or refactoring [15]. Similarly, surveys indicate that while developers use AI assistants for test generation, developers also express skepticism about the quality and coverage of AI-generated tests [16]. These findings suggest that while agents can generate tests on demand, they may neglect to test the new code paths introduced during autonomous development.

This disconnect has substantial implications for software maintenance. When agents introduce untested code paths, they create technical debt [17]–[19] that accumulates silently until regression failures emerge in production [20]. Understanding how agents currently approach testing and where they fall short is essential for establishing quality gates in AI-augmented development pipelines and for guiding the evolution of more testing-conscious agents.

In this paper, we present emerging results from an empirical analysis into how well-tested code generated by AI agents is in practice. We analyze 532 Java and 4350 Python pull requests from the AIDev dataset [1], spanning five prominent AI coding agents. Our analysis addresses two main research questions:

RQ_1: How often do agentic PRs contain test code changes? This RQ establishes a baseline of testing behavior. When agents create pull requests, do they include *test changes* (adding new/modifying existing tests) alongside code changes? We classify PRs into three categories: those that change ***both*** *code under test* and *test* files, those that change ***only*** *code under test*, and those that change ***only*** *test* files. Findings for this RQ show the "broad strokes" of agentic behavior.

RQ_2: What is the test coverage of code changes in agentic PRs? Beyond presence, we assess test effectiveness through coverage analysis, across two sources of tests:

RQ_{2a} (Coverage by existing tests): *What is the coverage of agent-written code for existing tests?* We analyze all merged

*This work is supported by the NSF CCF-2423813 & CCF-2338287 grants.

PRs that change code under test, i.e., both Code-Under-Test-Only and Code + Tests PRs. For the latter source, we measure coverage with the agent's test changes removed.

RQ_{2b} (Coverage gain by agent-written tests): *What is the coverage gain of agent-written code when agent-written tests are combined with existing tests?* We analyze Code + Tests PRs, comparing coverage with and without the agent's test changes. High gains indicate targeted, effective tests; low gains expose superficial testing.

Our analysis reveals four main findings: (i) 50.4% of agentic PRs that modify *code under test* include no test changes at all. (ii) Existing tests do not fully cover agent-written code: only 61.5% of agents' changed executable lines in Java and 27.0% in Python are covered, and 64.8% of Python PRs have none of their changed lines executed by any existing test. (iii) Agent-written tests significantly improve coverage of the agents' changes on average (+15.6 pp in Java, +9.6 pp in Python) over existing tests, but the gains are concentrated in a minority of PRs: only 35.9% of Java and 22.5% of Python Code + Tests PRs show any gain. (iv) Error-handling constructs are under-tested in both languages, with miss rates of 86.0% in Java and 81.0% in Python. Our results provide actionable insights for practitioners integrating AI agents into their workflows and for improving agent capabilities (Section V). Our results also illustrate notable shortcomings related to the "tested-ness" of agent generated code, signaling the need for new research that enable agents to better check and test their work.

This paper makes the following contributions:

- The first cross-language empirical study of testing behavior in real-world agentic PRs, spanning 4882 PRs across five AI coding agents and two languages.
- A coverage analysis of agent-generated code for existing tests and existing tests combined with agent-written tests.
- Identification of specific weak spots in agentic testing—error-handling constructs in both languages and Python's near-zero existing-test fallback—that point to concrete priorities for future agent design and continuous integration (CI) gating.
- A publicly available dataset [21], [22] of PR patches, test-only patches, coverage reports, and added lines labeled by syntactic construct (e.g., `Assignment`, `Return`, `Try-Catch`) to support future research.

II. Dataset

Java (1278) and Python (7191) PRs are taken from the AIDev dataset version 3 [1], spanning five agents across twelve conventional commit categories. Codex is the most popular agent in both languages, contributing 1018 Java and 5209 Python PRs; Copilot, Cursor, Claude Code, and Devin contribute the remainder. The category mix differs by language: Java is documentation-heavy (**`docs`** 504, **`fix`** 277, **`test`** 210, **`feat`** 180, **`refactor`** 53, **`chore`** 17, **`build`** 16), while Python is feature-heavy (**`feat`** 2315, **`fix`** 1802, **`docs`** 1376, **`test`** 449, **`refactor`** 438, **`build`** 300, **`ci`** 214, **`chore`** 202, **`style`** 46, **`perf`** 41, **`other`** 6, **`revert`** 2). To answer our RQs, we apply two filters to isolate PRs that modify executable code. First, we retain PRs that change at least one .java or .py file. Second, we parse the changed files with tree-sitter [23] and discard PRs whose modifications fall entirely within comments or docstrings. These filters yield 532 Java PRs across 68 repositories and 4350 Python PRs across 448 repositories, which we use for RQ_1. For the coverage analysis in RQ_2, we consider only merged PRs from repositories with at least 10 agentic PRs, discarding projects and repositories with too few agentic PRs to analyze reliably. This filter retains 14 Java and 55 Python repositories. The number of PRs used in RQ_2 is further discussed in Section III-A.

III. Methodology

A. Artifact Generation

PR patch. The AIDev dataset does not provide PR-level patches. We cloned each repository locally and reconstructed the unified diff for every PR using its base and head SHAs (**`git diff <base>..<head>`**), yielding a self-contained patch per PR.

Extracting test-only code from the patches. From each PR patch, we extract the changes corresponding to test files. The test-only patch enables us to remove those tests selectively during coverage measurement.

PR-level coverage. For each PR, we execute the repository's entire test suite, collecting line coverage with JaCoCo [24] for Java and `pytest-cov` [25] for Python. Of the repositories in the coverage subset (Section II), 10 of the 14 Java and 34 of the 55 Python repositories could be built and instrumented, yielding coverage results for 213 of the 532 Java PRs and 1664 of the 4350 Python PRs.

PR-level coverage (without tests). For each PR with a non-empty test-only patch, we reverse-apply the test-only patch (`git apply -R`) on the PR's **`<head>`** commit and re-run the suite. The difference between the two runs isolates the coverage contributed by agent-written tests. This paired analysis is applicable to the 64 Java and 605 Python PRs (of the 213 and 1664 merged PRs with coverage results) that contain test changes; the remaining 149 Java and 1059 Python PRs modify only code under test and are analyzed with developer's existing tests.

Identifying added *code under test* lines. For each PR patch, we record, for every *code under test* file, the line numbers of all added (+) lines as they appear in the head-commit version of the file. Test files are excluded by directory (**`test/tests)`** and by language-specific filename conventions: JUnit/Spring/Failsafe patterns (**`*Test`**, **`*Tests`**, **`*TestCase`**, **`Test*`**, **`*IT`**, **`*ITCase`**) for Java, and PyTest/unit test patterns (**`test_*.py`**, **`*_test.py`**) for Python.

Classifying added lines. For each PR, we snapshot every *code under test* file at the PR's head commit and parse it with **`srcml --position`** [26]. Following Zhu et al. [27], each added line is assigned the local tag of the deepest srcML element whose position range encloses its trimmed content (e.g., **`Assignment`**, **`If`**, **`Return`**). This typed inventory of added lines lets us characterize *which syntactic constructs* agents systematically leave untested.

B. RQ_1: Classifying Test Behavior

We categorize each PR by the type of test changes, using two criteria derived from the PR patch: (i) whether a hunk changes a test file (identified by the rules in Section III-A), and (ii) whether a hunk adds a new test.

Detecting newly added tests. For each hunk, we compute the set of test-method declarations in the post-change version of the hunk (i.e., the file content at the PR's head commit) minus those in the pre-change version (i.e., the content at the base commit); the difference identifies tests *introduced*, not merely edited, by the PR. A method counts as an added test if, in Java, it carries one of `@Test`, `@ParameterizedTest`, `@RepeatedTest`, `@TestFactory`, or `@TestTemplate` (JUnit 4/5, TestNG), or its name follows the JUnit 3 test name convention; in Python, a test is a `def` or `async def` whose method name begins with `test`, covering both `pytest` (`test_*`) and `unittest` (`test*`). Body-only edits and annotation-only changes leave the declared-name set unchanged and are not counted as added test.

PR classification. Using criterion (i), we assign each PR to exactly one of three categories: Code + Tests PRs, which change both code under test and test files; Tests-only PRs, which change test files only; and Code-Under-Test-Only PRs, which change code under test with no test-file changes. The distribution of these categories is in Section IV-A. Using criterion (ii), a PR *Adds Tests* if it introduces at least one new test, and *Modifies Tests* if it changes test files without introducing a new test. A PR may both add and modify tests. Figure 1 reports the category overlap.

C. RQ2: Measuring Diff Coverage and Analysis

To assess test effectiveness, we measure *diff coverage*: the percentage of added code lines that are executed by tests.

Coverage computation. For each modified *code under test* file, we check every added line against the JaCoCo or `pytest-cov` report and label it as covered (at least one test executed it) or missed. File-level diff coverage is then computed as:

$$\text{Diff Coverage}_{\text{file}} = \frac{|\text{Covered Lines}|}{|\text{Covered Lines}| + |\text{Missed Lines}|} \quad (1)$$

Aggregation. We aggregate file-level diff coverage to the PR level by summing covered and missed lines across all changed *code under test* files in the PR, then applying the same ratio. Doing so nets a single diff-coverage value per PR.

For Code + Tests PRs (RQ_{2a}), we additionally compute PR-level diff coverage on the without-tests run (Section III-A), denoting the two values as $\text{DiffCov}_{\text{with}}$ and $\text{DiffCov}_{\text{without}}$. We report their difference,

$$\Delta\text{DiffCov} = \text{DiffCov}_{\text{with}} - \text{DiffCov}_{\text{without}}, \quad (2)$$

as the diff-coverage gain attributable to the agent's own tests.

IV. Results

A. RQ1 Results: Test Inclusion in Agentic PRs

Figure 1 presents a Venn diagram of the intersection of PRs with three attributes: those that (i) add new tests, (ii) modify existing tests, and (iii) change only *code under test* files.

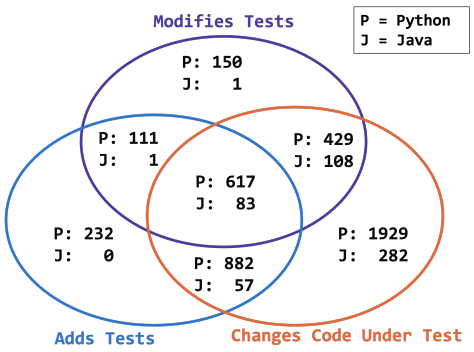

Fig. 1. PR classification results. Circles denote non-exclusive PR attributes; each region shows the number of PRs.

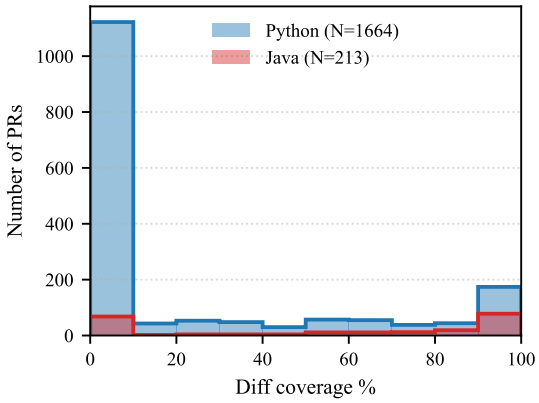

Fig. 2. Existing-test diff coverage per merged PR.

Overall test inclusion. Of the 4882 PRs in our dataset, 4387 modify *code under test* files (530 Java and 3857 Python); the remaining 495 change test files only, of which 430 are merged. Among the 4387 PRs that modify *code under test* files, 2176 (49.6%) include test changes, of which 1346 are merged, while 2211 (50.4%) have no such changes, of which 1692 are merged.

Test addition vs. modification. When agents do include tests, they more frequently add new tests than modify existing ones. A total of 1983 PRs add new tests, while 1500 PRs modify existing tests. 812 PRs both add and modify tests.

PR categories without test changes.

As a majority of PRs (53.0% Java, 44.3% Python) modify *code under test* files without any test changes, we examine their categories. Java is dominated by `docs` (37.6%), `fix` (34.4%), and `feat` (15.2%); Python by `feat` (40.5%), `fix` (30.3%), and `refactor` (11.8%).

B. RQ2 Results: Test Coverage in Agentic PRs

In this section, the results are from merged PRs (described in sections II and III-A) for both languages.

RQ_{2a}: What is the coverage of agent-written code for existing tests? For all merged PRs that change *code under test*, we measure the fraction of added executable lines covered by the repository's existing tests. For Code + Tests PRs, we measure coverage with the agent's test changes removed (Section III). This measurement indicates how much of the agent's diff the existing tests actually covers. Across the 213

TABLE I
DIFFCOV$_{\text{WITHOUT}}$ AND DIFFCOV$_{\text{WITH}}$ BY PR CATEGORY FOR CODE + TESTS PRS, COMPARED BETWEEN JAVA AND PYTHON. SIGNIFICANCE IS ASSESSED WITH THE ONE-SIDED WILCOXON SIGNED-RANK TEST [28] (H_1: DIFFCOV$_{\text{WITH}}$ > DIFFCOV$_{\text{WITHOUT}}$).

	Java					Python				
Category	**DiffCov$_{\text{without}}$**	**DiffCov$_{\text{with}}$**	**Δ (DiffCov)**	**% Improved**	***p*-value**	**DiffCov$_{\text{without}}$**	**DiffCov$_{\text{with}}$**	**Δ (DiffCov)**	**% Improved**	***p*-value**
feat	39.7%	87.0%	+47.3	82.4% (14/17)	$< 0.001^{***}$	18.3%	31.3%	+13.0	27.2% (99/364)	$< 0.001^{***}$
fix	74.8%	79.7%	+5.0	27.6% (8/29)	0.006^{**}	36.4%	42.9%	+6.6	17.7% (25/141)	$< 0.001^{***}$
test	62.5%	87.5%	+25.0	50.0% (1/ 2)	$0.500^{\text{n.s.}}$	25.7%	23.5%	-2.2	8.3% (3/ 36)	$0.633^{\text{n.s.}}$
refactor	—	—	—	— (0/ 0)	—	42.9%	46.6%	+3.7	11.4% (5/ 44)	$0.088^{\text{n.s.}}$
docs	97.1%	97.1%	+0.0	0.0% (0/14)	$n/a^{\text{n.s.}}$	27.3%	33.5%	+6.2	21.4% (3/ 14)	$0.231^{\text{n.s.}}$
build	—	—	—	— (0/ 0)	—	—	—	—	— (0/ 0)	—
chore	—	—	—	— (0/ 0)	—	19.4%	19.4%	+0.0	0.0% (0/ 3)	$n/a^{\text{n.s.}}$
perf	92.9%	92.9%	+0.0	0.0% (0/ 2)	$n/a^{\text{n.s.}}$	0.0%	0.0%	+0.0	0.0% (0/ 1)	$n/a^{\text{n.s.}}$
style	—	—	—	— (0/ 0)	—	1.0%	1.5%	+0.5	100.0% (1/ 1)	$0.500^{\text{n.s.}}$
ci	—	—	—	— (0/ 0)	—	0.0%	0.0%	+0.0	0.0% (0/ 1)	$n/a^{\text{n.s.}}$
Overall	**70.5%**	**86.1%**	**+15.6**	**35.9% (23/64)**	$< 0.001^{***}$	**24.8%**	**34.5%**	**+9.6**	**22.5% (136/605)**	$< 0.001^{***}$

Significance: $^{***}p < 0.001$; $^{**}p < 0.01$; $^{*}p < 0.05$; n.s. = not significant.

Java and 1664 Python merged PRs, existing tests cover 61.5% of changed executable lines in Java and only 27.0% in Python.

PR-level coverage. The two languages exhibit sharply different coverage from existing tests. In Java, the median PR has 71.1% of its changed executable lines covered by the existing tests, and 34.3% (73/213) are fully covered. Python tells a different story: the median PR has *zero* coverage, and existing tests cover none of the changed executable lines in 64.8% (1079/1664) of Python PRs. Figure 2 shows the underlying distribution: both languages are bimodal with mass at 0% and 100%, but Python concentrates overwhelmingly at the zero-coverage end while Java is more evenly distributed.

File-level coverage. Across all changed *code under test* files with at least one executable line in the diff, aggregate diff coverage is 61.5% in Java (259 files) versus 27.0% in Python (2696 files). In Java, 50.2% of files are fully covered and only 18.5% are uncovered; in Python, the proportions invert: 24.5% are fully covered and 54.2% of files are uncovered.

Takeaway. Existing tests are an incomplete safety net in both languages, and far weaker in Python: such tests cover 61.5% of changed lines in Java, but only 27.0% in Python. Existing tests also cover no changed lines in 64.8% of Python PRs.

RQ$_{2b}$: What is the coverage gain of agent-written code when agent-written tests are combined with existing tests? Table I reports DiffCov$_{\text{without}}$, DiffCov$_{\text{with}}$, and their delta across PR categories for Code + Tests PRs. Agent-written tests yield a statistically significant improvement: Java improves from 70.5% to 86.1% ($\Delta = +15.6$ pp; $p < 0.001$) and Python from 24.8% to 34.5% ($\Delta = +9.6$ pp; $p < 0.001$). The gain is concentrated in feature-implementing PRs, where agent tests deliver the largest deltas in both languages (Java: $+47.3$ pp on 17 PRs; Python: $+13.0$ pp on 364 PRs), followed by smaller but significant gains for bug fixes ($+5.0$ pp Java, $+6.6$ pp Python). Other categories—**`docs`**, **`refactor`**, **`chore`**, **`perf`**, **`style`**, **`ci`**—show little gain.

Improvement is concentrated in a minority of PRs. Only 35.9% of Java (23/64) and 22.5% of Python (136/605) Code + Tests PRs actually show any improvement. For the remainder, agent-supplied tests do not raise diff coverage. Inspecting these non-improving PRs reveals that the underlying causes differ sharply between the two languages. In Java, 42.2% already reach 100% diff coverage from the existing tests.

Among the rest, agents delete more tests than they add (82 deleted vs. 31 added, a 2.6× ratio), with another 51.2% editing only the bodies of existing tests – such edits modify existing test behavior but introduce no new tests targeting the changed code. In Python, the pattern inverts: only 8% hit the coverage ceiling and just 12.4% contain test deletions, yet 74.8% add new tests that nonetheless fail to cover the agent's own changed lines.

Takeaway. Agent-written tests significantly improve coverage on average, but only a minority of PRs drive this gain: just 35.9% of Java and 22.5% of Python Code + Tests PRs add coverage beyond the existing suite. Failures look different by language: Java agents often delete tests or hit a coverage ceiling, while Python agents add tests that exercise code other than the lines they introduced.

Categories of code that agents leave untested. Table II reports miss rates by syntactic category for the added executable lines. Two patterns dominate.

First, the cross-language gap in overall diff coverage extends to almost every category: Python miss rates run roughly 2–6× higher than Java's across **`Method Call`**, **`Assignment`**, **`If`**, **`For`**, and **`Return`**. Even common constructs like **`Assignment`** (16.1% Java vs. 63.1% Python) and **`Method Call`** (36.7% vs. 73.6%) are largely untested in Python.

Second, error-handling lines are under-tested in *both* languages. Newly added **`Throw`** statements miss 67.5% in Java and 82.3% of the time in Python; lines inside `Try-Catch` blocks miss 86.0% and 81.0%, respectively. Notably, Java's **`Try-Catch`** miss rate is the only category where Java performs *worse* than Python in absolute terms, despite its substantially higher overall coverage.

Takeaway. Agent test misses are not uniform across syntax. Error handling is the weakest spot in both languages, with most **`Throw`** statements and **`Try-Catch`** bodies left unexercised. In Python, the gap extends to routine constructs: **`Assignment`**, **`Method Call`**, and **`Return`** are all missed more than half the time—suggesting agents exercise happy paths,

TABLE II
DIFF-COVERAGE MISS COUNTS BY STATEMENT CATEGORY, COMPARED BETWEEN PYTHON AND JAVA. *Total* IS THE NUMBER OF EXECUTABLE LINES OF THAT CATEGORY ADDED BY AGENTS; *Miss%* IS THE FRACTION OF THOSE LINES THAT NO TESTS CAN COVER.

	Python		Java	
Category	**Total**	**Miss%**	**Total**	**Miss%**
Method Call	4,516	73.6%	403	36.7%
Assignment	9,472	63.1%	671	16.1%
If	3,094	62.5%	308	13.3%
For	710	54.1%	31	19.4%
While	94	69.1%	21	9.5%
Return	2,189	71.4%	367	39.8%
Try-Catch	1,356	81.0%	43	86.0%
Throw	536	82.3%	80	67.5%
Break	27	81.5%	3	33.3%
Continue	204	90.2%	2	0.0%
Switch	7	100.0%	1	0.0%
Definition	3,062	54.1%	35	22.9%
Import	2,930	58.4%	—	—
Other	1,171	58.3%	103	22.3%

while leaving large portions of their own diff unexercised.

V. DISCUSSION AND FUTURE DIRECTION

For practitioners. Do not assume agents write tests: 50.4% of *code under test*-modifying PRs include no test changes at all, and the safety net provided by existing tests is incomplete in both languages. Existing tests cover only 61.5% of agents' changed executable lines in Java and 27.0% in Python. with existing tests covering no changed lines in 64.8% of Python PRs. Teams using agentic PRs should not assume a passing run of tests means that the change has been tested. Review efforts should prioritize error-handling paths as 86.0% Java and 81.0% Python of the them are not covered regardless of whether the agent added tests.

For agent developers. Agents that do include tests still fail to test their own changes most of the time. Agent-written tests improve coverage on average, but add coverage of the agent's own changes in only 35.9% of Java and 22.5% of Python Code + Tests PRs. The two languages fail differently: Java agents sometimes *delete* more tests than they add (a 2.6× deletion-to-addition ratio in non-improving Java PRs), while Python agents add tests that exercise some other code than the code they introduced. Both patterns point to the same intervention: agents need a coverage-aware feedback loop that checks whether their own added lines are exercised by their own added tests before submitting a PR. As existing tests often leave agent changes not covered, agents cannot assume existing tests will test their newly added code. Error-handling constructs (e.g., `throw`/`raise`, `try`/`catch`) should be the priority target for agent code test generation as 86.0% Java and 81.0% Python of them are not covered.

For researchers. Our findings spur key new research:

(1) Coverage-aware agent benchmarks. SWE-bench [7] and similar benchmarks reward agents for passing existing tests, but do not analyze whether agents test the code they introduce. Future benchmarks should focus on diff coverage.

(2) Diagnose why agent tests miss the lines they introduce. For *Code + Tests* PRs that show no diff-coverage gain, our current analysis tells us that the gain is absent but not *why*. Diagnosing this behavior is an open problem whose answer can inform agent design and benchmark construction.

(3) Targeted test-generation for weak categories. Error-handling constructs miss 86.0% in Java and 81.0% in Python. How can targeted prompting shift agents' attention toward such constructs remains an important future work direction.

VI. RELATED WORK, LIMITATIONS

Related work. One existing body of work evaluates LLMs as test generators under controlled conditions. Pizzorno and Berger [11] and Ryan et al. [12] used coverage-guided prompting to drive LLM-generated tests toward higher branch and line coverage; Meta's TestGen-LLM [13] reported code coverage gains from augmenting existing tests with LLM-written tests; El Haji et al. [10] evaluated Copilot for Python test generation; and Mastropaolo et al. [9] studied the robustness of Copilot's code generation more broadly. These studies established that LLMs *can* produce useful tests when explicitly asked, but they evaluated models in a test-generation role rather than observing what agents do when given autonomy over a PR. Watanabe et al. [15] analyzed 567 agentic PRs and reported that 45.1% of merged PRs required human revisions, but do not measure test coverage. Coverage-based benchmarks such as SWE-bench [7] evaluate whether agents *pass* curated tests, treating the tests as fixed and externally provided, but none measure whether existing tests or tests an agent adds actually exercise the code that the agent introduced.

Limitations. Our coverage analysis is limited to merged PRs from repositories that could be built and instrumented, which may introduce sampling bias. To assess this threat, we compared lines of code (LOC), number of commits, age, stars, and forks between the analyzed repositories and the entire set of projects (median of analyzed vs. entire set). The 10 analyzed Java repositories are substantially smaller (29K vs. 102K LOC), less starred (437 vs. 745), and less forked (116 vs. 254) than the entire set of Java projects. That being said, the analyzed projects are comparable in number of commits (2271 vs. 3983 commits) and median age (4178 vs. 3785 days). Our Java coverage results may thus primarily reflect the testing practices of smaller projects. In contrast, the 34 analyzed Python repositories show no substantial differences from the entire set of Python projects on any metric (31K vs. 25K LOC, 1202 vs. 872 commits, 1065 vs. 1268 days, 676 vs. 734 stars, 120 vs. 141 forks). Full statistics and distribution plots are in our replication package.

VII. CONCLUSIONS

In this paper, we performed a cross-language study of how well tested agentic PRs are in the wild, which our results show are often poorly tested. Over half of *code under test*-changing PRs include no test changes; existing tests cover only 61.5% of agents' changed lines in Java and 27.0% in Python; agent-written tests improve coverage, but only in a minority of PRs; and error-handling constructs are often missed. Our findings spur new research to better help test agent generated code.

REFERENCES

[1] H. Li, H. Zhang, and A. E. Hassan, "The rise of AI teammates in software engineering (SE) 3.0: How autonomous coding agents are reshaping software engineering," *arXiv preprint arXiv:2507.15003*, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.15003

[2] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.

[4] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[5] P. Bourque, R. Dupuis, A. Abran, J. Moore, and L. Tripp, "The guide to the software engineering body of knowledge," *IEEE Software*, vol. 16, pp. 35–44, 1999. [Online]. Available: https://doi.org/10.1109/52.805471

[6] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering Methodology*, vol. 10, pp. 184–208, 2001. [Online]. Available: https://doi.org/10.1145/367008.367020

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

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chanez, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2107.03374

[9] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on GitHub Copilot," in *International Conference on Software Engineering*, 2023, pp. 2149–2160. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00181

[10] K. El Haji, C. Brandt, and A. Zaidman, "Using GitHub Copilot for test generation in Python: An empirical study," in *International Conference on Automation of Software Test*, 2024, pp. 45–55. [Online]. Available: https://doi.org/10.1145/3644032.3644443

[11] J. Altmayer Pizzorno and E. D. Berger, "CoverUp: Effective high coverage test generation for Python," *Proceedings of the ACM on Software Engineering*, vol. 2, pp. 2897–2919, 2025. [Online]. Available: https://doi.org/10.1145/3729398

[12] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using LLM," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 951–971, 2024. [Online]. Available: https://doi.org/10.1145/3643769

[13] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at Meta," in *Symposium on the Foundations of Software Engineering, industry track*, 2024, pp. 185–196. [Online]. Available: https://doi.org/10.1145/3663529.3663839

[14] N. Mündler, M. N. Müller, J. He, and M. Vechev, "SWT-bench: Testing and validating real-world bug-fixes with code agents," in *Advances in Neural Information Processing Systems*, 2024, pp. 81 857–81 887. [Online]. Available: https://doi.org/10.52202/079017-2601

[15] M. Watanabe, H. Li, Y. Kashiwa, B. Reid, H. Iida, and A. E. Hassan, "On the use of agentic coding: An empirical study of pull requests on GitHub," *ACM Transactions on Software Engineering Methodology*, 2026. [Online]. Available: https://doi.org/10.1145/3798166

[16] A. Sergeyuk, Y. Golubev, T. Bryksin, and I. Ahmed, "Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward," *Information and Software Technology*, vol. 178, p. 107610, 2025. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107610

[17] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, pp. 29–30, 1992. [Online]. Available: https://dl.acm.org/doi/10.1145/157709.157715

[18] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems*, 2015.

[19] A. Bhatia, F. Khomh, B. Adams, and A. Hassan, "An empirical study of self-admitted technical debt in machine learning software," *ACM Transactions on Software Engineering Methodology*, vol. 35, pp. 1–44, 2026. [Online]. Available: https://doi.org/10.1145/3785001

[20] F. Pecorelli, F. Palomba, A. De Lucia, and A. Bacchelli, "The relation of test-related factors to software quality: A case study on Apache systems," *Empirical Software Engineering*, vol. 26, pp. 1–42, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09891-y

[21] SageSELab, "Replication package," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2026, replication package: https://github.com/SageSELab/Agentic-Pull-Request-Test-Coverage/.

[22] A. K. Dipongkor, "SageSELab/Agentic-Pull-Request-Test-Coverage," Jul. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21419686

[23] M. Brunsfeld, "Tree-sitter: An incremental parsing system for programming tools," https://tree-sitter.github.io/tree-sitter/, accessed: 2026.

[24] EclEmma Team, "JaCoCo: Java code coverage library," https://www.jacoco.org/jacoco/, 2026.

[25] pytest-dev contributors, "pytest-cov: Coverage plugin for pytest," https://pypi.org/project/pytest-cov/, 2026.

[26] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *International Conference on Software Maintenance*, 2013, pp. 516–519. [Online]. Available: https://doi.org/10.1109/ICSM.2013.85

[27] X. Zhu, E. J. Whitehead Jr, C. Sadowski, and Q. Song, "An analysis of programming language statement frequency in C, C++, and Java source code," *Software: Practice and Experience*, vol. 45, no. 11, 2015. [Online]. Available: https://doi.org/10.1002/spe.2298

[28] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*, 1992, pp. 196–202. [Online]. Available: https://doi.org/10.1007/978-1-4612-4380-9_16