Change And Cover: Last-Mile, Pull Request-Based Regression Test Augmentation

Zitong Zhou*
UCLA
Los Angeles, USA
zitongzhou@cs.ucla.edu

Matteo Paltenghi*
University of Stuttgart
Stuttgart, Germany
mattepalte@live.it

Miryung Kim
UCLA
Los Angeles, USA
miryung@cs.ucla.edu

Michael Pradel
CISPA Helmholtz Center for Information Security
Stuttgart, Germany
michael@binaervarianz.de

Abstract

Software is in constant evolution, with developers frequently submitting pull requests (PRs) to introduce new features or fix bugs. Testing newly added or modified code in PRs is critical to maintaining software quality. Yet, even in projects with extensive test suites, some of the code modified in PRs may remain untested, leaving a "last-mile" regression test gap. Existing automated test generators mostly focus on improving overall code coverage, but do not specifically target the uncovered lines in PRs. This paper presents Change And Cover (ChaCo), a novel, LLM-based test augmentation technique that specifically addresses the last-mile regression test gap in PRs. Our approach is enabled by three key contributions: (i) Instead of focusing on overall code coverage, ChaCo considers a specific PR and the lines left uncovered after applying the PR, offering developers augmented tests for code just when it is on the developers' mind. (ii) We identify providing suitable test context as a crucial challenge for an LLM to generate useful tests, and present two techniques to extract relevant test content, such as existing test functions, fixtures, and data generators. (iii) To make augmented tests acceptable for developers, ChaCo carefully integrates them into the existing test suite, e.g., by matching the test's structure and style with the existing tests, and generates a summary of the test addition for developer review. We evaluate ChaCo on 145 PRs from three popular, complex, and well-tested open-source projects—SciPy, Qiskit, and Pandas. The approach successfully helps 30% of PRs achieve *full patch coverage*, at the affordable cost of $0.11 per PR, demonstrating its effectiveness and feasibility. A qualitative assessment of the generated tests shows that human reviewers find the tests to be worth adding (4.53/5.0), well integrated (4.20/5.0), and relevant to the PR (4.70/5.0). Ablation studies show test context is crucial for context-aware test generation, leading to 2× coverage. In a contribution study, we submitted 12 tests to these projects, of which 8 have already been merged, and two previously unknown bugs were discovered and fixed. We envision our approach to be integrated into CI workflows, automating the last mile of regression test augmentation.

CCS Concepts

• **Software and its engineering → Software defect analysis**.

Keywords

Pull requests, regression testing, large language models, software maintenance

ACM Reference Format:
Zitong Zhou, Matteo Paltenghi, Miryung Kim, and Michael Pradel. 2026. Change And Cover: Last-Mile, Pull Request-Based Regression Test Augmentation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787768

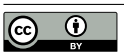
1 Introduction

Testing newly added or modified code in pull requests (PRs) is critical to maintaining software quality. For example, SciPy, a popular scientific computing library, claims that it "aims for a high coverage for all new code that is added" [29]. Tools like Codecov [30] that measure code coverage as projects evolve are used in Continuous Integration (CI) by projects, such as NodeJS, Keras, and scikit-learn [7, 19, 23]. We call the coverage of code changed in a PR *patch coverage*. While many developers already add tests to exercise changed code, we observe that often a few lines of uncovered code remain, which we refer to as the *last-mile regression test gap*.

There is an extensive body of prior work on test generation and fuzzing, yet few techniques sufficiently address the last-mile problem in PRs. One line of work focuses on improving the overall code coverage of a project [16, 17, 24, 26, 28, 36], but does not focus on PRs. Another line of work targets code changes, e.g., by fuzzing newly/frequently changed code [41, 43]. However, these approaches do not consider the context of a PR, such as the discussion between developers, the PR description, and links to related issues or PRs. Finally, Testora [25] tries to detect unintended changes introduced by a PR, but it does not aim to improve patch coverage.

Despite advances in automated test generation, we identify several key research gaps: (1) Prior work has not focused on generating regression tests specifically to improve patch coverage in PRs; (2) Existing approaches rarely leverage the rich context available in

*Both authors contributed equally to this work



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787768

PRs, such as links to related PRs and issues; (3) Test generation approaches often ignore the structure and utilities of the existing test suite, including fixtures, markers, mocks, and custom assertions.

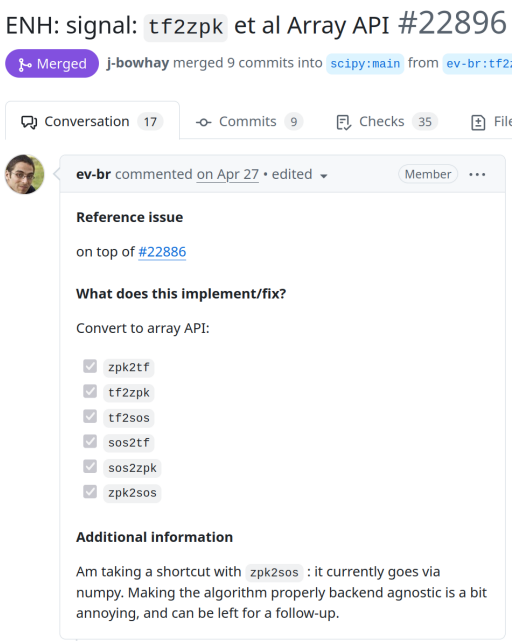
We address these gaps with Change And Cover (ChaCo), a novel, LLM-based test augmentation technique that specifically addresses the last-mile regression test gap in PRs. To address gaps (1) to (3), our approach (1) considers missing patch coverage of a specific PR, offering developers augmented tests for code just when it is on their mind; (2) provides suitable test context to the LLM, such as existing test functions, fixtures, and data generators; (3) carefully integrates tests into the existing test suite, e.g., by matching the test's structure and style with the existing tests. Furthermore, we evaluate ChaCo not only on automated metrics, such as coverage improvement, but also by manual review and real-world acceptability.

PRs are a natural playground for adding new tests because each PR represents a discrete unit of reviewable changes to a codebase. A PR typically introduces new functionality, bug fixes, or performs a refactoring, making it an ideal opportunity to ensure that the affected code is adequately tested before integration. By focusing on PRs, we can catch gaps in test coverage and prevent regressions early. Moreover, PRs typically provide rich contextual information, including change descriptions, related issues, and developer discussions, which can be leveraged to generate targeted and relevant tests.

Figure 1a shows a SciPy PR as a motivating example. The code changes in Figure 1b show the lines added by the PR, and the highlighted lines are the intersection between the changes, and those not covered by the test suite. ChaCo contributed a test to this PR, which exposed a bug in the implementation that was subsequently fixed. To further motivate the need for suitable *test context*, consider a requirement for new tests added to SciPy: If a feature under test does not support GPUs, its test functions must be marked with the test marker `@pytest.mark.skip_xp_backends(cpu_only=True)`. Without understanding this specific requirement, LLMs will likely fail to generate a test that developers would accept. As another example, consider an existing test function `test_foo` that already tests the function `foo`, to which the PR adds a new optional parameter. When prompted to produce a test for the new parameter, it is best for the LLM to reference `test_foo`, and generate a new test that matches the structure and style of `test_foo`.

Our evaluation applies ChaCo to generate tests for 145 recent real-world PRs from three open-source projects – SciPy, Qiskit, and Pandas. Our results show that ChaCo successfully helps 30% of PRs achieve *full patch coverage*, at an average cost of only $0.11 per PR. A qualitative assessment of the generated tests shows that human reviewers find the tests to be worth adding (4.53/5.0), well integrated (4.20/5.0), and relevant to the PR (4.70/5.0). We also conduct a contribution study that submitted 12 of ChaCo's generated tests to these projects, of which 8 have been merged and 4 are under review, demonstrating the practical utility of ChaCo in real-world development. ChaCo's added tests have already exposed two bugs in SciPy, both confirmed and fixed. Finally, our ablation study confirms that test context is crucial for generating high-quality tests: Compared to a variant of the approach without test context and without runtime feedback, ChaCo achieves a 2× and 5.6× higher total coverage increment, respectively.

In summary, our work makes the following contributions:



(a) PR with title, description, discussion, and links.

```
def zpk2tf(z, p, k):
    """
    Return polynomial transfer function ...
    """
+   xp = array_namespace(z, p)
+   z, p, k = map(xp.asarray, (z, p, k))
+   ...
+   if z.ndim > 1:
+       temp = _pu.poly(z[0], xp=xp)
+       b = xp.empty((z.shape...),...)
+       if k.shape[0] == 1:
            k = [k[0]] * z.shape[0]
        for i in range(z.shape[0]):
+           b[i] = k[i] * _pu.poly(z[i], xp=xp)
    else:
+       b = k * _pu.poly(z, xp=xp)
```

(b) PR diff with missing patch coverage highlighted.

Figure 1: Illustration of a PR's patch coverage gap. ChaCo runs the regression test suite on the PR to identify the missing patch coverage: missing branches in yellow, lines in red.

- **PR-specific test generation.** We introduce a novel approach that leverages PR context to generate regression tests specifically aimed at improving patch coverage.
- **Context-aware test generation.** We integrate dynamic analysis with LLM-based techniques to leverage test context, ensuring tests align with existing testing practices.
- **Test suite integration.** Our approach generates tests that are consistent with the structure and utilities of the existing test suite, increasing the likelihood of acceptance by maintainers.
- **Real-world evaluation.** We evaluate our approach by augmenting real-world PRs from large open-source software with additional tests and assessing their acceptance and feedback from developers, demonstrating the practical utility of our method.
- **Data availability.** ChaCo is available at https://github.com/UCLA-SEAL/Change-Cover.

2 Problem Statement

The overall goal of ChaCo is to augment a given PR with additional test cases that cover otherwise uncovered code that was newly introduced or modified code by the PR. ChaCo assumes the PR is bug-free and generates tests to catch future regressions. Exposing buggy code changes on the spot is an orthogonal problem that requires a different approach, such as Testora [25]. Isolating these two goals is deliberate: bug detection hinges on a change-intent oracle [25], which is not required for ChaCo.

ChaCo takes a PR as input, which consists of a title, a description, a sequence of discussion comments, and a diff. The diff consists of line-level additions, deletions, or modifications, as well as file-level additions or removals. Among all lines touched by the diff, we focus on the set $\mathcal{E}$ of *executable lines*, i.e., all non-comment, non-empty source code lines that are not in test files but in the main source code of the project. We partition $\mathcal{E}$ into two subsets: the set C of lines that are covered by the existing tests and the set $\mathcal{U}$ of lines that remain uncovered. Given these sets, we define the *patch coverage* of the PR as the fraction of modified lines covered by existing test: $pc = \frac{|C|}{|\mathcal{E}|}$. If a PR has a patch coverage of 100%, we call it *fully covered*. Based on the above definitions, the problem addressed by ChaCo consists of two subproblems, *PR-based test generation* and *test integration*, as detailed in the following.

2.1 Task 1: PR-Based Test Generation

The first subtask is to increase the patch coverage of PRs that are not yet fully covered:

Definition 2.1 (PR-based test generation task). Given a PR with patch coverage $pc < 100\%$, generate a test case t such that:

- t covers at least one line in $\mathcal{U}$.
- t is relevant to the changes introduced in the PR.

Meaningful tests should reuse project-specific or module-specific test utilities, including but not limited to test fixtures, markers, mocks, data generators, and custom assertions. For example, in SciPy, many APIs support input arrays of different backends, such as NumPy and PyTorch. To write backend-agnostic tests for these APIs, the project provides a fixture `xp`, and tests are expected to use `xp` to create arrays of different backends, instead of writing a test for each backend.

2.2 Task 2: Test Integration

The second subtask is to integrate the generated test case into the existing test suite of the project. To this end, the approach should identify the appropriate location within the existing test suite where the test logically belongs, ensuring that it adheres to the conventions and utilities used in the existing tests:

Definition 2.2 (Test integration task). Given a generated test case t, identify the appropriate placement (f_t, c_t, m_t) of t within the existing test suite, where

- f_t is the target test file in the test suite.
- $c_t :: m_t$ are the target test class (or $\emptyset$ if no such class exists) and the target test method/function.

Besides identifying the correct placement location, the integration step should reuse existing test utilities. Generating a test case that increases patch coverage is insufficient to ensure that the test is actually integrated into the codebase. Indeed, a survey of over 700 core project maintainers responsible for accepting contributions [13] reports that the most important factors influencing PR acceptance include *code style* and *technical fit*. For example, this includes "whether it adheres to the project conventions", ensure "keeping with the spirit of the project's other APIs", and that "its newly introduced code follows the total and functional style of the rest of the codebase" [13]. Consequently, beyond merely increasing patch coverage, a generated test must also be *well-integrated*, i.e., consistent with the style and conventions of the existing test suite and designed to reuse test utilities.

3 Approach

Figure 2 illustrates our approach. Given a PR, ChaCo produces a test case that increases patch coverage and integrates it into the existing test suite. The approach consists of three stages: (1) codebase analysis, (2) test generation, and (3) test integration.

The analysis stage collects information from three sources. First, patch coverage analysis computes the patch coverage of the PR diff, identifying uncovered lines that require additional testing (Section 3.1.1). If the PR's code changes are fully covered, ChaCo terminates as there is no contribution to make. Second, PR context analysis extracts information from the PR description and its linked resources, such as related issues and documentation (Section 3.1.2). By analyzing this PR context, generated tests should align with the PR's intent. Third, test context analysis identifies existing tests to serve as examples and the context where the newly generated tests should be inserted into (Section 3.1.3).

The test generation stage (Section 3.2) generates new test cases targeting uncovered lines. Each candidate test is executed against the post-PR code, and ChaCo verifies whether it improves patch coverage. If the test fails or does not enhance coverage, ChaCo iteratively refines the test using execution error messages as feedback, up to a predefined number of attempts. For successful tests, ChaCo prompts the LLM to maximize coverage improvement.

Finally, the test integration and report stage (Section 3.3) incorporates successful (i.e., passing and coverage-improving) test cases into the existing test files. It reuses the most relevant location to insert the test into, as identified by test context analysis. ChaCo employs an LLM to decide whether to add a new test method or extend an existing one, ensuring seamless integration into the project's testing conventions. To allow developers to quickly assess the newly added test cases, ChaCo also generates a report (Section 3.3) summarizing the added test cases, their purpose, and their impact on coverage. If multiple tests add the same coverage, ChaCo additionally prompts an LLM to select the best one for reporting, using three criteria: test worthiness, integration quality, and PR relevance.

From a developer's perspective, ChaCo automatically analyzes PRs, generates relevant test cases, and produces a summary if it finds a test that increases coverage. We envision the approach to be used on open PRs, e.g., implemented as a GitHub action that runs on new PRs, providing a low-friction way to improve the test suite.

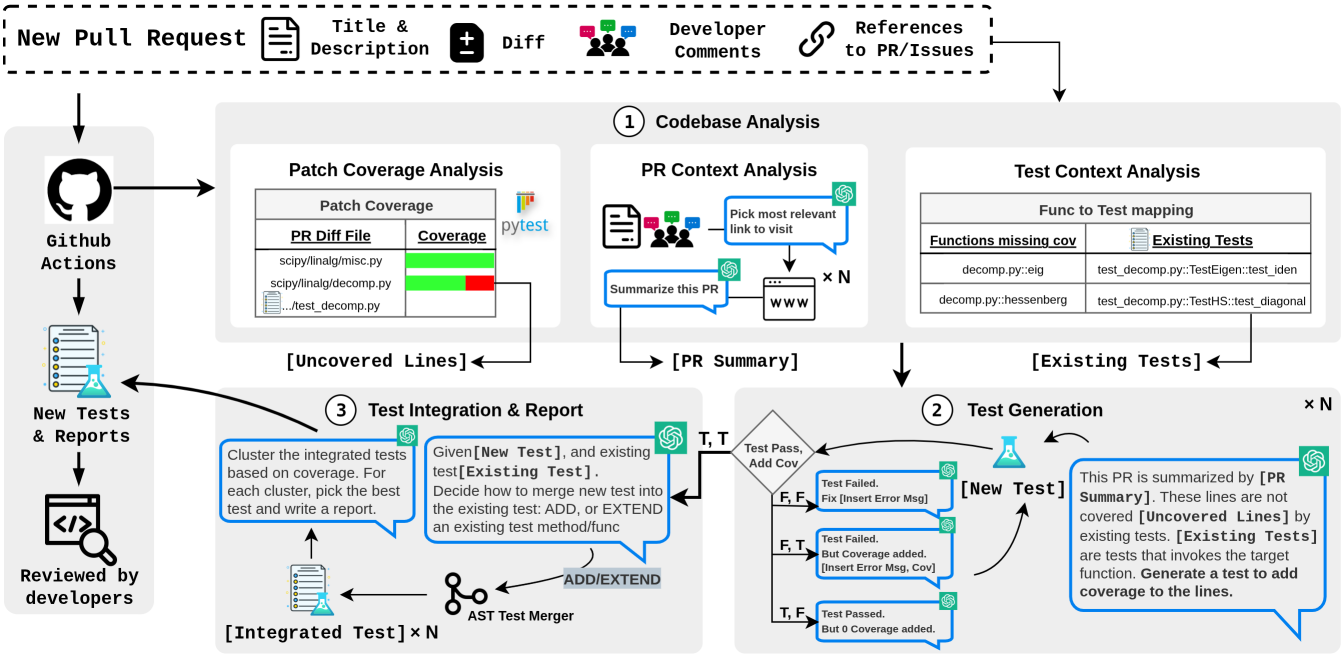


Figure 2: Flowchart of ChaCo showing the main processing stages from PR input to integrated test case output.

3.1 Codebase Analysis

The codebase analysis stage extracts contextual information that is useful for generating relevant test cases that are suitable for the existing codebase. The approach performs three analyses, which are conceptually independent, as shown in Figure 2.

3.1.1 Patch Coverage Analysis. To understand the coverage gap of the PR, ChaCo computes the patch coverage. This analysis identifies uncovered lines within the PR diff by running the entire regression test suite. If the patch coverage is 100%, the process terminates, deeming the PR sufficiently tested.

To convey the missing coverage to the LLM, the approach annotates each uncovered line using the comment `# UNCOVERED!`, ensuring the LLM focuses on these areas during subsequent stages. The summary spans multiple files, concatenated for clarity. An example is provided in Figure 3, which highlights uncovered branches in the `status` method of `PrimitiveJob`. ChaCo segments the lines with missing patch coverage into *focal functions*. It only considers lines that are inside functions or methods. This is because we observed that uncovered top-level lines often do not require dedicated tests, such as `if TYPE_CHECKING: from scipy._lib import utils`.

We also acknowledge that not all uncovered lines necessarily merit additional tests. Prior work [4, 32] shows that sometimes coverage gaps are considered acceptable by developers when the code is unlikely to be buggy, is legacy and no longer expected to change, or is exercised by regular fuzz-testing runs.

3.1.2 Pull Request Context Analysis. This analysis extracts contextual information from the PR description, developer discussions, and automated infrastructure messages, summarizing it into a format suitable for LLM input during the test generation stage. For

```
# qiskit/primitives/primitive_job.py
class PrimitiveJob(BasePrimitiveJob[ResultT, JobStatus]):
...
  def __init__(self, function, *args, **kwargs):
...
  def status(self) -> JobStatus:
    if self._status is None:
      self._check_submitted()
      if self._future.running():
        return JobStatus.RUNNING  # UNCOVERED!
      elif self._future.cancelled():
        self._status = JobStatus.CANCELLED  # UNCOVERED!
      elif self._future.done() and self._future.exception() is
           None:
        self._status = JobStatus.DONE
      else:
        self._status = JobStatus.ERROR  # UNCOVERED!
    return self._status
```

Figure 3: Code where lines that miss test coverage are annotated with `# UNCOVERED`.

example, a PR *A* may address a bug reported in issue *B*, where issue *B* attributes the bug to changes introduced in PR *C*. This creates a network of interconnected PRs or issues. De Souza et al. [9] highlight the importance of the "Is contextualized by" link type, observed in 20.5% of cases, which indicates that developers reference other PRs or issues to obtain configuration details, understand limitations, or find relevant examples.

Specifically, ChaCo retrieves the PR page's HTML content, converts it to markdown, and extracts embedded links. The PR page includes developer comments, code reviews, and CI messages. By consolidating this information, ChaCo provides rich context to inform subsequent stages, enabling targeted test generation. ChaCo enriches the initial summary by visiting selected links. ChaCo employs an LLM-based link selection mechanism that prioritizes official documentation, community forums, and technical guides.

ChaCo applies heuristic filtering to exclude irrelevant links, such as GitHub navigation pages, providing the LLM with a curated list of meaningful links. Each iteration selects an outgoing link from the PR (we set a maximum of three links), retrieves its content, and updates the summary.

Summarization and updates are performed using a structured LLM prompt. The prompt includes the HTML content of the PR, the current summary, the selected link content (if any), asking the LLM to generate or update the PR summary. ChaCo tracks provenance throughout the enrichment process, recording visited URLs and LLM calls for transparency and reproducibility. This iterative approach enables ChaCo to gather comprehensive information about the PR, enhancing the quality of the generated test cases. LLM prompts to select links and to summarize the content are available in the supplementary material.

3.1.3 Test Context Analysis. ChaCo's tests should be consistent with the style of existing tests. Specifically:

(1) *Test placement*: The new test should be placed in the correct test file, class, and be named appropriately.
(2) *Test utilities*: If suitable, the new test should reuse existing test utilities, such as fixtures, markers, and helper functions.
(3) *Test style*: The new test should follow the coding style of neighboring tests, such as use of type hints, trailing commas, and use of single vs. double quotes.

To satisfy these requirements, it is crucial to learn from existing tests relevant to a focal function with missing coverage. We refer to these existing tests as the *test context* of a focal function.

We present a motivating example in Figure 4. Suppose that a PR adds a new function for matrix decomposition as `scipy.linalg.mat_decomp`. Further suppose there are tests for matrix operations in the file `scipy/linalg/tests/test_matrix.py`. Figure 4a shows an example of the test context of the focal function. Test utilities include imports, top-level variable `dtypes`, the test class `TestMatrixOperations`, test fixture `setup_method`, as well as test methods `test_matrix_det` and `test_matrix_rank`. Test style includes the naming convention: `test_matrix_{func_name}`.

Figure 4b shows a test generated using the test context, while Figure 4c shows one generated without it. A developer reviewing the test in Figure 4c would immediately spot several issues requiring manual fixes before the test can be merged. First, the test needs to be moved into the correct test class. Second, it fails to reuse existing test utilities, such as the fixture that sets the random seed. It also hardcodes test inputs instead of using parametrization for matrix shapes and data types, limiting its scope. Third, the assertion is a generic equality assertion instead of the preferred NumPy-specific assertion for matrix equality. Lastly, the test fails to check for a warning on empty matrices. In contrast, the test in Figure 4b is well-placed, reuses test utilities and conforms to style, and can be accepted with minimal or no changes.

To extract test context of a focal function from the existing test suite, we employ a mixture of lightweight static analysis, dynamic analysis, and LLM-prompting. Algorithm 1 shows the steps taken for extracting test context analysis, as explained in the following.

Static Test Context. If the PR modifies test files, then ChaCo considers them as the candidate test files (*test_files'*) that may contain the test context (line 1-3). Alternatively, if the PR does not modify any test file, we rank all test files in the project by the lexical similarity of their path (e.g., `scipy/linalg/tests/test_matrix.py`) and paths of the source files modified by the PR (e.g., `scipy/linalg/matrix_decomp.py`) (line 4-5). We find this simple technique to be very effective, because large projects usually have well organized test suites. After obtaining a shortlist of candidate test files, *test_files'*, we give the LLM the PR and a summary of each file, and let it confirm and pick the relevant test files (line 6).

Dynamic Test Context (line 8-12). The next step aims to identify relevant existing tests via dynamic analysis. Once the test files (*test_files*) are identified, we run all of their tests with a profiler, constructing a dynamic call graph (line 8). If the dynamic call graph contains call chains where a test method invokes a focal function, we mark the test as a caller of the focal function (line 9-12).

LLM Test Context (line 13-17). When no test covers the focal function, dynamic call graph analysis cannot find its caller, and ChaCo *falls back to LLM-prompting* (line 13-16). We provide the LLM with the PR diff, a test file summary, and the focal function, asking it for the test class and the test method that is the most relevant to the focal function.

After the relevant test method is identified, we extract the top-level and class-level statements around the test, including any imports, fixtures, and other test utilities used by the caller test. Collectively they constitute the test context of a focal method (lines 11 and 16). The test context is saved for use by the test generator.

On a high level, test context analysis serves a dual purpose. First, it informs the test generation stage by providing insights into existing tests, enabling the generation of new tests that mirror the style and taking advantage of proper test utilities. Second, it aids the test integration stage by identifying the most relevant test files and test classes for incorporating the generated tests seamlessly (Section 3.3). To optimize efficiency, ChaCo maintains a cache of test context information, allowing it to reuse previously computed results and avoid redundant analysis in subsequent runs.

3.2 Test Generation

Once the codebase analysis is complete, we generate test cases targeting the uncovered lines identified during patch coverage analysis. The test generation process is iterative and adaptive, leveraging the outputs of the codebase analysis stage: uncovered lines, PR context, and test context. These inputs are combined into structured prompts for the LLM. We detail the test generation step below.

The test generation conversation begins with a prompt that includes the PR's diff, the PR context, and the focal function annotated with uncovered lines (as shown in Figure 3). The prompt instructs the LLM to "Inspect and summarize the lines modified by the PR that are uncovered by existing regression test suites." The LLM returns with a natural-language summary of why the lines are uncovered. Then, ChaCo uses a prompt that includes the PR context, the summary of uncovered lines, and one test context (pick $t \sim TC[foc]$, as provided by Algorithm 1). The prompt instructs the LLM to "generate test cases for the PR's changes" and to return an executable `pytest` test (Figure 4b).

```
"""scipy/linalg/tests/test_matrix.py"""
import pytest as pt
import numpy as np
from numpy.testing import (
    assert_matrix_almost_equal,
    suppress_warnings)

dtypes = [np.float32, np.complex64...]

class TestMatrixOperations:
    ...
    def setup_method(self):
        # Ensure reproducibility
        self.rng = np.random.
              default_rng(42)

    @pt.mark.parametrize...
    def test_matrix_det(self, shape,
          dtype):
        matrix = self.rng.random(shape)
              .astype(dtype)
        ...
```

(a) Test context for `mat_decomp`.

```
import ...
from scipy.linalg import mat_decomp
dtypes = ...

class TestMatrixOperations:
   def setup_method(self):
       ...
   @pt.mark.parametrize("shape",[(5,5),(10,5),(0,0)])
   @pt.mark.parametrize("dtype", dtypes)
   def test_matrix_decompose(self, shape, dtype):
       matrix = self.rng.random(shape).astype(dtype)

       if matrix.size == 0:
           with suppress_warnings() as sup:
               sup.filter(UserWarning,
                   "Empty_Matrix")
               U, V = mat_decomp(matrix)
           assert U.size == 0 and V.size == 0
           return

       U, V = matrix_decompose(matrix)
       assert_matrix_almost_equal(U @ V, matrix)
```

(b) Style-conforming test of `mat_decomp` generated with test context. Grey highlights test context's contribution.

```
import...
from scipy.linalg import mat_decomp

def test_matrix_decompose():
   # Matrix Shape & Type are
         hardcoded
   matrix = np.random.randn(5, 5)

   # Overlook edge case: empty matrix
         should emit warning
   U, V = mat_decomp(matrix)

   # Generic assertion
   assert np.allclose(U @ V, matrix)
```

(c) Style non-conforming test generated without test context.

Figure 4: Comparison of tests generated for `mat_decomp` with and without providing the LLM with test context.

Algorithm 1 Extract Test Context

Require: $\delta, \mathcal{U}, T$ ▷ PR code changes, Uncovered lines, All test-file paths
Ensure: TC ▷ Mapping from focal functions to test context

1: **for** $f \in \delta$ **do** ▷ ***Static Test Context***
2: **if** f is a test file **then**
3: $test\_files'$.append(f)
4: **if** $test\_files' = \emptyset$ **then** ▷ no direct test files found
5: $test\_files' \leftarrow$ **Jaccard**(T, δ)[1:K]
6: $test\_files \leftarrow$ **LLM**(PickTestFiles, $test\_files'$, δ)

7: $TC \leftarrow \emptyset$
8: $trace \leftarrow$ Profiler($test\_files$)
9: **for** $foc \in$ **ParseFunc**($\mathcal{U}$) **do** ▷ ***Dynamic Test Context***
10: $callers \leftarrow$ **AncestorsOf**($trace$, foc)
11: $TC_{foc} \leftarrow$ [**Extract**(c) | $c \in callers \wedge c$ is test function]
12: $TC[foc] \leftarrow TC_{foc}$
13: **for** $foc \in$ **ParseFunc**($\mathcal{U}$) **do** ▷ ***LLM Test Context (fallback)***
14: **if** $TC[foc] = \emptyset$ **then** ▷ no test invoked the focal function
15: $t \leftarrow$ **LLM**(PickTestFunction, $test\_files$, foc, δ)
16: $TC[foc] \leftarrow$ **Extract**(t, $test\_files$)
17: **return** TC

In our preliminary experiments, we discovered that it is extremely rare for LLMs to generate a correct and useful test case on first trial. It is intuitive since human developers also need "trial and error." In light of this and program repair approaches [2, 38, 42], ChaCo employs a refinement technique to iteratively fix and improve generated tests. First, ChaCo executes the test on the post-PR project to check whether it 1) passes and 2) covers previously uncovered lines. Refinement continues depending on the four different outcomes. If the test passes and adds new coverage, then the generated test is good and ChaCo moves on to the test integration stage. If the test failed and added no coverage, ChaCo prompts the LLM with the error message to fix the test. If the test failed but added coverage (which, based on our observations during the evaluation, often indicates that the test is correct except the oracle) ChaCo prompts the LLM with the error message, the code with both added and missing coverage annotated with special comments, and ask it to fix the runtime error while preserving the coverage. If the test passed but does not add coverage, ChaCo prompts the LLM to change the test to increase coverage. Three custom prompts contextualize the generated test with its runtime outcome, enabling the LLM to make informed improvements on the test.

```
"""scipy/linalg/tests/test_matrix.py"""
import pytest as pt
import numpy as np
from numpy.testing import (
    assert_matrix_almost_equal,
    suppress_warnings,
)
+from scipy.linalg import mat_decomp

dtype_list = [np.float32, np.complex64, ...]


class TestMatrixOperations:

    def setup_method(self):
        ...

    def test_matrix_determinant(self):
        ...

    def test_matrix_rank(self):
        ...

+   @pt.mark.parametrize("shape",[(5, 5),(10, 5),(0, 0)])
+   @pt.mark.parametrize("dtype", dtype_list)
+   def test_matrix_decompose(self, shape, dtype):
+       matrix = self.rng.random(shape).astype(dtype)
+
+       if matrix.size == 0:
+           ...
+
+       U, V = mat_decomp(matrix)
+       assert...
```

Figure 5: Integration of generated standalone test (shown in Figure 4b) into `scipy/linalg/tests/test_matrix.py`

3.3 Test Integration and Report

ChaCo's generated tests are standalone tests that need to be integrated into the test suite properly. With the help of test context, we

can directly integrate a standalone test into its test context. The test context provides the existing test file and test class into which our generated test should be incorporated. At first, for each test and its context, ChaCo prompts an LLM to decide on the integration mode: whether to add the test as a new test method/function or to extend the body of an existing test. Then, the generated test, existing test file, and integration mode are passed to an abstract syntax tree (AST) transformer that performs the merge. The transformer checks if the generated test includes new imports, top-level variable definitions, test fixture definitions, and merges them into the existing test file accordingly. Figure 5 shows how the generated test in Figure 4b is integrated into its test context. The test method `test_matrix_decompose` and the import of the focal method are inserted. Other imports and the helper method `setup_method` are skipped because they are already present in the existing test file.

ChaCo generates a report of its proposed test additions. For each PR, after generating N tests, ChaCo filters the successful (i.e., passing and coverage-improving) tests. In case of multiple successful tests, ChaCo clusters them by the coverage addition. Each cluster contains tests that add the same set of line coverage; tests that add a strict subset of any cluster are dropped.

While tests of a cluster provide identical coverage, they may still vary in quality. ChaCo automates test selection. We prompt the LLM with the PR context, PR diff, PR's patch coverage, the tests in the cluster, and instruct it to select the highest-quality test. The quality criteria include: 1) *worthiness*: how likely the test will catch regressions; 2) *integration*: how seamlessly the test integrates with existing tests, considering style and use of test utilities; and 3) *relevance*: how closely the test aligns with the PR's intention. We give hand-written positive and negative examples in the prompt to illustrate these criteria. Each selected test is then summarized using an LLM, which generates a concise report detailing the test, its purpose, and impact on coverage. At the end, ChaCo attaches the report to the open PR for developer review.

4 Implementation

ChaCo is implemented in Python and leverages containerization to ensure a consistent and isolated testing environment. For each project, a dedicated Docker container is created to execute the test suite as specified by documentation and continuous integration (CI) pipeline. This setup essentially simulates a CI pipeline locally.

To retrieve dynamic call chains and extract dynamic test contexts, we utilize viztracer [34] as the dynamic profiler. Additionally, ChaCo employs DsPy [15] to structure prompts. ChaCo's parameters include the choice of LLM, model-related parameters (e.g., temperature), # test cases to generate per PR, and # maximum feedback iterations—all exposed via a configuration file. ChaCo passes the model-related parameters to DsPy, which abstracts and handles the communication with the LLM. In total, ChaCo uses eleven structured prompts to achieve its objectives. Figure 2 illustrates eight of these prompts for simplicity. The other three prompts not shown are used for 1) picking the most relevant test file, 2) picking relevant test class and test method, and 3) summarizing the uncovered lines. We refer readers to ChaCo's repository for the complete set of prompts.

Table 1: Projects and PRs used in evaluation.

Project	Pull requests					
	Considered	Merged	With code changes	Keyword & scope	≤5 files	**Incomplete patch cov.**
SciPy	2,000	1,822	851	772	572	**121**
Qiskit	2,000	1,842	919	712	481	**135**
Pandas	2,000	1,763	813	372	273	**45**

5 Evaluation

We address the following research questions:

(1) **RQ1: Effectiveness**: How effective is ChaCo at producing tests that *pass* and that *add patch coverage*?
(2) **RQ2: Acceptability**: To what extent are the tests generated by ChaCo acceptable by developers in terms of their added value for detecting future bugs, their integration into the existing test suite, and their relevance to the PR?
(3) **RQ3: Ablation Study**:
   (a) **Test context component**: How do the two kinds of test context (dynamic and LLM-based) contribute to ChaCo's effectiveness?
   (b) **Feedback component**: What is the impact of ChaCo's use of runtime feedback on test pass rate and coverage?

Project Selection. We evaluate our approach on three open-source projects: SciPy, Qiskit, and Pandas. We selected them as they are: 1) complex and production-quality, necessitating thorough testing, 2) highly active, with many PRs and contributors, 3) already having a comprehensive test suite, which matches our "last-mile" approach of adding new tests to PRs. SciPy and Pandas are popular libraries for scientific computing and data science. Qiskit is the most popular compiler framework for quantum computing, which helps us understand ChaCo's effectiveness on newer domains. Compared to prior work on automated test generation for Python, which are evaluated on smaller projects [11, 24, 39], our selection is more challenging as it evaluates the augmentation of already strong test suites of complex codebases.

PR Selection. We systematically filter PRs to identify suitable ones for evaluation. We begin with the most recent 2,000 PRs in June 2025, and apply the following filters: 1) the PR is merged, 2) it contains code changes other than deletions and documentation changes, 3) the PR title must not contain specific keywords, such as "DOC" or "backport", and the PR must not modify files outside our coverage tracking scope (e.g., we ignore Pandas PRs that modify files in the `pandas/io` directory as they are responsible for (de)serialization to various formats), and 4) the PR modifies no more than five code files, which helps us filter out large refactorings. Finally, we run the regression test suite on the PR's branch, and retain the PRs whose patch coverage is not 100%. The final selection of PRs is shown in Table 1.

To conduct experiments and balance the projects, we randomly sample 50 PRs of Qiskit and Scipy from Table 1, plus all 45 PRs of Pandas, resulting in a total of 145 PRs to evaluate RQ1. The qualitative analysis and test submission (RQ2) and the ablation

Table 2: Test pass rate and cost.

Project	Tests Generated	Passed	**Pass Rate**	**Cost ($)**
SciPy	832	274	**32.9%**	**0.17**
Qiskit	689	170	**24.7%**	**0.08**
Pandas	737	161	**21.8%**	**0.08**

Table 3: Coverage increment. PRs is the total number of PRs with non-full patch coverage used in this experiment. Cov Added is the number of PRs that ChaCo successfully added coverage to. 100% Cov is the number of PRs that reached full patch coverage after the test addition. #lines ↑ is the sum of line coverage added by ChaCo on all PRs. Ratio is the average ratio of coverage added / coverage missing across PRs.

Project	PRs	Cov Added	**100% Cov**	#lines ↑	Ratio
SciPy	50	27	**15**	112	40.0%
Qiskit	50	21	**13**	57	34.0%
Pandas	45	17	**14**	20	33.5%

study (RQ3) are done on a smaller benchmark of 30 PRs, where we downsample to 10 PRs per project.

Model and Parameters. For our evaluation, we use GPT-4o-mini [21] for its cost-effectiveness in large-scale test generation. Using GPT-4o-mini also minimizes data contamination, as its training data predates all the 145 PRs used in our evaluation. We configure the model with a temperature of 0.7, which we empirically found to balance test diversity and quality. ChaCo exposes the choice of LLM and parameters via a configuration file, allowing easy parameter tuning and experimentation with different LLMs.

Metrics. These are the metrics used for evaluation.

(1) **Test pass rate**: The ratio of tests that pass, divided by the total number of tests generated.
(2) **Coverage increment**: The number of newly covered lines that generated tests add to the project.
(3) **Test review scores**: In RQ2.1, we manually review tests and score them on worthiness, integration, and relevance.
(4) **Test submission outcomes**: In RQ2.2, we track the number of tests submitted as PRs and their outcomes.
(5) **Cost**: The monetary cost (USD) of using the LLM.

5.1 RQ1: Effectiveness

We apply ChaCo on the 145 PRs and measure three key metrics: test pass rate, coverage increment, and cost. Table 2 shows the test pass rate of the tests generated by ChaCo on the 145 benchmark PRs. Table 3 shows the coverage increment. ChaCo fully covers 43 PRs with 100% patch coverage. Importantly, each line of coverage added by ChaCo *represents a readable, regression-quality test*, and should not be compared with coverage obtained from approaches, such as fuzzing or search-based software testing (SBST). We demonstrate in RQ2 that these tests are acceptable by project maintainers with high success rate. Studying the costs of running ChaCo, we find that augmenting one PR costs between $0.08 and $0.17, depending on the project, with an overall average of $0.11 per PR. We expect this cost to decrease as LLMs become cheaper in the future.

ChaCo reliably generates passing tests that add high-quality regression test coverage (189 lines) to complex, real-world projects, showing the benefit of integrating LLM-based test generation into CI workflows. A substantial portion (30%) of PRs reach full patch coverage, highlighting the value of targeted, PR-context-aware test generation. The cost per PR remains affordable ($0.11 per PR), supporting the feasibility of large-scale adoption. Overall, these findings confirm the effectiveness of ChaCo in bridging coverage gaps.

5.2 RQ2: Acceptability

We envision ChaCo to be incorporated into Continuous Integration (CI) pipelines, where it can automatically monitor and fix missing patch coverage in new PRs. We ran the full ChaCo pipeline to generate tests for each PR with missing test coverage. Then we performed two complementary experiments: (1) qualitative analysis, where reviewers score each generated test on worthiness, integration, and relevance; and (2) test submission, where tests are submitted to the upstream projects and developer feedback is analyzed.

Unlike RQ1, which focuses on quantitative metrics, RQ2 delivers a qualitative and fine-grained assessment of ChaCo. We also present a case study to illustrate how ChaCo helps increase test coverage, exposes issues, and motivates feature changes.

5.2.1 RQ2.1: Qualitative Analysis. We evaluate 10 PRs per project, selected from the 50 PRs used in RQ1. Two reviewers independently assess each test with three criteria: (1) *Worthiness*: the test's potential to detect regressions or bugs (e.g., tests for trivial getters are less valuable); (2) *Integration*: the test's conformity to the structure of the existing test suite, including its placement in files/classes and adherence to project-specific conventions, such as test utilities; and (3) *Relevance*: the test's alignment with the intent of the original PR. Each criterion is rated on a scale from 1 (poor) to 5 (good). Developer feedback (Section 5.2.2) confirmed that these three criteria are instrumental for accepting tests.

The two reviewers both have eight years of experience in software engineering. Reviewer A is a contributor to quantum computing platforms, including Qiskit, PennyLane, BQSKit and PyTket. Reviewer B has contributed tests to SciPy, Pandas, and is also experienced in software testing. To standardize the review process, we provide a step-by-step guideline that defines the task, the three scoring criteria (with examples). For example, the guideline instructs the reviewers to first review the PR itself and understand why patch coverage is incomplete; then the reviewers examine the test report generated by ChaCo, which includes a summary, coverage visualization, runtime log, test patch, and the complete test file.

We acknowledge that this qualitative assessment is inherently subjective. For instance, determining the "most appropriate" test utilities for the integration score depends on deep, project-specific knowledge that even experienced developers may not possess for all submodules. Therefore, the reviewers provided scores on a best-effort basis, reflecting their expert but not infallible judgment.

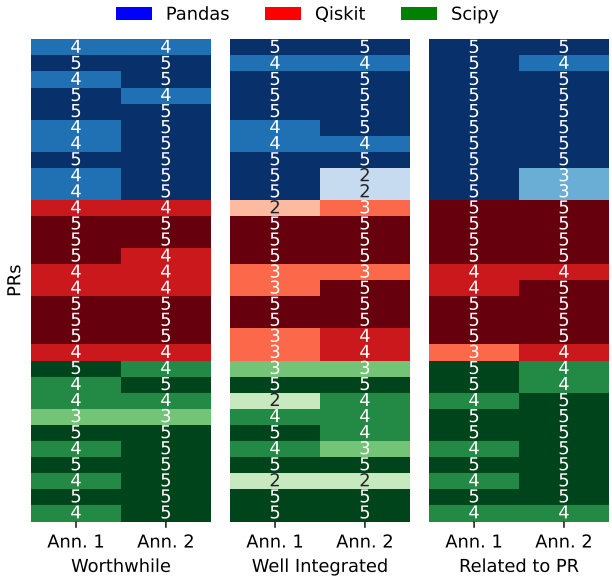


Figure 6: Heatmap of annotation scores provided by reviewers for each PR.

Figure 6 presents a heatmap summarizing the annotation scores assigned by reviewers for each PR. The color coding indicates the project, while the degree of shading represents the score (darker is better). We report a satisfactory level of agreement between reviewers accounting for 60.00% for worthiness, 66.67% for integration, and 66.67% for relevance. The average scores across all PRs are 4.53 for worthiness, 4.20 for integration, and 4.70 for relevance.

The worthiness score of 4.53 indicates that the tests were generally effective in addressing coverage gaps and preventing regression. Reviewers noted their utility in covering untested paths, branches, or exceptional cases, with comments like "nice to cover an extra branch." However, feedback also highlighted areas for improvement, such as weak or absent assertions, with critiques like "no oracle" or "`assertIsNotNone` is weak," suggesting that while some tests added coverage, their bug detection abilities could be further improved.

Integration scored 4.20, suggesting that tests generally adhered to the suite's structure and conventions, which supports maintainability. Comments like "Good location" or "right class" confirmed test placement. A common suggestion is to use better test parameterization. Inconsistent use of frameworks, such as `unittest` over `pytest`, was also noted.

Relevance scored highest at 4.70, indicating strong alignment with PR intent. This confirms our design decision to use PR context, such as the PR description and discussion, to guide test generation.

5.2.2 RQ2.2: Test Submission as Pull Requests. Our north-star goal is to submit these tests as PRs fully automatically. However, submitting a large volume of PRs with generated tests is impractical. Therefore, we submitted a subset of tests to obtain feedback from developers. Each PR submission includes the following. An example submission is shown in Figure 7.

- A manually written title and description.
- A reference to the original PR with missing coverage.
- A visualization of coverage addition.

Figure 7: Example Test Submission (PR 23265) to SciPy. This test uncovers a bug in the previously-untested branch of `signal.zpk2tf` when input array `z` is multi-dimensional. The test was accepted and merged after fixing the CI failures.

- A note on AI usage and human review before submission.

We submitted a total of 12 ChaCo's tests (9 for SciPy, 1 for Pandas, 2 for Qiskit) in a best-effort manner. All 12 tests received some developer feedback, 8 have been accepted and merged, 2 are open, 2 have been rejected.

Developers appreciated the increased test coverage. Interestingly, our submissions prompted discussion about AI usage for test generation; most developers were supportive, though some raised concerns about maintainability. We addressed these by highlighting our manual review before submission. Most tests did not raise concerns from developers regarding their worthiness (i.e., potential to catch future bugs) or relevance to the original PR. Two tests were rejected due to concern of worthiness. Developers commented that "this particular test is not very strong," and "not sure it's super valuable." Notably, 3/12 tests received developer suggestion to *use more appropriate test utilities and follow specific conventions*, the largest challenge, despite ChaCo's use of test context analysis. This is consistent with RQ2.1 (Section 5.2.1), where integration scored the lowest. Specifically, the three suggestions are 1) adding a custom test marker to skip on certain backends (Section 5.2.3), 2) switching

to an assertion customized for checking array equality, and 3) using a custom context manager for setting options.

The selection of test utilities is highly nuanced and context-dependent. Taking custom assertions as an example, there are too many in projects such as SciPy and Pandas. Choosing the appropriate one depends on several factors: the functionality under test (e.g., exact versus approximate matching), the types of arguments involved (e.g., integer vs. floating-point), whether or not FP exceptional values should be handled (e.g., `[NaN] == [NaN]`?), whether or not comparisons should be type-flexible (e.g., `[1] == [1.0]`?). This complexity means that LLMs often need to reason and adapt test utilities from the test context, rather than simply replicating.

The test utilities are distributed across multiple namespaces and locations, making retrieval challenging. For example, when adding a new test method, such as `scipy/module/test_something.py::TestSomething::test_method_new`, relevant test utilities may be defined in various places: global libraries like `pytest` and `numpy.testing`, project-specific modules such as `scipy._lib`, configuration files like `scipy/conftest.py`, submodule-level utilities (e.g., `scipy.signal`), as well as file-level and class-level utilities. This dispersion means that gathering all possible test utilities for consideration is non-trivial, and the correct utility may be missed or overlooked by automated approaches.

ChaCo's tests are generally well-received by developers (8/12 already merged). The tests score well on worthiness and relevance, though integration remains a challenge. Developer discussions highlight both enthusiasm for AI-assisted testing and concerns about maintainability, which we address through transparency and collaboration. Overall, these findings demonstrate the acceptability of ChaCo's tests in real-world development workflows.

5.2.3 Case Study of Test Submission. We present a case study to show how ChaCo can help increase project coverage, uncover issues in the codebase, and motivate feature changes. Figure 7 is a screenshot of the test submitted to SciPy. This test ensures the focal function `signal.zpk2tf` is correct when input array `z` is multi-dimensional. The entire branch `if z.ndim > 1` is untested, in which the original PR modified four lines.

The submitted test case passed locally (because we only submit passing, coverage-adding tests), but failed on two checks in Scipy's CI. This can happen because ChaCo's existing implementation is limited to a docker-based build of the project, while SciPy's CI builds the project across multiple library dependencies, operating systems, and architectures. The test fails 1) on GPUs, and 2) when array backend is JAX [33] or `array-api-strict` [20]. The test failed on GPUs as expected because the focal function `signal.zpk2tf` does not support running on GPUs as of 1.16. To solve this, as instructed by the developers, we manually added a marker `@skip_xp_backends(cpu_only=True)` to the test.

The other CI failure uncovers a bug in `signal.zpk2tf`. It is expected to work on backends JAX and `array-api-strict`. However, ChaCo's test exposed that the previously-untested branch fails on these two backends. This led the developers to fix the implementation of `signal.zpk2tf`. In the discussion, one developer proposed a patch that fixes the bug on `array-api-strict` backend. After the CI failures were resolved by adding test markers, the test PR was approved and merged into SciPy as part of its 1.17.0 milestone. The bug-fix patch was also submitted and merged.

5.3 RQ3: Ablation Study

We perform an ablation study to evaluate the contributions of ChaCo's two key components: (Dynamic) Test Context and Feedback Component. To do this, we created the following variants of ChaCo: **ChaCo (LLM Test Context)** uses exclusively LLM to generate test context (which is the fallback mode of ChaCo when no DTC is available). **ChaCo (No Test Context)** does not perform test context analysis and removed test context in all prompts. **ChaCo (No Feedback)** removes test generation stage's feedback loop that provides the LLM with runtime error and coverage information. By default, ChaCo uses a maximum of three rounds of feedback.

Table 4 shows the results on 30 PRs (10 per project). Overall, ChaCo outperforms all ablated variants in terms of coverage increment, in all three projects. In particular, ChaCo using dynamic test context outperforms two variants (i) purely LLM-generated test context, (ii) no text context, by the same 100% in coverage increment, highlighting the importance of precise retrieval of test context thanks to dynamic analysis. ChaCo (No Feedback) performs the worst in terms of both metrics, underscoring the importance of iterative test improvement using runtime feedback. In terms of pass rate, ChaCo (No Test Context) performs the best in Qiskit and Pandas, but adding fewer lines of coverage. This suggests that while test context helps generate tests that add more coverage, it may also introduce complexity that can lead to more test failures.

The use of (dynamic) test context significantly improves the chance of increasing coverage by 100%, compared to using (i) no test context and (ii) LLM-generated test context. The latter can be very imprecise that the performance is on par with not using test context at all. Iterative feedback significantly improves the test pass rate by 312% and coverage by 460%.

6 Threats to Validity

Construct Validity. RQ2.1 relies on two reviewers acting as proxies for the original developers. These reviewers may lack the deep contextual knowledge of the project and the specific intent behind a code change that the original developers possess. Consequently, their judgment on the quality (worthiness, integration, and relevance) of a generated test might differ from that of the code's author. Moreover, these quality metrics are subjective, and standards vary among developers (e.g., how strong a test must be to catch regressions), making consistent measurement difficult. These factors may affect the validity of the findings for RQ2.1.

Internal Validity. Our approach relies on automated setup for test execution environments across multiple projects. as changes to projects' build systems may affect the reproducibility of our tool's execution. For instance, we observed that SciPy periodically updates its build process, necessitating updates to our Docker configurations. Moreover, the effectiveness of the LLM-based test generation depends on the prompt engineering and the specific model used,

Table 4: Performance of ChaCo and its ablated variants.

Method	Qiskit			Scipy			Pandas		
	Pass Rate	**# Lines**	**Cost($)**	**Pass Rate**	**# Lines**	**Cost($)**	**Pass Rate**	**# Lines**	**Cost($)**
ChaCo	28.8%	**30**	0.15	**38.7%**	**18**	0.21	19.2%	**8**	0.09
ChaCo (LLM Test Context)	29.5%	19	0.14	28.8%	2	0.15	18.8%	7	0.09
ChaCo (No Test Context)	**31.5%**	21	0.14	31.3%	2	0.19	**27.7%**	5	0.09
ChaCo (No Feedback)	7.6%	5	0.04	12.4%	2	0.06	3.5%	3	0.03

introducing potential variability in the measured test quality. Additionally, the absence of guardrails to prevent LLM hallucinations could result in the generation of irrelevant or incorrect tests. Also, ChaCo assumes that the PR is bug-free, leading it to generate only passing tests. In practice, its utility could be enhanced by integrating it with other automated testing techniques that first identify and filter out buggy pull requests.

External Validity. Our evaluation focused on only three open-source Python projects (SciPy, Qiskit, Pandas), which may not represent the full spectrum of software projects. Their codebases were likely part of the LLMs' training data, so the effectiveness of ChaCo on projects not seen during training needs further investigation. The generalizability of our results to other languages (e.g., C++, Java) or project domains remains uncertain. Furthermore, development practices regarding test coverage vary across projects, e.g., sometimes coverage gaps are ignored or intentional [4, 32], which may influence the utility of our tool.

7 Related Work

Automated Test Generation Techniques. Traditional test generation techniques remain relevant alongside LLM-based approaches. Search-based software testing (SBST) employs metaheuristic algorithms to optimize test creation for specific goals like coverage. Tools like EvoSuite [11] and Pynguin [17] generate tests by exploring code structure and exercising diverse execution paths. Similarly, DSpot [8] aims to enhance the bug-finding ability of test suites by applying custom mutations to existing tests and using mutation score as a fitness function. This line of work, in principle, can be adapted to improve patch coverage. Automated test generation spans diverse methodologies, including specification-based approaches [3], feedback-directed random testing [22], and symbolic execution-guided techniques [12]. Transformer-based methods [35] leverage machine learning advancements to enhance test generation. In contrast, we propose a PR-specific test generation approach that uses PR context to improve patch coverage.

Test Generation with Large Language Models. LLMs have reshaped automated test generation [1, 10, 14, 16, 18, 24, 25, 27, 28, 37, 40]. CoverUp [24] uses LLMs for coverage-guided test generation, targeting uncovered code regions. SymPrompt [27] introduces code-aware prompting strategies, decomposing test generation into multi-stage sequences aligned with execution paths. TestPilot [28] generates tests using function signatures and documentation, eliminating the need for additional training data. UTGen [10] integrates an LLM into SBST to enhance the understandability of generated tests. Unlike these works, we integrate dynamic analysis with LLM-based techniques to extract and utilize test context, addressing integration challenges overlooked by prior methods.

Pull Request Testing. EvoSuiteR [31] generates regression tests by comparing two versions of a Java class, while Testora [25] uses LLMs to detect unintended behavioral changes in PRs. These approaches focus on regression or behavioral testing rather than patch coverage. CTG [6] combines EvoSuite with CI, focusing tests on on class complexity and project evolution. In contrast, we focus on generating tests for uncovered lines in PRs to fix patch coverage.

Developer Perceptions of Generated Tests. Recent studies [5, 8] investigated developer perceptions of automatically generated/augmented tests. Brandt et al. [5] submitted pull requests to 39 Java projects with one DSpot-augmented [8] test each, 19 PRs were accepted. The main findings were that manual edits are often necessary for the tests to be accepted; common edits include aligning assertion styles, relocating tests, and removing unnecessary code. The first two edits are addressed by ChaCo's use of test context. UTGen [10] carried out a controlled experiment with 32 developers and shows that LLM-improved test understandability leads to better bug-fixing from developer feedback.

8 Conclusion

We presented ChaCo, a novel approach for augmenting regression tests at the PR-level by targeting the "last-mile" gap in patch coverage. ChaCo leverages LLMs, enriched with PR and test context, to generate and integrate meaningful tests that align with project-specific conventions. Our evaluation across 145 PRs from SciPy, Qiskit, and Pandas proves ChaCo's effectiveness, achieving full patch coverage in 30% of cases, discovering previously unknown bugs, and receiving positive developer feedback. With low per-PR cost and high acceptability, ChaCo offers a practical path toward automating fine-grained test augmentation in CI workflows.

Acknowledgments

This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, and 2106404, by the European Research Council (ERC; grant agreements 851895 and 101155832), and by the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073). It is also supported in part by funding from Amazon and Samsung. We thank the reviewers for their constructive feedback that helped improve the work.

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang.

2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839

[2] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *International Conference on Software Engineering (ICSE)*.

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002), 123–133. doi:10.1145/566171.566191

[4] Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. 2024. Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 157–167. doi:10.1145/3639477.3639721

[5] Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. 2024. Shaken, Not Stirred: How Developers Like Their Amplified Tests. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1264–1280. doi:10.1109/TSE.2024.3381015

[6] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/2642937.2643002

[7] François Chollet et al. 2015. Keras. https://keras.io.

[8] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2019. Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects. *Empirical Software Engineering* (2019), 1–35. doi:10.1007/s10664-019-09692-y

[9] Cleidson R. B. de Souza, Emilie Ma, Jesse Wong, Dongwook Yoon, and Ivan Beschastnikh. 2024. Revealing Software Development Work Patterns with PR-Issue Graph Topologies. *Reproduction Package for Article "Revealing Software Development Work Patterns with PR-Issue Graph Topologies"* 1, FSE (July 2024), 106:2402–106:2423. doi:10.1145/3660813

[10] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2025. *Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests*. IEEE Press, 1449–1461. https://doi.org/10.1109/ICSE55347.2025.00032

[11] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[13] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 358–368. doi:10.1109/ICSE.2015.55

[14] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMS . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1475–1487. doi:10.1109/ICSE55347.2025.00098

[15] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs] doi:10.48550/arXiv.2310.03714

[16] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering, ser. ICSE*.

[17] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. doi:10.1145/3510454.3516829

[18] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test Intention Guided LLM-Based Unit Test Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1026–1038. doi:10.1109/ICSE55347.2025.00243

[19] Node.js contributors. 2025. nodejs/node. https://github.com/nodejs/node. Accessed: 2025-12-28.

[20] NumPy Developers. 2025. data-apis/array-api-strict: Strict implementation of the Python Array API (GitHub repository). https://github.com/data-apis/array-api-strict. Accessed: 2025-12-28.

[21] OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/. Accessed: 2025-12-28.

[22] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84. doi:10.1109/ICSE.2007.37

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Effective High Coverage Test Generation for Python. arXiv:2403.16218 [cs] doi:10.1145/3729398

[25] Michael Pradel. 2026. Testora: Using Natural Language Intent to Detect Behavioral Regressions. In *International Conference on Software Engineering (ICSE)*.

[26] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. In *FSE*.

[27] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (July 2024), 21 pages. doi:10.1145/3643769

[28] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Softw. Eng.* 50, 1 (Jan. 2024), 85–105. doi:10.1109/TSE.2023.3334955

[29] SciPy community. 2025. Detailed SciPy Roadmap. https://docs.scipy.org/doc/scipy/dev/roadmap-detailed.html#test-coverage. SciPy v1.16.0 Manual. Accessed: 2025-12-28.

[30] Sentry. 2025. Codecov Quick Start. https://docs.codecov.com/docs/quick-start. Accessed: 2025-12-28.

[31] Sina Shamshiri. 2015. Automated Unit Test Generation for Evolving Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 1038–1041. doi:10.1145/2786805.2803196

[32] Alexander Sterk, Mairieli Wessel, Eli Hooten, and Andy Zaidman. 2024. Running a Red Light: An Investigation into Why Software Engineers (Occasionally) Ignore Coverage Checks. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)* (Lisbon, Portugal) *(AST '24)*. Association for Computing Machinery, New York, NY, USA, 12–22. doi:10.1145/3644032.3644444

[33] The JAX Authors. 2025. Quickstart — JAX documentation. https://docs.jax.dev/en/latest/quickstart.html. Accessed: 2025-12-28.

[34] Tian Gao. 2025. Welcome to VizTracer's Documentation! — VizTracer 1.0.4 Documentation. https://viztracer.readthedocs.io/en/latest/. Accessed: 2025-12-28.

[35] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs] doi:10.48550/arXiv.2009.05617

[36] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1258–1268. doi:10.1145/3691620.3695501

[37] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. arXiv:2408.11324 [cs] doi:10.48550/arXiv.2408.11324

[38] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831. doi:10.1145/3650212.3680323

[39] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: A ChatGPT-based Automated Unit Test Generation Tool. arXiv:2305.04764 [cs] doi:10.48550/arXiv.2305.04764

[40] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 76 (July 2024), 24 pages. doi:10.1145/3660783

[41] Jia-Ming Zhang, Zhan-Qi Cui, Xiang Chen, Huan-Huan Wu, Li-Wei Zheng, and Jian-Bin Liu. 2022. DeltaFuzz: Historical Version Information Guided Fuzz Testing. *Journal of Computer Science and Technology* 37, 1 (Feb. 2022), 29–49. doi:10.1007/s11390-021-1663-7

[42] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

[43] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. doi:10.1145/3460120.3484596